Multi-Agent Systems and Agent-Oriented Programming

Andreas Viktor Hess (s103441)
Øyvind Grønland Woller (s103447)

Kongens Lyngby 2013
B.Sc.-2013-14

DTU Compute
Technical University of Denmark
Matematiktorvet, Building 303B
DK-2800 Kongens Lyngby, Denmark
http://www.compute.dtu.dk/

B.Sc.-2013-14

Summary (English)

This thesis concerns multi-agent systems and agent-oriented programming in relation to the Multi-Agent Programming Contest (MAPC). More specifically the MAPC scenarios of 2011 and 2012, namely the Agents on Mars scenarios, and the adaptation and improvement of the 2011 winner, HactarV2, to the 2012 scenario.

HactarV2 is written in the GOAL programming language. GOAL is an agent-oriented programming language for developing rational agents. The logic programming language Prolog is used as GOAL's knowledge representation language.

Our system, which is named HARDAC, is evaluated against an updated version of the Python-DTU system from the MAPC 2012, which is the strongest system for the 2012 contest we know.

The results are positive showing that while still marginally weaker than Python-DTU, HARDAC is competitive against Python-DTU and wins close to 40% of the time.

Summary (Danish)

Denne afhandling omhandler multi-agent systemer og agent-orienteret programmering i relation til Multi-Agent Programmeringskonkurrencen (MAPC). Mere specifikt MAPC scenarierne fra 2011 og 2012, altså Agenterne på Mars scenarierne, og adaption og forbedring af vinderen fra 2011, HactarV2, til 2012 scenariet.

HactarV2 er skrevet i GOAL programmeringssproget. GOAL er et agent-orienteret programmeringssprog for udvikling af rationelle agenter. Logik programmeringssproget Prolog er brugt som GOAL's vidensrepræsentationssprog.

Vores system, som er kaldt HARDAC, er evalueret mod en opdateret version af Python-DTU systemet fra MAPC 2012, hvilket er det stærkeste system fra 2012 konkurrencen som vi kender.

Resultaterne er positive og viser at selvom HARDAC er en smule svagere end Python-DTU, så er HARDAC stadigvæk konkurrencedygtig overfor Python-DTU og vinder omkring 40% af gangene.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the requirements for acquiring a B.Sc. in Software Technology.

This project was conducted from February 4th 2013 to July 1st 2013 under supervision of Jørgen Villadsen. The thesis deals with multi-agent systems, the GOAL agent-oriented programming language and the Multi-Agent Programming Contest.

Lyngby, 01-July-2013

Andreas Viktor Hess (s103441)
Øyvind Grønland Woller (s103447)

Contents

CHAPTER 1

Introduction

A multi-agent system is a distributed system with intelligent agents capable of sensing and acting which can be used to solve problems which are difficult or even impossible to handle with traditional approaches.

The Multi-Agent Programming Contest (MAPC) is a competition that aims to stimulate research in the area of multi-agent system development and programming by providing an annual competition where multi-agent systems compete in a scenario constructed to favor using multi-agent systems. This thesis considers a multi-agent system from the MAPC 2011 scenario, HactarV2.

The goal of this thesis is to identify and improve aspects of HactarV2 to make it competitive in the MAPC 2012 scenario. The multi-agent system which is the result of the improvements is named HARDAC. To test whether HARDAC is competitive in the MAPC 2012 scenario, HARDAC will be evaluated against a strong contestant from the MAPC 2012, Python-DTU.

The thesis begins with an introduction to GOAL (the agent-oriented programming language that HactarV2 is written in) and the MAPC 2011 and 2012 scenarios in chapters 3, 4, and 5.

This is followed by an analysis of the strategies used by HactarV2 and Python-DTU in chapters 6 and 7. The analysis is concluded with a description of the improvements and strategies to HactarV2 that will be implemented in HARDAC.

After the analysis, important implementation details of the improvements and strategies will be explained in chapter 8.

Then HARDAC will be tested against Python-DTU over 18 simulations, representing 6 tournaments. The results of the simulations will be evaluated, examining more closely the effect of some of the strategies. This happens in chapter 9.

Based on the evaluation, possible future improvements will be identified and presented in chapter 10.

A reflection on the development process and discussion about the strategies is then conducted in chapter 11.

The thesis ends with a conclusion in chapter 12.

CHAPTER 2

Problem statement and learning objectives

The purpose of the project is to define, implement and evaluate a prototype of a multi-agent system using the agent programming language GOAL, available as open source software.[Lø]

The learning objectives are:

1. Understand the GOAL programming language, as well as the Mars Scenario from the MAPC version 2011 and 2012.
2. Adapt the HactarV2 system, winner of the MAPC 2011 tournament, to the MAPC 2012 scenario and attempt to improve it.
3. Evaluate our multi-agent system against the updated Python-DTU 2012 system in the MAPC 2012 scenario.

CHAPTER 3

Basics of multi-agent systems

In this chapter the basic idea of a multi-agent system is explained. It begins with a look at some definitions of agents, and a discussion of what an agent is. This concept of an agent is then expanded to describe multi-agent systems.

3.1 Agents

To understand multi-agent systems, it is important to specify the term agent. Russel and Norvig define an agent as:

> anything that can be viewed as perceiving its *environment* through *sensors* and acting upon that environment through *actuators*. [RN09, p. 34]

Wooldridge is more specific, limiting his definition to computer systems:

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its delegated objectives. [Woo11, p. 21]

Both definitions are general and not limited to artificial intelligence. However, only software agents will be considered in this thesis. Software agents are situated in an environment; are *reactive* of that environment, they *perceive* the environment and are able to respond to perceived changes; are *proactive*, they perform actions in order to achieve their goals; and they are *social*, they are capable of communicating with other agents. [Woo11, p. 26-27]

An agents *percepts* are the information about the environment that the agent is able to perceive with its sensors. An agents *actions* are the possible movements of its actuators that the agent is able to perform to manipulate its environment. Actions therefore also change the state of the instantiated agent in the environment but not just the agents mental state as the mental state of the agent is not part of the environment.

An agent's *belief base* is the set of its *beliefs*, that is, the statements that the agent believes are correct about the environment, itself, and other agents. An agent's *knowledge base* is the set of *knowledge*, that is, facts about the environment and agents. The difference between beliefs and knowledge is that beliefs can be incorrect while knowledge is always correct.

To prevent performing impossible actions and to update the mental model after an action has been performed, with the immediate effects of the action on the belief base, an *action schema* ([RN09, p. 367]) can be used. It consists of *action rules*, which consist of an action (the name of the action, including any parameters the action may have), a set of preconditions (conditions that must be met before the action can be executed), and a postcondition (effects of the action on the mental model of the agent). An action schema is a set of such rules for each possible action.

A *rational agent* is by definition an agent that acts so as to achieve the best outcome, or at least the expected outcome when there is uncertainty. [RN09, p. 4]

3.2 Multi-Agent systems

The relationship between agents and multi-agent systems (abbreviated MAS) is now clear.

> *Multiagent systems* are systems composed of multiple interacting computing elements, known as *agents*. Agents are computer systems with two important capabilities. First, they are at least to some extent capable of *autonomous action* - of deciding *for themselves* what they need to do in order to satisfy their design objectives. Second, they are capable of interacting with other agents - not simply by exchanging data, but by engaging in analogues of the kind of social activity that we all engage in every day of our lives: cooperation, coordination, negotiation, and the like. [Woo11, preface, p. xiii]

A multi-agent system is a system comprised of several agents that may cooperate, negotiate, and/or compete with each other to achieve their goals.

CHAPTER 4

Agent-oriented programming and GOAL

This chapter contains an introduction to agent programming and the GOAL programming language. A short introduction to the agent-oriented programming paradigm is followed by a more in-depth guide to the GOAL programming language, explaining what a GOAL agent is, what it consists of, how it reasons, and how it communicates. It is concluded by a short summary of important bugs present in the GOAL version used during this project.

For a short introduction to the GOAL IDE see the appendices.

4.1 The agent-oriented programming paradigm

To design agent systems, programming using the paradigm of *agent-oriented programming* can be useful. The key idea of agent-oriented programming is that agents are programmed in terms of mentalistic notions (such as belief, desire, intention) that represent the properties of agents ([Woo11, p. 55]). The design of a program is therefore centered around designing intelligent agents that have the characteristics defined in the previous section. Primarily that the agents are autonomous, reactive, proactive, and social.

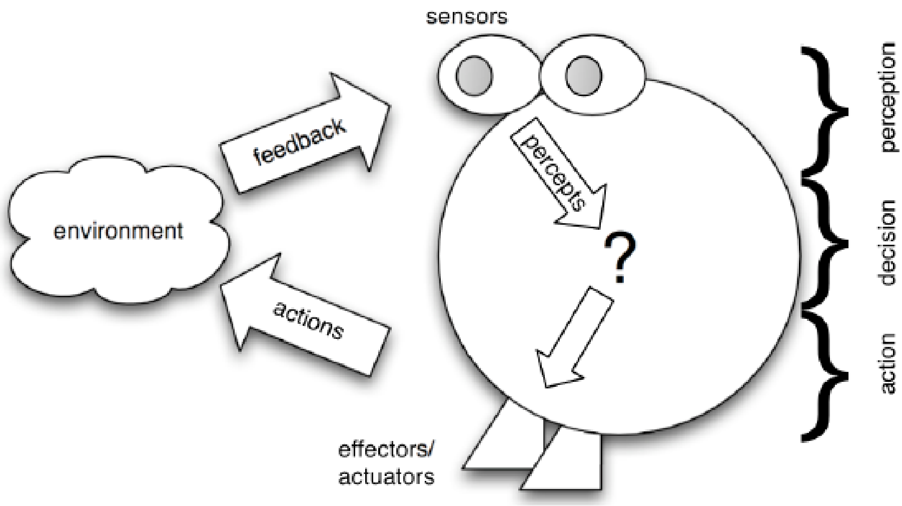


Figure 4.1: An illustration of an agent in its environment including the sense-decide-act loop. Agents perceive their environment and act accordingly. [Woo11, p. 22]

When designing agents, a useful interpretation of the agent's behavior is that the agent is in a close-coupled, continual interaction with its environment. It perceives its environment and then decides what actions to perform based on its percepts and its beliefs about the environment, indefinitely. This is called the *sense-decide-act loop* (see Figure 4.1).

The HactarV2 multi-agent system is written in the GOAL programming language. What follows is an introduction to the language.

4.2 The GOAL programming language

GOAL is an agent programming language for programming *rational agents*. GOAL agents derive their choice of *action* from their *beliefs* and *goals*. The language provides the basic building blocks to design and implement rational agents. The language elements and features of GOAL allow and facilitate the manipulation of an agent's beliefs and goals and to structure its decision-making. The language provides an intuitive programming framework based on common sense notions and basic practical reasoning. [MPI]

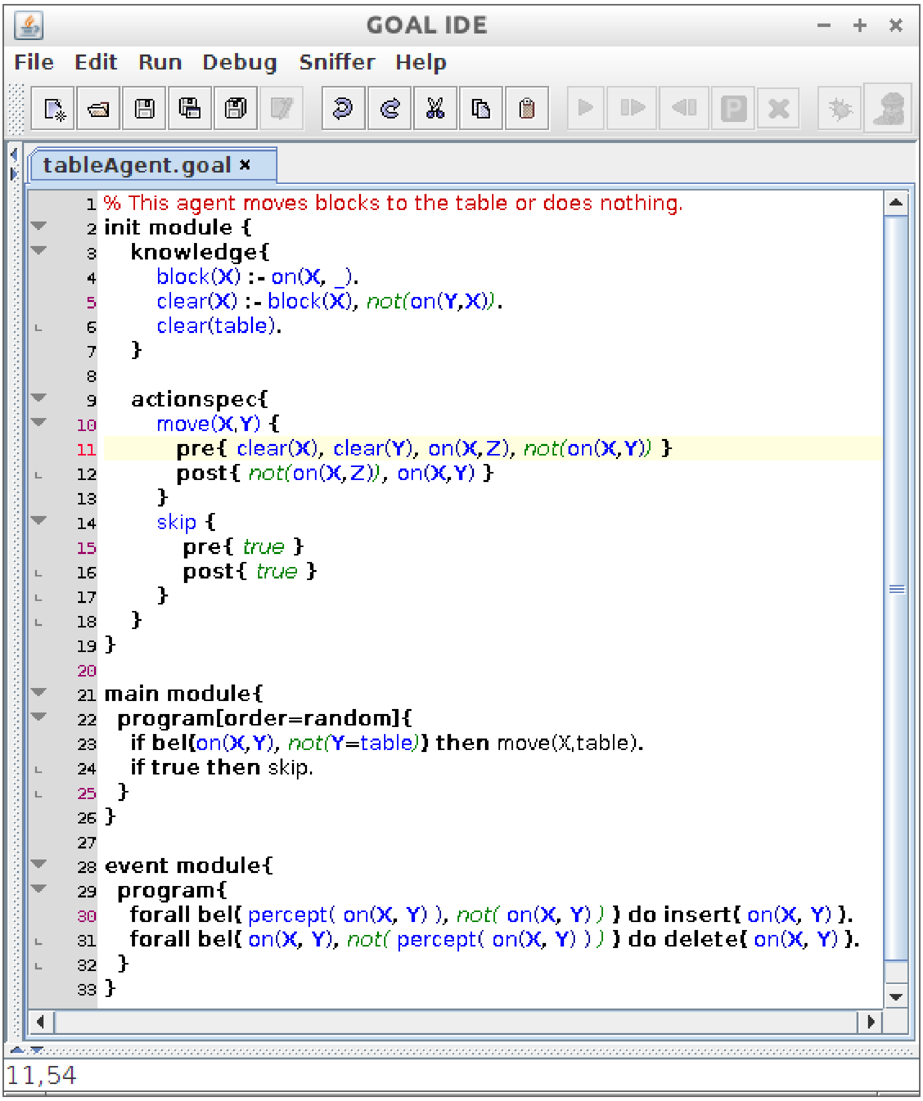


Figure 4.2: An example of GOAL's syntax. In the figure is shown a simple program for an agent in the Blocks World environment, opened in the GOAL IDE.

GOAL is a programming language for writing multi-agent systems. It is built using Java and therefore executes on the JVM. It is based on the idea that agents have *declarative goals* (i.e. states consisting of statements about themselves, the environment, and other agents) that they would like to fulfill. After a goal has been achieved it is discarded automatically.

The agents of GOAL are based on the sense-decide-act loop. In GOAL, the agents first receive percepts then decide on an action and then execute the action on the environment. The decision phase includes processing of percepts and updating the internal mental state of the agent, as well as communication with other agents in the multi-agent system run by GOAL. Communication is done by sending and receiving messages in each sense-decide-act loop.

Each agent has five databases: a belief base, a knowledge base, a goal base, a message/mail base, and a percept base. These databases together comprise the mental state of the agent. The mental state of the agent is declarative and is written using a declarative programming language such as Prolog. This language is called the *knowledge representation language* ([Hin, p. 19]). It contains atoms and predicates that the agent uses to decide on actions and update its mental state based on any new percepts received during the decide phase of the sense-decide-act loop.

There are a number of built-in actions that can be executed multiple times for each iteration of the sense-decide-act loop because they do not interact with the environment. These actions are `insert`, `delete`, `adopt`, `drop`, and `send`. Their usage will be explained in the following sections. All other actions, that are part of the environment, end the current iteration of the loop, with the result that the agent has requested the action be performed.

All agents are executed once per round, i.e. all agents in the system have finished one iteration of their sense-decide-act loop before they execute the next iteration of their loop. That is, they are synchronized at the end of the loop. This implies that all agents always execute the same number of iterations. A *round* is therefore defined as the execution of one iteration for all the agents in the multi-agent system. Furthermore, it seems that all agents are executed sequentially and deterministically (they have a randomly predefined order, such that the order of execution of the agents is always the same) when debugging ("stepping") the system.

4.2.1 Structure of GOAL programs

The overall structure of a GOAL agent program looks like Table 4.1.

A user-defined module has the syntax `module <NAME> {}` where `<NAME>` is the name of the module. Modules can have parameters after their name, e.g. `module <NAME>(X,Y) {}` where `X` and `Y` are variables that must be instan-

```
init module{ <sections> }

main module{ <sections> }

event module{ <sections> }

<other user-defined modules>
```

Table 4.1: The structure of GOAL programs (from [MPI]).

tiated when the module is executed.

Each module can contain sections such as knowledge, beliefs, goal, and program. See Figure 4.2 for an example. The contents of the knowledge, beliefs, and goal sections are added to the corresponding database of the agent when the system is executed. It is important to note that the queries to the databases can be connected by conjunction, but not disjunctions.

The program section contains the actual reasoning code that is executed when the agent enters the module. The program section consists of if <BELIEF> then <ACTION>. sentences (such that if the agent believes that <BELIEF> then it executes <ACTION>) or forall <BELIEF> do <ACTION> (such that for all substitutions the agent believes <BELIEF> the agent executes <ACTION>). <BELIEF> and <ACTION> usually contain variables that can be unified with atoms. The first such substitution triggers the $if - then$ construct while the $forall - do$ construct triggers all the possible substitutions. $forall - do$ is therefore usually used when processing percepts and messages. For example, if bel(has(X)) then use(X). means that if the agent believes that $has(\mathbf{X})$ for an atom that can be substituted by $\mathbf{X}$ then it does the action $use(\mathbf{X})$. If use is a module it enters the module use with the instantiated variable $\mathbf{X}$ instead. $if - then$ and $forall - do$ can also be nested. These constructs are imperative, in contrast to the declarative syntax when reasoning about the mental state.

The order in which the constructs are evaluated can be specified at the beginning of the section as program[order=<ORDER>] where <ORDER> can be linear, linearall, or random. linear means that they are evaluated from top to bottom until one of the conditions is applicable for instantiation or none of them are. random randomizes the evaluation order. linearall evaluates all of them, from top to bottom. The default order is linear.

A project in GOAL consists of:

1. A `.mas2g` file containing information about how to set up the environment, when to start executing the agents, and which files contain the programming for the agents (files ending in `.goal`).
2. `.goal` files containing the actual implementation of the agents. Each type of agent preferably has its own `.goal` file.
3. Optional `.mod2g` files (containing modules) and `.pl` (containing predicates and atoms), that are imported in the relevant `.goal` files using the `#import "<FILE>"` statement, where `<FILE>` is the filename.

GOAL has multiple default modules that are executed at multiple times in the lifecycles of the agents. The `init` module is executed when the agents are instantiated. Then the `event` and `main` modules are executed, in that order, in each sense-decide-act loop. The `event` module is supposed to handle any new percepts and send and receive messages from other agents in each loop while the `main` module is the actual decision phase of the agent where the agent decides on an action.

As Prolog is the language of choice for modeling the mental state of each agent, the predicates and atoms of the mental state are written using the usual Prolog syntax. For example, the knowledge for an implementation of an agent in the Blocks World environment [1] can be seen in Table 4.2.

```
% only blocks can be on top of another object.
block(X) :- on(X, _).
% a block is clear if nothing is on top of it.
clear(X) :- block(X), not( on(_, X) ).
% the table is always clear.
clear(table).
% the tower predicate holds for any stack of blocks that sits on the table.
tower([X]) :- on(X, table).
tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
```

Table 4.2: Knowledge for the Blocks World MAS.

[1] This is one of the demonstration multi-agent systems that is included in the GOAL package. GOAL can be downloaded at [MPI].

4.2.2 Mental state of agents

> A rational agent maintains a *mental state* to represent the current state of its environment and the state it wants the environment to be in. The representation of the current state determines the informational state of the agent, and consists of the *knowledge* and *beliefs* of an agent. The representation of the desired state determines the *motivational* state of the agent, and consists of the *goals* of an agent. A mental state thus is made up of the knowledge, beliefs and goals of an agent. [Hin, p. 19]

The belief and knowledge base can be accessed by the `bel` keyword. See for example line 22 in Figure 4.2.

The goals can be accessed by the `goal`, `a-goal` (short for achievement goal, `a-goal(X)` is the same as `goal(X), not(bel(X))`), and `goal-a` (short for goal achieved, `goal-a(X)` is the same as `goal(X), bel(X)`) keywords. Goals can be adopted using `adopt` and dropped by `drop`. Goals are automatically dropped when they are fulfilled.

The belief base can also be modified by the `insert` and `delete` actions, which inserts or deletes atoms in the belief base, respectively. It is not possible to modify predicates.

The negation-as-failure operator `not` can also be used on the `goal` and `bel` operators, in addition to using it on an atom or predicate.

GOAL programs are *first-order intentional systems* [Hin, p. 14]. Agents can reason about beliefs and goals but not beliefs and goals *about* beliefs and goals. So the mental content of agents can be represented by sentences such as `bel(p)` (the agent *believes* that `p`) and `goal(p)` (the agent *wants* that `p`), but not `bel(a, bel(p,b))` (agent `a` believes that `b` believes `p`). This also implies that the belief and goal operators in GOAL, `bel` respectively `goal`, cannot be nested.

Messages and percepts are accessed through the `bel` operator. Messages can be deleted using the `delete` operator, but percepts cannot be modified as they are added and removed automatically at the beginning and end of each round, respectively. This is in accordance with the sense-decide-act model as new percepts are received each round.

4.2.3 Environment and agent communication

Agents can communicate with each other using the send action. The syntax is send(<ID>, <ATOM>) where the atom <ATOM> is sent to the agent <ID>. It is also possible to use the ID allother to send the atom to all the other agents in the multi-agent system.

It is only possible to send and receive atoms, not predicates, and only one atom per send action. But it is possible to chain multiple actions together by using the + operator. E.g. send(<ID>, vertex(X)) + send(<ID>, location(Y)) sends the atoms vertex(X) and location(Y), with instantiated variables X and Y, to the agent called <ID>.

Messages are received at the beginning of each sense-decide-act loop as a received(<FROM>, <ATOM>) predicate that can be accessed through the bel operator, where <FROM> is the agent that has sent the atom <ATOM>. Any send actions executed also results in a sent(<TO>, <ATOM>), that is inserted into the message database. As messages are sent during the execution of an agent, the messages can be delayed. For example, if an agent A_1 sends a message to an agent A_2 that has already acted on the environment in the given round, the message to A_2 is delayed by one round.

The environment is initialized in the .mas2g file in a section called environment, where the actual environment interface is specified as a string. Parameters for initializing the environment can also be specified in this section. The section called launchpolicy contains information about how and when to launch agents. An agent is usually launched when there exists a necessary embodiment of the agent in the environment, an entity, that the agent can connect to and control. As an example of the environment setup, see Table 4.3 for the .mas2g file for the Blocks World MAS as seen in Figure 4.2.

Communication between the environment and the multi-agent system occurs through the Environment Interface Standard (EIS)[2].

The percepts from the environment are sent to the multi-agent system through the EIS interface. Each agent receives its own set of percepts that it is able to perceive at the given time. In GOAL the percepts are represented as percept(X) predicates where X is the actual percept from the environment.

The environment actions, those that are meant to be executed on the environment and change the state of the entity that the agent controls instead of modifying the internal state of the agent, usually have an action schema. These

[2] See http://sourceforge.net/projects/apleis/ for more information about EIS.

```
environment{
  "blocksworld.jar" .

  init[configuration="bwconfigEx1.txt"] .
}

agentfiles {
  "stackBuilder.goal" .
  "tableAgent.goal" .
}

launchpolicy {
  when entity@env do launch stackbuilder:stackBuilder, tableagent:
      tableAgent.
}
```

Table 4.3: The `.mas2g` file for the Blocks World MAS.

are called action specifications in GOAL and are written in the `actionspec` section of the `init` module. These actions are written using the usual syntax for executing actions as described in the previous sections. When these actions are executed the action is sent to the environment and the current iteration in the sense-decide-act loop ends. The result of the action is received as a percept in the next round. As an example, see the `actionspec` section in the `init` module in Figure 4.2.

4.3 Relevant GOAL bugs

As GOAL is in the alpha stage of its development there are a lot of bugs and the syntax is not finalized. Unfortunately, because of bugs and syntax changes in the latest revision of GOAL that we have tested at the time of writing (GOAL revision 5738), our system must be run using *GOAL revision 4941*[3]. Revision 4941 is not devoid of bugs however. An unfortunate and critical bug is that the agents are randomly disconnected from the environment after some time with no possibility to reconnect again. It seems to be triggered when there is high disk activity. A partial workaround that alleviates the problem somewhat is to disable writing logs to the disk from the GOAL IDE.

[3] Available at `http://mmi.tudelft.nl/trac/goal/raw-attachment/wiki/Releases/goal20120705v4941/goal20120705v4941.jar`

Chapter 5

The Multi-Agent Programming Contest

This chapter explains the Multi-Agent Programming Contest's 2012 scenario, Agents on Mars, and gives an introduction to the multi-agent systems that will be examined in this thesis.

> [The Multi-Agent Programming Contest] competition is an attempt to stimulate research in the area of multi-agent system development and programming [...]. The performance of a particular system will be determined in a series of games where the systems compete against each other. While winning the competition is not the main point, we hope it will shed light on the applicability of certain frameworks to particular domains. [MAP]

5.1 Agents on Mars scenario

It is the Agents on Mars scenarios from the MAPC competitions in 2011 and 2012 that are relevant in this project. For both scenarios the main task of the agents is to find the best water wells and occupy the best zones of Mars. Some-

Role	Actions	Energy	Health	Strength	Visibility range
Explorer	`skip`, `goto`, `probe`, `survey`, `recharge`, `buy`	35	4	0	2
Repairer	`skip`, `goto`, `parry`, `survey`, `repair`, `recharge`, `buy`	25	6	0	1
Saboteur	`skip`, `goto`, `parry`, `survey`, `attack`, `recharge`, `buy`	20	3	1	1
Sentinel	`skip`, `goto`, `parry`, `survey`, `recharge`, `buy`	30	1	0	3
Inspector	`skip`, `goto`, `inspect`, `survey`, `recharge`, `buy`	25	6	0	1

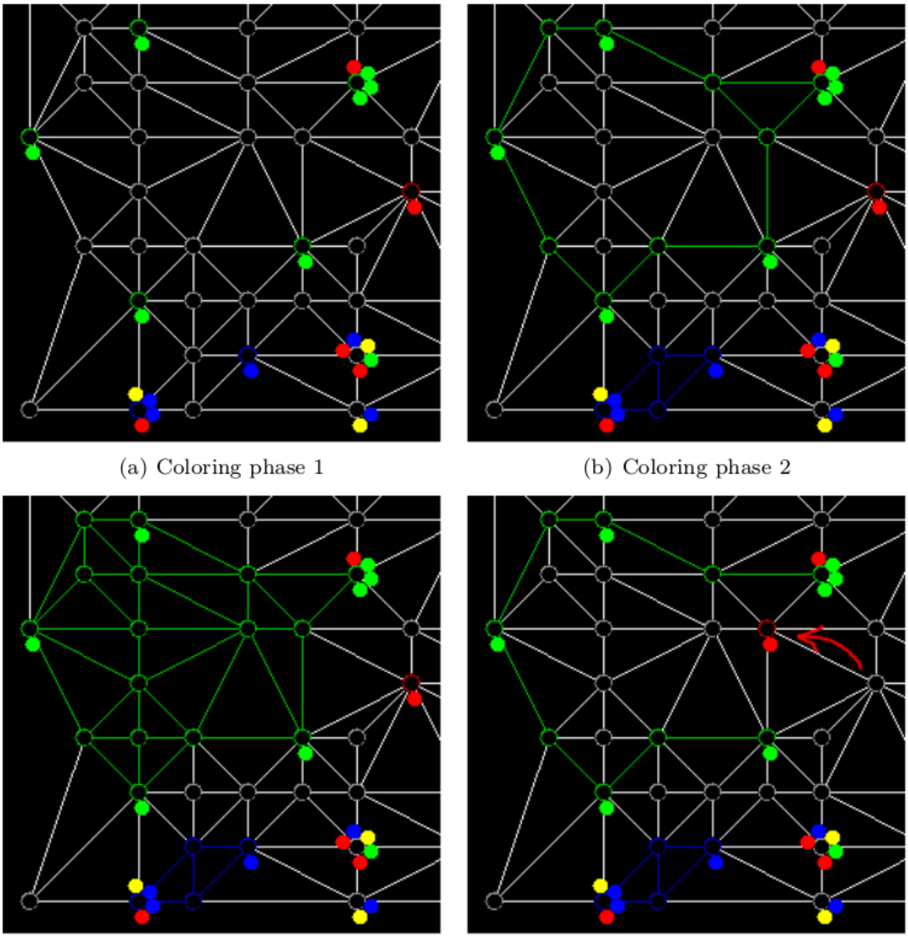
Table 5.1: The different roles in the Agents on Mars scenario for the MAPC 2012. Adapted from [BKS+, p. 6].

