Functional Programming and Multi-Agent Systems

Thor Helms

Kongens Lyngby 2011
IMM-BSC-2011-13

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-BSC: ISSN 0909-3192

Summary

This project revolves around a multi-agent contest [1] in which several competing teams, each consisting of several agents of different types, try to occupy territory on Mars, which is represented by a graph.

A simulation program for the above scenario has been implemented in a functional programming language (F# via Mono), exploring the advantages and disadvantages of using a functional programming language when making a GUI.

An artificial intelligence (AI) has been created for use in the simulator. This AI is documented and analyzed in this report.

Resumé

Dette projekt omhandler en konkurrence i multi-agent systemer [1] i hvilken adskillige hold, hver bestående af adskillige agenter af forskellige typer, forsøger at erobre territorie på Mars, som er repræsenteret af en graf.

En simulator for det ovenstående scenario er blevet implementeret i et funktionelt programmerings sprog (F# via Mono), og fordele og ulemper ved at bruge et funktionelt sprog til at lave en grafisk brugergrænseflade er blevet undersøgt.

En kunstig intelligens er blevet implementeret til simulatoren. Denne kunstige intelligens er dokumenteret og analyseret i denne rapport.

Preface

This is the bachelor project for Thor Helms, student at the Technical University of Denmark (DTU), with Jørgen Villadsen as counselor. The project period is february to june 2011.

Knowledge about functional programming has been retrieved from the course 02157 Functional Programming at DTU, autumn 2010.

Some knowledge about multi-agent systems has been gathered during a previous attempt at a bachelor project on multi-agent systems in autumn 2010.

Kgs. Lyngby, June 2011

Thor Helms

Contents

Chapter 1

Introduction

This report describes the development of a simulator for a multi-agent system (MAS), and an artificial intelligence for use in the simulator. The scenario is from an international competition in multi-agent systems [1]. The scenario is described in section 1.1, and in more detail in chapter 2.

Both the simulator and the artificial intelligence is developed in the language F# [7], which is created by Microsoft for their .NET framework, and supported by the cross-platform Mono [4]. Section 1.2 describes the F# language and the reasons for using it in this project.

1.1 About the scenario

The scenario simulates a number of robots, operating on Mars. The purpose of these robots are to find and occupy the best water wells on the planet. There are multiple teams, consisting of multiple types of robots (agents) each. All teams consist of the same number and types of robots. Some robots might be able to gather information about the world, and some robots might be able to sabotage opponent robots. There are a number of predefined roles, which are described in section 2.2 on page 6.

The world is represented by a graph, where the nodes in the graph represent water wells, and the edges represent roads between water wells. Even though it is on Mars and the robots should be able to drive directly from any one point to any other point on the surface, the graph isn't complete, but it is guaranteed to be connected.

1.2 About functional programming and F#

F# is Microsoft's take on functional programming, and is in fact multi-paradigm, encompassing functional, objective-oriented and imperative programming [7]. It targets Microsofts .NET framework, and also runs on the open source, multi-platform Mono framework. When compiled, it will produce the exact same code as C#, which enables the two languages to be used together once compiled. This also means that F# is able to use all .NET and Mono frameworks, which are mostly object-oriented code.

Functional programming is often based on lambda calculus [10]. This is true for the functional parts of F# also. Functional functions have no side-effects, but in F# it is possible to define functions with side-effects, which is necessary for the object-oriented parts of the language.

Functions are first-class citizens in F#, and it is possible to define higher order functions, which means that a function takes another function as argument and/or returns a function.

In functional languages, types are usually inferred by the compiler, which means that the programmer doesn't have to explicitly define types. In F#, this is true, but the programmer is able to define the types if needed. Types can also be generic, in which case the type will be inferred at compile-time.

Values are usually immutable in functional languages, but in F# it is possible to define a value to be mutable. There are some limitations when using a mutable value, but some can be overcome by using the *ref* type, which is a wrapper for a mutable value [8].

Values in F#, and most other functional languages, can be defined as tuples and as discriminated unions. An example [7] of a discriminated union can be seen below:

```
type A =
```

```
    | ConstructorX of string
    | ConstructorY of int
```

Discriminated unions allow the programmer to easily define compilers, or tree's of types, which in an object oriented style would have to implemented as a hierarchy of classes, allowing functional code to be more succinct.

F# and most other functional languages, also support lists and records as first class citizens, which means that collections of values can be easily ordered and used in calculations.

Chapter 2

Scenario details

In this chapter, the details of the scenario are explained, and any decisions to change the official scenario are argued for.

2.1 The world as a graph

The world in which the agents operate is represented as a graph, with a number of nodes and edges. Internally, the nodes are represented with a coordinate, making it possible to draw the graph. A restriction is placed on the edges, in that no two edges may cross when using the node-coordinates. This will allow the graph to be viewed as a 2D map, where the edges represent roads, and there are no intersections except at the nodes. The nodes represent points of interest, in this case water wells.

Each node in the graph will have a weight value, determining the quality of the water well. Higher is better.

Each edge in the graph will likewise have a weight value, determining the quality of the road and how much energy is required to traverse it.

2.2 Teams, agents and roles

In a simulation, several teams will operate. Each team consists of a number of agents. Agents can have different roles, and the different roles have different stats such as strength, visibility, energy and health. All teams consist of the same number of agents, with the same roles on each team for fairness.

Energy is used to perform most actions. The agent can recharge its energy, and the recharge rate will be expressed in percentage of the agent's maximum energy level.

Health determines how robust the agent is, and thus how difficult it is to destroy. Health can only be restored when an agent is repaired by another agent from the same team. If an agent has no health left, it is referred to as being disabled. Otherwise, it is referred to as being active.

Visibility determines how far the agent can see. All nodes that a graph search would find when going to at most the same depth as the visibility range, will be visible for the agent. All edges connecting visible nodes, and all agents on visible nodes, will be visible as well.

Strength determines how hard an agent can attack, and thus destroy enemy agents. For instance, an agent with 4 strength would be able to reduce enemy agents' health by 4 per attack.

The different roles are able to perform different actions. Actions are defined in section 2.4.

Table 2.2 lists the predefined agent roles.

2.3 Disabled agents

When an agent has no health left (because its been sabotaged by enemy agents), it is disabled and various limitations is placed on it. Its health can be restored fully when repaired by another agent, and the agent will thus cease being disabled.

Role	Energy	Health	Visibility range	Strength	Actions
Explorer	12	4	2	0	Skip, goto, probe, survey, buy, recharge
Repairer	8	6	1	0	Skip, goto, parry, survey, buy, repair, recharge
Saboteur	7	3	1	4	Skip, goto, parry, survey, buy, attack, recharge
Sentinel	10	1	3	0	Skip, goto, parry, survey, buy, recharge
Inspector	8	6	1	0	Skip, goto, inspect, survey, buy, recharge

Table 2.1: Predefined agent roles

As mentioned earlier, when an agent is not in the disabled state, it is in the active state.

2.4 Agent actions

As mentioned earlier, there are different actions the agents can perform. When performing an action, there is a certain percentage chance that the action will fail.

The actions are defined below.

2.4.1 Skip

Does nothing. This action will always succeed. Actions that have failed, for any reason, will be considered to be of this type.