Action:	`attack`	`parry`	`goto`	`probe`	`survey`	`inspect`	`buy`	`repair`
Cost:	2	2	*Travel cost*	1	1	2	2	2

Table 5.2: Action cost for the MAPC 2012 scenario Agents on Mars. [BKS+, p. 6-7].

times they have to sabotage their rivals to achieve their goal (while the opponents will most probably do the same) or defend themselves. When an agent is sabotaged (i.e. its health drop to zero) then it is disabled and it is only allowed to execute the actions `goto`, `repair`, `skip`, and `recharge` (the recharge rate is set to 10 %). Of course the agents' vehicle pool contains specific vehicles. Some of them have special sensors, some of them are faster and some of them have sabotage devices on board. Last but not least, there are the repair agents, that are capable of fixing agents that are disabled, i.e. have been sabotaged. In general, each agent has a special expert knowledge and is thus the only one able to perform a certain action. So the agents have to find ways to cooperate and coordinate themselves. The different vehicle roles, their attributes, and possible actions are shown in Table 5.1. The agents are able to execute their respective actions only if they have the necessary amount of energy. Action costs are shown in Table 5.2.

In Agents on Mars the environment is represented by a graph. Vertices denote



(a) Coloring phase 1

(b) Coloring phase 2

(c) Coloring phase 3

(d) Breaking a frontier.

Figure 5.1: An illustrated example of the first three phases of coloring. [BKS+, p. 3]

water wells of different value and are possible locations for the agents. The weights of the edges denote the costs of traversing the edge. In order to score points the agents have to control zones. A *zone* is a subgraph that is colored in one's team's color. The coloring algorithm follows 4 phases, see Figure 5.1 for a visual example of the first 3 phases.

1. **Phase 1.** A vertex is given the color of the team which has the majority of agents standing on it.

2. **Phase 2.** Vertices which are neighbors to two previously colored vertices(of the same color) are colored.

3. **Phase 3.** If part of the map is separated by one teams colored vertices, the separated area is colored in that teams color.

4. **Phase 4.** If all of one teams agents are disable, the opposing team colors all the vertices on the map.

Each round the team scores points based on the values of the nodes in the zone it controls. This score will be referred to as the *zone score*. However, the team needs to have an agent of the Explorer role probe a vertex in order to receive points equal to the node's value. Otherwise it receives one point for that node. It is also possible to score points each round through achievement points. These achievements are acquired when a team reaches certain milestones, e.g., having attacked enemy agents 10 times, or probed 20 vertices, etc. Each achievement gives two achievement points. These points count for two normal points every round. This will be referred to as the *achievement score*. The achievement points can also be spent on upgrades that will give the agents an edge over the opponent, but then the team no longer receives the 2 points each round.
The goal of the game is to maximize the score. The map is unknown in the beginning, so it is necessary to explore the area first before attempting to control zones. [MAP]

The total score for each team is calculated as

$$\texttt{score} = \sum_{s=1}^{\texttt{steps}} (\texttt{zones}_s + \texttt{money}_s)$$

where `steps` are the number of steps in the simulation, $\texttt{zones}_s$ is the zone score at step s, and $\texttt{money}_s$ is the achievement score at step s.

The environment is supplied with the MAPC package [BKS+] as a server that must be executed to start the simulation. A Unix shell script called `startServer.sh` in [BKS+] can be used to execute the server where it is also possible to choose between different teams and simulations. It utilizes EIS to communicate between the MAPC server and the multi-agent system. This interface is called EISMASSIM. The file `eismassim-2.0.jar` is the environment that GOAL must load. To connect to the server the multi-agent systems must be authorized by means of a username and password specified in one of the configuration files for the server and the configuration file for EISMASSIM called `eismassimconfig.xml`.

The environment also has a deadline for each step, from when the agents receive their percepts to they send their action requests and the server receives them. If an agent does not send an action in time before the deadline is reached then the agent does not perform an action that step. So this should be avoided. The deadline can be changed through `eismassimconfig.xml`

Between 2011 and 2012 the Agents on Mars scenario underwent some important changes. The number of agents on each team doubled from 10 to 20. And most importantly the distribution on the value of nodes changed. In 2011 the high value nodes would always be in the center of the map, and therefore it was important to find and control the center. It also meant that once the Explorers found a higher value node, it was certain to be near the center, so the team could focus its efforts on that area. In 2012 this changed. The 2012 graph generator randomly distributes a random number of the highest value (10) nodes, and then "blurs" the areas surrounding these nodes. That is, the neighbors of the highest value nodes have a value less than 10 and their remaining neighbors have an even smaller value. This continues until the value of the remaining nodes are 1. It then flips the graph symmetrically. Considering the values of the vertices as a height map, the topography in 2011 was guaranteed to always be a single hill, whereas in 2012 it can be anything from a mountain range to two solitary hills in an otherwise flat environment. This poses a lot of challenges, but this will be discussed in the analysis chapter.

The simulation state transition is as follows: [BKS$^+$, p. 9]

1. collect all actions from the agents,
2. let each action fail with a specific probability,
3. execute all remaining `attack` and `parry` actions,
4. determine disabled agents,
5. execute all remaining actions,
6. prepare percepts,
7. deliver the percepts.

So attacks and parries have higher priority than the other actions. This implies that it is impossible to run from an attack without being hurt. Also note that disabled agents are determined before any repair actions are executed. This is important as it can be exploited, which will be described in the analysis section.

After the actions have been executed the percepts for the next round are prepared and delivered for each agent. The percepts include: the current state of the simulation, team, and vehicle (i.e. the embodiment of the agent); all the visible edges, vertices, and vehicles; any probed vertices, surveyed edges, and inspected vehicles. We refer to [BKS+, p. 8] for the complete listing of the percepts.

5.2 The Python-DTU and HactarV2 systems

In this project we have developed a multi-agent system, called HARDAC, based on the multi-agent system HactarV2[1]. HactarV2 is a multi-agent system written in GOAL developed in 2011 by students from the Delft University of Technology, Netherlands. It won the Multi-Agent Programming Contest in 2011.

The system that we are evaluating HARDAC against is Python-DTU[2], written in Python. It was developed by students from the Technical University of Denmark for the 2012 MAPC competition. This system reached second place. It is important to mention that the Python-DTU code used in this thesis is an updated version[3] of the team that reached second place in the 2012 MAPC competition. The update consists of a small change in the buying strategy that prevents Python-DTU from overreacting when a single enemy Saboteur buys a lot of upgrades. With this update the Python-DTU team is the strongest multi-agent system developed for the MAPC Agent on Mars scenario that we know of.

As there are several challenges to overcome to be competitive against the other multi-agent systems in the MAPC (e.g. which upgrades to buy and when, how to choose zones to control, if the system should be aggressive or defensive), an overall overview of the behavior of system can be made by identifying the problems and the solutions implemented in the system. A solution to any one of these problems will henceforth be referred to as a *strategy*.

[1]HactarV2 can be downloaded from `http://multiagentcontest.org/downloads/Multi-Agent-Programming-Contest-2011/Sources/HactarV2_Code.zip/`.

[2]Python-DTU can be downloaded from `http://multiagentcontest.org/downloads/Multi-Agent-Programming-Contest-2012/Sources/Python-DTU/`

[3]A diff of the changes can be found in the appendices.

CHAPTER 6

Analysis of MAPC 2012, HactarV2, and Python-DTU

This chapter begins with a description of the small changes needed to make HactarV2 run on the MAPC 2012 server. It is followed by a short analysis of the MAPC 2012 scenario map. Then the limitations of HactarV2, as a multi-agent system, caused by the messaging system are discussed. The chapter is concluded by a thorough analysis of the strategies used by HactarV2, and comparisons to relevant strategies used by Python-DTU.

Before beginning to analyze, it is important to define a few terms. The use of the word optimum varies slightly between the MAPC scenario and HactarV2. The MAPC scenario defines optimum as any vertex of the highest value, in this case 10. HactarV2 however uses optimum as the term for the vertex it will "swarm" around. In this thesis it will be used to mean any vertex of the highest value.

Swarm is used by HactarV2 to denote the group of agents that constitute the area around the optimum. However it is used by Python-DTU and the scenario to mean any group of agents that are controlling an area. This is the meaning it will have in this project. Swarming is the behavior of any agent who is in a swarm.

6.1 Agents on Mars 2012 scenario maps

In the 2012 MAPC scenario, three different map sizes are used. The largest map has 300 vertices, the second largest 240 vertices, and the smallest map has 200 vertices.

More interesting is the change in distribution of vertex values. In the 2011 scenario the high valued nodes were always clustered in the middle of the map. In 2012 however the graph generation has a random chance of creating an optimum at any give node. Once it has iterated over all possible nodes, it "blurs" the areas around the placed optimums, decreasing the value by some amount the further away it gets from the optimum. It then flips the map symmetrically across the vertical axis, making the left and right sides equal but opposite. The agents are distributed in much the same way, so that if one team starts with a Saboteur in the top left corner of the map, then the opposing team has a Saboteur in the top right corner of the map.

Considering the varying vertex values as a height map, the Agents on Mars scenarios can be viewed topographically. Seen this way the 2011 distribution was a single hill, whereas in 2012 the distribution can be anything from a hill, to a mountain range, to two solitary peaks. See Figure 6.1 and Figure 6.2 for examples of the difference between the 2011 and 2012 scenarios. This is actually a big obstacle for HactarV2. It has implications for the most important of HactarV2's strategies, as will be seen in this chapter.

6.2 The messaging system of HactarV2

It is important to discuss the messaging system HactarV2 uses, and how it limits the possibilities of HactarV2 as a multi-agent system. As explained in the GOAL programming language section of chapter 4, the agent's mails are not synchronized before deciding on an action, leading to agents receiving mail that is one round old.

This messaging system prevents many options for coordination. It prevents the agents from agreeing on actions before performing them. This means that it is not possible to prevent agents from moving to the same vertex with the same purpose. As an example, several Explorers will often move to the same unprobed vertex with the intention of probing it. They will not realize the redundancy until they are standing on the same vertex at which point only one will actually probe, effectively wasting a turn for the other Explorers that

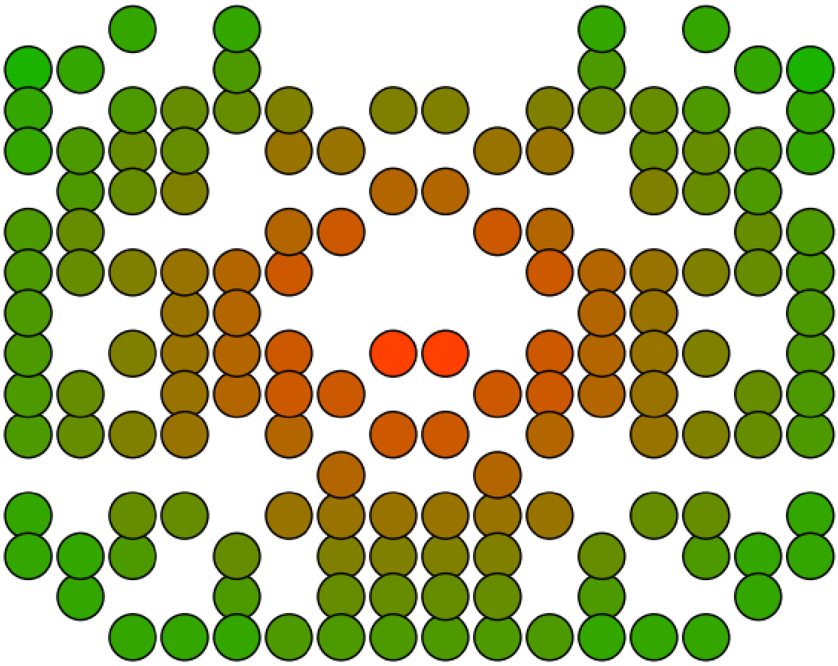


Figure 6.1: An example of the topography of a 2011 scenario map. A red color indicates a higher value than a green color. The red vertices in the middle therefore have value 10 while the green vertices at the corners have a value of 1.

moved to the vertex.

This is also a problem for decision making when attempting to control a zone, as it often leads to two agents leaving their vertex to make the zone bigger, which results in them losing their connection to the zone. This makes the zone smaller, and forces the agents who moved out of the zone to move back, wasting their turn and causing HactarV2 to lose points because their zone is smaller.

Besides the limitations on communication, the messaging system is also very computationally heavy. Sending more than a bare minimum of messages slows the system down too much to be able to make the deadline imposed by the server.

Python-DTU does not suffer from these problem. At the beginning of every step, all agents handle perceptions and mail any new beliefs to the other agents. This way, all agents have the same knowledge before they begin deciding on

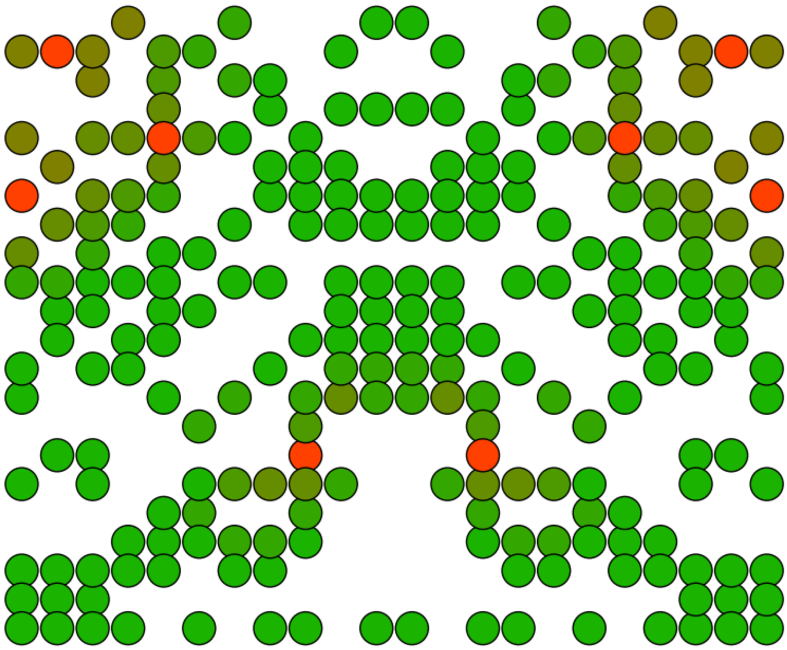


Figure 6.2: An example of the topography of a 2012 scenario map. A red color indicates a higher value than a green color. The completely red vertices have value 10 while the completely green vertices have a value of 1.

an action, allowing much greater levels of cooperation. To decide which agent does what, Python-DTU uses an auction based negotiation which prevents several of the cases mentioned above. For example, Explorers will never move to the same unprobed vertex with the intention of probing and agents in their swarm collectively decide where to stand.

6.3 Analysis of HactarV2

The general idea of HactarV2 is to use hill-climbing[1] for the Explorers to quickly find the center of the map, and the optimum area. Once this is found the location is sent to all other agents, and they move towards the optimum until they reach the optimum zone they control. When they reach the zone all the agents swarm around the optimum until the end of the game.

This strategy will not work as intended in the 2012 scenario, as it results in HactarV2 deciding the first optimum it finds must be *the* optimum, and the center of the map, and begins to swarm around it. Due to the updated scenario, where the highest valued nodes are placed randomly, this can be anywhere on the map. Therefore the area chosen by HactarV2 is rarely the best area, and will usually swarm around a lower valued area than Python-DTU. Occasionally HactarV2 is lucky and picks a good zone which is far enough away from Python-DTU to be left alone. However, even in the best case, HactarV2 will never be able to win against Python-DTU.

6.3.1 Probing

As mentioned HactarV2's Explorers use hill-climbing to search for the highest value nodes. The hill-climbing algorithm itself functions very well, but the Explorers stop probing once an optimum node has been found. When an optimum is found the Explorer sends a mail to all other agents telling them to converge on the optimum and begin to swarm. At this point the Explorers will only probe the area around the optimum. This is a consequence of the general strategy of finding the center of the map, as mentioned above. This strategy leaves HactarV2 with very few probed vertices, very little knowledge about the map, and fewer achievement points.

HactarV2's Explorers are also programmed to always survey a vertex first, before performing any other action. This is backwards, as probing first may give a few extra points the next round, whereas surveying first yields no advantage. This is a small point, however, the fact that Explorers survey at all may be unnecessary. Traversing an edge which has not been surveyed costs the same amount of energy as traversing one where the weight is known. Therefore, it may be worthwhile to completely remove surveying from Explorers.

[1] That is, continuously moving towards a higher-valued vertex until no such vertex exists.

6.3.2 Swarming

HactarV2 uses a swarming strategy once it has decided which highest value node to control. The basic algorithm for an agent in the swarm is to consider its position and possible neighboring positions for expanding the swarm. Depending on the calculations the agent will move to one of these neighboring vertices, or stay where it is. The algorithm only allows agents to expand the swarm to vertices which are not owned by either team. This can lead to an unfortunate circumstance where all of HactarV2's agents become trapped inside of Python-DTU's swarm. When this happens, it counts as Python-DTU separating the rest of the map from HactarV2, resulting in Python-DTU owning nearly all the vertices on the map. The swarm has other issues as well.

There are two more reasons the swarm is not very effective. First the agents often become confused about where they should stand, sometimes moving to the same node as another HactarV2 agent, or moving too far away from the swarm. When this happens they are told to move back towards the optimum and try again. This causes the swarm to be very volatile and unstable. Moving around like this means that HactarV2 controls fewer vertices than it has the opportunity to.

Secondly, a single enemy Saboteur can be enough to heavily disrupt HactarV2's swarm. When an agent in the swarm feels threatened by enemy agents it runs away, and unless there is an allied Saboteur nearby, the enemy Saboteur will eventually drive the whole HactarV2 swarm away. Another way it disrupts the swarm is by causing *large battles* (see Figure 6.3. Large battles happen when all of HactarV2's Saboteurs move to the same node as the enemy Saboteurs. This causes Python-DTU's and HactarV2's Repairers to come to the node. When many Saboteurs and Repairers gather on a node, HactarV2's poor targeting allows the enemy agents to stall all of HactarV2's Saboteurs and Repairers.

When these battles happen inside of HactarV2's zone, the vertex where the battle happens is often occupied by equal numbers of allied and enemy agents, causing HactarV2 to lose control of the node, and through the coloring algorithm, possibly other nearby nodes as well. It also means that the distance from the battle to other HactarV2 agents is short, allowing Python-DTU to disable large portions of HactarV2's swarm while the Repairers slowly repair each other and the Saboteurs at the site of the battle. These scenarios cause HactarV2 to lose a lot of points over the course of the simulation.

The result of the cases above is shown clearly on the zone stabilities statistic generated by the monitor (see Figure 6.4).

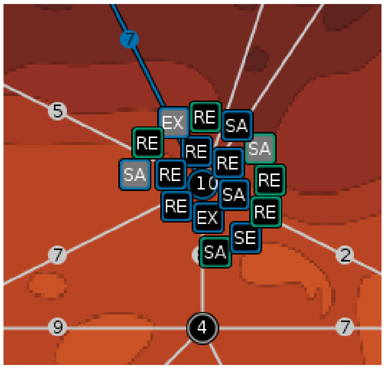


Figure 6.3: An example of a large battle

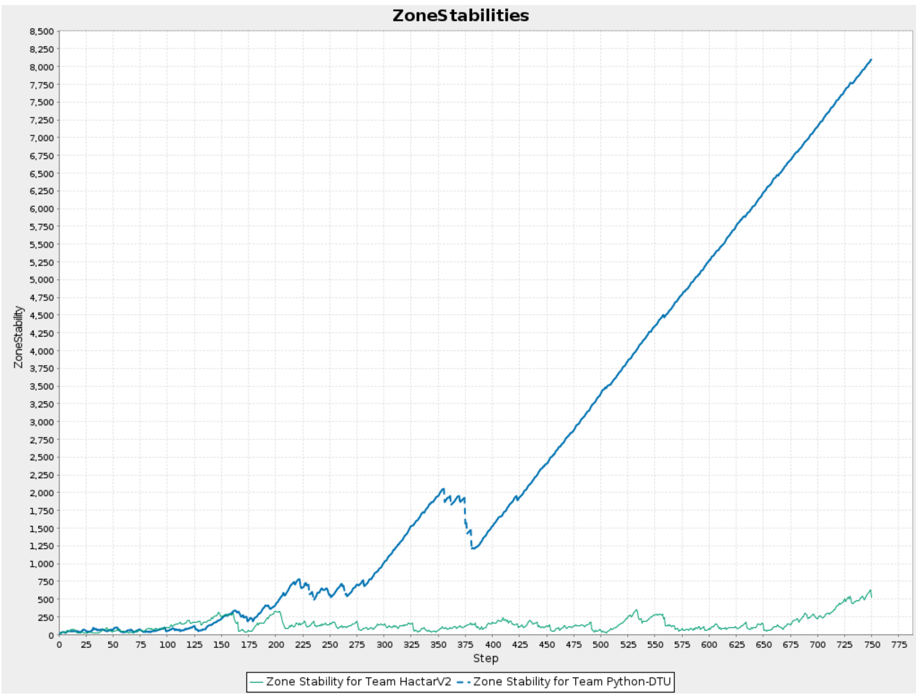


Figure 6.4: The zone stabilities graph output from the MAPC 2012 server showing HactarV2 zone stability versus Python-DTU zone stability.

6.3.3 Repairing and attacking

Something that was noticed when examining simulations, is that when there are several Repairers, or Saboteurs, on the same node, they end up choosing the same target. As an example, if there were three Repairers on a node, and three disabled agents, all three Repairers would attempt to repair the same agent, instead of them repairing one each. The same happens when there are several enemy agents on a node with several of HactarV2's Saboteurs. This is a limitation of the current HactarV2 code, which they did not address in the 2011 scenario. We have noticed that versus Python-DTU this is actually a pressing issue for HactarV2's swarm, both when attacking and defending. Especially as when a large battle arises the Repairers congregate on the battle vertex and become trapped, indefinitely repairing disabled agents on the node. This causes problems as it allows enemy Saboteurs to completely destroy HactarV2's swarm while the Repairers and Saboteurs do nothing to stop it.

6.3.4 Buying

HactarV2 uses a very aggressive buying strategy. As soon as it has achievement points to spend, it begins to upgrade its Saboteurs with strength and shields. The advantage gained does not seem to be worth the cost however, as being stronger than Python-DTU for the first 150 steps does not seem to make a difference in the overall outcome. Also, the cost is quite large, often HactarV2 has spent around 12 achievement points in the first 20 steps. These achievement points spent add up to several thousands of points lost during the simulation. (see Figure 6.5)

6.3.5 Superiority

When HactarV2 controls more than a certain percentage of the map, it activates its superiority strategy. In this strategy each Saboteur is assigned a Repairer that it is to hunt down and follow. This should prevent the opposing team from ever getting back into the game, while HactarV2's other agents explore the map for more achievement points. This state will never be entered versus Python-DTU.

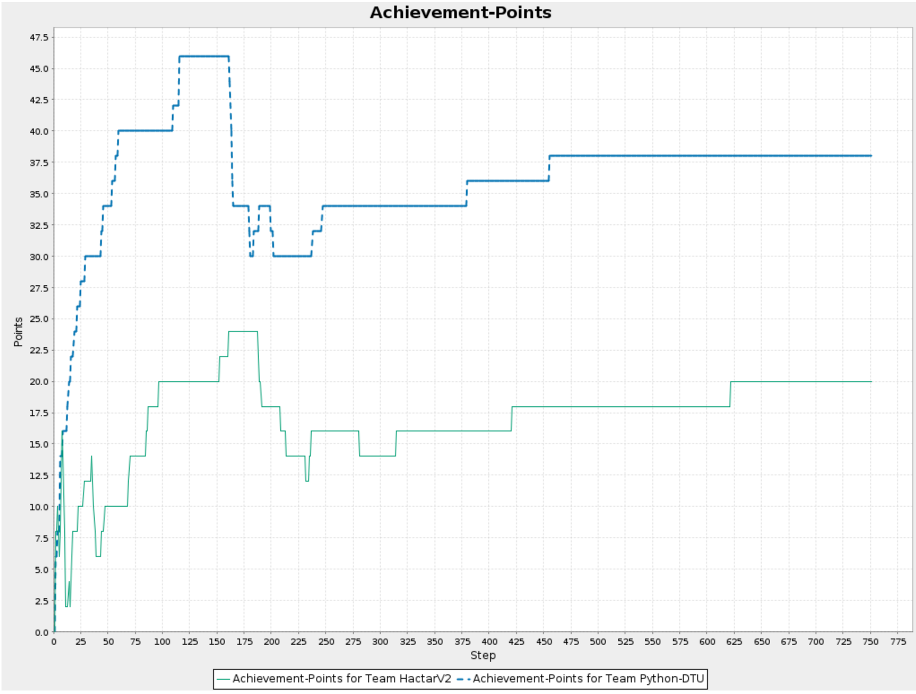


Figure 6.5: The achievement points graph showing HactarV2's aggressive buying strategy and its consequences.

6.3.6 Bugs and needed updates

Separate from these strategies are a couple issues that should be addressed. First is the fact that some of the algorithms used, e.g. for deciding whether or not to parry, expect only two agents of each role. The MAPC 2012 server also returns more precise feedback for failed actions, which needs to be updated in the HactarV2 code.

The other bug is in the Inspector code. The bug causes them to behave poorly in the swarm, not keeping to their positions, but rather moving towards enemy Saboteurs. Looking at the statistics we see that our Inspectors inspect around 1000 times per simulation. However, teams only receive achievement points for how many of the enemy teams agents have been inspected, this is a big waste of time.

This behavior often results in the Inspectors getting caught up inspecting only a few enemy agents over and over. This often leads to not inspecting all of the enemy teams agents, which reduces the number of achievement points received, as well as causing other agents to move, usually in fear of enemy Saboteurs, due to not knowing the role of some nearby enemy agent.

6.3.7 Storing information

In the beginning of a simulation the agents have very little information about the map causing one severe issue. When one of HactarV2's agents is disabled early on in the simulation, there is a high probability that it will not know of any paths to allied Repairers. Looking for solutions it was discovered that the agents receive a lot of information about their surrounding area during the map, which HactarV2 does not store. Storing this information and sending it to allied agents would help reduce the probability of not having a path to an allied Repairer. This will not be enough in all cases, so a secondary solution is needed. One possibility would be to try to survey as much of the map as quickly as possible with the Sentinels, spreading them out to cover as large a portion of the map as possible. It would be difficult to avoid them moving towards each other, as the agents don't know how the map is connected. A simpler solution would be to have the agents play more cautiously during the early stages of the simulation. This would not remove the problem entirely, but would hopefully make it less dangerous.

6.4 Relevant strategies from Python-DTU

6.4.1 The greedy algorithm

Returning to Python-DTU's solution regarding finding the best zones. Python-DTU runs a greedy algorithm on all the nodes in the map. It begins by calculating an approximate best optimum area, and then calculates approximate best locations for their agents to stand, taking into account the coloring algorithm used by the environment. The agents then decide through an auction-based negotiation who will go where. This results in Python-DTU selecting, not perfect, but very good vertices to stand on so that the zone they control is large. This strategy will not always be optimal, but in practice it works very well.

6.4.2 Probing

Python-DTU's Explorers use a random exploration algorithm for finding vertices to probe. They never survey, instead they make sure that if the edge is unknown they do not attempt to traverse it without 9 energy (the max weight of an edge). Their random probing solution works well because they probe for 200 steps, allowing them to probe a lot of vertices. Probing for the first 200 steps allows Python-DTU to probe a far greater number of vertices than HactarV2. However, HactarV2's hill-climbing algorithm should be more efficient at finding high valued nodes quickly than Python-DTU's random search.

6.4.3 Buying

Python-DTU uses a less aggressive buying strategy than HactarV2. It waits until step 150 before using knowledge gained from inspecting the opposing team's Saboteurs to decide whether or not to buy upgrades, and how many. Python-DTU uses the second highest enemy Saboteur strength, and second highest enemy Saboteur health, to decide when to buy upgrades. This is the update that was made after the MAPC 2012. Previously Python-DTU considered the highest enemy Saboteur health and strength, when deciding to buy upgrades. This was exploited by the winners of the MAPC 2012 competition by buying lots of upgrades on a single Saboteur, causing Python-DTU to spend up to four times as many achievement-points on upgrades as the winners.

Both waiting until step 150 and using knowledge of the second highest enemy

Saboteur health and strength seem to be good solutions. The 150 step timeout may not be optimal, but very little would be gained by adjusting it. Therefore, Python-DTU's solution will inspire the solution for HARDAC.

6.4.4 Repairing and attacking

Another point is how Python-DTU uses its Saboteurs. When Python-DTU agents detect enemy agents around an area they control, they request help from a Saboteur. Python-DTU then sends a single Saboteur to disable HactarV2's agents. And if Python-DTU discovers an enemy swarm it will send a Saboteur to disrupt it.

Examining Python-DTU's swarm it is noticeable that when swarming, the Repairers will move to the disabled agents if they are far away, whereas the agent will move to the Repairer if it is nearby. This allows Python-DTU to have a greater zone stability, and waste less time moving disabled agents around.

Chapter 7

Possible Strategies for HARDAC

Based on the analysis in the previous chapter, there are several improvements that could be implemented and a few bug fixes that have to be implemented. The bug fixes are for mistakes in the original code, such as role names being spelled incorrectly, and a few fixes for the new server. For example the server now returns more precise messages when an action fails, so a few lines in the original code must be replaced. In broad categories the improvements deal with messaging, probing, swarming, buying upgrades, attacking, repairing, and defending.

7.1 Messaging

The messaging system could be rewritten so that all agents process their percepts and send any mails before any agent begins to decide what action to take. This would be very beneficial, also for implementing many other strategies, but it requires a complete rewriting of the original messaging system and would probably require a large restructuring of the main agent logic. Therefore it will not be improved as a part of this thesis. It will be discussed in the extensions chapter as a possibility for future development, as well as the possibilities it opens for new or improved strategies.

7.2 Probing

An important strategy to improve is probing. First of all, the Explorers need to deal with having multiple optimums. Having HactarV2 swarm around a the first found optimum, which likely has low valued surrounding vertices, is not good enough. Therefore the Explorers should continue to probe the map, even after they have found one optimum and have a new way of handling optimums. Handling multiple optimums could be handled by finding any probed vertices with value 10, instead of having a single Explorer send information on "the" optimum.

The hill-climbing algorithm used by HactarV2 works well. However, once an Explorer has found an optimum, hill-climbing is no longer useful as the agent is already at the top of the hill. There are two clear possibilities; start to probe nearby unprobed vertices randomly, or start probing the area surrounding the optimum. Given enough time the random search should cover all the vertices, but high-value vertices near optimums may be missed. To get the most out of the positions HARDAC chooses for its agents, probing the surrounding area should be the better solution and will be the solution used in HARDAC. The efficiency of the Explorers can also be improved by stopping them from surveying.

7.3 Swarming

Swarming could be improved in several ways. HactarV2's single swarm is often a bad solution. This is because a vertex's value decreases the further from an optimum it is. Therefore agents swarming far from the optimum would probably contribute more points, each round, by swarming around a separate optimum instead. Another possibility for improvement would be to make the swarm less frightened of enemy agents. Having agents that can stay and parry on their vertex instead of running away would make the swarm more stable. To further control the erratic movement, agents who believe they have found a good position to stand on, could be told to stay there for a certain number of turns. The agents should also be allowed to move to vertices that are controlled by the enemy team, not just unoccupied vertices. Finally, the agents should not all move if there is an enemy agent on a node. For example, there are situations where four HactarV2 agents are standing on an optimum, along with an enemy agent. All four HactarV2 agents conclude that they should move to expand the swarm. This causes HactarV2 to lose control of the optimum, so next turn they all move back to the optimum. This continues indefinitely or until the enemy agent moves.

To prevent this, HARDAC could take into account the number of enemy agents on a node, and leave enough agents to keep control of the node. A timeout could also be used to prevent erratic movement in the agents once they have found a good place to stand.

Another solution would be to use Python-DTU's greedy algorithm and have a single agent calculate positions and send a position to each swarming agent. This could potentially make HARDAC's swarm equally as good as Python-DTU's. Hopefully this would then allow improvements in other areas to tip the balance in HARDAC's favor. The advantages of this solution is that we know it works, and would take of the problems of erratic movement. HARDAC will use a Prolog implementation of Python-DTU's greedy algorithm.

7.4 Buying upgrades

The buying strategy needs improvement, both for conserving points, but also in choosing upgrades. HactarV2 uses too many achievement points, for little gain in the early stages of the simulation. One partial solution would be to only buy strength to conserve achievement points. However, this would leave HARDAC's Saboteurs very vulnerable to enemy Saboteurs. Using a time-out of some number of steps would solve the problem. Python-DTU waits 150 steps before buying upgrades. Waiting longer may not be safe, as it would leave HARDAC weaker than Python-DTU for some number of steps. The simplest option is to use the same time-out as Python-DTU, which should put HARDAC on equal footing with Python-DTU.

Using the opponents second strongest Saboteur's upgrades to decide what improvements to buy would prevent the updated system from falling into the same trap that Python-DTU did during the MAPC 2012 competition. Some guidelines for choosing upgrades may be necessary as Python-DTU might be the first to buy, in which case they will buy strength first. If this happens, HARDAC would buy health, which would cause a feedback loop ending in HARDAC's Saboteurs having a lot of health and no strength, while Python-DTU has a lot of strength. This is a problem as it would prevent HARDAC's

Saboteurs from being able to disable any enemy agent in a single attack, while Python-DTU would be able to.

7.5 Attacking

The targeting of the Saboteurs could be improved heavily by coordinating attacks when on the same node as other allied Saboteurs. A solution that does not rely on sending messages is necessary. A possibility would be using the agent name to determine which agent does what. A priority list of which agent role to attack would also be useful for ensuring that high-priority targets are disabled.

To prevent large battles, the Saboteurs could consider the number of, allied and enemy, Saboteurs and Repairers that are on their node and move away if they are not needed. To prevent the large battles from happening in our zone, it would also be beneficial to send some Saboteurs to the enemy swarm to harass it. This should force some of the enemies' Saboteurs back to defend, as well as disrupt the enemy swarm, and help keep HARDAC's swarm safe. This requires that HactarV2's Saboteurs stop swarming, and instead focus on keeping Python-DTU occupied.

7.6 Repairing

As with Saboteurs, Repairers could be greatly improved by coordinating repairs when on the same node as other allied Repairers. And because of the order that actions are handled by the server, namely repairs occur after attacks, we can improve them even further. By knowing that repairs happen after attacks, and knowing Python-DTU's attack priorities, the Repairers could attempt to predict which allied agent on a node will be attacked that turn, and perform a repair on that agent the same turn(see Figure 7.1). In this way allied agents can avoid becoming disabled even though they are attacked. This makes it possible to effectively fight battles with fewer Saboteurs than the opponents. Needing fewer Saboteurs in large battles allows the surplus Saboteurs to move around and attack other enemy agents, disrupting their swarm.

To deal with getting stuck in large battles, there are a couple of options. It would be possible for the Repairers to calculate whether or not they are in a large battle, and move away. This may not be ideal as if there are no allied

(a) The Saboteurs will attack each other this turn. Knowing this, HARDAC's Repairer will preemptively repair HARDAC's Saboteur.

(b) HARDAC's Saboteur is not disabled, whereas Python-DTU's Saboteur is, allowing it to attack Python-DTU's Repairer.

(c) Again the Saboteurs will attack each other, causing HARDAC's Repairer to preemptively repair HARDAC's Saboteur.

(d) Both Python-DTU agents are disabled, while both HARDAC agents are fully operational.

Figure 7.1: An example showing how the new preemptive repairing works, with Python-DTU(green) and HARDAC(blue) in an even battle with one Repairer and one Saboteur each, and the result.

disabled agents, then the Repairers would be better off helping in the battle. They could also make repairing an agent a goal, to force them to commit to repairing a disabled agent, while still allowing them to participate in the battle when they are not needed elsewhere.

7.7 Defending

Knowing the priorities of Python-DTU's Saboteurs can also be used to improve the defense algorithms of HactarV2. If an agent is standing on the same node as a Python-DTU Saboteur, it is possible to calculate whether or not it might attack the agent. This can be used to prevent running away, needless parrying, and for calling for help if the swarm is under attack. While in the swarm agents that can parry should stay on their node when an enemy Saboteur comes and parry for its attacks for as long as possible. This allows allied Saboteurs the chance of coming to the rescue and disabling the enemy Saboteur before it disrupts the swarm.

CHAPTER 8

Implementation

In the first half of this chapter different ways to control an agents behavior are described. In the second half, the implementation details of strategies chosen for HARDAC are described.

For the complete program listing see the appendices.

8.1 Controlling the behavior of agents

The behavior of the overall system must be consistent such that the behaviors of the agents are determined by the state of the simulation and the teams. Several methods have been utilized to realize this goal. The use of these methods will be explained in more detail in the relevant sections later on in this chapter.

8.1.1 Timeouts

For some strategies timeouts have been used to determine if it is time to perform some specific behavior. These timeouts are for the most part triggered when certain steps in the simulation are reached or if certain knowledge about the match has been obtained.

These triggers have been chosen and tweaked so as to be competitive against Python-DTU. Some of the triggers therefore also coincide with those used by Python-DTU. However, it is important to note that the triggers of HARDAC do not exploit any shortcomings Python-DTU may have, so they should also be competitive against other opponents when battling on the MAPC 2012 scenario.

In HARDAC there are timeouts for deciding when to decide on swarms (`timeToDecideSwarm`), when to swarm (`timeToSwarm`), when to disrupt enemy swarms (`timeToHarass`), when to hunt an enemy Saboteur that is attacking HARDAC's swarms (`timeToHunt`), and when to buy upgrades (`timeToBuy`).

8.1.2 Agent ranks

```
% 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), !,
  findall(Agent, visibleEntity(Agent,Here,Team,normal), Agents),
      agentRank(Agents,Name,Rank).

% An agents rank (i.e. index) in the list List
agentRank(List,Agent,Rank) :- nth0(Rank, List, Agent), !.
```

Figure 8.1: An example a predicate computing the rank of an agent on the current vertex; `agentRankHere/1`.

Sometimes, to prevent performing large computations unnecessarily or for optimizing behavior, agents can order the agents on the team by a rank where the rank depends on the situation. These ranks are usually used to determine which target to attack or repair to prevent attacking or repairing the same target, and who needs to calculate the swarming positions.

8.1.3 Messages

Messages are expensive and are delayed by one simulation step, so they are kept to a minimum and used only for synchronizing certain beliefs and when requesting services from other agents. Sometimes they are also necessary, however, such as when sharing relevant information about the simulation state between the agents. A compromise between sending messages and performing complex calculations for each agent therefore had to be reached.

Messages for new vertices are sent to each agent when found, not only when a vertex has been probed or surveyed as in the HactarV2 system. This dramatically reduces the performance in the first couple of steps of the simulation because a large amount of vertices are visible at each step. The advantage is that paths between vertices are found significantly earlier than otherwise.

Messages for synchronizing and updating the status (i.e. health, name, and position) of allies and enemies are also shared between the agents as this information is used for deciding when to do certain behavior.

8.1.4 Goals and predicates

Goals in GOAL are useful for forcing agents to commit themselves to reach a desired state of the simulation or until a certain condition is met. Goals are used in HARDAC for swarming, repairing, harassing, and probing.

Sometimes a predicate is used instead to determine if a certain condition is met. This serves a similar purpose as goals but may be easier to implement or has already been implemented in HactarV2 and refactoring would be a waste of time.

8.1.5 Predicting other agent's behavior

If the relevant beliefs are consistent between the agents of HARDAC then each agent can potentially predict the behavior of the other agents. This is exploited by the Saboteurs when choosing targets to attack so that when multiple Saboteurs are at the same vertex they will not attack the same target if it is unnecessary. It is also used by Repairers for the same reason, i.e. delegating tasks for the Repairers at a vertex. This is achieved in each agent by doing the same calculations and then choosing targets depending on the rank of the agent.

8.2 The strategies used by HARDAC

8.2.1 Buying upgrades

```
module upgrades{
  knowledge{
    shouldBuyStr(S) :- enemySaboteurSecondMaxHealth(Health), S < Health, !.
    shouldBuyStr(S) :- me(Me), hasLowestRoleRank(Me), S < 6, !. % At least one Saboteur should be able to kill anybody in one round.
    shouldBuyHP(H)  :- enemySaboteurSecondMaxStrength(Strength), H =< Strength, !.
  }
  program{
    if bel(timeToBuy, not((enabledEnemyHere(ID), dangerousEnemy(ID))), strength(S), maxHealth(H), money(M), M >= 4) then {
      if bel(shouldBuyStr(S)) then {
        if true then buy(sabotageDevice).
        if true then recharge.
      }
      if bel(shouldBuyHP(H)) then {
        if true then buy(shield).
        if true then recharge.
      }
} } }
```

Figure 8.2: The buying module for Saboteurs.

The buying strategy of HARDAC has been made similar to the buying strategy used by Python-DTU. The current strategy is to:

1. Only buy upgrades for the Saboteurs, and only buy health and strength upgrades.

2. Ensure that the health of HARDAC's Saboteurs is always one more than the enemy's strength. This ensures that the enemy cannot disable HARDAC's Saboteurs in one step.

3. Consider the health and strength of the enemies Saboteurs before buying upgrades. HARDAC only buys upgrades if the second highest health and strength among the enemy Saboteurs is higher (or equal in the case of the enemy's strength) than HARDAC's Saboteurs, using predicates called `secondMaxHealth` and `secondMaxStrength`. This ensures that the enemy cannot trick HARDAC into buying upgrades for all its Saboteurs while the enemy only buys for one Saboteur.

4. Use a timeout to prevent buying upgrades early in the simulation. This timeout is called `timeToBuy` and is constructed such that the Saboteurs first considers buying upgrades after at least 140 steps have passed.

8.2.2 Probing

The motivation for this strategy is the fact that Python-DTU probes a lot more than HactarV2 and therefore is able to determine better nodes to control for its zone score.

To reduce this discrepancy, HARDAC's Explorers do not survey unless they have probed the whole map, and the behavior have also been changed significantly.

The Explorers of HARDAC go through three stages when probing the map:

1. **Finding optimums.** This stage is controlled by using a goal, `optimum`, that is achieved when the Explorer itself has probed a vertex of value 10.
2. **Probing the area around the found optimum.** This stage is controlled by a timeout. When it is not time to swarm and the `optimum` goal has been achieved, the agent will probe the area around the optimum so the potential zone score can be calculated for when the team later on chooses to swarm. The Explorers calculate a list of vertices around their respective optimum that have not been probed and then probes all the vertices in the list. This is accomplished by using a predicate called `needExploring/1`, where the variable is the list in question, that is updated at each step.
3. **Probing the rest of the map.** When it is not time to swarm and all nodes in the `needExploring/1` predicate has been probed, the Explorer will find any vertices left on the map that is not probed and then probe them.

After the three stages have been completed, or the `timeToSwarm` timeout is reached, the Explorers will swarm.

8.2.2.1 Hill-climbing to find optimums

The hill-climbing algorithm from HactarV2 for finding optimums has been kept in HARDAC because it will also find the highest valued vertices in MAPC

2012. The implementation is in the `searchOptimal` module of `explorer.mod2g`. It works by probing a random neighbor of the current vertex and then advancing to that neighbor if it has a higher value. Otherwise it goes back to the previous vertex and tries another neighbor. If no higher-value vertices are neighbor to the current vertex then the current vertex must be an optimum.

8.2.2.2 Probing around optimum

```
calculateNeedExploring(L) :- swarmPosition(MOpt), MOpt \= unknown,
    findall(V1,(member(O,Opts), neighbour(O,V1), needProbe(V1)), A),
    findall(V2,(member(N,A),    neighbour(N,V2), needProbe(V2)), B),
    append(A,B,C), append(Opts,C,D), sort(D,L).

updateNeedExploring(A,B) :- findall(V, (member(V,A), needProbe(V)), B).
```

Figure 8.3: Predicates used for updating and calculating the list contained in the `needExploring` predicate.

When an Explorer has found an optimum it must change its behavior as hill-climbing no longer suffices because it is already at the top of the hill. We have chosen to probe all vertices surrounding the found optimum.

The Explorers uses a predicate `needExploring/1` to ensure that they probe around a certain radius of the optimum that they have found. This ensures that a zone value can be approximated for each optimum when it is time to decide where to swarm. See also the above explanation of `needExploring/1`.

This behavior is implemented in the module `searchPostOptimal` in `explorer.mod2g`.

8.2.2.3 Probing afterwards

After having probed all vertices around the optimum, the Explorer will probe the rest of the map until it is time to swarm. This is also implemented in the module `searchPostOptimal`.

8.2.3 Attacking

HactarV2's Saboteurs have some deficiencies that we have focused on improving. These deficiencies of HactarV2's Saboteurs are:

1. **They do not delegate targets among themselves**, thereby not preventing them from attacking the same enemy.
2. **They do not reason about the state of the swarms**, both the enemy's swarms and their own team's swarms. It is important to disrupt the enemy swarms and prevent the enemy from disrupting our swarms. Otherwise they can gain a considerable advantage.

8.2.3.1 Keeping track of the enemy

```
if bel(agentRankHere(0), enemyTeam(T), me(Me), currentPos(V), findall([
    E,P,X], (visibleEntity(E,P,T,X), not(enemyStatus(E,P,X))), L), L \=
    []) then {
  forall bel(agent(ID), ID \= Me, statusUser(ID), not(visibleEntity(ID,
     V,_,_))) do send(ID,enemyStatusPack(L)).
```

$20 \cdot 4$, and not $20^2 \cdot 4 \cdot N_i$, i.e. one for each enemy

Figure 8.4: How the agents inform each other about the status of the enemy. Note how the agents send a list of `enemyStatus` rather than one message per `enemyStatus`. This reduces the penalty associated with messaging. From `common.mod2g`.

The known states of the enemy agents are sent to the Saboteurs (and Repairers, see the section about repairing) at the beginning of each step in a predicate called `enemyStatus/3`. It contains the ID, the position, and the status (disabled, normal) of the enemy agent. The Saboteurs need to know the location and status of enemy agents such that they can reason about where the enemy is swarming and if the enemy Saboteurs are a potential threat to HARDAC's swarms.

Alternatively, the agents themselves could request tasks from the Saboteurs, resulting in fewer messages sent. But this would require writing additional code, thereby raising the complexity, and synchronizing the status of the enemies is also relevant for Repairers.

For performance purposes, all the visible enemy agents are sent to the Saboteurs in one message instead of multiple messages. Furthermore, the messages are sent only if the agent believes the relevant Saboteur cannot see the enemies visible to the agent. So the maximum number of messages sent each round about the status of the enemy is reduced significantly. See Figure 8.4.

8.2.3.2 Handling large battles

```prolog
% If there are N-1 ally Saboteurs and N enemy Saboteurs at a vertex then we should not go there because we are not needed (i.e. if not(notLargeBattle))
largeBattleCalculator(V,AN,EN,AL) :- findall(EID, (enemyStatus(EID,V,_), dangerousEnemy(EID)), EL), !, findall(AID, (teamStatus(AID,V,_), role(AID,'Saboteur')), AL), !, length(EL,EN), length(AL,AN).
largeBattle(V,AL) :- largeBattleCalculator(V,AN,EN,AL), AN >= EN, AN \= 0, EN \= 0, !.
notLargeBattle(V) :- largeBattleCalculator(V,_,0,_), !.
notLargeBattle(V) :- largeBattleCalculator(V,AN,EN,_), ANPlusUs is AN + 1, ANPlusUs < EN, !. % AN+1 to prevent us from creating a large battle
```

Figure 8.5: How large battles are identified. From `roleKnowledge.pl`.

A vertex V is said to contain a large battle if the number of ally Saboteurs on V $\geq$ the number of enemy Saboteurs on $V > 0$. This definition assumes that the Saboteurs of each team are evenly matched, which is a reasonable assumption.

If a vertex is determined to contain a large battle, it may be useful to move a Saboteur away from the battle. When HARDAC detects a large battle the Saboteur with the lowest rank among the Saboteurs at the vertex will try to either harass an enemy swarm, find an enemy on a neighboring vertex to attack, or move to a random neighboring vertex, in that order.

To prevent the Saboteur from moving back into the large battle, it determines if moving to the vertex would create a large battle, and if true, will not move to that vertex.

These changes hopefully forces some Saboteurs to disable swarming enemies instead of fighting indefinitely at the same vertex.

The predicates in HARDAC are called `largeBattle/2` and `notLarge Battle/1`. `largeBattle` has as parameters the vertex and the number of allied Saboteurs at the vertex, as this information is needed to determine if the Saboteur is of the lowest rank among the Saboteurs at the vertex. `notLargeBattle` also has the vertex in question as parameter. The reason for the predicate notLargeBattle, instead of utilizing the negation-as-failure operator in GOAL on largeBattle, is that notLargeBattle also considers if the Saboteur is moving to the vertex and if its presence in the immediate future will create a large battle.

8.2.3.3 Harassing enemy swarms

```
% A possible harassment vertex is a high-value vertex that is owned by the enemy and therefore probably contains a swarm
timeToHarass :- me(Me), hasLowestRoleRank(Me), step(N), N > 60.
possibleHarassVertex(Pos) :- findall(V, (enemyStatus(EID,V,normal), not(inspectedEnemy(EID,'Saboteur')), not(inspectedEnemy(EID,'Repairer'))), notLargeBattle(V)), L), L \= [], randomElement(L,Pos), !.

% If we have defeated the enemy near the harass vertex then the harass is over.
harass(V) :- (currentPos(V) ; (currentPos(P), neighbour(V,P))), not(enemyStatus(_,V,normal)), not((neighbour(V,N), enemyStatus(_,N,normal))), harassStart(S), step(Cur), N is Cur - S, N < 75.
harass(V) :- harassStart(S), S \= 0, step(Cur), N is Cur - S, N > 50, N < 75, vertex(V,_,_).

% When the enemy is disabled, the hunt is over
timeToHunt :- me(Me), hasLowRoleRank(Me), not(hasLowestRoleRank(Me)), step(N), N > 100.
hunt(ID) :- enemyStatus(ID,_,disabled).
```

Figure 8.6: The timeouts for controlling when the harass and hunt should start, and the predicates controlling when they have been achieved. The timeouts also takes into consideration if the Saboteur has the correct rank. From roleKnowledge.pl.

For disrupting enemy swarms, a goal called harass(V) has been created, where V is a vertex. It seems that Python-DTU predominantly uses Inspectors, Explorers, and Sentinels for swarming. The rest of the agents, Repairers and Saboteurs, do not as they repair and attack respectively.

So for determining a vertex to harass, which is hopefully a place with a swarm, we have chosen to choose a random vertex with an enemy Inspector, Explorer, or Sentinel whose position do not contain a large battle.

If the harassing Saboteur is occupied with attacking enemy Saboteurs and Repairers, in the worst case while fighting in a large battle, the Saboteur would probably never move towards its harass vertex to achieve its goal. Harassing therefore has a higher priority than nearly anything else, with the exception being buying upgrades. Of course, running from a battle includes a risk of becoming disabled. Only one Saboteur is therefore allowed to harass.

The `harass` goal is achieved (i.e. the predicate `harass/1` is true) when the enemies at the harass vertex and its neighbors have been defeated. It is also achieved when 50 steps of the simulation have elapsed from when the harass goal is adopted. To implement that, an atom which records at which step the harassment strategy has begun, called `harassStart/1`, is inserted when adopting the goal.

Goals are achieved when their respective predicate is true, in this case if the `harass/1` predicate is true. An important point is that goals in GOAL cannot be adopted unless the predicate is false. So we have decided that after 75 steps have elapsed, the predicate is false. This means that the agent has 25 steps to drop the goal when it is achieved, which should be done automatically by GOAL when the goal is achieved anyway. It of course cannot adopt a new harass goal in this interval as the predicate is true.

To prevent confusion, the agent is only allowed to have at most one harass goal at any time.

8.2.3.4 Delegating targets

To prevent the Saboteurs from attacking the same targets, the targets are delegated amongst the enabled Saboteurs by rank. As all enemies on a vertex are visible to all the Saboteurs at vertex, it is possible to sort the enemies and choose a target in the resulting list by order. This ensures that the Saboteurs do not choose the same targets.

Furthermore, the targets are sorted by their role as not all roles are equal. The ordering is as follows, in descending order of importance: Saboteurs, Repairers, Explorers and Inspectors, Sentinels.

Sentinels have lower priority than Explorers and Inspectors, because they are able to parry. The actual sorting is performed by the built-in `sort/2` predicate.

As an example, if two Saboteurs, S_0 and S_1 with rank 0 and 1 respectively, are at a vertex where the sorted list of targets is $[E_0, E_1]$ then S_0 will attack E_0 and S_1 will attack E_1.

8.2.3.5 Rescuing swarms from attacks

When an enemy Saboteur is at the same vertex as a swarming HARDAC agent, then one of the Saboteurs will attempt to remove the threat. That is, if a known enemy Saboteur E is (probably) attacking an ally Inspector, Explorer, or Sentinel, then the Saboteur with the second lowest rank among all the Saboteurs will adopt a goal, `hunt/1`, with E as the parameter. If there are multiple such E's then it will choose one at random.

As with the harassment strategy the `hunt` goal has a high priority, and the Saboteur with this goal will disregard all other possible course of actions than to reach and disable the enemy. Buying upgrades, however, has a higher priority. Of course, if the agent is disabled itself, it will not attempt to fulfill its goal until it is repaired again. Unless a path to the enemy in question does not exist, in which case the goal will be dropped.

The goal is accomplished when the enemy has been disabled. For determining this, the `enemyStatus/3` predicate is used.

8.2.4 Repairing

```
roleSelectRepairTargetHere(Target) :-
    me(Me), agentEnabledRoleRankHere(Me,Rank),
    roleSortedDisabledHere(RL), nth0(Rank,RL,Target).

roleSortedDisabledHere(L) :-
    findall(ID,(disabledAllyHere(ID),role(ID,'Saboteur')),SLTmp),
    findall(ID,(disabledAllyHere(ID),role(ID,'Repairer')),RLTmp),
    likelyTargets(TLTmp), sort(TLTmp,TL),
    findall(ID,(disabledAllyHere(ID),role(ID,R),not(member(R,['Saboteur
        ','Repairer'])))),OLTmp),
    findall(ID,damagedAllyHere(ID),DLTmp),
    sort(SLTmp,SL), sort(RLTmp,RL), sort(OLTmp,OL), sort(DLTmp,DL),
    append(SL,TL,Tmp), append(Tmp,RL,Tmp2), append(Tmp2,OL,Tmp3),
        append(Tmp3,DL,L).
```

Figure 8.7: The implementation of the target-selection process for repairing.

Two major issues with the way HactarV2 handles repairing is that:

1. Multiple Repairers on the same vertex could accidentally repair the same agent, which is unnecessary.

2. Repairers will sometimes repair agents on the current vertex that they are standing on indefinitely, ignoring any nearby disabled agents. This happens when enemy Saboteurs and Repairers are fighting against HactarV2's Saboteurs and Repairers at the same vertex.

8.2.4.1 Delegating targets on vertex

To solve the first problem, the Repairers should somehow be able to delegate repair tasks among the Repairers on the same vertex. As messages are a big performance hit and delayed by one round, communication between the Repairers is not feasible. Instead the Repairers of HARDAC predict the behavior of the other allied Repairers on the same vertex, thereby making it possible to avoid repairing the same agent twice.

As it is only possible to repair agents on the same vertex as the Repairer, and because all agents can perceive all other agents on the same node, the exact same calculations can be performed, resulting in the exact same output for each Repairer. The solution in HARDAC therefore consists of building a list of agents to repair, sorting this list, and then choosing a target depending on the rank of the agent among the allied Repairers at the vertex. Of course, if there are more Repairers than repair targets available and the agent was not able to select a unique target, then it will continue its program and not repair any agents on the vertex.

The list is constructed such that disabled Saboteurs and Repairers are prioritized higher than other agents. The Repairers also consider repairing agents that are hurt but not disabled.

This list is created by the `roleSelectRepairTargetHere/1` predicate in `repairer.mod2g`.

8.2.4.2 Predicting enemy attacks

Frequently the Saboteurs and Repairers of each team will converge to the same vertex where they will fight until the end of the simulation. It is difficult to escape if the teams are evenly matched as attack actions are processed before goto actions by the server. By the same reason disabling Repairers may seem like the most beneficial course of action, because this would prevent the Repairers from repairing. But attacking and disabling all enemy Repairers in one step requires a large amount of strength upgrades to be bought beforehand as the Repairers have at least 6 health and the Saboteurs have at least 3 strength. Even if successful, the enemy could do the same or disable all Saboteurs which would accomplish mostly the same; preventing at most 4 agents from repairing or attacking for one step.

As repair actions are processed after attacks, a Saboteur can attack, become disabled by being attacked, and be repaired in one step. Python-DTU seems to favor attacking Saboteurs so repairing enabled Saboteurs that the Repairers successfully predict will be attacked would give HARDAC an advantage over Python-DTU. Unless the prediction was wrong, in which case the Repairers would have wasted energy and time *if* the Saboteurs were at full health, i.e. not damaged or disabled, before the repair action is processed.

The implementation is based on the same predicate, `roleSelectRepairTargetHere/1`, with the difference being that this predicate now accounts for non-disabled ally Saboteurs that are likely to be attacked.