2.4.2 Recharge

Attempt to recharge the agent's energy. This action will fail if the agent has been successfully attacked by another agent. The amount of energy restored depends on whether the agent is disabled or not, and on the maximum amount of energy the agent is able to store.

2.4.3 Attack

Attempt to sabotage an enemy agent, identified by team and agent ID. A target agent on the same node as the attacking agent is required. An attack can be parried. Default energy cost is 2. A disabled agent can't attack.

2.4.4 Parry

Parry any expected attacks. Default energy cost is 2, and will be charged regardless of number of incoming attacks. A disabled agent can't, and wouldn't gain anything from performing the parry action, as its health can't be lower than 0.

2.4.5 Goto

Go to a neighbor node, determined in a parameter with the ID of the neighbor node. The weight of the edge connecting the current node and the neighbor node determines the energy cost. If the agent doesn't have enough energy to traverse the edge, its energy will be reduced (if any energy is left). The default energy cost for a failed goto-action is 1.

2.4.6 Probe

Request the weight of the node the agent is at. Once a node has been probed, its weight is visible for the entire team at all times, otherwise its weight is regarded as unknown. Default energy cost of the probe action is 1. A probe action will fail if the probing agent is disabled or being attacked by another agent.

2.4.7 Survey

Request the weight for all edges connected to the node the agent is at. Once an edge has been surveyed, its weight is visible for the entire team at all times, otherwise its weight is regarded as unknown. Default energy cost of the survey action is 1. A survey action will fail if the surveying agent is disabled or being attacked by another agent.

The official scenario description defines the survey action as requesting the weight for "some" visible edges. As this is an unspecified amount, its been chosen to see the edges directly connected to the node the agent is at.

2.4.8 Inspect

Request the various stats for all enemy agents on the same node and on neighboring nodes, compared to the agent's position. Once an enemy agent has been inspected, its stats and possible actions are visible for the entire team at all times, otherwise all stats and actions for the agent is regarded as unknown. Default energy cost of the inspect action is 2. An inspect action will fail if the inspecting agent is disabled or it has been successfully attacked by another agent.

2.4.9 Buy

Attempt to perform an upgrade. There are four types of upgrades, each of which upgrades either the agent's maximum energy, maximum health, visibility range or strength. Only agents who can perform the attack action are able to upgrade its strength (As it is useless for all other agents). When performing an upgrade, the chosen stat is increased by 1. In case of upgrading maximum energy or health, the current energy and health is also increased by 1. Default energy cost of the buy action is 2. It is not possible to perform the buy action if the agent is disabled or is being attacked.

2.4.10 Repair

Repairs another agent on the same team. The agent that needs to be repaired must be identified with its agent ID, and both agents must be on the same

node. This action fully restores the target agent's health, and brings it out of the disabled state if it was previously in it. It is not possible for an agent to repair it self. Default energy cost for the repair action is 2. The action will fail if the repairing agent is being attacked.

2.5 Occupying a node

A node is occupied by the team who has the most active agents on the node. In the case of a tie, the node is not occupied. A node occupied in this manner will be referred to as being directly occupied, or directly dominated.

Nodes can also be occupied if there are no active agents on them, and one of two conditions are met. The first condition is that the node has at least two neighbor nodes that are directly occupied by the same team, in which case it gets dominated by the same team. In case of a draw in the amount of occupied neighbor nodes by team, the node will not be dominated. If a node is occupied in this manner, it will be referred to as being indirectly occupied, or indirectly dominated. The second condition is that the node lies within an otherwise unoccupied zone, defined below. All nodes directly next to the nodes in the unoccupied zone, but not in it, will be referred to as a frontier. All nodes in the frontier must be occupied by the same team, for the zone to be occupied, in which case the team dominating the frontier will also dominate all nodes in the zone. No enemy agents must be able to move into the zone without crossing the frontier (Enemy agents standing on a node on the frontier doesn't count). This means that no nodes in the zone are allowed to have any active agents.

If an enemy enters a zone dominated by one team, or breaks the frontier, the zone will no longer be dominated.

2.6 Zones

A zone is defined as a connected subgraph within the graph, all dominated by the same team, or all dominated by no team, where the subgraph can't be extended with more nodes while still maintaining the requirements.

2.7 Graph coloring

Graph coloring is an algorithm that determines which team, if any, occupies the various nodes. The color of a node represents the team dominating it. An example of a colored graph may be seen in figure 2.1.

Coloring of the graph, i.e. determining the domination of the various nodes, takes place in five steps. First the existing coloring is reset so no nodes or edges are dominated by any team. Second the directly occupied nodes are colored. Third the indirectly dominated nodes are colored. Fourth all unoccupied zones are determined, and colored if they comply with the restrictions given above. Fifth all edges connecting two nodes of the same color, are colored with the same color. Edge colors have no meaning, and they are only colored for making it visually easier to determine zones in the simulator.

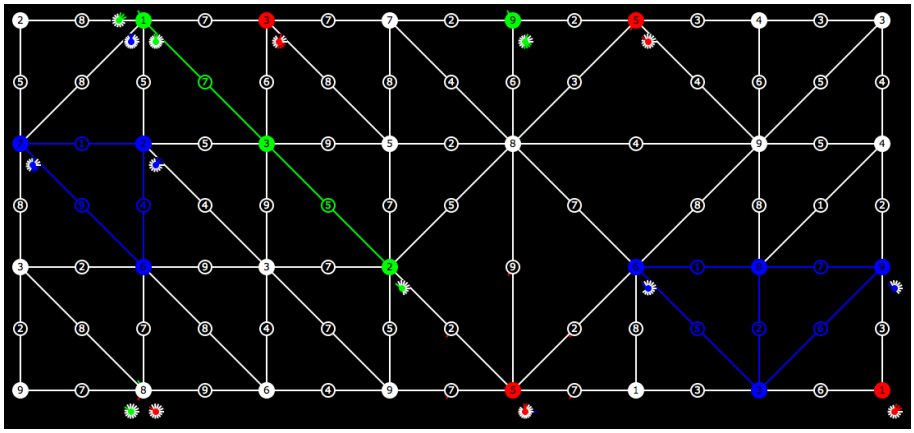

Figure 2.1: An example of a colored graph

2.8 Zone scores

After the graph has been colored, the zone scores can be calculated using the colors. A zone score is the sum of node weight's in an occupied zone. The scores are not actually calculated per zone, but per team. The zone scores are used when calculating the step score for each team.

2.9 Milestones and money

In a simulation, certain milestones may be defined. For instance, a milestone could be that a team has a total zone score of at least 50 in one simulation step. When a team reaches a milestone, the team will receive a reward in the form of money, which can be used to buy upgrades for the agents. The reward size may vary, depending on whether the team reaching a milestone is the first team reaching that particular milestone or not. Milestones should be defined before the simulation starts.

Several of the same type of milestone may be defined, for instance one for 5 successful attacks and one for 20 successful attacks, yielding (possibly) different amounts of money.

There are six types of milestones. These are:

- Zone values in a single step.
- Probed nodes in total.
- Surveyed edges in total.
- Inspected enemy vehicles in total.
- Successful attacks.
- Successful parries.

2.10 Agent perceptions

In the beginning of each simulation step, each agent is given a perception, i.e. their view of the world. The contents of the perceptions are:

- The current step number.
- The team score.
- The amount of money the team possesses.
- The stats for the agent it self, including the actions its role permits it to do.