The complete listing of `roleSelectRepairTargetHere/1` can be seen in Figure 8.7.

`likelyTargets/1`, also seen in Figure 8.7, creates a list of enabled ally Saboteurs at the vertex if there are at least as many enabled enemy Saboteurs on the vertex as enabled ally Saboteurs. Sorting the lists ensures that they are identical across all the Repairers at the vertex. Appending them fulfills the purpose that the agents at the beginning of the resulting list `L` have a higher priority than the rest.

One case that the above does not consider, is if a disabled Repairer selects itself for repairing. Then no other Repairer at the vertex would repair the disabled Repairer and it would still be disabled the next step. However, fixing this quirk would require predicting what target the other Repairers choose, which would make the predicate more complex than it already is.

8.2.4.3 Committing to repair an agent

To prevent the second point in the table above, the Repairers can adopt a goal called `repairing(ID)` where `ID` is a disabled ally not on the same vertex as the Repairer. Choosing an agent to repair is done at random, but agents that can parry are weighted double. This is because those agents should be able to survive longer after being repaired.

Three of the Repairers will only commit themselves to repairing an agent if they have nothing else to do. They will also only repair the agent if it is close by. This prevents most of the Repairers from sporadically leaving spots where agents are frequently disabled. The lowest ranking Repairer, when ranking by role, works differently. If it detects that it is at a large battle then it will commit to repair an agent regardless of the current situation around it. By only allowing one Repairer to leave the large battle, HARDAC hopefully prevents the enemy from gaining a significant advantage. The lowest ranking Repairer, when ranking by role, will even go towards the disabled agent regardless of the distance between them.

With this goal the enemy should hopefully not be able to destroy whole swarms, because the disabled agents would be repaired faster than otherwise.

8.2.5 Swarming

The swarming behavior of HactarV2 had to be changed significantly to address the following problems:

1. There are multiple optimums in the MAPC 2012 scenario as opposed to the single optimum in MAPC 2011. This implies that swarming probably has to be spread out around multiple optimums, creating the need for multiple swarms.

2. Swarming agents in HactarV2 are easily disturbed by the enemy and frequently moves around, breaking the swarm and decreasing the zone score. The agents have to remain calm to be competitive against Python-DTU.

As the Saboteurs and Repairers are frequently preoccupied with attacking and repairing, the only agents able to swarm in HARDAC are the Inspectors, Explorers, and Sentinels. This results in a total of 12 swarming agents.

8.2.5.1 Calculating swarms

HARDAC utilizes the same algorithm as Python-DTU for calculating swarms, as this algorithm has proved sufficient for generating valuable swarms in the MAPC 2012 competition.

function CALCSWARMS
$Opts \leftarrow$ BESTOPTIMUMS()
$chosen_{opt} \leftarrow$ CALCAREACONTROL(*opt*) for all $opt \in Opts$
$opt_{best} \leftarrow \text{argmax}_{opt \in Opts} \left\{ \sum_{v \in \text{CALCOWNED}(chosen_{opt})} value(v) \right\}$
The swarms are then decided by CALCAREACONTROL(opt_{best}).
end function
function CALCAREACONTROL(*opt*)
Place an agent $a \in A$ on *opt*
$chosen \leftarrow \{opt\}$
for all $\alpha \in A \setminus \{a\}$ **do**
$owned \leftarrow$ CALCOWNED($chosen$)
$best \leftarrow$ BESTPOSITION($chosen, owned$)
Place α on $best$
$chosen \leftarrow chosen \cup \{best\}$
end for
return $chosen$
end function

Figure 8.8: Pseudo-code of the swarming algorithm (CALCAREACONTROL) from Python-DTU. A is the set of agents that should swarm, $value(v)$ is the value of vertex v. In HARDAC CALCSWARMS is used.

The algorithm is a simple, greedy algorithm. It works by trying all possibilities for placing one agent at a time on the map, and calculates the position that results in the best outcome. It cannot place multiple agents on the map at the same time, which is unfortunate as this would result in better swarms. Due to time constraints, the algorithm has not been improved significantly. However, the implementation in HARDAC calculates swarms from multiple optimums and selects the best swarm from these possibilities, which could in some cases result in a higher-value swarm.

See Figure 8.8 for an overview of the algorithm. See the appendix for the actual implementation in GOAL used by HARDAC.

An explanation of the additional functions in Figure 8.8 is as follows ($value(v)$ is the value of the vertex v and $neighbors(v)$ are the set of vertices that are connected to v by an edge):

- BESTOPTIMUMS(): calculates $value_{opt} = \sum_{v \in N} value(v)$, and $value_{max} = \text{argmax}_{opt \in Opts} \{value_{opt}\}$, where N is the set of neighbors and neighbor's neighbors to the optimum, for all optimums $opt \in Opts$. It then finds all *opt*s for which $value_{opt} \leq limit$ where $limit$ is the nearest integer to $0.65 \cdot value_{max}$.

- CALCOWNED($chosen$): computes a set of the vertices who have at least two neighbors $n_1, n_2 \in chosen$ where $n_1 \neq n_2$ and then unions it with $chosen$, as these are exactly the vertices that can be owned by HARDAC assuming that HARDAC only has agents on the vertices in $chosen$.

- BESTPOSITION($chosen, owned$): computes and returns

$$\text{argmax}_{v \in V \setminus chosen} \left\{ value(v) + \sum_{w \in neighbors(v) \setminus owned} value(w) \cdot N \right\}$$

  where V is the set of all vertices and $N = |neighbors(w) \cap chosen|$. So BESTPOSITION finds a vertex that is not chosen yet such that the sum of its value and the neighbors w that will be owned after v is chosen is maximal. It has two peculiarities however:

  1. It is possible for v to be in $owned$.
  2. N can be larger than 1 so the value of each neighbor w of v that are connected to at least one vertex in $chosen$ are weighted by its connections, but the zone score provided by w will, of course, never be higher than $value(w)$.

  Both of these properties contribute to more resilient swarms as it promotes redundancy by sacrificing a potential higher zone score. However, it remains to be tested in HARDAC if the absence of these properties would be useful.

To prevent multiple agents from computing the swarms and to ensure that all the agents are positioned correctly, the only agent computing the swarm is the Explorer with the highest role rank amongst the Explorers.

After the swarm has been computed, the Explorer will message the other agents with their swarm position.

Only the agents that should actually swarm at the time of computation are considered. This is important as not all agents should swarm at the same time. See the timing for swarming below.

8.2.5.2 Swarming behavior

When the agents have received their swarm position, they will adopt the `swarm` goal. With this goal, the agents will move towards their swarm position. This goal never succeeds, so as to ensure that the agents are swarming for the rest of the simulation.

To prevent unnecessary movements, the agents will not flee from an enemy when they are standing at their swarm position. However, they will defend themselves, and with the hunting strategy for Saboteurs described in the previous section, help should hopefully reach the swarming agents in time before most of the swarm has collapsed.

8.2.5.3 Timing

```
timeToDecideSwarm :- decidedSwarmAt(OldS), step(NewS), D is NewS - OldS
    , D >= 60, !.
timeToDecideSwarm :- swarmPosition(unknown), !, optimum(X),
    calcZoneValue(X,V), V >= 70.

timeToSwarm :- not(role('Explorer')), swarmPosition(Opt), Opt \=
    unknown, !.
timeToSwarm :- role('Explorer'), optimum(_), step(Cur), Cur > 150.
```

Figure 8.9: The timeouts for the swarm strategy.

The timeout `timeToSwarm` is used to determine when the agents should swarm. It differs from the Explorers and the rest of the agents; the Explorers do not swarm for an extended period of time after the other agents are swarming. This is to ensure that the Explorers can complete most of their post-optimal probing phases, thereby locate better swarming positions if any are available, while the other agents swarm earlier to increase the zone score.

For deciding when to compute the swarm, a timeout called `timeToDecideSwarm` is used. This timeout is designed such that new swarms are computed for each 60 steps. The swarms are of course only updated if they can provide a higher zone score. Swarms are also frequently calculated in the first 150 steps of the simulation, so HARDAC can swarm early if any swarms of significant value are found.

The atoms, `decidedSwarmAt/1` and `currentSwarmValue/1`, are updated when new swarms have been calculated to provide the functionality used by the timeouts and the swarms-calculating Explorer. They keep track of when the current swarms were calculated and the value of these swarms.

The code for the timeouts can be seen in Figure 8.9.

8.2.6 Smaller strategies

8.2.6.1 Cautious behavior at the beginning of the match

To account for the fact that there may not exist a path between the agent and a Repairer before a sufficient amount of the map has been explored, the agents avoids dangerous enemies as long as there does not exist such a path. Dangerous enemies are those that are either Saboteurs or not inspected yet if not all Saboteurs are inspected.

8.2.6.2 Record more vertices

As the agents of HactarV2 only record and inform other agents of vertices that have been probed or surveyed, a lot of vital information is lost. This is because the amount of vertices visible to agents are large. This enables the agents to compute paths drastically sooner than before, at the expense of the agents sending and handling many messages in the first few steps of the simulation.

CHAPTER 9

Evaluation

This chapter deals with the setup and the results of testing HARDAC against Python-DTU in the MAPC 2012 scenario. An explanation of how the testing has been conducted and how the performance of the systems has been measured and compared with each other follows, followed by the results from the testing.

An ordinary tournament from MAPC 2012 consists of exactly three simulations, each with a different map size. Therefore the tests include all three map sizes. The three map sizes are, in order and numbered: (1) 300 vertices, (2) 240 vertices, and (3) 200 vertices. The maps will henceforth be referred to by their number.

Unfortunately, because of the bug that sometimes prevents agents from communicating with the server and each other for the rest of the simulation, the testing had to be performed one simulation at a time.

It is important to note that HARDAC had a timeout of 30 seconds during the simulations, instead of the usual 2 seconds which is the standard for the MAPC 2012 tournament. This was thought necessary because of the large performance hit stemming from the increased amount of messages being sent between agents in HARDAC. However, HARDAC seems to be able to send all its action within 6 to 10 seconds. With a faster machine and some additional optimizations HARDAC may be able to comply with the ordinary 2 second deadline.

6 tournaments have been run, resulting in 18 simulations. Running additional tournaments were not possible due to time constraints.

9.1 Setup

9.1.1 Dependencies

The following dependencies are needed for running the simulations:

- The MAPC 2012 package, `massim-2012-2.0-bin.zip`, for running the scenario.
- GOAL revision 4941 for running HARDAC.
- Python version >= 3.0 for running Python-DTU.
- The `eismassimconfig.xml` configuration file, to be placed in the GOAL installation folder.
- The configuration file for the simulation[1], `evaluations-hardac.xml`.
- Configuration files[2]: `config_HARDAC.dtd`, `accounts-HARDAC-longtimeout.xml`, `accounts-Python-DTU-2012.xml`.

The environment interface, `eismassim-2.0.jar`, resides in the `eismassim/target` subfolder in the MAPC 2012 package and has to be moved to the `environments` subfolder of the GOAL installation folder.

The configuration files are available in the appendices.

[1]This file should be placed in the `massim/scripts/conf` subfolder of the MAPC 2012 package.

[2]These files should be placed in the `massim/scripts/conf/helpers/2012` subfolder of the MAPC 2012 package.

9.1.2 Running the simulation

To run the simulation the MAPC 2012 Server[3] is started and a configuration is chosen. The configuration holds information about which teams are playing, which maps to use, each agent's timeout etc. Once the server is running, the MAS's are started. Python-DTU through the command line, and HARDAC through the GOAL IDE. The server outputs whether or not the authentication of each agent is successful. Before beginning the simulation the Mars Monitor[4] is started, because the monitor generates useful statistics for each map. When all agents have been authenticated and the monitor has started, the simulation is begun by hitting the enter key at the terminal where the server is running.

9.1.3 Measuring performance

As it is not possible to specify the random seed for the map generator in the 2012 scenario, multiple simulations have been run to even out any advantage or disadvantage that the map topography may provide. That is, if the best swarm positions are far from each other then it may favor a more mobile team. Conversely, if the best swarm positions are close to each other, then it may favor the team that has the best attack strategy as there would inevitably be large battles if the swarms from both teams overlap.

The performance of the systems has been evaluated using the final scores for each simulation. A single measure is created from these scores for each simulation; the difference of the scores over the sum of the scores. By convention the difference is calculated as the opposing team's final score minus HARDAC's final score.

So this measure is a real number between -1 and 1. It is a percentage of how unbalanced the scores are. That is, if the measure α is positive then the opposing team had won the match by $\alpha \cdot 100$ percent. If the measure α is negative then HARDAC had won the match by $-\alpha \cdot 100$ percent. The reason for this measure is that it is a single number for each match that can be compared across matches regardless of the difference in the zone scores available in each map.

Additionally, the MAPC 2012 Server provides statistics images generated for each match. These are used to identify possible places for improvement. The

[3] By executing `startServer.sh` from the `massim/scripts` subfolder of the MAPC 2012 package.

[4] By executing `startMarsMonitor.sh` from the `massim/scripts` subfolder of the MAPC 2012 package.

Mars Monitor generates files for each step throughout the simulation so the match can be replayed by the Mars File Viewer[5].

9.2 Results

The results of the 18 simulations are summarized in Table 9.1.

Tournament	*Map*	*Total score*		*Difference*	*Sum*	$\frac{Difference}{Sum}$ *measure*
		HARDAC	*Python-DTU*			
1	1	**151628**	129901	-21727	281529	-7.72%
1	2	74358	**89021**	14663	163379	8.97%
1	3	**49124**	44090	-5034	93214	-5.40%
2	1	50525	**92606**	42081	143131	29.40%
2	2	45300	**54330**	9030	99630	9.06%
2	3	65247	**88612**	23365	153859	15.19%
3	1	**66874**	57534	-9340	124408	-7.51%
3	2	**81231**	73169	-8062	154400	-5.22%
3	3	80352	**130895**	50543	211247	23.93%
4	1	132308	**138695**	6387	271003	2.36%
4	2	**51318**	41507	-9811	92825	-10.57%
4	3	**87497**	69884	-17613	157381	-11.19%
5	1	92623	**105268**	12645	197891	6.39%
5	2	102456	**104881**	2425	207337	1.17%
5	3	61679	**89707**	28028	151386	18.51%
6	1	**146182**	125935	-20247	272117	-7.44%
6	2	56956	**59150**	2194	116106	1.89%
6	3	68520	**72737**	4217	141257	2.96%
Total		1464178	**1567922**	103744	3032100	3.42%

Table 9.1: The final score of each team for each simulation and the corresponding difference-over-sum measure. The score of the winning team is bolded. The difference is calculated by subtracting HARDAC's score from Python-DTU's score.

The results show that Python-DTU won 11 of the 18 simulations, while HARDAC won 7. However, if considered as tournaments the score is 3 wins each. In most of the simulations the difference between the two scores is less than 10%. If the results of the evaluations are representative of the actual performance

[5]By executing `startMarsFileViewer.sh` from the `massim/scripts` subfolder of the MAPC 2012 package.

of HARDAC against Python-DTU then it shows that HARDAC would win against Python-DTU $\frac{7}{18} \approx 40\%$ of the time, and therefore also about 40% of the tournaments as the matches are independent of each other.

In some simulations Python-DTU wins by over 20%, implying that there may be cases where Python-DTU is significantly superior to HARDAC.

The boxplot in Figure 9.1 shows that the median match is close to 0. However, the matches where HARDAC won are closer to 0 than the matches where Python-DTU won. That is, Python-DTU sometimes win with a considerably larger margin than what HARDAC seems to be capable of, as seen in the barplot in Figure 9.1.

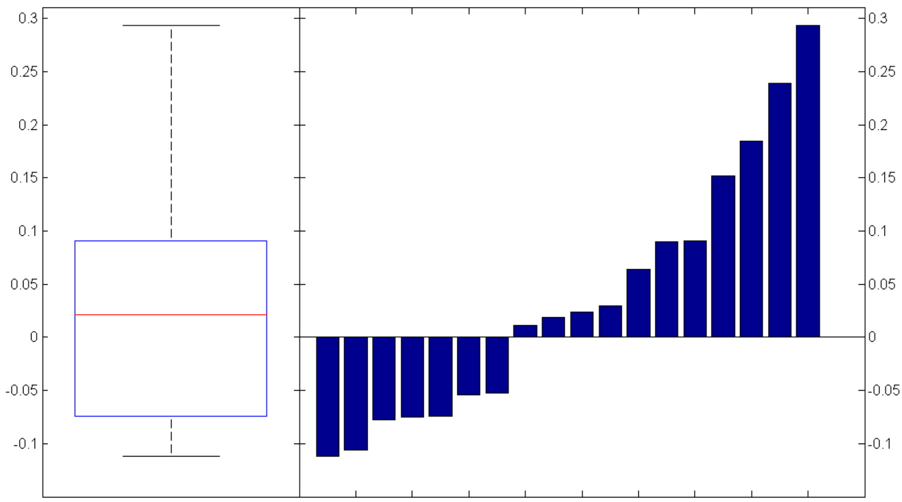
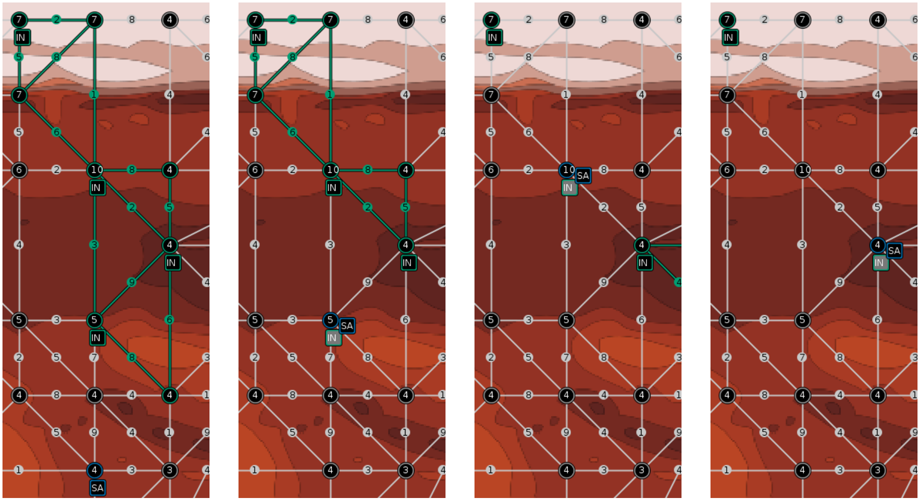


Figure 9.1: A box- and barplot of the difference-over-sum measures for the 18 simulations from Table 9.1.

9.3 Selected observations

9.3.1 Successful harass example

HARDAC prevents Python-DTU's Inspectors from swarming by disabling them, showcasing the harass strategy. See Figure 9.2.



(a) The Saboteur moves towards the swarm.
(b) An Inspector has been disabled.
(c) A second Inspector is disabled, breaking the swarm.
(d) A third Inspector is attacked and disabled.

Figure 9.2: From tournament 1 simulation 1. A HARDAC Saboteur (blue agent) successfully harassing and destroying one of Python-DTU's swarms (green agents).

9.3.2 Analysis of the most unbalanced match

Tournament 2 simulation 1 is the most unbalanced match with respect to the difference-over-sum measure. Examining the output from the server and the Mars Monitor gives clues to what went wrong. The scores graph (see Figure 9.3 shows that HARDAC never managed to get a foothold during this simulation. However, it does not show a certain step at which something went wrong for HARDAC, so what happened? Using the Mars File Viewer it is possible to watch the simulation again step-by-step. It shows that it was a combination of factors that caused problems for HARDAC:

- At around step 75, Python-DTU and HARDAC begin to swarm. They choose the same areas to swarm. Two symmetrically opposed areas at the bottom of the map, with several optimums each, and two symmetrically opposed areas at the side, with a single optimum each. Both teams swarm at the side optimums, but Python-DTU gets to the best area at the bottom of the map first. HARDAC's attempt to capture the oppo-

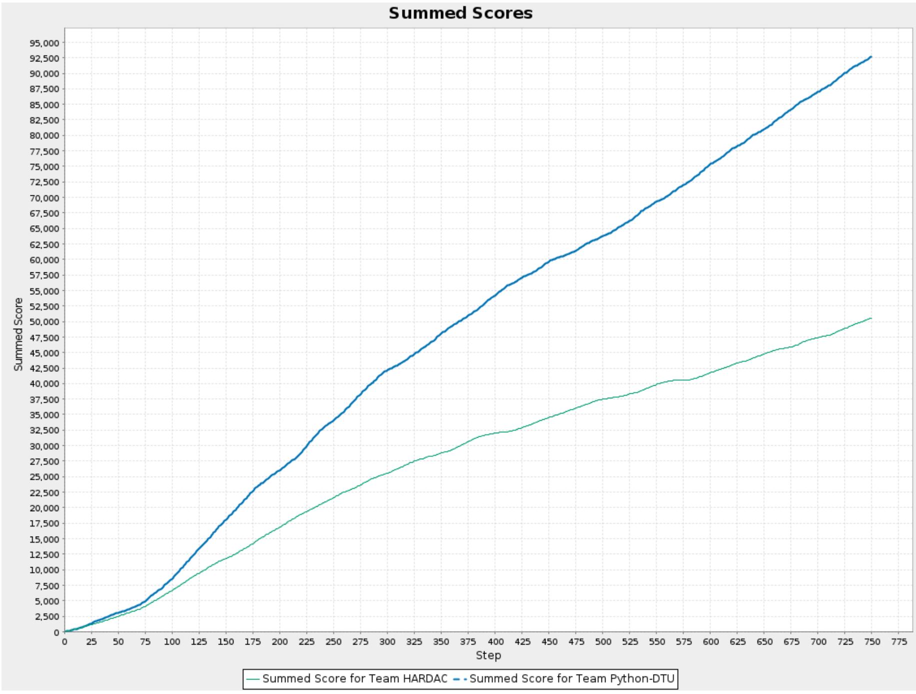


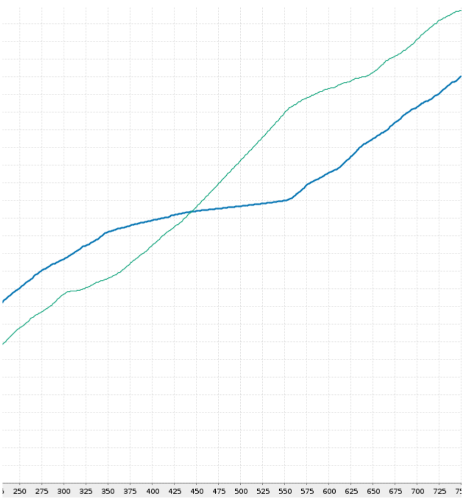
Figure 9.3: The scores graph for tournament 2 simulation 1.

site area is stopped by Python-DTU's Saboteurs which are already in the area, because of their swarm. The result of this is that all large battles happen inside HARDAC's desired zone, completely disrupting HARDAC's swarm, while at the same time Python-DTU's swarm is left untouched. This is the main cause of Python-DTU's dominance in this simulation.

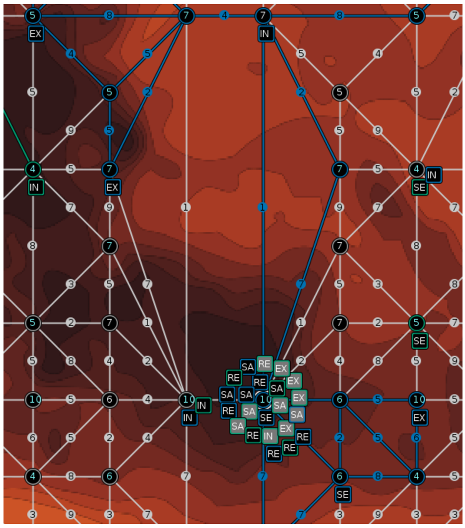
- A little later, one of HARDAC's Saboteurs sets out to find the rest of Python-DTU's swarm, but goes in the completely wrong direction, moving towards the top of the map.

- The last problem is that an Inspector becomes the central piece of HARDAC's swarm, but whenever it becomes time for it to inspect Python-DTU's Saboteurs, it moves away from its position. This causes the swarm to lose nearly 30 points each step it is missing, which is a third of HARDAC's total step score.

9.3.3 Disabled swarming agents



(a) Summed score for each step between step 225 and step 750. HARDAC's score is green, Python-DTU's is blue.



(b) A large battle from step 455 showing how HARDAC has disabled Python-DTU's Explorers while Python-DTU fails to repair them for a long time.

Figure 9.4: The turning point for HARDAC in tournament 3 simulation 1, occurring approximately at step 445.

The outcome of tournament 3 simulation 1 is curious as HARDAC overtakes Python-DTU in the middle of the match where Python-DTU's score suddenly ceases to increase significantly. This is because HARDAC manages to disable a large amount of the enemy's swarming agents while Python-DTU fails to repair them again for an extended period of time. This also happens to HARDAC sometimes, but it shows the importance of disabling swarming agents. See Figure 9.4.

9.3.4 Repairers parrying unnecessarily

HARDAC's Repairers seem to parry when at a large battle, probably when they have nothing to repair, which uses a lot of energy. This may be detrimental for HARDAC if this would result in Python-DTU gaining the upper hand in these battles. See for example Figure 9.5.

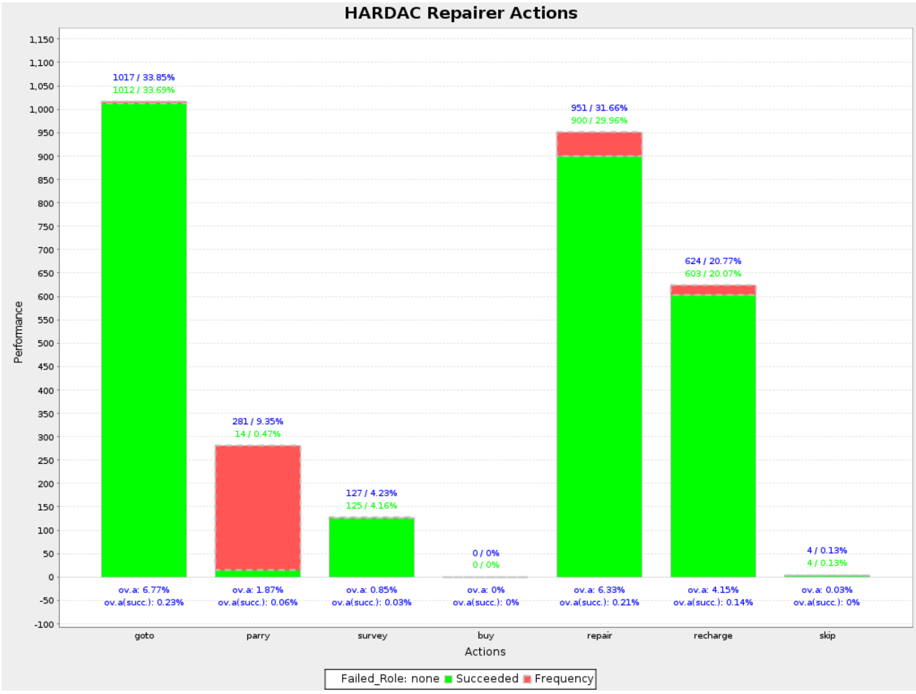


Figure 9.5: The actions for the Repairers from tournament 6 simulation 1. The red portions of each action are the amount of the action that failed to be performed correctly. Note the large number of failed parries (second from the left), which should only occur if the Repairer is not attacked while parrying.

9.3.5 Large battle inside swarms

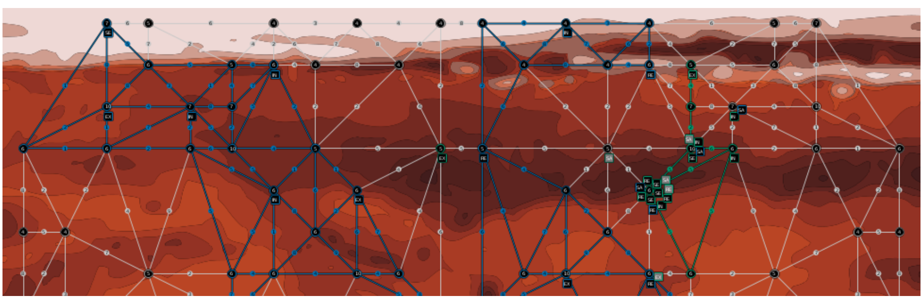


Figure 9.6: Tournament 3 simulation 3. Large battles happen in HARDAC's swarms (green agents).

There is a common theme in tournament 3 simulation 3 and tournament 5 simulation 3 (where Python-DTU wins by a large margin) and tournament 4 simulation 2 and tournament 4 simulation 3 (where HARDAC wins by a large margin). See for example Figure 9.6.

In the above mentioned simulations where HARDAC loses it seems that large battles happens inside HARDAC's swarms, for most of the simulation, preventing HARDAC from receiving the zone score it needs to win.