- A list of visible nodes.
- A list of visible edges.
- A list of enemy agents that have been inspected by any agent on the same team.
- A list of messages, containing the ID of the sending agent and the content of the message.

The list of visible nodes and edges are shared with any friendly agent in the same zone as the agent it self. This encourages team work by creating and keeping zones. Nodes, edges and agents are anonymized if necessary (see below) before the perception is sent to the agent.

In the official scenario description, agents would also receive a list of all nodes and edges that has been surveyed or probed by its team. This has been disregarded, and it will be up to the agents to remember this information, and share it with the team.

In the official scenario description, agents would also be told the result of its last attempted action. This has been disregarded as well, and it is up to the agent it self to determine the success of the last action.

2.11 Anonymous objects

In perceptions, some nodes, edges and enemy agents may be anonymous to the agent receiving the perception. When nodes haven't been probed, edges haven't been surveyed or agents haven't been inspecting by any agent on the same team as the receiving agent, they are anonymous. That is, unknown objects visible to the agent are anonymous. Anonymous nodes and edges will have their weight set to 1 when they are anonymous. All stats of an anonymous enemy agent will be set to 1, and their actions will not be visible.

When zone scores are determined, the weight of an anonymous node for a team will be regarded as being 1.

2.12 Communication

Communication agent-to-agent can and should preferably be done through the simulator. Communication consists of messages, where each message contains a receiver and some content. As mentioned earlier, the agent perceptions contains a list of incoming messages. When agents respond to the simulator with their requested action, they should also send a list of outgoing messages. Because of this structure, there will be a delay in the reception of the messages of one simulation step.

The receiver of a message can be either a single agent, or a broadcast to all agents – in either case, only agents on the same team as the sender can receive the message, simulating safe communication.

In the first scenario draft [2], communication was to happen through the simulation server, and there was a limitation on the amount of messages an agent could send in each simulation step. In the latest version of the scenario description [3], both of these requirements has been removed. In this implementation of the simulator, the first requirement is kept and the second has been removed.

2.13 Winning condition

The winning team is the team with the highest score after the last simulation step. The score is calculated as the sum of zone scores and money in the previous steps:

$$Score = \sum_{s=1}^{steps} Zones_s + Money_s \tag{2.1}$$

Chapter 3

User interface

As the scenario is somewhat abstract, the simulator should somehow display the status of the simulation. For that, a graphical user interface (GUI) has been created. This section describes the design and implementation of the GUI.

3.1 Requirement specification

The following is a list of requirements for the simulator:

- Automatically run a simulation calculation.
- View already calculated simulation steps.
- Start and stop the playback of a calculated simulation.
- Add new artificial intelligences to the program without recompiling the simulator, and allow the program to automatically discover new artificial intelligences.
- The simulation must be displayed graphically.

- All agents on the same team should have the same color – each team should have different colors.
- Change various simulation variables:
  - Number of teams.
  - Number of agents on each team.
  - Number of simulation steps to be calculated.
  - Chance that agent actions will fail, in percentage.
  - Maximum response time for agents when requesting actions, in milliseconds.
  - Recover rate for agents in percentage, both when active and when disabled.
  - Number of nodes on the map.
  - Grid size of the map/graph.
  - Min/max node weight.
  - Min/max edge weight.
  - Energy cost of the various actions.
  - Price of the various upgrades.
  - Roles or types of the agents – either selected from the preset roles, or the user should be able to precisely define the stats and actions.
  - Colors for the different teams.
  - Select from a list of artificial intelligences, which one controls which role on each team.
  - Define milestones in the simulation. Select from one of the six types, along with what amount is needed to trigger the milestone, and the reward for the first team(s) and subsequent teams to reach the milestones.
- Generate a random map/graph using the simulation variables.
- Display the progress of the simulation:
  - The score and amount of money for the different teams.
  - The map/graph, colored by which teams dominate which areas.
  - Stats for the various agents:
    - Strength.
    - Current energy.
    - Max energy.

        - Current health.
        - Max health.
        - Visibility range.
        - Which teams have inspected the agent.
    - Calculation progress in number of steps.
    - Which teams has probed which nodes.
    - Which teams has surveyed which edges.
- Save/load simulation configurations.
- Enable/disable view of the agent's perceptions.
- Allow the agents to use the simulation window to display their world model and/or graphical debug information.
- Save/load an entire simulation after its been calculated.

3.2 Graphical user interface

Given that the simulation needs to be visually available, a GUI has been made. Included in the GUI is the ability to change all simulation variables, as a contrast to loading the simulation variables from a configuration file, which would have to be created beforehand. The GUI consists of two windows: The main window, which displays the progress of the running simulation; and the simulation settings window, which enables the user to change the simulation variables.

Throughout the simulator, some generic functions are used, which aren't simulation specific but rather just nice to have. The source code for these can be seen in appendix B.8 on page 118.

3.2.1 Main simulator window

The main simulator window, figure 3.1, consists of a menubar in the top, view-selection in the left side, simulation graphics view in the right side, and some simulation-view controls in the bottom of the window. Furthermore, it contains a status bar in the very bottom of the window, displaying the progress of the simulation calculation. The main simulator window is implemented as a class named *SimulatorWindow*, located in a module of the same name. The source code can be seen in appendix B.24 on page 178. The simulator window, as well

as the simulation settings window, use some helper functions to more gracefully handle certain functions in the Gtk library, as seen in appendix B.10 on page 128. A more detailed description of the various parts follows.

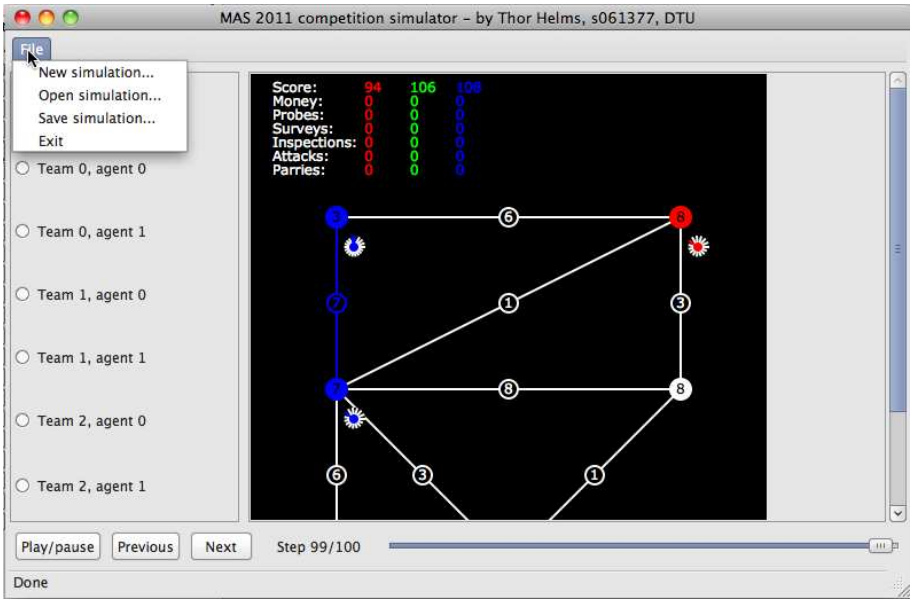

Figure 3.1: Main simulator window

3.2.1.1 Menubar

The menubar contains four items. The first, "New simulation", opens the simulation settings window, which can start a new simulation calculation. The second, "Open simulation", opens an already calculated and saved simulation. The third, "Save simulation", saves the progress of the current simulation – it will save only the view of the simulation, and thus a simulation calculation can't be resumed. Simulation views are saved as vector graphics, and will take up a lot of space for even rather small simulations. The fourth item, "Exit", will stop the current simulation and quit the program.

3.2.1.2 View-selection

The view-selection pane in the left side of the window allows the user to change which world-view to see. In figure 3.1, the "Simulation" view is selected (The selection is hidden behind the menubar), and the user can thus see the status of the entire simulation. In figure 3.2, the user has selected to view the perception for an agent. Note that both figures display the same simulation in the same step.

The user has to enable agent perception views in the simulation settings window in order to see these.

If an artificial intelligence wishes to display its world view, it can be added to and can be selected from the view-selection pane.

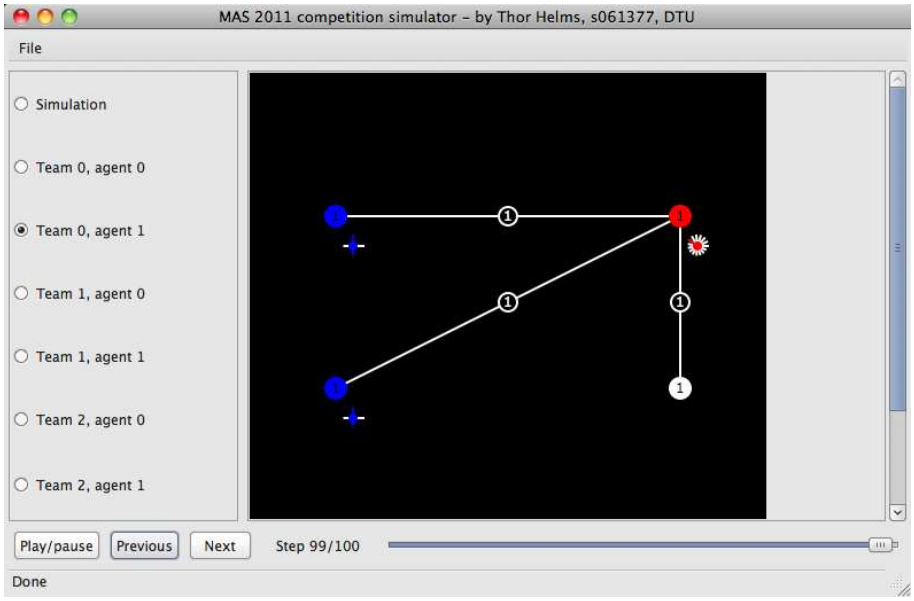

Figure 3.2: Agent perception in main simulator window

3.2.1.3 Simulation graphics view

In the right side of the window is the simulation view, visually displaying the simulation status as seen in figure 3.1. In the top left corner of the simula-

tion view is the scores, money, etc. for the various teams displayed when the "Simulation" view is selected.

If the simulation view is too big to be displayed completely in the window, a scrollbar will appear and the user will be able to see the entire simulation view anyway.

The simulation graphics view is implemented as a class named *SimulationView*, located in a module of the same name. The source code for this can be seen in appendix B.23 on page 175. The graphics view draws a number of shape primitives, consisting of rectangles, lines, circles and text. Along with a description of each shape, is a description of the color to be used. Text and lines have a single color, but circles and rectangles may have two colors, one for the stroke color and one for the fill color.

The shape types are defined in a module; the source code can be seen in appendix B.14 on page 134.

A description of the various entities in the simulation view follows later in section 3.2.2.

3.2.1.4 Simulation-view controls

The simulation-view controls in the bottom of the window consist of three buttons and a scale-selector.

The first button, "Play/pause", toggles the playback of existing simulation views. When enabled, it will automatically go to the next simulation step until no more exists, displaying three simulation steps per second.

The other two buttons, "Previous" and "Next", displays the previous and next simulation steps respectively, in respect to the currently viewed step.

The scale-selector allows the user to select which simulation step to display in larger steps than the "Previous" and "Next" buttons. The user can drag the slider to select a simulation step, or click on the scale to the left/right of the slider, in order to go 10 steps forward or backward.

3.2.1.5 Status bar

The status bar, in the very bottom of the window, displays the progress of the simulation calculation.

3.2.2 Graphics

A description of how the various entities in the simulation is displayed in the simulator follows here. There are three types of entities when viewing the simulation: Agents, nodes and edges.

Nodes are displayed in a grid, the size of which is defined by the user. The position of the nodes have no meaning for the agents in the simulation, and it is used only to display the simulation.

3.2.2.1 Agent graphics

The different agents distinguish in their stats, their color, which teams they have been inspected by and their ID. All but the ID is possible to see in the simulator view. Figure 3.3 gives an example of how an agent could be displayed. The color determines its team, so the displayed agent is on the blue team.

Figure 3.3: An example of an agent as viewed in the simulator

Agents are displayed as a circle, with its team color in the center. From the center of the circle, some amount of lines extend, forming a larger circle. These lines displays the various stats of the agent. There is one line for each one stat, for instance a visibility range of 1 will be represented with a single line.

The long white lines represent the agent's health, and as such some lines may be missing, meaning that the agent is not at full health.

The short white lines represent the agent's energy, and as such some lines may be missing, meaning that the agent is not at full energy.

The long colored lines represent the agent's visibility range.

The short colored lines represent the agent's strength.

Around the stats-circle, there may be some amount of colored lines, displaying which teams the agent has been inspected by. The agent in figure 3.3 has been inspected by the red team, and only the red team.

3.2.2.2 Edge graphics

Edges in the graph distinguish by their weight, the team dominating the edge as described earlier, the nodes they connect and which teams they have been surveyed by. All of this information is visible in the simulation view, an example of which is displayed in figure 3.4 (Nodes are not visible).

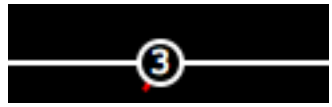

Figure 3.4: An example of an edge as viewed in the simulator

A line is drawn between the two nodes the edge connects, thus representing the edge it self. The line has the color of the dominating team, i.e. the same color as the two nodes if they have the same color. If an edge isn't dominated by a team, its color is white.

At the center of the line, the circumference of a circle is displayed in the same color as the line, with the weight of the edge in the center of the circle.

Around the circle, some colored lines may be displayed, displaying which teams the edge has been surveyed by. The edge in figure 3.4 has been surveyed by the red, and only the red team.

3.2.2.3 Node graphics

The various nodes distinguish by their weight, their ID, their neighbors, the team dominating the node and which teams the node has been probed by. All of this information, except for the ID, is visible in the simulation view, as seen in figure 3.5.

Figure 3.5: An example of a node as viewed in the simulator

The node it self is displayed as a colored circle, in the same color as the dominating team. If a node isn't dominated by any team, its color is white. In the center of the circle, the node's weight is displayed.