In the other two the same happens but inside Python-DTU's swarms instead, resulting in HARDAC winning the match.

It therefore seems important to prevent large battles from forming inside HARDAC's swarms, or at least to relocate the swarms to a more quiet region of the map.

CHAPTER 10

Extension

This chapter contains thoughts and ideas about development which were outside the scope of this project and possible extensions of HARDAC for future work, as well as a brief look at MAPC 2013.

- **Improving the messaging system.** The messaging should be rewritten so that all agents have received all other agents messages before deciding on an action to perform. One possibility for achieving this would be to have a predicate `doneMessaging` which becomes true when an agent has received a mail with a keyword, such as `doneMailing`. Until this predicate was true, the agent would skip through the main section, not deciding on an action.
- **Implementing negotiation** With an updated messaging system it would be possible to implement different forms of negotiation. Negotiation could also help in one of the other areas that need improvement, making decisions about which agent does what more efficient. Implementation of negotiation is beyond the scope of this project.
- **Handling large battles in HARDAC's zone.** The main weakness for HARDAC is not being able to handle large battles happening within its zones. There are two solutions. Move the zone, or move the battle. Moving the battle may be possible if HARDAC manages to send Saboteurs to Python-DTU's swarm.

- **Reducing Repairers failed parries.** As seen in the evaluation chapter, HARDAC's Repairers have a disproportionate amount of failed parries. This is caused by Repairers choosing to parry when there are a greater or equal number of allied Saboteurs compared to enemy Saboteurs on the same node. The solution would be to calculate which agents are likely to be attacked by the enemy, and only parry if necessary.

- **Repairers leaving large battles.** In an attempt to control large battles and mitigate them, HARDAC Repairers move away from large battles when they detect they are a part of them. This allows the Repairers to repair other disabled agents, preventing the swarm from collapsing due to the swarm being attacked while the Repairers are caught up. The idea works well, except when the Repairers have no other tasks. In this case, the Repairer will identify being in a battle and leave it, but will then walk around aimlessly, as it knows not to join the battle again. This can be fixed by only leaving the battle if the agent has something worthwhile to do.

- **Dynamic strategies.** An interesting possibility for improvement would be dynamic strategies. It may be possible to adjust strategies during a simulation or tournament depending on the enemy team. Adapting to the enemy team would require testing several strategies during the simulation to find which one works best, or require information extracted from observing the opposing team in previous matches. Another possibility would be to estimate zone values for the opposing team, and playing more defensive if HARDAC believes to have the better zone or more aggressive if HARDAC believes the enemy has the better zone.

- **Improving the swarm algorithm.** The swarming algorithm, taken from Python-DTU, is a greedy algorithm. It has not been tested thoroughly enough to claim it is the best solution. There are cases where the algorithm does not cover the largest possible area of the graph. It may be possible to spread the agents out more by considering vertices which are not already colored only.

CHAPTER 11

Discussion

This chapter contains discussion and reflection about the development process and HARDAC's strategies.

11.1 The development process

The first one and half months of this thesis were spent on getting HactarV2 to connect and run on the MAPC 2012 server. Bugs in new releases of GOAL, new syntax in GOAL, and a bug in EISMASSIM caused problems that were hard to debug. The result was having to use an older version of GOAL, namely version 4941.

The original intention was to evaluate the performance of the implemented strategies individually. So the development of the strategies were performed by splitting up the strategies into multiple folders, which had a single base directory (HactarV2 with bugfixes and adapted to the MAPC 2012) from which the strategies were merged to, and which were independent systems when merged with the base directory and which could be merged into the final system. As the strategies spanned multiple files, some which would often have to be changed in multiple strategies, some Python scripts, using the diff and patch tools available in Linux, were made which automated the merging process. However, it became too rigid as the number of strategies rose. Especially when hierarchies of the strategies were made as some strategies depended on other strategies. But it also gave a good overview of the different strategies of HARDAC and it was easy to test a specific strategy.

11.2 Working with GOAL

Working in GOAL has both advantages and disadvantages. A very positive advantage is the use of declarative languages for knowledge representation, in our case Prolog. Declarative languages are a natural abstraction for writing an agent's reasoning process and mental state, that feels intuitive. Prolog also makes for very robust systems. When a piece of Prolog code in GOAL does not work as intended, the agent will usually still be able to make decisions, albeit poor decisions, rather than crash. In the case of the competition this is not always positive as efficiency is important for the agents to reach a decision before the deadline, and it is not good when the agents perform actions that are not as expected.

The major disadvantage is that GOAL is still in alpha. This is a problem as it means the syntax is not finalized and the language and IDE have many bugs. Some examples of bugs are the IDE randomly becoming unresponsive, panels only being fully re-sizable while there is no simulation running, and the most problematic bug where one of our agents would suddenly lose contact with the server. The syntax changes forced us to use an outdated version of the language, when updating to a newer version may have gotten rid of the disconnection bug.
Using declarative languages with no feedback from the IDE about usage of variables also makes debugging the code difficult, as a misspelled variable name is simply accepted as a new variable that can be unified, no matter the context.
Furthermore, most of the predicates in HARDAC do not ensure the variables have the correct type. In hindsight using the built-in Prolog predicates such as `atom/1` and `int/1`, would have prevented difficult bugs from emerging.

11.3 Reflections on strategies

During the project period we implemented and tested many strategies, not all of which were successful. For a long time we hoped that HactarV2's swarming algorithm was good enough, if we managed to improve the other strategies to support it. We spent a lot of time creating a second Sentinel swarm that would be separate from the main swarm, in the hopes that having a second swarm around a different optimum would allow HARDAC to get the upper hand. This assumed that the harass strategy and HARDAC's main swarm would be so disruptive to Python-DTU's swarm that neither team would receive many points each step, with the Sentinel swarm tipping the score in HARDAC's favor. In practice this resulted in many special cases, that all had to be handled. The Sentinel swarm had to choose a different optimum to swarm around than the rest of the swarm. The area the Sentinels chose should be as high-valued as possible. If the Sentinel swarm is attacked it should find a new place to swarm. The area should not be near Python-DTU's swarm. These are just some of the things the Sentinel swarm had to take into account when deciding where to swarm.

In an attempt to reduce the erratic movement that took place in HactarV2's swarm, we implemented a delay on movement in the swarm, so that an agent that believed it had found that best vertex for it to stand on would be encouraged to stay until some number of steps had passed. Unfortunately this caused several agents to stand on top of each other all believing they had found the best vertex for them to stand on, instead of spreading out. The attempted solutions only slightly helped with the problem.

Eventually we realized that the swarming algorithm simply was not good enough, or consistent enough, versus Python-DTU. Attempting to improve it caused significant increases in computation time, and large amounts of very specific code for various cases that would need to be handled. In hindsight, it would have been beneficial to rewrite the swarm code from the beginning. It would have saved a lot of time spent on trying to improve code that would never be adequate against Python-DTU.

Inserting more vertices during the first steps of the simulation is very computationally heavy. There is a lot of redundancy in the information being sent, and since sending and receiving messages is very slow, this causes HARDAC to miss the 2 second deadline during the first few steps of the simulation.

The final strategies of HARDAC have a lot of the same ideas that Python-DTU's have, with some of the algorithms strongly inspired by Python-DTU; the swarming algorithm is a re-implementation of Python-DTU's swarming algorithm; the ideas for hunting enemy agents and harassing enemy swarms are present in both HARDAC and Python-DTU; the delayed buying of upgrades; and repairing damaged agents. Other strategies and ideas are quite different; HARDAC's Repairers predicting Python-DTU's attacks; and a general decrease in the amount of surveying done by HARDAC.

Despite our attempts to prevent large battles, they still appear in every simulation. Whose zone the large battles happen in, seems to be the major factor in whether Python-DTU or HARDAC wins a match. When the large battles happen inside HARDAC's zone Python wins convincingly. When the large battles happen inside Python-DTU's zone the simulations are very even, often ending in a victory for HARDAC. This may be a result of Python-DTU being better at continuing to harass outside of the large battles than HARDAC. Improving the handling of large battles, i.e., finding a way to force the battle into enemy territory, would benefit HARDAC immensely.

CHAPTER 12

Conclusion

In this thesis we have worked with HactarV2, the winner of MAPC 2011, and Python-DTU, the strongest team for the 2012 MAPC scenario that we know of. Our goal was to adapt and improve HactarV2 to make it competitive versus Python-DTU, while learning the GOAL programming language and gaining an understanding of the MAPC 2011 and 2012 scenarios.

The thesis begins by clarifying the meaning of the term multi-agent system. Then examines the GOAL programming language, the language that HactarV2 is written in. We showed the syntax and basic structure of an agent in GOAL and how to write a multi-agent system in it.

Relevant changes that were made to the Agents on Mars scenario between 2011 and 2012 were analyzed, most importantly the new graph generation, which results in randomly placed optimums. The analysis continued with an analysis of HactarV2's strategies, and their advantages and shortcomings. Where relevant, the shortcomings were compared with Python-DTU's strategies. The analysis lead to many ideas for new strategies and improvements. By analyzing the possible rewards and the time that would be involved in rewriting the code, some of these strategies were implemented.

The important details about the implemented strategies have been explained, including swarming by utilizing the same algorithm as Python-DTU, predicting attacks for repairing, delegating targets for repairing and attacking without sending messages, and harassing and defending swarms. Once all the new strategies had been tested they were combined, into a new multi-agent system named HARDAC.

HARDAC was tested against Python-DTU through 18 simulations, representing 6 tournaments. The results showed the major improvement of HARDAC versus HactarV2, moving from never winning versus Python-DTU to winning nearly 40% of the time. Through the evaluations of the results, and the possibilities for future work, possible future improvements to HARDAC have been explained. Lastly the development process and the results of the evaluation were discussed.

APPENDIX A

The GOAL IDE

GOAL projects are usually developed, executed, and debugged using the GOAL IDE.

To run an MAS project, first load the `.mas2g` file in the IDE and then press the green run button. The agents are paused by default, so it is necessary to execute the agents by pressing the run button a second time.

When an MAS is running, it is possible to inspect the mental state of an agent by right-clicking an agent in the IDE and clicking the Introspector item. Alternatively, pressing Ctrl+D when an agent is selected accomplishes the same. In the Introspector, actions for both the environment and belief base can be manually executed. Additionally, the belief base can be queried. It is important to surround queries with the `bel` operator.

Agents can be debugged by stepping through them by clicking the "Step agent" button. The agent windows at the lower portion of the IDE provides information about the current step during the stepping.

The console window at the lower portion of the IDE contains information about the connection to the agent and any errors and warnings that may occur during the execution of the MAS.

When the MAS is finished, click the "Kill multi-agent system" while the MAS

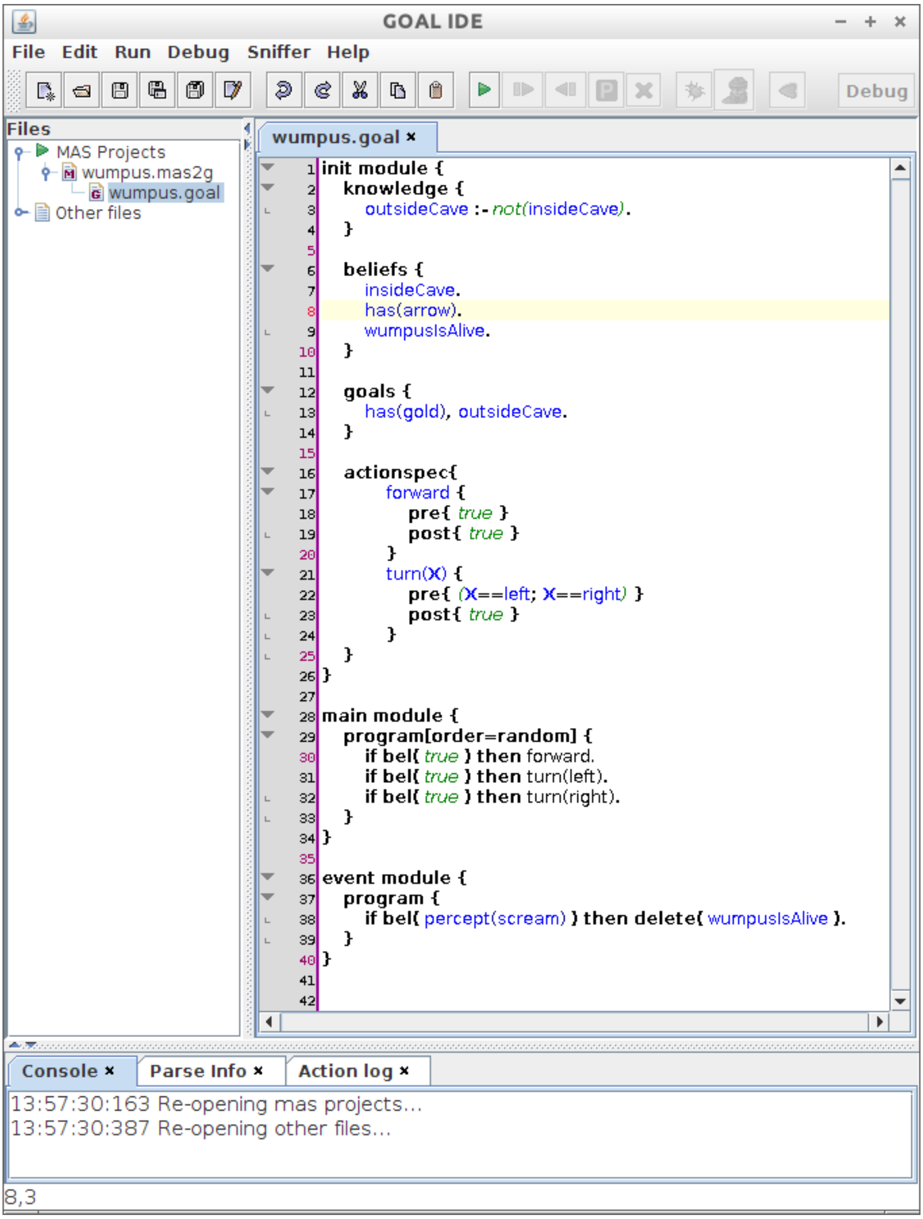


Figure A.1: The GOAL IDE for GOAL revision 4941 with an opened project, also demonstrating the syntax of GOAL.

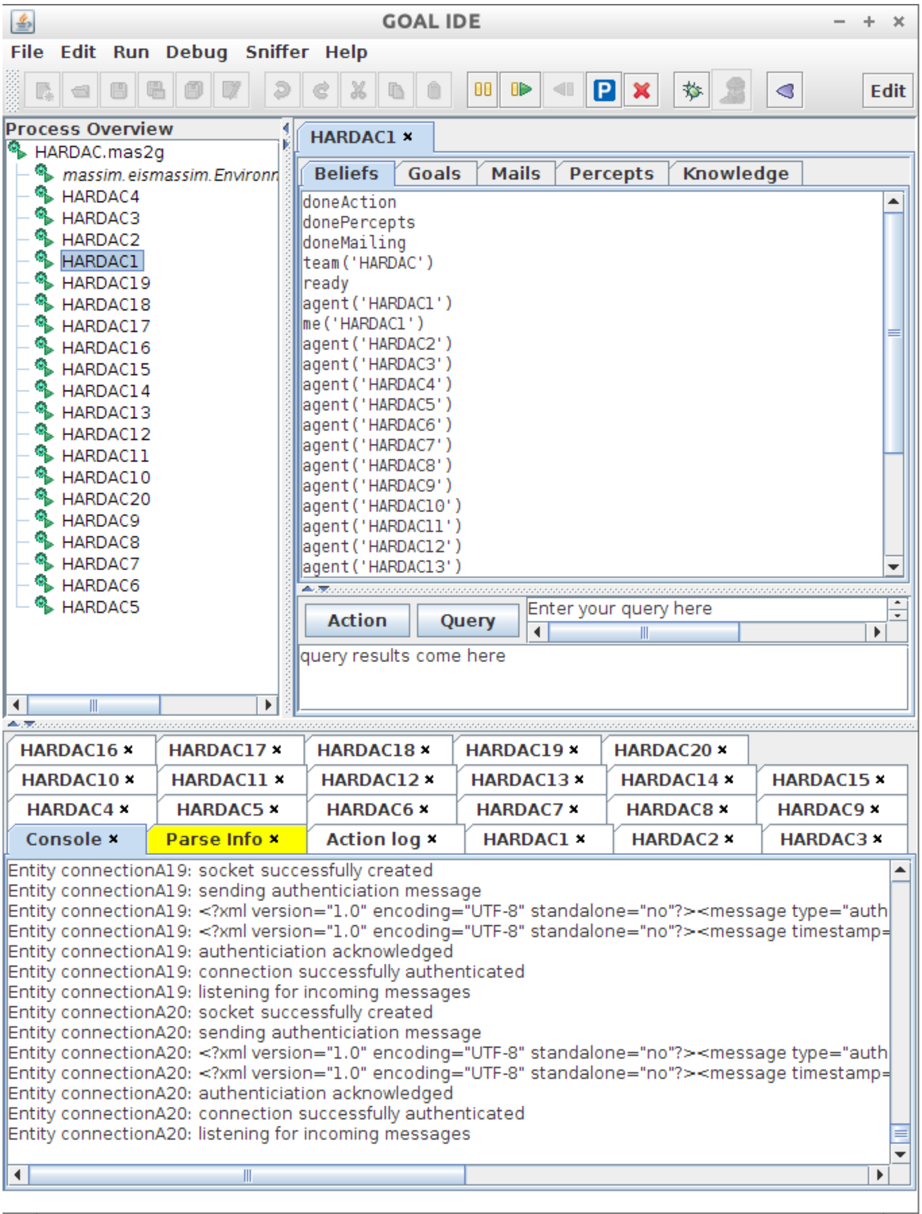


Figure A.2: The GOAL IDE while running HARDAC.

entry (in the "Process overview" pane, together with the agents and environment interface) is selected.

Appendix B

Swarming algorithm

The swarming algorithm from Python-DTU, `calcAreaControl`, reimplemented in GOAL and changed slightly to calculate swarms from multiple promising optimums instead of only the best. The predicate to call when deciding the swarms is `calcSwarms/2`, which creates a list of vertex-agent pairs and the value of the swarm.

```
calcSwarms(Chosen, Value) :- decideOptimums(Opts), findall((Val,Swarm), (member(Opt,Opts), calcAreaControl(Opt,Swarm,Val)), L), sort(L,S), length(S,N), nth1(N,S,(Value,Chosen)), !.

% Python-DTU's algorithm for calculating swarm positions, reimplementation in GOAL

% calcAreaControl returns pairs of agents and vertices which determine where the agents shall stand when swarming
% Chosen = agent-vertex pairs
calcAreaControl(Opt,Chosen,Value) :- allVertices(Tmp), delete(Tmp,Opt,Vs), swarmAgents([A|AT]), cacAux(Vs,AT,[Opt],Rest), Chosen = [(A,Opt)|Rest], swarmValue(Chosen, Value), !.

cacAux(_,[],_,[]).
cacAux(Vs,[A|T],Chosen,[(A,Best)|Rest]) :- calcOwned(Chosen, Owned), bestPosition(Vs,Chosen,Owned,Best), cacAux(Vs,T,[Best|Chosen],Rest).

allVertices(Vs) :- findall(V, (vertex(V,Val,_), Val \= unknown, Val \= 1), L), sort(L,Vs), !.
```

```
swarmAgents(As) :- timeToSwarm, swarmAgents(As,['Saboteur','Repairer'])
    , !.
swarmAgents(As) :- swarmAgents(As,['Saboteur','Repairer','Explorer']),
    !.
swarmAgents(As,IgnoredRoles) :- findall(A, (agent(A), role(A,R), not(
    memberchk(R,IgnoredRoles))), L), sort(L,As), !.

swarmValue(Chosen,Val) :- findall(V, member((_,V), Chosen), L),
    calcOwned(L,Owned), swarmValueAux(Owned,Val).
swarmValueAux([],0).
swarmValueAux([V|T],Val) :- swarmValueAux(T,Part), vertexValue(V,Tmp),
    (Tmp == unknown -> VVal = 1 ; VVal = Tmp), !, Val is Part + VVal.

bestPosition([],_,_,_) :- fail, !.
bestPosition(Vs,Chosen,Owned,Best) :- subtract(Vs,Chosen,NewVs), bpAux(
    NewVs,Chosen,Owned,_,Best).

bpAux([],_,_,0,_).
bpAux([V1|R],Chosen,Owned,MaxVal,Best) :- bpZoneVal(V1, Chosen, Owned,
    Val1), bpAux(R,Chosen,Owned,Val2,V2), (Val1 > Val2 -> (Best = V1,
    MaxVal = Val1) ; (Best = V2, MaxVal = Val2)).

bpZoneVal(V,Chosen,Owned,Val) :- vertex(V,VVal,_), neighbours(V,Ns),
    subtract(Ns,Owned, Ws), bpZoneValAux(Ws,Chosen,ValPart), Val is
    ValPart + VVal.

bpZoneValAux([],_,0).
bpZoneValAux([W|R], Chosen, ValSum) :- neighbours(W,Tmp), intersection(
    Tmp,Chosen,Zs), vertex(W,Tmp2,_), (Tmp2 == unknown -> WVal = 1 ;
    WVal = Tmp2), bpZoneValAuxAux(Zs,WVal,ValPart1), bpZoneValAux(R,
    Chosen,ValPart2), ValSum is ValPart1 + ValPart2.

bpZoneValAuxAux([],_,0).
bpZoneValAuxAux([_|R],WVal,ValSum) :- bpZoneValAuxAux(R,WVal,ValPart),
    ValSum is WVal + ValPart.

calcOwned([],[]).
calcOwned(Chosen,Owned) :- Chosen = [_|T], coAux(Chosen,T,O), union(
    Chosen,O,Owned).

coAux([H],[],[H]).
coAux([H|T],T,Owned) :- T = [N|R], neighborIntersect(H,T,O), coAux([N|R
    ],R,O2), union(O,O2,Owned).

neighborIntersect(_,[],[]).
neighborIntersect(V,[H|T],NI) :- neighbours(V,NV), neighbours(H,NH),
    intersection(NV,NH,X), neighborIntersect(V,T,Y), union(X,Y,NI).


% Finds the optimum nodes that can contain a swarm with the largest
% potential values as defined by calcZoneValue
bestOptimums(List,Opts) :- findall((ValSum,Swarm), (member(Swarm,List),
    calcZoneValue(Swarm,ValSum)), L), sort(L,S), length(S,N), nth1(N,S
    ,(MaxVal,_)), Limit is round(0.65*MaxVal), bestOptimumsAux(S,Limit,
    L2), sort(L2,Opts).
```

```
bestOptimumsAux([],_,[]).
bestOptimumsAux([(Val,Opt)|T],Limit,[Opt|Rest]) :- Val >= Limit, bestOptimumsAux(T,Limit,Rest).
bestOptimumsAux([(Val,_)|T],Limit,Rest) :- Val < Limit, bestOptimumsAux(T,Limit,Rest).

% Calculates the sum of the values for all the neighbors, and their neighbors, and the vertex O, around the vertex O
calcZone(O,S) :- findall(N, (neighbour(O,_,N)), L1), findall(N, (member(M,L1), neighbour(M,_,N)), L2), union(L1,L2,L3), sort(L3,S).
calcZoneValue(O,V) :- calcZone(O,L), vertexListSum(L,V).

vertexListSum([], 0).
vertexListSum([H|T], Sum) :- vertexValue(H,V), V == unknown, vertexListSum(T,S), Sum is S+1.
vertexListSum([H|T], Sum) :- vertexValue(H,V), V \== unknown, vertexListSum(T,S), Sum is S+V.

% Find all optimums that are not already in use
allOptimums(Opts) :- allOptimums(Opts,[]).
allOptimums(Opts,Ignore) :- findall(V, (optimum(V), not(member(V,Ignore)), not((neighbour(V,N),member(N,Ignore)))), Opts), length(Opts,N), N > 0, !.

% Choose the best optimums
decideOptimums(Opts) :- allOptimums(L), !, bestOptimums(L, Opts), !.
```

Appendix C

Updates to Python-DTU from MAPC 2012

```
23a24
>           self.counter = args.counter
60a62
>           self.max_opponent_health_list = [INITIAL_MAX_HEALTH, "",
    INITIAL_MAX_HEALTH, ""]
504,507c506,524
<               if role == SAB and maxHealth > self.max_opponent_health:
<                   self.max_opponent_health = maxHealth
<               if role == SAB and strength > self.max_opponent_strength:
<                   self.max_opponent_strength = strength
---
>               if self.counter and role == SAB:
>                   if  maxHealth >= self.max_opponent_health_list[0]:
>                       if name != self.max_opponent_health_list[1]:
>                           self.max_opponent_health_list[2] = self.
    max_opponent_health_list[0]
>                           self.max_opponent_health_list[3] = self.
    max_opponent_health_list[1]
>                           self.max_opponent_health_list[0] = maxHealth
>                           self.max_opponent_health_list[1] = name
>                       else:
>                           self.max_opponent_health_list[0] = maxHealth
>
>                   elif maxHealth > self.max_opponent_health_list[2]:
>                       self.max_opponent_health_list[2] = maxHealth
>                       self.max_opponent_health_list[3] = name
```

```
>                   self.max_opponent_health = self.max_opponent_health_list[2]
>               elif role == SAB:
>                   if maxHealth > self.max_opponent_health:
>                       self.max_opponent_health = maxHealth
>                   if strength > self.max_opponent_strength:
>                       self.max_opponent_strength = strength
589a607
> parser.add_argument('-c', '--counter', help='Counter UFSCs counter', action='store_true')
```

This update can be applied to the file `bagent.py` from the Python-DTU code downloaded from `http://multiagentcontest.org/` by using the command `patch bagent.py < update.diff` on a Linux system, where `update.diff` is a file containing the above `diff` output.