Around the node circle, some colored lines may be seen. These lines display which teams the node has been probed by. The node seen in figure 3.5 has been probed by the green, and only the green team.

3.2.3 Simulation settings

In this section, the window for changing the simulation settings is described. It consists of three panes, where only one at a time is visible, and five buttons in the bottom of the window, as can be seen in figure 3.6. The three panes describe the general settings, the role definitions and the milestone definitions respectively. Which pane is visible can be changed by clicking on the appropriate pane name in the top of the window.

When the simulation settings window is opened, it will have some default values entered for all variables. The default values are loaded from the text file "DefaultSettings.txt", which is expected to be located in the same folder as the simulator. Each time the simulation settings window is opened, it will have its values set to the contents of the above file.

The simulation settings-window is implemented as a class named *SimSettingsWindow*, located in a module of the same name. The source code can be seen in appendix B.18 on page 147. Each of the three panes is implemented as a class, located in their own modules.

3.2.3.1 General settings

In the general settings-pane, most of the simulation variables can be selected, as seen in figure 3.6. The pane consists of three columns.

The left-most column contains the number of teams and agents, the simulation length in ticks, the chance in percentage that agent actions will fail, the maximum response time for the artificial intelligences when queried about their wanted action, and the recharge rate in percentage for both active and disabled agents.

The middle column contains the map variables, which are the number of nodes, grid width and height, and node and edge minimum and maximum weights. It also contains a checkbox, where the user can toggle viewing of the agents' perceptions. Per default, viewing the agents' perceptions is disabled.

The right-most column contains the cost of performing the various actions, and the price of buying the various upgrades.

The class for the general settings pane is named *SimSettingsGeneral*, and it is located in a module of the same name. The source code for the general settings-pane can be seen in appendix B.15 on page 135.

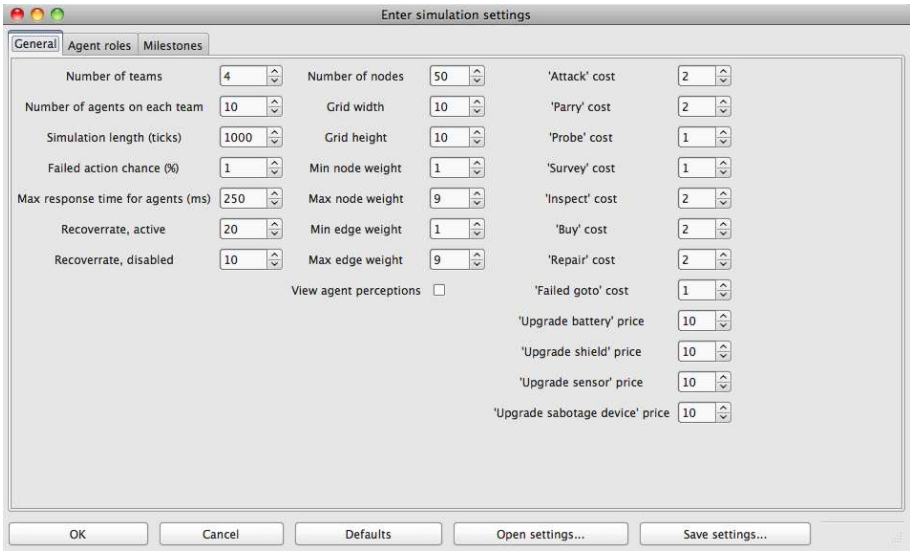

Figure 3.6: General simulation settings

3.2.3.2 Defining roles

In the agent roles-pane, the user is able to precisely define all of the roles used in the simulation, as seen in figure 3.7, by either choosing from a predefined role, or defining a custom role with custom stats and available actions. The three actions “Skip”, “Goto” and “Recharge” are always available for the agents, and thus can’t be toggled for the agents. The reason for this is, that the scenario describes robots moving around on Mars, and thus requires them to be able to move, including getting more energy or choosing not to move. Other than these three actions, all remaining actions can be enabled or disabled for all agents.

The user is also able to set the colors for the various teams, and choose which artificial intelligence controls which agent on the different teams.

Note that the visible amount of roles and teams in the agent roles-pane, is defined in the general settings-pane. If the user changes the number of teams or roles, the agent roles-pane will reflect this change.

The roles-pane is defined as a class named *SimSettingsRoles*, and located in a module of the same name. The source code for the agent roles-pane can be seen in appendix B.17 on page 141.

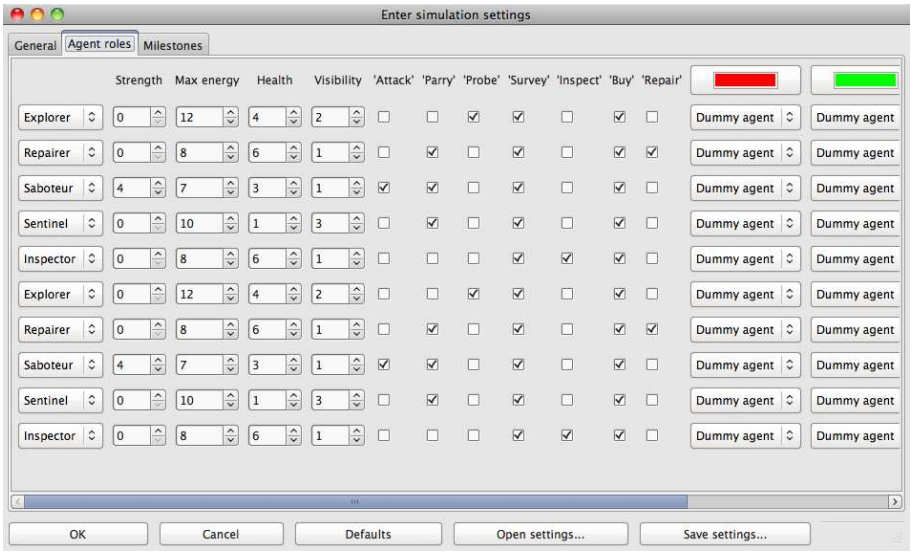

Figure 3.7: Defining the milestones to be used in a simulation

3.2.3.3 Defining milestones

In the milestones-pane, the user can define the various milestones to be used in the simulation, as seen in figure 3.8.

The user can add a milestone by clicking the "Add milestone" button in the top of the window, or remove the last milestone by clicking the "Remove last milestone" button. The milestones are displayed in a table layout, with each row defining a single milestone.

For each milestone, there are four settings: The type of milestone, the number needed to trigger the milestone, and the reward for the first team(s) and the subsequent teams to achieve the milestone. There are six types of milestones, each selectable in a drop-down box in the left-most column. The three other settings for each milestone can be set in the other three columns.

Note that the same type of milestone can appear several times, and even with the same trigger-value if the user so chooses.

The milestones-pane is defined as a class named *SimSettingsMilestones*, and located in a module of the same name.The source code for the milestones-pane can be seen in appendix B.16 on page 138.