Appendix D

MAPC 2012 configuration files used during evaluation

D.1 eismassimconfig.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interfaceConfig scenario="mars2012" host="localhost" port="12300" scheduling="yes" times="no" notifications="no" queued="yes" statisticsFile="no" statisticsShell="yes" submitStatistic="no">
  <entities>
    <entity name="HARDAC1" username="HARDAC1" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC2" username="HARDAC2" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC3" username="HARDAC3" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC4" username="HARDAC4" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC5" username="HARDAC5" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC6" username="HARDAC6" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC7" username="HARDAC7" password="y4D76cpW" iilang="yes" xml="yes"/>
    <entity name="HARDAC8" username="HARDAC8" password="y4D76cpW" iilang="yes" xml="yes"/>
```

```
    <entity name="HARDAC9" username="HARDAC9" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC10" username="HARDAC10" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC11" username="HARDAC11" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC12" username="HARDAC12" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC13" username="HARDAC13" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC14" username="HARDAC14" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC15" username="HARDAC15" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC16" username="HARDAC16" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC17" username="HARDAC17" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC18" username="HARDAC18" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC19" username="HARDAC19" password="y4D76cpW"
        iilang="yes" xml="yes"/>
    <entity name="HARDAC20" username="HARDAC20" password="y4D76cpW"
        iilang="yes" xml="yes"/>
  </entities>
</interfaceConfig>
```

D.2 config_HARDAC.dtd

```
<!ENTITY teamPythonDTU2012 SYSTEM "accounts-Python-DTU-2012.xml">
<!ENTITY teamHARDACLongTimeout SYSTEM "accounts-HARDAC-longtimeout.xml"
    >

<!ENTITY simulation1 SYSTEM "sim1.xml">
<!ENTITY simulation2 SYSTEM "sim2.xml">
<!ENTITY simulation3 SYSTEM "sim3.xml">

<!ENTITY actionclassmap SYSTEM "actionclassmap.xml">
<!ENTITY sim-server SYSTEM "sim-server.xml">

<!ENTITY actions SYSTEM "sim-actions.xml">
<!ENTITY roles SYSTEM "sim-roles.xml">
<!ENTITY achievements SYSTEM "sim-achievements.xml">
<!ENTITY agents SYSTEM "sim-agents.xml">

<!ATTLIST conf
  backuppath CDATA "backup"
  launch-sync-type CDATA "key"
  reportpath CDATA "./backup/"
  time CDATA "18:06"
```

```
  time-to-launch CDATA "10000"
  tournamentmode CDATA "0"
  tournamentname CDATA "Mars2012"
  debug-level CDATA "normal"
  >

<!ATTLIST simulation
  configurationclass CDATA "massim.competition2012.
      GraphSimulationConfiguration"
  rmixmlobsserverhost CDATA "localhost"
  rmixmlobsserverport CDATA "1099"
  rmixmlobserver CDATA "massim.competition2012.
      GraphSimulationRMIXMLDocumentObserver"
  simulationclass CDATA "massim.competition2012.GraphSimulation"
  xmlstatisticsobserver CDATA "massim.competition2012.
      GraphSimulationXMLStatisticsObserver"

  visualisationobserver CDATA "massim.competition2012.
      GraphSimulationVisualizationObserver"
  visualisationobserver-outputpath CDATA "output"
  rmixmlobserverweb CDATA "massim.competition2012.
      GraphSimulationRMIXMLDocumentObserverWebInterface"
  xmlobserver CDATA "massim.competition2012.GraphSimulationXMLObserver"
  xmlobserverpath CDATA "./backup/xmls"

  statisticsobserver CDATA "massim.competition2012.
      GraphSimulationStatisticsObserver"
  statisticsobserverpath CDATA "statistics"
  >

<!ATTLIST configuration
  xmlns:meta CDATA "http://www.tu-clausthal.de/"
  maxNumberOfSteps CDATA "750"
  numberOfAgents CDATA "40"
  numberOfTeams CDATA "2"
  gridWidth CDATA "21"
  gridHeight CDATA "21"
  cellWidth CDATA "100"
  minNodeWeight CDATA "1"
  maxNodeWeight CDATA "10"
  minEdgeCost CDATA "1"
  maxEdgeCost CDATA "10"
  mapGenerator CDATA "GraphGeneratorTriangBalOpt"
  >
```

D.3 evaluations-hardac.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conf SYSTEM "helpers/2012/config_HARDAC.dtd">

<conf>
  &sim-server;
  <match>
    &simulation1;
    &simulation2;
    &simulation3;
  </match>

  <accounts>
    &actionclassmap;

    &teamHARDACLongTimeout;
    &teamPythonDTU2012;
  </accounts>
</conf>
```

D.4 accounts-HARDAC-longtimeout.xml

```
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="y4D76cpW" team="HARDAC" timeout="30000" username="HARDAC1" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="y4D76cpW" team="HARDAC" timeout="30000" username="HARDAC2" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="y4D76cpW" team="HARDAC" timeout="30000" username="HARDAC3" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="y4D76cpW" team="HARDAC" timeout="30000" username="HARDAC4" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="y4D76cpW" team="HARDAC" timeout="30000" username="HARDAC5" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="y4D76cpW" team="HARDAC" timeout="30000" username="HARDAC6" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
```

```
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC7" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC8" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC9" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC10" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC11" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC12" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC13" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC14" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC15" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC16" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC17" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC18" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC19" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="
    massim.competition2012.GraphSimulationAgentAction" maxpacketlength=
    "65536" password="y4D76cpW" team="HARDAC" timeout="30000" username=
    "HARDAC20" />
```

D.5 accounts-Python-DTU-2012.xml

```
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU1" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU2" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU3" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU4" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU5" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU6" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU7" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU8" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU9" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU10" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU11" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU12" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU13" />
```

```
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU14" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU15" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU16" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU17" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU18" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU19" />
<account actionclassmap="Graph" auxtimeout="500" defaultactionclass="massim.competition2012.GraphSimulationAgentAction" maxpacketlength="65536" password="1" team="Python-DTU" timeout="2000" username="Python-DTU20" />
```

APPENDIX E

Test scores

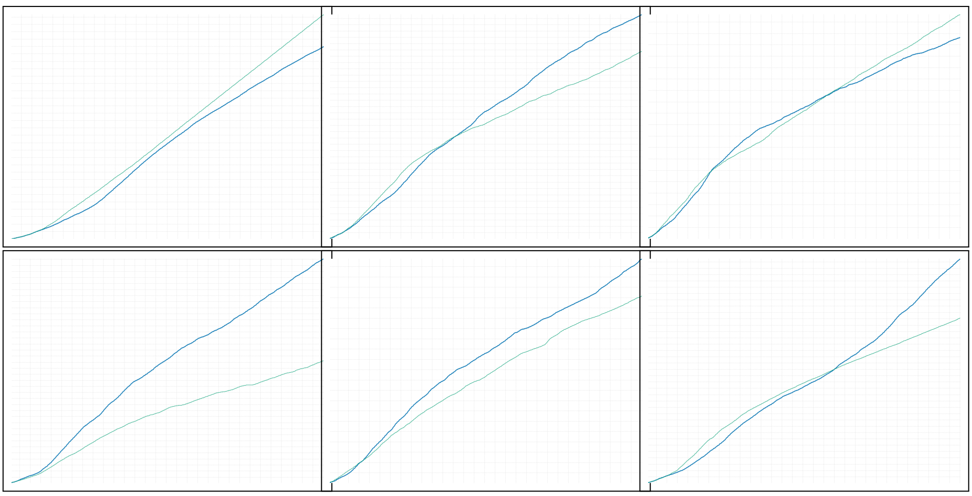


Figure E.1: Scores for the first 2 test tournaments. Rows are simulations (1, 2, 3), columns are tournaments (1 to 2). HARDAC's score is green, Python-DTU's is blue. Horizontal axis are the steps, vertical axis are the total score at a step.

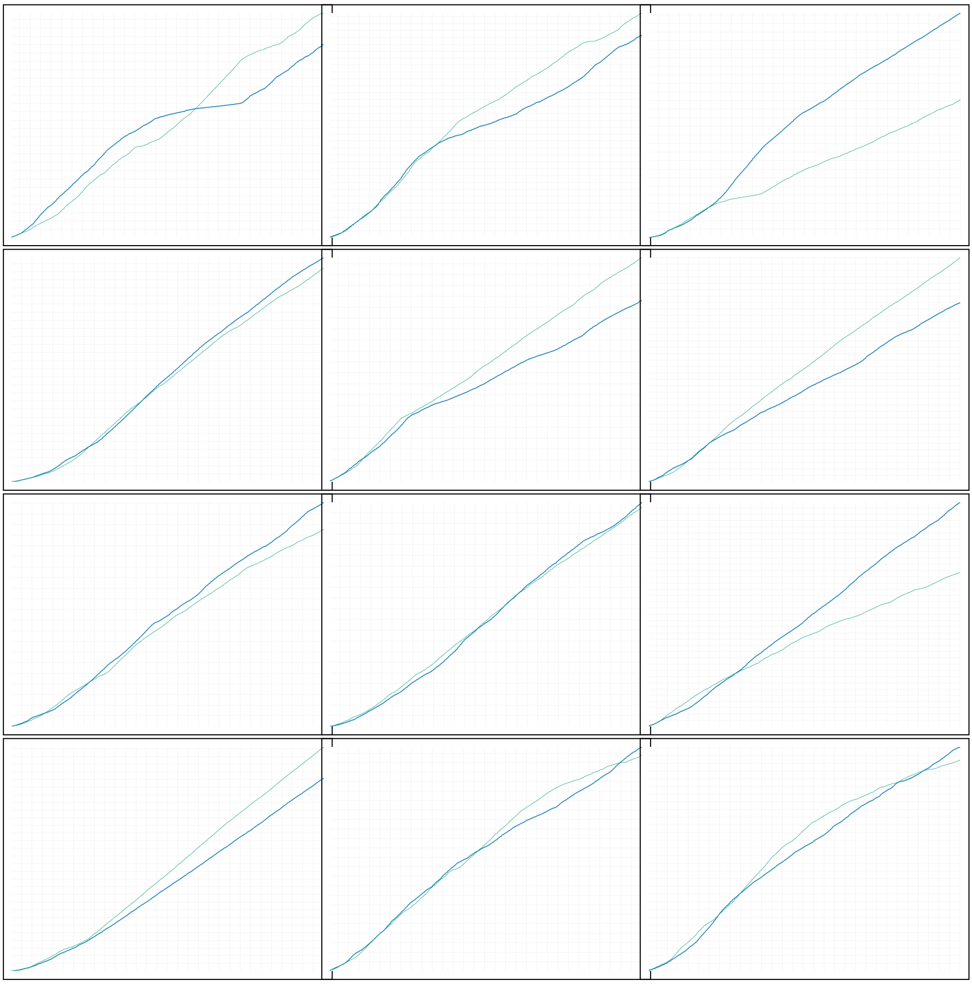


Figure E.2: Scores for the last 4 test tournaments. Rows are simulations (1, 2, 3), columns are tournaments (3 to 6). HARDAC's score is green, Python-DTU's is blue. Horizontal axis are the steps, vertical axis are the total score at a step.

Appendix F

The source code of HARDAC

F.1 HARDAC.mas2g

```
%% The agent team's mas2g file
%% This file contains several parameters required for launching the
    GOAL agent team

environment {
  "eismassim-2.0.jar".
}

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

launchpolicy {
  % Launch all the agents with a name corresponding to the one they
      have in the simulation
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC1:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC2:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC3:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC4:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC5:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC6:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC7:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC8:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC9:mapc.
  when [type=mars2012entityunknown,max=1]@env do launch HARDAC10:mapc.
```

```
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC11:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC12:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC13:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC14:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC15:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC16:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC17:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC18:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC19:mapc.
    when [type=mars2012entityunknown,max=1]@env do launch HARDAC20:mapc.
}
```

F.2 HARDAC.goal

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

    % Contains some rules that allow the agent to extract information
        from the percepts
    #import "perceptKnowledge.pl".

    % Contains role specific knowledge rules
    #import "roleKnowledge.pl".

    % Contains algorithms used for pathfinding
    #import "dijkstra.pl".

    % Contains rules about navigational subjects
    #import "navigationKnowledge.pl".
  }

  beliefs{
    % Makes sure the agent doesnt try to execute actions while the
        server is not started on startup
    doneAction.
    donePercepts.
    doneMailing.

    % Our team name
    team('HARDAC').

    ready.
  }

  goals{
    % Goals are dynamically inserted in the percept rules later on
  }

  % Define actions that can be sent to the environment interface
  % Also specify what needs to be true in order to perform the actio
  % and what should be inserted into the belief base afterwards
  actionspec{
    % Insert doneAction after each action to make sure no new actions
        are performed in this step(manual scheduling)
    % 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

    recharge {
      pre { true }
      post { doneAction }
    }
    buy(Upgrade) {
      pre { not(disabled), moneyGE(2), energyGE(2), role('Saboteur') }
```

```
      post { doneAction }
    }
    probe {
      pre { not(disabled), energyGE(1), role('Explorer') }
      post { doneAction }
    }
    parry {
      pre { not(disabled), energyGE(2), not(role('Explorer')), not(role('Inspector')) }
      post { doneAction }
    }
    survey {
      pre { not(disabled), energyGE(1) }
      post { doneAction }
    }

    % Only move over an edge when you actually have enough energy to do so
    % OR: Sometimes the edge you want to cross is not surveyed yet, but do make sure you try to move to a neighbour
    % (see canGoto/2 in navigationKnowledge.pl)
    goto(There) {
      pre { currentPos(Here), canGoto(Here, There) }
      post{ doneAction }
    }

    skip {
      pre { true }
      post { doneAction }
    }
    % Only repair agents of the same team, on your location, and not yourself!
    repair(Agent) {
      pre { energyGE(3), currentPos(Here), team(Team), me(Me), visibleEntity(Agent, Here, Team, _), Agent \= Me, role('Repairer') }
      post { doneAction }
    }
    % Only attack enemies on your location. Keep track of who you last attacked for strategic purposes
    attack(Agent) {
      pre { not(disabled), energyGE(2), currentPos(Here), visibleEntity(Agent, Here, Team, _), enemyTeam(Team), lastAttacked(X), role('Saboteur') }
      post { not(lastAttacked(X)), lastAttacked(Agent), doneAction }
    }
    inspect {
      pre { not(disabled), energyGE(2), role('Inspector') }
      post { doneAction }
    }
  }
}

% Main module which is executed every cycle, rules are considered linearly by default
```

```
main module{
  knowledge {
    dangerousPosition :- currentPos(Here), not(safePos(Here)), not(pathClosestRepairer(Here,_,_,[Here,_|_],_)).
  }
  program {
    % Only try to find a new action when one was not chosen in this step yet
    if bel(not(doneAction)) then {

      % If disabled get yourself fixed as soon as possible
      if bel(disabled, not(role('Repairer'))) then disabled.

      % Perform specific behavior when we have the entire map
      %if bel(allMapAreBelongToUs) then superioritySelect.

      % We should be very cautious if we risk being disabled and cannot find a repairer
      if bel(dangerousPosition, not(role('Saboteur')), not(role('Repairer'))) then defense.

      % Otherwise enter your role specific module to do something useful with your role
      if bel(role('Repairer')) then repairerAction.
      if bel(role('Inspector')) then inspectorAction.
      if bel(role('Explorer')) then explorerAction.
      if bel(role('Saboteur')) then saboteurAction.
      if bel(role('Sentinel')) then sentinelAction.

      % Aparently you had nothing role specific to do, so do some exploring
      if bel(true) then explore.

      % If no action could be found just send a skip to 'no valid action received in time'
      if bel(true) then skip.
    }
  }
}

% Importing all the modules that are used for choosing an action

% This is a module that contains common behavior that each agent should perform
#import "common.mod2g".

% The following modules contain role specific behavior
#import "explorer.mod2g".
#import "saboteur.mod2g".
#import "repairer.mod2g".
#import "sentinel.mod2g".
#import "inspector.mod2g".

% This module contains general behavior for disabled agents, but not Repairers
```

```
#import "disabled.mod2g".

% This module contains some administrative rules that have to be
    performed after specific actions
#import "actionProcessing.mod2g".

% This module contains rules that allow for pathfinding and moving
#import "pathing.mod2g".

% This module contains rules required by an agent to defend itself in
    times of danger
#import "defense.mod2g".

% 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
event module{
  program{
    % When a new step is detected allow the program to process the
        percepts, mails from other agents and choose a new action
    if bel(percept(step(Current)), step(Old), !, Old \= Current) then {
      if bel(Old == unknown)
        then insert(not(step(Old)), not(donePercepts), not(doneMailing)
            , not(doneAction), step(Current)).
      % The integer part is to keep unknown from getting in the
          arithmetic.. should be catched by the rule above but
          sometimes isn't
      if bel(integer(Old), Current > Old)
        then insert(not(step(Old)), not(donePercepts), not(doneMailing)
            , not(doneAction), step(Current)).
    }

    % simEnd, reset the agents belief base and also stop the agent from
        sending actions
    if bel(percept(simEnd)) then resetBeliefs.

    % if the percepts and mails are not handled do so, and make sure it
        doesn't happen again before the next step
    if bel(not(donePercepts)) then selectPercepts + insert(donePercepts
        ).
    if bel(donePercepts, not(doneMailing)) then selectReceiveMail +
        insert(doneMailing).

    % simStart percepted, but im not ready for a new match! Quickly
        prepare for a new match
    % BUG: agents reset themselves during the tournament! <--- seems to
        be fixed in MAPC 2012
    if bel(percept(simStart), not(ready)) then resetBeliefs.

    % simStart percepted and ready, handle the simStartpercepts and
        allow the program to send actions again
    if bel(percept(simStart), ready) then delete(ready) +
        simStartPercepts.
  }
}
```

F.3 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)).
    if true then insert(currentPos(unknown), zoneScore(unknown), health
        (unknown)).
    if true then insert(decidedSwarmAt(0), currentSwarmValue(0),
        swarmPosition(unknown), harassStart(0)).
```

```
    if true then insert(needExploring(unknown)).

    % Insert a dummy value for our teammates' positions
    forall bel(me(Me), !, agent(Agent), Me \= Agent) do insert(
        teamStatus(Agent, unknown, 10)).

    % 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('')).

    % Drop any goals that we may have
    if goal(optimum) then drop(optimum).
    if goal(swarm) then drop(swarm).
    forall goal(harass(X)) do drop(harass(X)).
    forall goal(hunt(X)) do drop(hunt(X)).
    forall goal(repairing(X)) do drop(repairing(X)).

    % 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(lastAttacked(X)) then delete(lastAttacked(X)).
    forall bel(lastInspect(Id, X)) do delete(lastInspect(Id, X)).
    forall bel(needExploring(X)) do delete(needExploring(X)).
    if bel(decidedSwarmAt(X)) then delete(decidedSwarmAt(X)).
    if bel(currentSwarmValue(X)) then delete(currentSwarmValue(X)).
    if bel(repairing(X)) then delete(repairing(X)).
    if bel(harassStart(X)) then delete(harassStart(X)).

    % Throw out information from the previous match
    if bel(health(H)) then delete(health(H)).
    if bel(steps(X)) then delete(steps(X)).
    if bel(vertices(X)) then delete(vertices(X)).
    if bel(edges(X)) then delete(edges(X)).
    if bel(swarmPosition(X)) then delete(swarmPosition(X)).

    % Forget your mates status (in case of a new random assignment)
    forall bel(role(Id, Role)) do delete(role(Id, Role)).
    forall bel(teamStatus(Id, Pos, HP)) do delete(teamStatus(Id, Pos,
        HP)).
```

```

    % More garbage deleting
    forall bel(enemyStatus(Id, Vertex, State)) do delete(enemyStatus(Id, Vertex, State)).
    if bel(currentPos(X)) then delete(currentPos(X)).
    if bel(lastPos(X)) then delete(lastPos(X)).
    if bel(step(X)) then insert(not(step(X))).
    if bel(zoneScore(X)) then delete(zoneScore(X)).
    if bel(oldZone(X)) then delete(oldZone(X)).
    forall bel( vertex(Id, Value, List) ) do delete( vertex(Id, Value, List) ).
    forall bel(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange))
      do delete(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange)).
    forall bel(doneProbing(V)) do delete(doneProbing(V)).

    % After deleting all garbage make sure no new actions are sent, and the agent is ready for a new simstart
    if true then insert(donePercepts, doneMailing, doneAction, ready).
  }
}

% Module that processes percepts that are received from the environment
module commonPercepts{
  knowledge{
    statusUser(ID) :- role(ID,'Saboteur').
    statusUser(ID) :- role(ID,'Repairer').
    statusChanged  :- lastAction(goto), lastActionResult(successful), !.
    statusChanged  :- oldHealth(OHP), health(HP), OHP \= HP.
  }
  program[order=linearall]{
    % Record any new vertices
    forall bel(percept(visibleVertex(V,_)), vertex(V,unknown,OldNBs), needSurvey(V), visibleEdgesList(V,NBs), union(OldNBs,NBs,Tmp), sort(Tmp,NewNBs), length(OldNBs,M1), length(NewNBs,M2), M1 < M2)
      do insert(not(vertex(V,unknown,OldNBs)), vertex(V,unknown,NewNBs)) + send(allother,newPerceivedVertex(V,NewNBs)).
    forall bel(percept(visibleVertex(V,_)), not(vertex(V,_,_)), visibleEdgesList(V,Tmp), sort(Tmp,NBs), length(NBs,M), M > 1)
      do insert(vertex(V,unknown,NBs)) + send(allother, newPerceivedVertex(V,NBs)).


    %Keep track of zoneScore
    if bel(percept(zoneScore(Z)), zoneScore(X), oldZone(Y)) then insert( not(zoneScore(X)), not(oldZone(Y)), oldZone(X), zoneScore(Z)).

    % Keep track of the vertex you were on before you got here.
    if bel(percept(position(Cur)), currentPos(Old), !, Old \= Cur) then {
      if bel(tookShortcut) then delete( tookShortcut ).
```

```
        if bel(lastPos(P)) then insert (not(lastPos(P)), lastPos(Old)).
    }

    % Update current location
    if bel(percept(position(Cur)), currentPos(Old))
      then insert(not(currentPos(Old)), currentPos(Cur)).

    % Swarm goal managing, when we have received a swarm position
    if not(goal(swarm)), bel(getOptimum(_), timeToSwarm) then adopt(
        swarm).

    % Check if the found optimum wasn't wrong
    forall bel(optimum(O), currentPos(Here), vertex(Here, Value, _),
        vertex(O, OValue, _), vertexValueGT(Value,OValue))
      do insert(not(optimum(O)), optimum(Here)) + send(allother,
          optimum(Here)).

    % Temporarily record our previous health
    if bel(health(HP)) then insert(oldHealth(HP)).

    % Update the agents health
    if bel(percept(health(H)), health(Current), !, H \= Current) then
        insert(not(health(Current)), health(H)).

    % If you can see an enemy and an ally cannot then inform the agent
    if bel(agentRankHere(0), enemyTeam(T), me(Me), currentPos(V),
        findall([E,P,X], (visibleEntity(E,P,T,X), not(enemyStatus(E,P,X
        ))), L), L \= []) then {
      forall bel(agent(ID), ID \= Me, statusUser(ID), not(visibleEntity
          (ID,V,_,_))) do send(ID,enemyStatusPack(L)).
    }

    % Keep track of the status of enemy agents
    forall bel(enemyTeam(T), visibleEntity(ID,Vertex,T,Status), not(
        enemyStatus(ID,_,_)))
      do insert(enemyStatus(ID,Vertex,Status)).
    forall bel(enemyStatus(ID,StoredVertex,StoredStatus), visibleEntity
        (ID,ActualVertex,_,ActualStatus), (StoredVertex \= ActualVertex
         ; StoredStatus \= ActualStatus))
      do insert(not(enemyStatus(ID,StoredVertex,StoredStatus)),
          enemyStatus(ID,ActualVertex,ActualStatus)).

    % Tell the others where you are
    if bel(percept(position(Pos)), statusChanged, health(HP)) then send
        (allother,teamStatus(Pos,HP)).

    % Delete temporary atom
    if bel(oldHealth(X)) then delete(oldHealth(X)).
  }
}

% Module that processes messages from other agents
module commonReceiveMail{
  knowledge{
```

```
    neighborUnion(L, NBs) :- flatten(L,A), findall([unknown,X], (member
        (X,A), atom_chars(X,Chrs), append([v,e,r,t,e,x],_,Chrs)), B),
        sort(B,NBs).
  }
  program[order=linearall]{
    % Record any new vertices
    forall bel(received(A,newPerceivedVertex(V,NBs)), not(vertex(V,_,_)
        )) do insert(vertex(V,unknown,NBs)) + delete(received(A,
        newPerceivedVertex(V,NBs))).
    forall bel(received(A,newPerceivedVertex(V,NBs)), vertex(V,unknown,
        OldNBs), length(OldNBs, M1), length(NBs,M2), M1 < M2) do insert
        (not(vertex(V,unknown,OldNBs)), vertex(V,unknown,NBs)) + delete
        (received(A,newPerceivedVertex(V,NBs))).

    % Fix any inconsistencies, because if multiple agents sends
        messages concerning the same vertex but with different
        neighbors (happens because not all edges are visible from the
        furthest visible vertex) then GOAL inserts the vertex for each
        unique list of neighbors!
    forall bel(vertex(V,unknown,_), findall(X, vertex(V,unknown,X), Tmp
        ), sort(Tmp,L), L = [_,_|_], neighborUnion(L,RealNBs), member(
        FalseNBs, L)) do insert(not(vertex(V,unknown,FalseNBs)), vertex
        (V,unknown,RealNBs)).

    % Update edge/node values for (non)existing vertices
    forall bel(received(A,vertex(Id,Value,NewList)), not(vertex(Id,_,_)
        )) do
      delete(received(A,vertex(Id,Value,NewList))) + insert(vertex(Id,
          Value,NewList)).
    forall bel(received(A,vertex(Id,Value,NewList)), vertex(Id,Value,
        OldList)) do
      delete(received(A,vertex(Id,Value,NewList))) + insert(not(vertex(
          Id,Value,OldList)), vertex(Id,Value,NewList)).

    % Update probe values for (non)existing vertices
    forall bel(received(A,vertexProbed(Id,Value,List)), not(vertex(Id,_
        ,_)))
      do delete(received(A,vertexProbed(Id,Value,List))) + insert(
          vertex(Id,Value,List)).
    forall bel(received(A,vertexProbed(Id,Value,TheirList)), vertex(Id,
        unknown,List))
      do delete(received(A,vertexProbed(Id,Value,TheirList))) + insert(
          not(vertex(Id,unknown,List)), vertex(Id,Value,List)).

    % Swarm location receiving
    if bel(received(Agent, swarmPosition(Opt)), swarmPosition(Old))
      then insert(not(swarmPosition(Old)), swarmPosition(Opt)) + delete
          (received(Agent, swarmPosition(Opt))).

    % Agent roles
    forall bel(received(Agent,role(Role))) do insert(role(Agent,Role))
        + delete(received(Agent,role(Role))).

    % Agent locations and status
```

```
    forall bel(received(Agent,teamStatus(Pos,HP)), teamStatus(Agent, OldPos,OldHP), (Pos \= OldPos ; HP \= OldHP))
      do insert(teamStatus(Agent,Pos,HP)) + delete(teamStatus(Agent, OldPos,OldHP), received(Agent,teamStatus(Pos,HP))).
    forall bel(received(Agent,teamStatus(Pos,HP))) do delete(received(Agent,teamStatus(Pos,HP))).

    % inspectedEntities
    % When you get a percept of an inspected enemy, replace the last inspection of that entity.
    forall bel(received(_, inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange)),
        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)).
    % When you get a percept of an inspected enemy, and it has never been inspected before, insert it.
    forall bel(received(_, inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange)),
        not(inspectedEntity(Id, _, _, _, _, _, _, _, _, _)))
      do insert(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange)).

    % Save any new information about the enemy.
    forall bel(received(ID,enemyStatusPack(L))) do {
      forall bel(member([E,V,X], L)) do {
        if bel(not(enemyStatus(E,_,_))) then insert(enemyStatus(E,V,X)).
        if bel(enemyStatus(E,OV,Y), (V \= OV ; X \= Y), not(visibleEntity(E,_,_,_))) then insert(not(enemyStatus(E,OV,Y)), enemyStatus(E,V,X)).
      }
    }
    forall bel(enemyStatus(A,B1,C1), !, enemyStatus(A,B2,C2), (B1 \= B2 ; C1 \= C2)) do delete(enemyStatus(A,B2,C2)).
  }
}

% Clears out received messages and sent messages, these are now processed and irrelevant, hence slowing down the queries for no reason
module clearMailbox{
  program[order=linearall]{
    forall bel(received(Agent,Message)) do delete(received(Agent, Message)).
    forall bel(sent(Agent,Message)) do delete(sent(Agent,Message)).
  }
}

% Behavior when swarming
module swarm{
  program{
    if bel(getOptimum(Pos), neighbour(Pos)) then advancedGoto(Pos).
```

```
    if bel(getOptimum(Pos), currentPos(V), V \= Pos, path(V,Pos,[V,Next|_],_)) then advancedGoto(Next).
    if bel(currentPos(V), getOptimum(V), enemyHere(ID), dangerousEnemy(ID)) then defense.
    if true then recharge.
  }
}


% 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 neighbor, 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), vertexValueGE(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).
  }
}
```

F.4 defense.mod2g

```
module defense{
  knowledge{
    needToParry :- currentPos(Here), !, enemyTeam(T), visibleEntity(ID,
        Here,T,normal), dangerousEnemy(ID).
  }

  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 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 cannot parry (and is not swarming) then just run away
    if not(goal(swarm)) then defenseFlee.
  }
}

module defenseParry{
  program{
    % randomly pick flee or parry when last parry was useless.
    if bel(lastAction(parry), lastActionResult(useless)) then
        randomDefense.
    if true then parry.
    if true then recharge.
  }
}

module randomDefense{
  program{
    % Keep 75% chance to parry, 25% to flee
    if bel(randomFloat(R), R > 0.25) then {
      if true then parry.
      if true then recharge.
    }
    if not(goal(swarm)) then defenseFlee.
  }
}

module defenseFlee{
  program{
    % run away if needed.
    if bel( currentPos(Here), not(needSurvey(Here))) then {
      % to a safe spot.
      if bel( neighbour(N), safePos(N)) then advancedGoto(N).
      % to a safer spot which isn't where I was last step.
      if bel( neighbour(N), not((visibleEntity(_, N, Team, _),
          enemyTeam(Team))), not(lastPos(N))) then advancedGoto(N).
```

```
      % to a safer spot.
      if bel( neighbour(N), not((visibleEntity(_, N, Team, _),
          enemyTeam(Team)))) then advancedGoto(N).
    }

    % max edge Weight is 9.
    if bel( energyGE(9), currentPos(Here) ) then {
      % to a safe spot.
      if bel( visibleEdge(Here,N), safePos(N)) then advancedGoto(N).
      % to a safer spot which isn't where I was last step.
      if bel( visibleEdge(Here,N), not((visibleEntity(_, N, Team, _),
          enemyTeam(Team))), not(lastPos(N))) then advancedGoto(N).
      % to a safer spot.
      if bel( visibleEdge(Here,N), not((visibleEntity(_, N, Team, _),
          enemyTeam(Team)))) then advancedGoto(N).
    }

    if true then recharge.
  }
}
```

F.5 disabled.mod2g

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

module disabled{
  program{
    % If we are at a vertex with a Repairer (and we are not a Repairer) then we should just wait
          if bel(not(role('Repairer')), currentPos(V), visibleEntity(ID,V,_,_), role(ID,'Repairer')) then recharge.

    % Wait for nearby Repairer when you are a Repairer and other Repairer has a higher priority.
    if bel(role('Repairer'), role(Agent,'Repairer'), me(Name), Agent \= Name, visibleEntity(Agent,Pos,_,_), (neighbour(Pos) ; currentPos(Pos)), compareAgents(Name,Agent,Agent)) then recharge.

    % Wait for nearby Repairer when you are not a Repairer
    if bel(not(role('Repairer')), role(Agent,'Repairer'), visibleEntity(Agent,Pos,_,_)) then {
      if not(goal(swarm)), bel(randomFloat(X)) then {
        if bel(X > 0.25, neighbour(Pos)) then recharge.
        if bel(X =< 0.25, neighbour(Pos)) then advancedGoto(Pos).
        if bel(currentPos(Pos)) then recharge.
      }
    }

    % Find nearest Repairer.
    if bel(neighbour(V), team(T), visibleEntity(ID,V,T,_), role(ID,'Repairer')) then advancedGoto(V).
    if bel(currentPos(Here), pathClosestRepairer(Here,_,_,[Here,Next|_],_)) then advancedGoto(Next).

    % Goto nearest unknown vertex to expand the known graph, hopefully enabling a path to a Repairer
    if bel(neighbour(N), not(vertex(N,_,_))) then advancedGoto(N).
    if bel(currentPos(Here), pathClosestUnknownVertex(Here,_,[Here,Next|_],_), not(lastPos(Next))) then advancedGoto(Next).

    % If there are no unknown vertices then why can't you find a path to the nearest Repairer?
    if true then recharge.
  }
}
```

F.6 saboteur.mod2g

```
%Saboteur specific Percept handeling
module saboteurPercepts{
  program[order=linearall]{
    % When an agent is under attack (and hopefully swarming) then help
        it if we are the right Saboteur
    if not(goal(harass(_))), not(goal(hunt(_))), bel(timeToHunt, !,
        findall(EID, (teamStatus(ID,V,_), role(ID,R), member(R,['
        Inspector','Sentinel','Explorer']), enemyStatus(EID,V,normal),
        inspectedEnemy(EID,'Saboteur')), L), randomElement(L,Enemy))
      then adopt(hunt(Enemy)).
  }
}

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

module saboteurAction{
  knowledge {
    enabledAllySaboteursHere :- currentPos(V), me(Me), team(T),
        visibleEntity(ID,V,T,normal),
                                role(ID,'Saboteur'), ID \= Me.
    enabledEnemiesHere(Role,S) :- currentPos(V), enemyTeam(T), !,
              findall(ID,(visibleEntity(ID,V,T,normal),inspectedEnemy(
                  ID,Role)),L), !, sort(L,S).
    enabledEnemiesHereNotInList(Ignored,S) :- currentPos(V), enemyTeam(
        T), !,
              findall(ID,(visibleEntity(ID,V,T,normal),not(memberchk(ID
                  ,Ignored))),L), !, sort(L,S).

    % Prioritize Saboteurs and Repairers over the others and prioritize
         Sentinel lowest
    roleSortedEnemyList(L) :- enabledEnemiesHere('Saboteur',SL),
        enabledEnemiesHere('Repairer',RL),
              enabledEnemiesHere('Sentinel',SeL), append(SL,RL,Tmp1),
                  append(Tmp1,SeL,Tmp2),
              enabledEnemiesHereNotInList(Tmp2,OL), append(Tmp1,OL,Tmp3
                  ), append(Tmp3,SeL,L).

      roleSelectAttackTarget(Target) :-
                    me(Me), agentEnabledRoleRankHere(Me,Rank),
                        roleSortedEnemyList(EL), nth0(Rank,EL,Target)
                        .

    % Prevent a gigantic endless battle between enemy Repairers and us
        at this vertex
    notRepairBlob :- enabledEnemyHere(EEID), dangerousEnemy(EEID), !.
    notRepairBlob :- enabledEnemiesHere('Repairer',ERL), length(ERL,N),
        N < 3.

```

```
    lowestRank(L) :- me(Me), length(L,N), agentRank(L,Me,Rank), M is N -1, Rank == M.
  }
  program{
    % Determine if it is time to buy upgrades
    if true then upgrades.

    % Hunt if we are supposed to help somebody
    if not(goal(swarm)), a-goal(hunt(ID)) then hunt.

    % Try to harass the enemy's swarms if we are harassing
    if a-goal(harass(There)) then harassGoto.

    % Harass sometimes
    if bel(randomFloat(X), !, X > 0.5) then harassBegin.

    % If we have been attacking a Sentinel, and it parries, and there are an Explorer or Inspector nearby then attack that enemy instead
    % (This can actually happen a lot when harassing)
    if bel(not((enemyHere(X), dangerousEnemy(X))), lastActionResult(failed_parry), lastActionParam(SID), inspectedEnemy(SID,'Sentinel'), Rs = ['Explorer','Inspector']) then {
      if bel(enabledEnemyHere(ID), currentPos(V), inspectedEnemy(ID,R), member(R,Rs)) then saboteurAttack(ID,V).
      if bel(enabledEnemyNear(ID,V), inspectedEnemy(ID,R), member(R,Rs)) then saboteurAttack(ID,V).
    }

    % If we are at a large battle (i.e. we some of us Saboteurs are not needed) then we should move
    if bel(currentPos(V), largeBattle(V,AL), (not(roleSelectAttackTarget(_)) ; (sort([Me|AL],AL2), lowestRank(AL2)))) then {
      % Harass so we can get away
      if true then harassBegin.

      if bel(enabledEnemyNear(ID,Vertex), Vertex \= V) then saboteurAttack(ID,Vertex).
      if bel(findall(N,neighbour(N),L), randomElement(L,X)) then advancedGoto(X).
    }

    % Attack enemy by rank if there are more ally Saboteurs here
    if bel(currentPos(V), enabledAllySaboteursHere, roleSelectAttackTarget(ID), notRepairBlob) then saboteurAttack(ID,V).

    % Attack enemy on this vertex.
    % Preference to hit Saboteur over other targets.
        % We prefer Explorers over Inspectors because they have less health
        % We normally prefer to hit Inspectors and Explorers over Repairers because they cannot parry.
        if bel(enabledEnemyHere(ID), currentPos(V)) then {
```

```
                if bel(inspectedEnemy(ID,'Saboteur')) then saboteurAttack(ID,V).
                if bel(inspectedEnemy(ID,'Inspector')) then saboteurAttack(ID,V).
                if bel(inspectedEnemy(ID,'Repairer'), notRepairBlob) then saboteurAttack(ID,V).
                if bel(inspectedEnemy(ID,'Explorer')) then saboteurAttack(ID,V).
                if bel(notRepairBlob) then saboteurAttack(ID,V).
            }

        % if the other saboteur is also at your location split up.
        if bel(currentPos(Vertex),enabledEnemyNear(_,Y),!, visibleEntity(ID,Vertex,_,_), role(ID,'Saboteur'), not(me(ID)),!,
          enabledEnemiesNear(List), agentRankHere(Rank) ) then gotoSplit(Rank,List).

        %Attack enemy on nearby vertex
        if bel(enabledEnemyNear(ID,Vertex), currentPos(V), Vertex \= V, not(inspectedEnemy(ID,'Sentinel')), notLargeBattle(Vertex)) then saboteurAttack(ID,Vertex).
        if bel(enabledEnemyNear(ID,Vertex), currentPos(V), Vertex \= V, notLargeBattle(Vertex)) then saboteurAttack(ID,Vertex).

        % Attack enemies on optimums
        if bel(currentPos(V), optimum(Opt), enemyStatus(ID,Opt,normal), inspectedEnemy(ID,R), member(R,['Inspector','Explorer']), path(V,Opt,[V,Next|_],_)) then advancedGoto(Next).
        if bel(currentPos(V), optimum(Opt), enemyStatus(ID,Opt,normal), path(V,Opt,[V,Next|_],_)) then advancedGoto(Next).

        %attack nearest visible enemy (only works in zones because otherwise it would have already been handled above)
        if bel( currentPos(Start), pathClosestVisibleEnemy(Start, LocationEnemy, NameEnemy, [Here,Next|Path], Dist),! )
          then advancedGoto(Next).

        % Harass if nothing else, if we can find a suitable vertex
        if true then harassBegin.

        %Fail save
        if true then explore.
    }
}

module saboteurAttack(ID,Vertex){
  program{
    % Attack target if on this location.
    if bel( currentPos(Vertex) ) then {
      %If your last attack action was at the same target who parried and there is another active target hit the other instead
      if bel( lastActionResult(failed_parry), lastAttacked(ID),!, enabledEnemyHere(AID), AID \== ID ) then attack(AID).
      if true then attack(ID).
      if true then recharge.
```

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

%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,Next|List],_))
        then advancedGoto(Next).
      % if you can't find target then drop the hunt
      if goal(hunt(ID)) then drop(hunt(ID)).
    }
}

% Used to determine if we need to buy upgrades
module upgrades{
    % We should probably not buy more health than strength because it is less useful. And if we have more health than them then they will buy more strength, which in turn would make us buy more health (3 is the default strength)
    knowledge{
      shouldBuyStr(S) :- enemySaboteurSecondMaxHealth(Health), S < Health, !.
      shouldBuyStr(S) :- me(Me), hasLowestRoleRank(Me), S < 6, !. % At least one Saboteur should be able to kill anybody in one round.
      shouldBuyHP(H)  :- enemySaboteurSecondMaxStrength(Strength), H =< Strength, !.
    }
    program{
      if bel(timeToBuy, not((enabledEnemyHere(ID), dangerousEnemy(ID))), strength(S), maxHealth(H), money(M), M >= 4) then {
        % buy strength upgrade according to second highest inspected enemy Saboteur health
        if bel(shouldBuyStr(S)) then {
          if true then buy(sabotageDevice).
          if true then recharge.
        }

        % buy health upgrade according to second highest inspected enemy Saboteur strength
        if bel(shouldBuyHP(H)) then {
          if true then buy(shield).
          if true then recharge.
        }
      }

    }
}

% Attempt to find enemy "swarms" and harass them
```

```
module harassGoto {
  program{
    % If we are near the harassment vertex then we should just proceed
        as usual
    % Otherwise go towards the vertex
    if goal(harass(There)), bel(currentPos(Here), Here \= There, (path(
        Here, There, [Here,Next|_], _) ; (neighbour(Here,There), Next =
         There)), !) then {
      % Maybe we are at a neighbour to the harass vertex and there are
          an enemy here (in which case don't go towards the harass
          vertex)
      if bel(enabledEnemyHere(ID), not(neighbour(Here,There))) then {
        % Maybe we are at a vertex together with an enemy and we might
            want to move towards our goal instead of attacking
        if bel(enabledEnemyHere(ID), randomFloat(X), !, X > 0.5) then
            advancedGoto(Next).

                % Otherwise, exit the module
      }
              % Otherwise move towards the vertex
              if bel(not(enabledEnemyHere(ID))) then advancedGoto(Next).
    }
  }
}

module harassBegin {
  program {
    if not(goal(harass(_))), bel(timeToHarass, possibleHarassVertex(Pos
        ), step(Step), harassStart(Old))
      then adopt(harass(Pos)) + delete(harassStart(Old)) + insert(
          harassStart(Step)).
  }
}
```

F.7 repairer.mod2g

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

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

module repairerAction{
  knowledge {
    isDamaged('Saboteur',HP)  :- HP < 3.
    isDamaged('Repairer',HP)  :- HP < 6.
    isDamaged('Inspector',HP) :- HP < 6.
    isDamaged('Explorer',HP)  :- HP < 4.
    disabledAllyHere(ID) :- currentPos(Here), team(Team), me(Me),
        visibleEntity(ID,Here,Team,disabled), ID \= Me.
    damagedAllyHere(ID) :- currentPos(Here), me(Me), teamStatus(ID,Here
        ,HP), ID \= Me, role(ID,Role), isDamaged(Role,HP).
    disabledImportantAllyHere(ID) :- disabledAllyHere(ID), role(ID,R),
        member(R,['Repairer','Saboteur']).
    damagedImportantAllyHere(ID)  :- damagedAllyHere(ID),  role(ID,R),
        member(R,['Repairer','Saboteur']).
    allyRepairersAt(V, RN) :- me(Me), findall(Id, (teamStatus(Id,V,HP),
         HP \= 0, Id \= Me, role(Id, 'Repairer')), RL), !, length(RL,RN
        ).
    enabledEnemySaboteursHere(L) :- currentPos(V), enemyTeam(T),
        findall(ID, (visibleEntity(ID,V,T,normal), dangerousEnemy(ID)),
         Tmp), sort(Tmp,L).
    enabledAllySaboteursHere(L) :- currentPos(V), team(T), findall(ID,
        (visibleEntity(ID,V,T,normal),role(ID,'Saboteur')), Tmp), sort(
        Tmp,L).

    % Likely targets for the enemy are our enabled Saboteurs. If the
        enemy has enough enabled Saboteurs to target all our enabled
        Saboteurs then our enabled Saboteurs are likely to be attacked
        the next round.
    likelyTargets(L) :- enabledEnemySaboteursHere(EL),
        enabledAllySaboteursHere(AL), length(EL,EN), length(AL,AN), ((
        EN >= AN, L = AL) ; (EN < AN, L = [])).
    disabledAt(V, DN) :- me(Me), findall(Id, (teamStatus(Id,V,0), Id
        \= Me), DL), !, length(DL,DN).
    insufficientRepairersAt(V) :- currentPos(H), H \= V,
        allyRepairersAt(V,RN), disabledAt(V,DN), DN > RN.
    enabledAllyRepairersHere :- currentPos(V), me(Me), team(T),
        visibleEntity(ID,V,T,normal), role(ID,'Repairer'), ID \= Me, !.

    % Create a list of ally agents to determine who to repair first. It
         is important to note that non-disabled agents can still be
```

```
      repaired if they are deemed to be attacked next round because both attack actions and determining disabled agents are processed before repair actions by the server.
    roleSortedDisabledHere(L) :- findall(ID,(disabledAllyHere(ID),role(ID,'Saboteur')),SLTmp),
                                findall(ID,(disabledAllyHere(ID),role(ID,'Repairer')),RLTmp),
              likelyTargets(TLTmp), sort(TLTmp,TL),
                                findall(ID,(disabledAllyHere(ID),role(ID,R),not(member(R,['Saboteur','Repairer']))),OLTmp),
                                findall(ID,damagedAllyHere(ID),DLTmp),
                                sort(SLTmp,SL), sort(RLTmp,RL), sort(OLTmp,OL), sort(DLTmp,DL),
                                append(SL,TL,Tmp), append(Tmp,RL,Tmp2), append(Tmp2,OL,Tmp3), append(Tmp3,DL,L).

    allDisabledNear(L) :- findall((V,ID),(teamStatus(ID,V,0),neighbour(V)),X),!,sort(X,S),findall(A,member((_,A),S),L).

    roleSelectRepairTargetHere(Target) :- me(Me), agentEnabledRoleRankHere(Me,Rank), roleSortedDisabledHere(RL), nth0(Rank,RL,Target).
    roleSelectRepairTargetNear(Target) :- me(Me), agentEnabledRoleRankHere(Me,Rank), allDisabledNear(RL), nth0(Rank,RL,Target).

    disabledAgentToRepair(Agent,There) :- me(Me), currentPos(Pos),
      findall((ID,V), (teamStatus(ID,V,0), ID \= Me, V \= Pos), L1),
      findall((ID,V), (teamStatus(ID,V,0), ID \= Me, V \= Pos, role(ID,R), member(R,['Sentinel','Repairer','Saboteur'])), L2),
      append(L1,L2,L), randomElement(L,(Agent,There)), !.

  }
  program{
    % Repair the agents that I have committed myself to repair if they are close by
    if a-goal(repairing(ID)), bel(not(disabled), currentPos(V), teamStatus(ID,Pos,_)) then {
      if bel(Pos == V) then repairerRepair(ID,V).
      if bel(neighbour(Pos)) then repairerRepair(ID,Pos).
      if bel(me(Me), hasLowestRoleRank(Me), path(V,Pos,[V,N|T],_)) then repairerRepair(ID,N).
    }

    % It is necessary to repair the other agents that are not at a large battle
    if bel(currentPos(V), largeBattle(V,_), me(Me), hasLowestRoleRank(Me)) then repairCommitBegin.

    % Fix ally here, delegating the repair tasks among all the ally Repairers at this vertex
    if bel(currentPos(V), enabledAllyRepairersHere, roleSelectRepairTargetHere(ID)) then repairerRepair(ID,V).
```

```

    % Fix ally here, when there are no other enabled ally repairers here, prioritizing Saboteurs and Repairers
    if bel(currentPos(V), not(enabledAllyRepairersHere)) then {
      if bel(disabledImportantAllyHere(ID)) then repairerRepair(ID,V).
      if bel(damagedImportantAllyHere(ID)) then repairerRepair(ID,V).
      if bel(disabledAllyHere(ID)) then repairerRepair(ID,V).
      if bel(damagedAllyHere(ID)) then repairerRepair(ID,V).
    }

    % Fix a nearby ally. It is important that not all Repairers at the vertex moves to repair the same target.
    % It is also important that not too many Repairers move a lot if they currently are in a large battle, because this could shift the battle towards our swarms as the disabled allies are probably coming from the swarms
    % (they cannot come from large battles and not many agents are doing much else than swarming or fighting at a vertex).
    if bel(disabledAllyNear(ID,Vertex), insufficientRepairersAt(Vertex), currentPos(Pos)) then {
      if bel(enabledAllyRepairersHere, not(roleSelectRepairTargetHere(_)), roleSelectRepairTargetNear(ID2)) then repairerRepair(ID2,Vertex).
      if bel(not(enabledAllyRepairersHere)) then repairerRepair(ID,Vertex).
    }

    % Go towards the agent that I want to repair
    if a-goal(repairing(ID)), bel(not(disabled), currentPos(H), teamStatus(ID,T,_), path(H,T,[H,N|R],_)) then repairerRepair(ID,N).

    % Find an ally to repair and commit to it
    if true then repairCommitBegin.

    % Find help, because I am disabled.
    if bel(disabled) then disabled.

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

    % Swarm if I should swarm
    if a-goal(swarm) then swarm.

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

module repairerRepair(ID,Vertex){
  program{
    if bel(me(Me), ID \= Me) then {
      % Repair target at this vertex if possible. Defend yourself if necessary.
      if bel(currentPos(Here), visibleEntity(ID,Here,_,_)) then {
```

```
        if true then repair(ID).
        if bel(not(safePos(Here))) then defense.
        if true then recharge.
      }
      % Goto vertex with disabled/injured agent.
      if true then advancedGoto(Vertex).
    }
  }
}

module repairCommitBegin {
  program {
    % Find an ally to repair and commit to it
    if not(goal(repairing(_))), bel(not(disabled), currentPos(Here),
        disabledAgentToRepair(Agent,There)) then {
      if bel(neighbour(There)) then repairerRepair(Agent,There) + adopt
          (repairing(Agent)).
      if bel(path(Here,There,[Here,Next|_],_)) then repairerRepair(
          Agent,Next) + adopt(repairing(Agent)).
    }
  }
}
```

F.8 explorer.mod2g

```
% Belief base management specific to the Explorers
module explorerPercepts{
  knowledge {
    calcSwarms(Chosen, Value) :-
        decideOptimums(Opts), findall((Val,Swarm), (member(Opt,Opts), calcAreaControl(Opt,Swarm,Val)), L),
        sort(L,S), length(S,N), nth1(N,S,(Value,Chosen)), !.

    % Python-DTU's algorithm for calculating swarm positions, reimplementation in GOAL

    % calcAreaControl returns pairs of agents and vertices which determine where the agents shall stand when swarming
    % Chosen = agent-vertex pairs
    calcAreaControl(Opt,Chosen,Value) :-
        allVertices(Tmp), delete(Tmp,Opt,Vs),
        swarmAgents([A|AT]), cacAux(Vs,AT,[Opt],Rest), Chosen = [(A,Opt)|Rest],
        swarmValue(Chosen, Value), !.

    cacAux(_,[],_,[]).
    cacAux(Vs,[A|T],Chosen,[(A,Best)|Rest]) :-
        calcOwned(Chosen, Owned), bestPosition(Vs,Chosen,Owned,Best), cacAux(Vs,T,[Best|Chosen],Rest).

    allVertices(Vs) :- findall(V, (vertex(V,Val,_), Val \= unknown, Val \= 1), L), sort(L,Vs), !.
    swarmAgents(As) :- timeToSwarm, swarmAgents(As,['Saboteur','Repairer']), !.
    swarmAgents(As) :- swarmAgents(As,['Saboteur','Repairer','Explorer']), !.
    swarmAgents(As,IgnoredRoles) :- findall(A, (agent(A), role(A,R), not(memberchk(R,IgnoredRoles))), L), sort(L,As), !.

    swarmValue(Chosen,Val) :- findall(V, member((_,V), Chosen), L), calcOwned(L,Owned), swarmValueAux(Owned,Val).
    swarmValueAux([],0).
    swarmValueAux([V|T],Val) :-
      swarmValueAux(T,Part), vertexValue(V,Tmp), (Tmp == unknown -> VVal = 1 ; VVal = Tmp), !, Val is Part + VVal.

    bestPosition([],_,_,_) :- fail, !.
    bestPosition(Vs,Chosen,Owned,Best) :-
        subtract(Vs,Chosen,NewVs), bpAux(NewVs,Chosen,Owned,_,Best).

    bpAux([],_,_,0,_).
    bpAux([V1|R],Chosen,Owned,MaxVal,Best) :-
      bpZoneVal(V1, Chosen, Owned, Val1), bpAux(R,Chosen,Owned,Val2,V2),
      (Val1 > Val2 -> (Best = V1, MaxVal = Val1) ; (Best = V2, MaxVal = Val2)).

```

```
    bpZoneVal(V,Chosen,Owned,Val) :-
      vertex(V,VVal,_), neighbours(V,Ns), subtract(Ns,Owned, Ws),
      bpZoneValAux(Ws,Chosen,ValPart), Val is ValPart + VVal.

    bpZoneValAux([],_,0).
    bpZoneValAux([W|R], Chosen, ValSum) :-
      neighbours(W,Tmp), intersection(Tmp,Chosen,Zs), vertex(W,Tmp2,_),
           (Tmp2 == unknown -> WVal = 1 ; WVal = Tmp2),
      bpZoneValAuxAux(Zs,WVal,ValPart1), bpZoneValAux(R,Chosen,ValPart2
           ), ValSum is ValPart1 + ValPart2.

    bpZoneValAuxAux([],_,0).
    bpZoneValAuxAux([_|R],WVal,ValSum) :-
      bpZoneValAuxAux(R,WVal,ValPart), ValSum is WVal + ValPart.

    calcOwned([],[]).
    calcOwned(Chosen,Owned) :- Chosen = [_|T], coAux(Chosen,T,O), union
         (Chosen,O,Owned).

    coAux([H],[],[H]).
    coAux([H|T],T,Owned) :- T = [N|R], neighborIntersect(H,T,O), coAux
         ([N|R],R,O2), union(O,O2,Owned).

    neighborIntersect(_,[],[]).
    neighborIntersect(V,[H|T],NI) :- neighbours(V,NV), neighbours(H,NH)
         , intersection(NV,NH,X), neighborIntersect(V,T,Y), union(X,Y,NI
         ).

    validSwarms(Swarms) :- not(memberchk((_,unknown),Swarms)), !.

    % Updates the list of nodes that still need to be probed
    updateNeedExploring(A,B) :- findall(V, (member(V,A), needProbe(V)),
         B).
  }

  program[order=linearall]{
    % If our last goto failed we are potentially under attack, fleeing
         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), safePosForProbing(Here), noFlee) then
         delete(noFlee).

    % Update needExploring if necessary
    if bel(not(needExploring(unknown)), needExploring(L), L \= [],
         updateNeedExploring(L,NewL)) then insert(not(needExploring(L)),
          needExploring(NewL)).

    % Decide on swarm positions if it is time to swarm
```

```
    if bel(timeToDecideSwarm, !, me(Me), hasHighestRoleRank(Me),
        calcSwarms(Swarms,NewVal), validSwarms(Swarms), step(NewS),
        decidedSwarmAt(OldS), NewS \= OldS, currentSwarmValue(CurVal),
        NewVal > CurVal) then {
      forall bel(member((A,Pos), Swarms)) do send(A,swarmPosition(Pos))
          + insert(not(decidedSwarmAt(OldS)), decidedSwarmAt(NewS)) +
          insert(not(currentSwarmValue(CurVal)), currentSwarmValue(
          NewVal)).
    }
  }
}

% Sending messages specific for the Explorers
module explorerReceiveMail{
  program{
    if true then exit-module.
  }
}

% Module that makes sure an action is chosen for the Explorer
module explorerAction{
  knowledge {
    % If all this nodes neighbours are probed, it is to be considered
        as 'doneProbing'
    doneProbing(Here) :- findall(V, (vertex(Here, _, L), member([_, V],
        L), needProbe(V)), []).

    % The list of vertices that still need to be probed
    calculateNeedExploring(L) :- swarmPosition(MOpt), MOpt \= unknown,
               findall(V1,(member(O,Opts), neighbour(O,V1), needProbe(
                   V1)), A),
               findall(V2,(member(N,A),    neighbour(N,V2), needProbe(
                   V2)), B),
               append(A,B,C), append(Opts,C,D), sort(D,L).

    % This predicate determines when a node is to be considered safe to
         stand on, this means no unknown role agent (we have decided
        that there are 25% chance of still being safe, because there
        are only 4 out of 20 agents that are Saboteurs) or Saboteur can
         be at this location. We do not use safePos because we need to
        take chances when probing.
    safePosForProbing(P) :- randomFloat(X), !, (safePos(P) ; X > 0.75).
  }
  program{
    % If we are at an optimum then we shouldn't look for an optimum
        anymore
    if a-goal(optimum), bel(currentPos(Pos), optimum(Pos), not(
        timeToSwarm)) then drop(optimum).

    % Agent is not safe, defend yourself
    if bel(not(noFlee), currentPos(Here), not(safePosForProbing(Here)))
         then defense.

    % probe your node if it is unprobed
```

```
    if bel( not(disabled), currentPos(Here), needProbe(Here), me(Name), team(Team),
        findall(Agent, (visibleEntity(Agent,Here,Team,_), role(Agent,'Explorer')), Agents), agentRank(Agents,Name,Rank), Rank == 0)
      then selectProbe(Rank).

    % When optimum is found but certain nodevalues still need exploring enter the module that makes sure this happens
    if not(goal(optimum)), bel(not(timeToSwarm)) then {
      if bel(needExploring(unknown), calculateNeedExploring(List)) then insert(not(needExploring(unknown)), needExploring(List)) + searchPostOptimal.
      if true then searchPostOptimal.
    }

    % If we are looking for an optimum enter the module that has optimum finding behavior
    if a-goal(optimum) then searchOptimal.

    % When swarming then swarm
    if a-goal(swarm) then swarm.
  }
}

% Module that contains behavior for Explorers to find the optimal value node
module searchOptimal {
  program {
    % if this vertex has a lower value than the last, track back to an unprobed neighbor of the last node
    if bel(lastPos(Last), currentPos(Here), !, vertexValue(Here, Value), vertexValue(Last, OldValue), vertexValueGE(OldValue,Value),!,
        neighbour(Last, New), needProbe(New), neighbour(Here, New), safePosForProbing(New))
        then advancedGoto(New) + insert(tookShortcut).

    % if this vertex has a lower value than the last, track back to the last node
    if bel(lastPos(Last), currentPos(Here), !, vertexValue(Here, Value), vertexValue(Last, OldValue), vertexValueGT(OldValue,Value),!, safePosForProbing(Last))
      then advancedGoto(Last).

    % find a probed neighboring vertex with a higher value and go to there
    if bel(currentPos(Here), vertexValue(Here, Value), !, neighbour(ElseWhere),
        vertexValue(ElseWhere, EWValue), vertexValueGT(EWValue, Value), safePosForProbing(ElseWhere) )
      then advancedGoto(ElseWhere).

    % find an unprobed neighboring vertex
    if bel(neighbour(There), needProbe(There), safePosForProbing(There)) then advancedGoto(There).
```

```

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

    % find an unprobed neighboring 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).
  }
}

% Module to search randomly after we have found an optimum
module searchPostOptimal {
  program {
    % Find the closest unprobed vertex which is a neighbor of a vertex which needs to be explored
    if bel(currentPos(Here), pathClosestNonProbedWithExtraChecks(Here, _, [Here, Next | _], _))
      then advancedGoto(Next).

    % find an unprobed neighboring vertex
    if bel(neighbour(There), needProbe(There), safePosForProbing(There)) then advancedGoto(There).