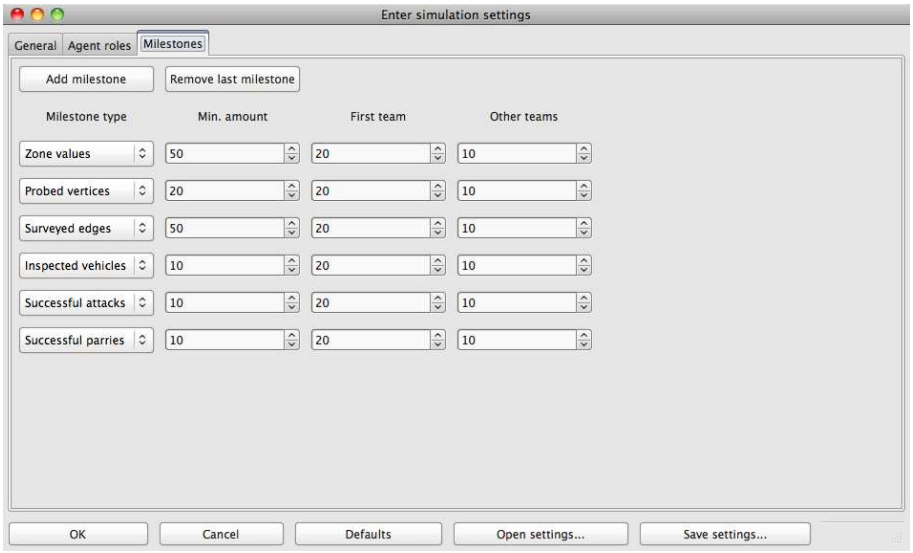

Figure 3.8: Defining the roles to be used in a simulation

3.2.3.4 Buttons

In the bottom of the simulation settings-window, there are five buttons.

The “OK” button will close the simulation settings-window and start a simulation using the entered values.

The “Cancel” button will close the simulation settings-window, and throw away the entered values.

The “Defaults” button will set all values to the default values, which are loaded from the file “DefaultSettings.txt”. The file is expected to be located in the same folder as the simulator.

The “Open settings” button will open a file-chooser window, where the user can select a file with simulation settings to open. When opening a such file, all values in the simulation settings-window will be replaced with the values from the chosen file, assuming the correct file format.

The “Save settings” button will open a file-chooser window, where the user can select where to save the entered values. The program will generate a settings-file that can be opened later. Note that it is possible to overwrite the default settings file, in order to get a new set of default values.

Chapter 4

Internals

This section describes some internal parts of the simulator. In section 4.1 the interface for external AI's is described. In section 4.2 the different custom types used in the simulator is described.

4.1 Agent interface and artificial intelligence

In order to be able to automatically load agents into the program without having to recompile the simulator, Microsoft's MEF [14] framework is used. For this framework to work, an interface has been declared, that the artificial intelligences must implement in order to hook up to the simulator. This means that at least some part of the artificial intelligences has to be a class.

4.1.1 Agent interface

The agent interface defines three methods that any artificial intelligence wanting to use the simulator, must implement. The source code can be seen in appendix B.11 on page 130.

The first function in the interface is called *MakeStep*. This function is used when the agent needs to perform an action. The agent is given a perception, and must return an action and a list of messages it wishes to send. This function is called once per simulation tick, for each agent.

The second function, *SetMaxResponseTime*, will tell the agent how many milliseconds it is allowed to use when calculating a single action response.

The third function, *SetNumNodesEdgesSteps*, will tell the agent the number of nodes and edges in the simulation, and for how many steps the simulation will take place.

Its usage, in F#, is as follows:

```
[<ExportMetadata("Name", "Some name for the agent here")>]
[<Export(typeof<IAgent>)>]
type SomeAgent() = ... // class definition
  ... // Variables and functions
  interface IAgent with
    ... // Interface functions
```

4.1.2 Automatically loading agents

Defining an interface for the agents is not enough, they must also be loaded into the program. For this, a module has been created, the source code for which can be seen in appendix B.4 on page 108.

The module will automatically load the agents, and enables the rest of the program to get an agent object based on its name, and get a list of all available artificial intelligences.

The simulator settings-window uses the list of artificial intelligences from this module to allow the user to choose which artificial intelligence to use for each agent on each team.

4.1.3 Dummy agent

In the program, a dummy agent is included. This is the default agent to be used in simulations. It implements the agent interface, but it is included directly in

the program. The source code for the dummy agent can be seen in appendix B.6 on page 115.

The dummy agent will randomly move around, probe, survey, inspect and recharge. It will only attempt to perform actions it is allowed to perform according to its role. Each of the actions mentioned above will be chosen with the same probability. When choosing to move, a random neighbor node is chosen.

4.2 Simulation specific types

In the simulation calculation, some custom types are used. The most notable of these are the *Agent*, *Edge* and *Node* types, which represent the objects seen in the GUI. Another important type is the *Graph* type, which represents a list of nodes and a matrix with edges, and thus defines a mathematical graph with an adjacency matrix for the edges.

The source code for the type definitions can be seen in appendix B.19 on page 150. Some functions have been developed to convert the *Agent*, *Edge*, *Node* and *Graph* types to a list of shapes to be used by the simulation view. The source code for these functions can be seen in appendix B.20 on page 154.

4.2.1 Action type

The *Action* type defines the different actions the simulation supports. This type is to be used by agents when answering with their requested action.

Some actions take one or two arguments, but most take none.

The *Goto* action needs an integer argument, representing the ID of the node the agent wants to move to.

The *Attack* action needs two integer arguments, representing the team ID and agent ID of the wanted target.

The *Repair* action takes an integer argument, representing the agent ID of the wanted target. Since repairs can only be performed to an agent on the same team as the repairing agent, only one argument is needed.

The *Upgrade* action takes an argument of a custom type. The custom type

defines the type of upgrade wanted, and can be either *Battery* (energy), *Sensor* (visibility range), *Shield* (health) or *SabotageDevice* (strength).

The other actions are *Skip*, *Recharge*, *Parry*, *Probe*, *Survey* and *Inspect*.

4.2.2 Agent type

The *Agent* type is defined as a record, containing the team and agent ID, the node ID of the node the agent is currently at, variables for the agent stats, a list of possible actions the agent can perform and a list describing which agents have inspected the given agent.

A module for manipulation of the *Agent* type has been created, which can be seen in appendix B.2 on page 97.

4.2.3 Edge type

The *Edge* type is defined as an *EdgeInfo Option* type. The reason for this is that not all possible edges will be present in the edge adjacency matrix of a graph, and thus there must be a way to distinguish existing from non-existing edges.

The *EdgeInfo* type is defined as a record, and contains the values describing an edge, which in this simulation is the two nodes it connects, plus its weight and a list of teams that the edge has been surveyed by. It also contains a variable determining the team that dominates the two nodes it connects, and thus the edge, if any.

A module for manipulation of the *Edge* and *EdgeInfo* types has been created, which can be seen in appendix B.7 on page 116.

4.2.4 Node type

The *Node* type is defined as a record, containing the information relevant for the simulation, which is the node ID, the weight of the node (i.e. the quality of the water well it represents), the agents that are currently located at the node, a list of the teams that has probed the node, the coordinate of the node when displayed and which team, if any, controls the node.

The *Node* type has an associated module for manipulation of it, the source code for which can be seen in appendix B.13 on page 132.

4.2.5 Graph type

The *Graph* type consists of a number of nodes and edges. The nodes are kept in an array, with their respective node ID as the index in the array. An array is used instead of a list, because many random access calls are expected, which has a runtime of O(1) in an array and O(n) in a list.