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

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

    % If all neighboring vertices has been probed determine if we need to survey
                if bel(not(disabled), currentPos(Here), needSurvey(Here), agentRankHere(Rank))
                    then selectSurvey(Rank).

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

F.9 inspector.mod2g

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

module inspectorReceiveMail{
  program[order=linearall]{
    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 towards the swarm position
    if bel(getOptimum(X), currentPos(Pos), path(Pos,X,[Here,Next|Path],
        _)) then advancedGoto(Next).

    % Randomly explore if nothing else
    if true then explore.
  }
}
```

F.10 sentinel.mod2g

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

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

module sentinelAction{
  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 swarm position if I am not in it.
    if bel(getOptimum(X), currentPos(Pos), path(Pos,X,[Here,Next|Path],
        _))
      then advancedGoto(Next).

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

F.11 pathing.mod2g

```
module gotoSplit(Rank, List){
  knowledge{
    % Data reformatting
    stripList([],[]).
    stripList([[Value,Vertex]|List],[Vertex|SList]) :- stripList(List,
        SList).
  }
  program{
    % List = [[Value,Vertex],[...]] Highest after!
    if bel(stripList(List,SList), selectDestination(SList,Rank,Vertex))
        then advancedGoto(Vertex).
    % List = [Vertex,...,Vertex]
    if bel(selectDestination(List,Rank,Vertex)) then advancedGoto(
        Vertex).
  }
}

module gotoNeighbour(Rank,Unknown,Safe){
  program{
    if bel( Unknown == true, Safe == true, maxEnergy(E), energyGE(E),
        currentPos(Here), setof(Neighbour, (visibleEdge(Here,Neighbour)
        ,
      safePos(Neighbour)), Neighbours), selectNeighbour(Neighbours,Rank
        ,Vertex))
      then advancedGoto(Vertex).

    if bel( Unknown == true, maxEnergy(E), energyGE(E), currentPos(Here
        ),setof(Neighbour, visibleEdge(Here,Neighbour), Neighbours),
      selectNeighbour(Neighbours,Rank,Vertex))
      then advancedGoto(Vertex).

    if bel( Safe == true, X is Rank + 1, setof(Neighbour, (neighbour(
        Neighbour), safePos(Neighbour)), Neighbours), selectNeighbour(
        Neighbours,X,Vertex) )
      then advancedGoto(Vertex).

    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 neighbor 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 neighbor if I am not rank 0
    if true then gotoNeighbour(Rank,true,false).
    if true then recharge.
  }
}
```

F.12 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 so 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,List)).
    % Other statement is false so do not search. Insert new vertex.
    if bel(percept(probedVertex(Id1,Value)), visibleEdgesList(Id1,List)) then
      insert(vertex(Id1,Value,List)) + send(allother,vertexProbed(Id1,Value,List)).
  }
}

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',_,_,_,_,_,_,_)), not(lastInspect(Id,_)), step(S)) then insert(lastInspect(Id,S)).
  }
}
```

F.13 dijkstra.pl

```
%% Code for the different algorithms presented here is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm

%% Dijkstra from S to T
% path(Vertex0, Vertex, Path, Dist) is true if Path is the shortest path from Vertex0 to Vertex, and the length of the path is Dist. The graph is defined by e/3. e.g. path(penzance, london, Path, Dist)
path(Start, Target, Path, Dist) :-
  dijkstra2(Start, Target, s(Target,Dist,Path)), !.

% Helping predicates
dijkstra2(Start, Target, ResultingS):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_2(Ds, ERecharge, [s(Start,0,[])], Target, ResultingS).

dijkstra_2([], _, _, _, _) :- !, fail.
dijkstra_2([D|Ds], ERecharge, _, Target, s(Target,Distance2,Path1)):-
  best(Ds,D,s(Target,Distance,Path)),
  delete2([D|Ds], [s(Target,Distance,Path)], _),
  reverse([Target|Path], Path1),
  Distance2 is Distance + ERecharge, !. % The first solution is the shortest, so '!'

dijkstra_2([D|Ds], ERecharge, Ss0, Target, ResultingS):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_2(Ds5, ERecharge, Ss1, Target, ResultingS).


%% Dijkstra for closest unknown vertex
pathClosestUnknownVertex(Start, UnknownVertex, Path, Dist) :-
  dijkstra7(Start, s(UnknownVertex, Dist, Path)), !.

% Helping predicates
dijkstra7(Start, ResultingS):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_7(Ds, ERecharge, [s(Start,0,[])], ResultingS).

dijkstra_7([], _, _, _) :- !, fail.
dijkstra_7([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
  best(Ds,D,s(Vertex,Distance,Path)),
```

```
  not(vertex(Vertex,_,_)),
  delete2([D|Ds], [s(Vertex,Distance,Path)], _),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge, !.

dijkstra_7([D|Ds], ERecharge, Ss0, ResultingS):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_7(Ds5, ERecharge, Ss1, ResultingS).


%% Dijkstra for closest non-probed vertex
pathClosestNonProbed(Start, NonProbedVertex, Path, Dist) :-
  dijkstra3(Start, s(NonProbedVertex, Dist, Path)), !.

% Helping predicates
dijkstra3(Start, ResultingS):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_3(Ds, ERecharge, [s(Start,0,[])], ResultingS).

dijkstra_3([], _, _, _) :- !, fail.
dijkstra_3([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
  best(Ds,D,s(Vertex,Distance,Path)),
  needProbe(Vertex),
  delete2([D|Ds], [s(Vertex,Distance,Path)], _),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge, !.

dijkstra_3([D|Ds], ERecharge, Ss0, ResultingS):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_3(Ds5, ERecharge, Ss1, ResultingS).


%% Dijkstra for closest non-probed vertex, with some additional checks
pathClosestNonProbedWithExtraChecks(Start, NonProbedVertex, Path, Dist) :-
  dijkstra9(Start, s(NonProbedVertex, Dist, Path)), !.
```

```

% Helping predicates
dijkstra9(Start, ResultingS):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_9(Ds, ERecharge, [s(Start,0,[])], ResultingS).

dijkstra_9([], _, _, _) :- !, fail.
dijkstra_9([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
  best(Ds,D,s(Vertex,Distance,Path)),
  not(needExploring(unknown)), needExploring(List), member(Vertex,List)
      , needProbe(Vertex),
  delete2([D|Ds], [s(Vertex,Distance,Path)], _),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge.

dijkstra_9([D|Ds], ERecharge, Ss0, ResultingS):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_9(Ds5, ERecharge, Ss1, ResultingS).


%% Dijkstra for closest non-surveyed vertex
pathClosestNonSurveyed(Start, NonSurveyedVertex, Path, Dist) :-
  dijkstra4(Start, s(NonSurveyedVertex, Dist, Path)), !.

% Helping predicates
dijkstra4(Start, ResultingS):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_4(Ds, ERecharge, [s(Start,0,[])], ResultingS).

dijkstra_4([], _, _, _) :- !, fail.
dijkstra_4([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
  best(Ds,D,s(Vertex,Distance,Path)),
  needSurvey(Vertex),
  delete2([D|Ds], [s(Vertex,Distance,Path)], _),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge, !.

dijkstra_4([D|Ds], ERecharge, Ss0, ResultingS):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
```

```
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_4(Ds5, ERecharge, Ss1, ResultingS).


%% Dijkstra for closest Repairer
pathClosestRepairer(Start, LocationRepairer, NameAgent, Path, Dist) :-
  dijkstra5(Start, s(LocationRepairer, Dist, Path), NameAgent), !.

% Helping predicates
dijkstra5(Start, ResultingS, NameAgent):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_5(Ds, ERecharge, [s(Start,0,[])], ResultingS, NameAgent).

dijkstra_5([], _, _, _, _) :- !, fail.
dijkstra_5([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1), NameAgent)
    :-
  best(Ds,D,s(Vertex,Distance,Path)),
  teamStatus(NameAgent,Vertex,_), role(NameAgent, 'Repairer'),
  delete2([D|Ds], [s(Vertex,Distance,Path)], _),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge, !.

dijkstra_5([D|Ds], ERecharge, Ss0, ResultingS, NameAgent):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_5(Ds5, ERecharge, Ss1, ResultingS, NameAgent).


%% Dijkstra for closest Visible Enemy
pathClosestVisibleEnemy(Start, LocationEnemy, NameEnemy, Path, Dist) :-
  dijkstra8(Start, s(LocationEnemy, Dist, Path), NameEnemy), !.

% Helping predicates
dijkstra8(Start, ResultingS, NameAgent):-
  create(Start, [Start], Ds),
  recharge(ERecharge),
  dijkstra_8(Ds, ERecharge, [s(Start,0,[])], ResultingS, NameAgent).

dijkstra_8([], _, _, _, _) :- !, fail.
dijkstra_8([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1), NameAgent)
    :-
  best(Ds,D,s(Vertex,Distance,Path)),
  enabledEnemy(NameAgent, Vertex),
```

```
  visibleEntity(NameAgent,_,_,_), %Enemy must be visible in current
      step.
  delete2([D|Ds], [s(Vertex,Distance,Path)], _),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge, !.

dijkstra_8([D|Ds], ERecharge, Ss0, ResultingS, NameAgent):-
  best(Ds, D, S),
  delete2([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  Distance2 is Distance + ERecharge,
  merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete2(Ds2, Ss1, Ds3),
  incr(Ds3, Distance2, Ds4),
  merge2(Ds1, Ds4, Ds5),
  dijkstra_8(Ds5, ERecharge, Ss1, ResultingS, NameAgent).


%% General Dijkstra helping predicates

% create(Start, Path, Edges) is true if Edges is a list of structures s
    (Vertex, Distance, Path) containing, for each Vertex accessible
    from Start, the Distance from the Vertex and the specified Path.
    The list is sorted by the name of the Vertex.
create(Start, Path, Edges):- maxEnergy(E), setof(s(Vertex,Edge,Path), (
    e(Start,Vertex,Edge), Edge =< E ), Edges), !.
create(_, _, []).

% best(Edges, Edge0, Edge) is true if Edge is the element of Edges, a
    list of structures s(Vertex, Distance, Path), having the smallest
    Distance.  Edge0 constitutes an upper bound.
best([], s(A,B,C), s(A,B,C)).
best([s(A,B,C)|Edges], Best0, Best):- shorter(s(A,B,C), Best0), !, best
    (Edges, s(A,B,C), Best).
best([_|Edges], Best0, Best):- best(Edges, Best0, Best).

shorter(s(_,X,_), s(_,Y,_)):-X < Y.

% delete2(Xs, Ys, Zs) is true if Xs, Ys and Zs are lists of structures
    s(Vertex, Distance, Path) ordered by Vertex, and Zs is the result
    of deleting from Xs those elements having the same Vertex as
    elements in Ys.
delete2([], _, []).
delete2([X|Xs], [], [X|Xs]):-!.
delete2([X|Xs], [Y|Ys], Ds):- eq(X, Y), !, delete2(Xs, Ys, Ds).
delete2([X|Xs], [Y|Ys], [X|Ds]):- lt(X, Y), !, delete2(Xs, [Y|Ys], Ds).
delete2([X|Xs], [_|Ys], Ds):- delete2([X|Xs], Ys, Ds).

% merge2(Xs, Ys, Zs) is true if Zs is the result of merging Xs and Ys,
    where Xs, Ys and Zs are lists of structures s(Vertex, Distance,
    Path), and are ordered by Vertex.  If an element in Xs has the same
     Vertex as an element in Ys, the element with the shorter Distance
    will be in Zs.
```

```
merge2([], Ys, Ys).
merge2([X|Xs], [], [X|Xs]):-!.
merge2([X|Xs], [Y|Ys], [X|Zs]):- eq(X, Y), shorter(X, Y), !, merge2(Xs, Ys, Zs).
merge2([X|Xs], [Y|Ys], [Y|Zs]):- eq(X, Y), !, merge2(Xs, Ys, Zs).
merge2([X|Xs], [Y|Ys], [X|Zs]):- lt(X, Y), !, merge2(Xs, [Y|Ys], Zs).
merge2([X|Xs], [Y|Ys], [Y|Zs]):- merge2([X|Xs], Ys, Zs).

eq(s(X,_,_), s(X,_,_)).

lt(s(X,_,_), s(Y,_,_)):-X @< Y.

% incr(Xs, Incr, Ys) is true if Xs and Ys are lists of structures s(Vertex, Distance, Path), the only difference being that the value of Distance in Ys is Incr more than that in Xs.
incr([], _, []).
incr([s(V,D1,P)|Xs], Incr, [s(V,D2,P)|Ys]):- D2 is D1 + Incr, incr(Xs, Incr, Ys).

% Predicate that finds all surveyed edges, and checks both ways to make sure not an edge is missed
e(X, Y, Z):- vertex(X, _, List), member([Z,Y], List), Z \= unknown.
e(X, Y, Z):- vertex(Y, _, List), member([Z,X], List), Z \= unknown.
e(X, Y, 5):- vertex(X, _, List), member([unknown,Y], List), vertex(Y, _, List2), member([unknown, X], List2).
e(X, Y, 5):- vertex(Y, _, List), member([unknown,X], List), vertex(X, _, List2), member([unknown, Y], List2).
```

F.14 generalKnowledge.pl

```
% Energy/money checks
energyGE(Nr) :- Nr = unknown, energy(E), E >= 9.
energyGE(Nr) :- Nr \= unknown, energy(E), E >= Nr.
moneyGE(Nr) :- money(M), M >= Nr.
maxEnergy(E) :- disabled, maxEnergyDisabled(E), !.
maxEnergy(E) :- not(disabled), maxEnergyWorking(E).
recharge(Nr) :- not(disabled), maxEnergy(E), Nr is round(0.5*E).
recharge(Nr) :- disabled, maxEnergy(E), Nr is round(0.3*E).

% Role of the agent
role(Role) :- me(Id), role(Id, Role).

% 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 its neighbor needs surveying
needSurvey(Vertex) :- vertex(Vertex,_,NBs), (NBs = []; member([unknown,_], NBs)), !.
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 and it is time to swarm
optimum :- optimum(_), !, timeToSwarm.

% Random predicates. random/3 with float inputs should work, but it doesn't!
randomFloat(R) :- R is (random(65391)/65391). % There seems to be a bug when using the built-in random/3 predicate
randomElement(List, Elem) :- length(List,N), N > 0, random(0,N,R), nth0(R,List,Elem).

% Defines whether an enemy is to be considered dangerous for sure
dangerousEnemy(Id) :- inspectedEnemy(Id, 'Saboteur'), !.
dangerousEnemy(Id) :- not(inspectedEnemy(Id, _)), !, findall(Id2, inspectedEnemy(Id2, 'Saboteur'), List), length(List,N), N < 4.
% 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,_)), !, findall(Id2, inspectedEnemy(Id2, 'Saboteur'), List), length(List, N), N == 4.

% Short predicate to extract the most useful information from an inspected enemy
```

```
inspectedEnemy(Id,Role) :- inspectedEntity(Id, _, Role, _, _, _, _, _,
    _, _).

% Vertex value checks (checks for unknown before evaluating arithmetic
    operation)
vertexValueGT(A, B) :- A \= unknown, B \= unknown, A > B.
vertexValueGE(A, B) :- A \= unknown, B \= unknown, A >= B.

% Sum of the values of all vertices in a list
vertexListSum([], 0).
vertexListSum([H|T], Sum) :- vertexValue(H,V), V == unknown,
    vertexListSum(T,S), Sum is S+1.
vertexListSum([H|T], Sum) :- vertexValue(H,V), V \== unknown,
    vertexListSum(T,S), Sum is S+V.

% Get your swarm position
getOptimum(X) :- swarmPosition(X), X \= unknown.

% (Optimums are now calculated from the belief base.)
% Optimums are vertices that are maximas such that no other vertex with
     a higher value exists, which is true of the vertices with value
    10.
% No local maxima n with value < 10 exists (with very high probability)
     because of the map generation algorithm.
optimum(X) :- vertex(X,10,_).
```

F.15 navigationKnowledge.pl

```prolog
% Finds a list of all neighboring nodes of a given node
neighbours(V,Ns) :- findall(N, neighbour(V,N), Ns), !.

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

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

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

% 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'))).

% Determines if the agent is the only agent on its position
foreverAlone :- not( (currentPos(Pos), me(Me), team(Team), !,
    visibleEntity(ID, Pos, Team ,_), Me \= ID ) ).

% Compares agents names to find which name has a higher 'value'
compareAgents(Agent1,Agent2,Agent2) :- Agent1 @< Agent2.
compareAgents(Agent1,Agent2,Agent1) :- Agent1 @> Agent2.

% 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), !,
  findall(Agent, visibleEntity(Agent,Here,Team,normal), Agents),
      agentRank(Agents,Name,Rank).

% An agents rank (i.e. index) in the list List
agentRank(List,Agent,Rank) :- nth0(Rank, List, Agent), !.

% 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 neighbor
selectNeighbour(List, Number, Neighbour) :- length(List, Size), Num is
    mod(Number,Size), nth1(Num, List, Neighbour), !.

% 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
selectDestination(List, Number, Destination) :- length(List,Size), Num
    is mod(Number,Size), nth0(Num, List, Destination), !.

% Short predicates for vertex information
```

```
vertexValue(Id,Value) :- vertex(Id,Value,_).
vertexValue(Id,unknown) :- not(vertex(Id,_,_)).

% Workaround for the action specifc warning from the action "goto":
% "WARNING: getPrecondition for UserSpecAction does not support
    multiple specifications"
% Saboteurs need to have >=11 energy because it would be unwise to not
    be able to attack after moving. Otherwise they would die if they
    walk to a vertex with an enemy Saboteur.
canGoto(Here, There) :- role('Saboteur'), neighbour(Here,Weight,There),
     enemyTeam(T), visibleEntity(ID,There,T,normal), dangerousEnemy(ID)
    , (Weight == unknown -> W is 11 ; W is Weight+2), energyGE(W), !.
canGoto(Here, There) :- neighbour(Here,Weight,There), energyGE(Weight),
     !.
canGoto(Here, There) :- not(neighbour(Here,There)), visibleEdge(Here,
    There).

% The agent's rank amongst its peers on the team with the same role
agentRoleRank(Agent, Rank) :- role(Agent, Role), findall(A, role(A,
    Role), L), sort(L, S), agentRank(S, Agent, Rank).
agentEnabledRoleRankHere(Agent, Rank) :- currentPos(Pos), team(T), role
    (Agent, Role), findall(A, (role(A, Role), visibleEntity(A, Pos, T,
    normal)), L), sort(L, S), agentRank(S, Agent, Rank).
hasHighestRoleRank(Agent) :- agentRoleRank(Agent, Rank), Rank is 0.
hasLowRoleRank(Agent) :- agentRoleRank(Agent, Rank), Rank > 1.
hasLowestRoleRank(Agent) :- agentRoleRank(Agent, Rank), Rank is 3.

% Used to find all visible edges around a vertex
visibleEdgesList(Id1, Array) :- findall([unknown, Id2], (percept(
    visibleEdge(Id1,Id2)); percept(visibleEdge(Id2,Id1))), Array).
```

F.16 perceptKnowledge.pl

```
% 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)).

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

% Round information from the percepts
lastAction(Action) :- percept(lastAction(Action)).
lastActionParam(Param) :- percept(lastActionParam(Param)).
lastActionResult(failed_parry) :- percept(lastActionResult(failed_parry
    )), !.
lastActionResult(failed) :- percept(lastActionResult(Result)),
    atom_chars(Result,Chrs), append([f,a,i,l,e,d],_,Chrs), !.
lastActionResult(Result) :- percept(lastActionResult(Result)).
```

F.17 roleKnowledge.pl

```
%% Explorer specific knowledge

needProbe(Vertex) :- vertex(Vertex,unknown,_).
needProbe(Vertex) :- not(vertex(Vertex,_,_)).

% True when we should decide swarm positions
timeToDecideSwarm :- decidedSwarmAt(OldS), step(NewS), D is NewS - OldS, D >= 60, !.
timeToDecideSwarm :- swarmPosition(unknown), !, optimum(X), calcZoneValue(X,V), V >= 70.

% True when it is time to swarm. It differs from Explorers and the others
timeToSwarm :- not(role('Explorer')), swarmPosition(Opt), Opt \= unknown, !.
timeToSwarm :- role('Explorer'), optimum(_), step(Cur), Cur > 150.

% Finds the optimum nodes that can contain a swarm with the largest
% potential values as defined by calcZoneValue
bestOptimums(List,Opts) :- findall((ValSum,Swarm), (member(Swarm, List), calcZoneValue(Swarm,ValSum)), L), sort(L,S),
        length(S,N), nth1(N,S,(MaxVal,_)), Limit is round(0.65*MaxVal), bestOptimumsAux(S,Limit,L2), sort(L2,Opts).
bestOptimumsAux([],_,[]).
bestOptimumsAux([(Val,Opt)|T],Limit,[Opt|Rest]) :- Val >= Limit, bestOptimumsAux(T,Limit,Rest).
bestOptimumsAux([(Val,_)|T],Limit,Rest) :- Val < Limit, bestOptimumsAux(T,Limit,Rest).

% Calculates the sum of the values for all the neighbors, and their neighbors, and the vertex O, around the vertex O
calcZone(O,S) :- findall(N, (neighbour(O,_,N)), L1), findall(N, (member(M,L1), neighbour(M,_,N)), L2), union(L1,L2,L3), sort(L3,S).
calcZoneValue(O,V) :- calcZone(O,L), vertexListSum(L,V).

% Find all optimums that are not already in use
allOptimums(Opts) :- allOptimums(Opts,[]).
allOptimums(Opts,Ignore) :- findall(V, (optimum(V), not(member(V,Ignore)), not((neighbour(V,N),member(N,Ignore)))), Opts), length(Opts,N), N > 0, !.

% Choose the best optimums
decideOptimums(Opts) :- allOptimums(L), !, bestOptimums(L, Opts), !.


%% Saboteur specific knowledge

% Used by the harassment strategy
% A possible harassment vertex is a high-value vertex that is owned by the enemy and therefore probably contains a swarm
timeToHarass :- me(Me), hasLowestRoleRank(Me), step(N), N > 60.
```

```
possibleHarassVertex(Pos) :- findall(V, (enemyStatus(EID,V,normal), not(inspectedEnemy(EID,'Saboteur')), not(inspectedEnemy(EID,'Repairer')), notLargeBattle(V)), L), L \= [], randomElement(L,Pos), !.

largestZone([],0,_).
largestZone([H|T],Val,V) :- calcZoneValue(H,X), largestZone(T,XT,VT), (X > XT -> (V = H, Val = X); (V = VT, Val = XT)).

% If we have defeated the enemy near the harass vertex then the harass is over.
% Alternatively if we have harassed for a long time (> 50 steps) then we should do something else.
% It is important note that the harass/1 predicate cannot be true when adopting a harass goal.
% Otherwise the agent won't adopt the goal! As a workaround, if N >= 75 then the predicate fails.
% So in the time between 50 =< N < 75 the agent cannot adopt a new harass goal.
harass(V) :- (currentPos(V) ; (currentPos(P), neighbour(V,P))), not(enemyStatus(_,V,normal)), not((neighbour(V,N), enemyStatus(_,N,normal))), harassStart(S), step(Cur), N is Cur - S, N < 75.
harass(V) :- harassStart(S), S \= 0, step(Cur), N is Cur - S, N > 50, N < 75, vertex(V,_,_).

% When the enemy is disabled, the hunt is over
timeToHunt :- me(Me), hasLowRoleRank(Me), not(hasLowestRoleRank(Me)), step(N), N > 100.
hunt(ID) :- enemyStatus(ID,_,disabled).

% To determine which enemies are on the current position
enemyHere(ID) :- currentPos(Vertex), visibleEntity(ID,Vertex,Team,_), enemyTeam(Team).

% To deterine which enemies are close to the current position
enemyNear(ID,Pos) :- currentPos(Pos), enemyHere(ID).
enemyNear(Id,Pos) :- neighbour(Pos), visibleEntity(Id,Pos,Team,_), enemyTeam(Team).

% To determine when a non-disabled enemy is at your position
enabledEnemyHere(Id) :- currentPos(Vertex), visibleEntity(Id,Vertex,Team,normal), enemyTeam(Team).

% when an non-disabled enemy is at or next to your position
enabledEnemyNear(ID,Pos) :- currentPos(Pos), enabledEnemyHere(ID).
enabledEnemyNear(Id,Pos) :- neighbour(Pos), visibleEntity(Id,Pos,Team,normal), enemyTeam(Team).

% A list of all locations near where there are enemies
enabledEnemiesNear(List) :- findall(Vertex,enabledEnemyNear(ID,Vertex),L), sort(L,List), List = [_|_].

% Short predicate for finding enemies worth attacking
enabledEnemy(ID,Vertex) :- enemyStatus(ID,Vertex,normal).

```

```
% Used for buying upgrades. Only buy if the second highest enemy Saboteur has a strength or health advantage and if enough time has elapsed.
timeToBuy :- step(Step), Step >= 140.
enemySaboteurSecondMaxStrength(Strength) :- findall(Str, inspectedEntity(_, _,'Saboteur', _, _, _, _, _, Str, _), L), msort(L, S), length(S, N), N > 1, sort([N,3],A), nth1(1,A,Index), nth1(Index, S, Strength), !.
enemySaboteurSecondMaxHealth(Health) :-        findall(Hp, inspectedEntity(_, _,'Saboteur', _, _, _, _, Hp, _, _), L),  msort(L, S), length(S, N), N > 1, sort([N,3],A), nth1(1,A,Index), nth1(Index, S, Health), !.

% If there are N-1 ally Saboteurs and N enemy Saboteurs at a vertex then we should not go there because we are not needed (i.e. if not(notLargeBattle))
largeBattleCalculator(V,AN,EN,AL) :- findall(EID, (enemyStatus(EID,V,_), dangerousEnemy(EID)), EL), !, findall(AID, (teamStatus(AID,V,_), role(AID,'Saboteur')), AL), !, length(EL,EN), length(AL,AN).
largeBattle(V,AL) :- largeBattleCalculator(V,AN,EN,AL), AN >= EN, AN \= 0, EN \= 0, !.
notLargeBattle(V) :- largeBattleCalculator(V,_,0,_), !.
notLargeBattle(V) :- largeBattleCalculator(V,AN,EN,_), ANPlusUs is AN + 1, ANPlusUs < EN, !. % AN+1 to prevent us from creating a large battle


%% Repairer specific knowledge

% Predicate that returns disabled allies near or on the current position
disabledAllyNear(ID,Here) :- currentPos(Here), team(Team), me(Me), visibleEntity(ID,Here,Team,disabled), ID \= Me, !.
disabledAllyNear(ID,Vertex) :- team(Team), neighbour(Vertex), visibleEntity(ID,Vertex,Team,disabled), !.
disabledAllyNear(ID,Vertex) :- currentPos(Here), team(Team), visibleEdge(Here,Vertex), visibleEntity(ID,Vertex,Team,disabled), !.

% The repairing(ID) goal.
% When the injured agent is no longer disabled, then the goal has been achieved.
% A repair goal should never be adopted unless there exists a path between the repairer and the injured agent.
repairing(Agent) :- agent(Agent), not(teamStatus(Agent,_,0)).


%% Inspector specific knowledge

% Predicate that returns uninspected agents close to the inspector
% This also makes sure enemy saboteurs are suitable for inspection again when last inspection is older than 50 steps
uninspectedNear :- visibleEntity(Agent,Vertex,Team,_), enemyTeam(Team), (currentPos(Vertex) ; neighbour(Vertex)),
```

```
  (not(inspectedEntity(Agent,_,_,_,_,_,_,_,_,_)); ( inspectedEnemy(
      Agent, 'Saboteur'), lastInspect(Agent,LI), step(S), LI2 is LI +
      50, LI2 < S)).
uninspectedEntity(Agent) :- not(inspectedEntity(Agent,_,_,_,_,_,_,_,_,_
    )).
```

Bibliography

[BKS+] Tristan Behrens, Michael Köster, Federico Schlesinger, Jürgen Dix, and Jomi Hübner. scenario.pdf. http://www.multiagentcontest.org/downloads/func-startdown/730/ 06-06-2013.

[Hin] Koen Hindriks. Programming rational agents in goal. https://mmi.tudelft.nl/trac/goal/raw-attachment/wiki/WikiStart/GOAL.pdf May 2011.

[Lø] Hans Henrik Løvengreen. Bachelorprojekt softwareteknologi. http://www2.imm.dtu.dk/courses/02125/0118.html 05-04-2013.

[MAP] Multi-agent programming contest. http://www.multiagentcontest.org/ 12-04-2013.

[MPI] MPI Forum. GOAL Webpage. https://ii.tudelft.nl/trac/goal/wiki/ (05-04-2013).

[RN09] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, December 2009.

[Woo11] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2011.