The outgoing edges for a single node is kept in an array, and all arrays of outgoing edges are kept in another array. Thus the data structure for the edges in a graph is a 2D-array, with all arrays having the same length (the number of nodes). This ensures that the 2D-array can be seen as an adjacency matrix [9]. The edges could also be kept in a 2D-list instead of an array, but for the same reasons as the nodes, it is kept in a 2D-array. As all edges are considered bidirectional, the number of edges in a graph is half the amount of edges in the adjacency matrix.

The *Graph* type has an associated module for manipulation of it, the source code for which can be seen in appendix B.9 on page 122. Unlike the modules for *Edge*, *Node* and *Agent* manipulation, the functions in the *Graph* module manipulates the actual *Graph* objects, instead of returning a new and modified version. The reason for this is, that, for the given *Graph* object in the functions to remain immutable, a copy of the entire graph would have to be made. As the size of a graph is O(n2), and there are many manipulations in each simulation step, this would very quickly lead to much redundant copying and writing of data. This of course means that the state of a graph can't be kept unless the entire graph is copied, unlike immutable datatypes where a reference to the object is all that needs to be saved. This has little influence though, as the previous simulation steps as seen in the graphical user interface is saved as lists of shape primitives.

Notable functions in the *Graph* module are the various functions used by the coloring algorithm, a function to find a zone for a starting node, a few functions for performing actions, and a function to create a new graph based on certain variables, as described below.

4.2.5.1 Find zone algorithm

The algorithm for finding a zone in a graph requires a starting node and two functions: One for determining when to accept a node into the zone; and one for determining whether or not a zone is invalidated by a found node. For instance, in the graph coloring algorithm, no active enemy agent should be able to move into the zone without passing a directly or indirectly colored node in the same color as the zone. This means that a found zone is invalidated if there is an active enemy agent in it. When determining a zone for sharing perceptions between friendly agents in the same zone, there is no way to invalidate a zone, and the zone is determined strictly by zones of the same color as the agents.

The algorithm maintains a list of accepted nodes, and a list of nodes to be investigated. Only if a node is accepted into the zone will its neighbors be added to the list of nodes to be investigated.

4.2.5.2 Coloring algorithms

The coloring algorithms consist of five different algorithms, used by the coloring algorithm in the simulation calculation.

The algorithm to reset coloring simply sets the dominating team for all nodes and edges to be non-existing.

The algorithm to color directly colored nodes calls a function from the *Node* module, which will determine the dominating team for the individual nodes, and set the dominating team in the nodes appropriately.

The algorithm for coloring indirectly dominated nodes will look at all uncolored nodes with no active agents, and determine whether or not they need to be colored, according to the color of its neighbor nodes.

The algorithm for coloring zones will find the zone for all uncolored nodes with no active agents that lie next to a colored node. The color of the neighbor node will determine which team the zone should belong to, if one such is found. The zone algorithm will accept nodes with no color and no active agents into the zone, and invalidate the zone if a non-colored node with an active agent is found, or if a colored node of an enemy team is found.

The algorithm for coloring edges will call a function from the *Edge* module on all edges, to determine whether or not they should be colored.

4.2.5.3 Action algorithms

The *Graph* module contains three functions that perform an agent action: A function for moving an agent from one node to another, a function for performing the survey-action from a given position and a function for performing the inspect-action from a given position.

The algorithm for moving an agent will check to see if the agent has enough energy to traverse the appropriate edge. If so, the agent will be moved. If not, the agents energy will be reduced by an amount defined in the simulation settings (The default value is 1).

The algorithm for surveying from a given position for a certain agent, will find all the outgoing edges from the given position, and then find the reverse edges as well. It will then set all of these edges to be surveyed by the agent if they aren't already surveyed by an agent on the same team. The number of successfully surveyed edges will then be counted and added to the appropriate counter for that team.

The algorithm for inspecting from a given position for a certain agent, will find all the agents on the neighbor nodes as well as the agents on the same node. It will then set all of these agents to be inspected by the agent if they aren't already inspected by an agent on the same team. The number of successfully inspected vehicles will then be counted and added to the appropriate counter for that team. Note that friendly agents will not be able to inspect each other.

4.2.5.4 Create graph algorithm

The algorithm for creating a new graph takes several arguments, such as maximum and minimum node and edge weights, the grid size and the number of nodes. It will create a list of all possible coordinates available within a grid of the given size, shuffle it and use it as the coordinates for the nodes it will create. This ensures that no two nodes have the same coordinate.

When the nodes have been given coordinates, it will perform a Delaunay triangulation [12] in the nodes, which will create the edges needed. For this, an external library is used [13]. It will then add these edges to the graph, which initially had no edges.

When an edge and a node is created, it will be assigned a random weight within their respective limits.

4.2.6 Milestone type

The *Milestone* type denominates one of the six predefined milestones available in the simulation, as well as three values: The ́trigger-́value and the rewards for the first team(s) and all subsequent teams to reach the milestone.

In the simulation calculation, an extended type, *MilestoneStatus*, is used. *MilestoneStatus* consist of a *Milestone* and a set of integers. The set of integers denominates which teams has achieved the milestone already, if any.

4.2.7 Message type

The *Message* type is what the agents use to communicate with each other. The *Message* type consists of a recipient, which can either be a single friendly agent or all friendly agents, plus a *MessageContent* value.

The *MessageContent* type should enumerate all possible types of communication the agents could be imagined to send. As such, the content of the MessageContent is to be defined along with the development of the artificial intelligence. Unfortunately, a change of the *MessageContent* type would require the simulator to be recompiled.

4.2.8 Percept type

The *Percept* type is defined as a record type, and represents the perception given to the agents in each simulation step. It contains the current step number, the team score, the amount of money the team has, the stats and possible actions of the receiving agent using the *Agent* type, a list of visible nodes and edges (including the shared perceptions), a list of all inspected agents, a list of all milestones in the simulation using the *MilestoneStatus* type, and a list of messages consisting of an integer representing the agent ID of the sender, and the *MessageContent* from the sent message.

4.2.9 TeamStatus type

The *TeamStatus* type is defined as a record, containing information about the score, money and number of successful probes, inspections, surveys, attacks and parries a team has made.

The *TeamStatus* type has an associated module for manipulation of it, the source code for which can be seen in appendix B.25 on page 182.

Chapter 5

Simulation calculation

A simulation is calculated in subsequent steps, with the results of one step being used in the next, until the desired simulation length has been reached. This chapter describes the algorithm used for calculating a single step. The source code for all algorithms in this section can be seen in appendix B.21 on page 159.

5.1 Simulation step algorithm

The algorithm for calculating a single simulation step proceeds as follows:

1. If the desired simulation length hasn't been reached:
2. Prepare the agent perceptions
3. Send the perceptions to the agents, and retrieve their requested actions and messages they wish to send
4. Perform all attack and parry actions
5. Perform all other actions

6. Color the graph
7. Update team scores and money
8. Draw the resulting graph
9. Recursively calculate the next simulation step

Step 8 in the algorithm will convert the graph to a list of shapes, and add it to the module that stores the different simulation steps, the source code for which can be seen in appendix B.22 on page 170.

Note that all actions other than attack and parry, are executed in the same step of the algorithm. The actions are of course executed one at a time, and not all at once, and thus a few of the actions may have influence on each other, such as the *Goto* and *Inspect* actions. If an agent performs the *Inspect* action, another agent can move one step away from the inspecting agent, and thus out of range, in the same simulation step. Which action will be executed first will thus have influence in the simulation calculation. However, this is how the official scenario description describes the algorithm. The same is true for the *Repair* and *Goto* actions.

5.2 Prepare perceptions

The algorithm for preparing the agents' perceptions start by finding all the agents in the entire graph, and finding all the nodes and edges their visibility allows them to see. It does so by maintaining a list of nodes, and extending this list with all neighbors of all nodes in the list, until the correct visibility range (depth) has been reached. During this, it maintains the list of nodes as a set, so as to not add the same nodes several times. The algorithm will then find the zone for all agents, if any, and retrieve the perception for all friendly agents in the same zone.

The algorithm will also make sure to fill in all other fields of the *Percept* type correctly. It will return a *Percept* list with the perception for all agents.

5.3 Send perceptions and receive actions

The algorithm to send perceptions to agents takes a map of artificial intelligences to be used, and a list of *Percept* values, as created in the above algorithm. It will

give each agent its perception, determining the appropriate artificial intelligence to query from the agent map, and collect their requested actions. These actions will be removed from the resulting list of actions with a certain percent chance (the action failed), determined in the simulation settings.

The algorithm will return a list of the requested actions for all agents that haven't failed to perform their action, and a list of all messages sent by all agents, regardless of whether or not their action failed.

In the simulation settings, a max response time for agents can be defined. However, it is completely up to the artificial intelligences to respect this limit, as the algorithm for sending percepts and gathering actions won't enforce it.

5.4 Execute attack and parry actions

The algorithm for performing attack and parry actions, will in reality only perform the attack actions. It will reduce the energy for all parrying agents though, regardless of whether or not they were attacked.

At first, the algorithm will create a list of all parrying agents and reduce their energy. It will then execute each attack, while determining whether or not the target is parrying. If so, the number of successful parries for the target's team will be increased. Else, the number of successful attacks for the attackers team will be increased, and the target's health will be reduced.

5.5 Execute all other actions

The algorithm for executing all other actions, other than attack, parry and of course skip actions, will determine which type of action is attempted to be performed by an agent. It will then determine whether or not the agent is able to perform the action, based on different parameters for all actions. Some actions can't be performed if the agent has been attacked, and some actions can't be performed when the agent is disabled. If the agent can perform the requested action, it will be executed accordingly.

5.6 Color the graph

The coloring algorithm consists of five steps, each calling a function from the *Graph* library:

1. Reset the coloring
2. Color all directly dominated nodes
3. Color all indirectly dominated nodes
4. Color all zones
5. Color all edges

5.7 Update milestones and scores

The algorithm for updating milestones and scores will first calculate the zone scores for all teams, not distinguishing between individual zones though. It will do so using the colors determined in the coloring algorithm. If a team hasn't probed a node it dominates, its value will count as 1, otherwise it will achieve its weight as value.

Having calculated the zone scores, the milestones are updated one at a time. An entire milestone is updated at once, to ensure that two teams reaching the same milestone at the same time as the first teams, both will receive the reward for being the first team.

After updating the milestones, and subsequently the different team's money, the scores will be calculated, as the sum of the score from the previous step, the zone score achieved in the current step and the total amount of money in the current step.

Chapter 6

Executing the simulator

The complete simulator consists of many different files, some of which are to be accessible by the artificial intelligence and thus need to be separated from the main executable created.

The executable needs a starting point, which in this case is a file created for this specific purpose. This file will simply create an instance of the main simulator window, show it and start the Gtk run-loop. The source code for this file can be seen in appendix B.12 on page 131.

6.1 Compiling the simulator

The simulator, when compiled properly, will consist of a DLL file and an EXE file. The DLL file contains all functions needed by the artificial intelligence, including functions allowing them to draw their world view in the GUI. The EXE file contains the files relevant for displaying the simulation, editing the simulation settings and performing the simulation calculation.

Compiling the simulator is easy when using the attached *Makefile*, see appendix B.1 on page 95. This will compile both the DLL and the EXE files to a folder

specified in the *Makefile*. It requires the following files to be in the specified folder:

- Triangulator.dll (Delaunay triangulator)
- System.ComponentModel.Composition.dll (MEF)

Furthermore, the simulator expects the file "DefaultSettings.txt" to be in the folder, containing the default simulator settings in the format used by the simulator.

It is also required that the compiling/running system has the Gtk library installed, including the Pango and Cairo modules.

6.2 Running the simulator

In a Windows system, the simulator can be run by executing the executable file. Note that the simulator may attempt to print to a console.

On other systems, Mono [4] is required in order to run the simulator. In this case, the application can be run by entering a terminal, going to the appropriate folder, and executing the file by writing mono followed by the name of the executable.

6.3 Testing/verification

To ensure that the simulator works correctly, some tests has been performed. However, as the simulation is very complicated and F# is a very high level and robust language, only graphical tests has been performed. The tests performed are described in this section.

6.3.1 Graph coloring

It is essential that the graph coloring algorithm works correctly, due to the fact that it determines the scores and thus the winning team.

Only active agents should be able to dominate nodes. In figure 6.1, an example of this can be seen. As seen, only the active agents can dominate nodes. Note: There is one disabled and two active agents on each team.

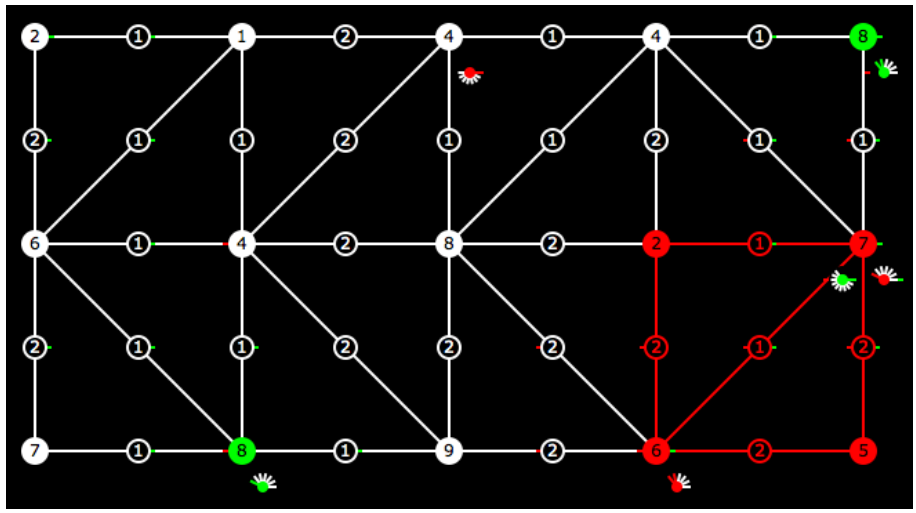

Figure 6.1: Only active agents can dominate nodes

Ties in the graph coloring algorithm should result in undominated nodes. An example of this can be seen in figure 6.2, in which an indirect tie is occurring on the top, number two from the right, node. Note that some of the agents are disabled.

In figure 6.3, an example of the zone coloring can be seen. The top and bottom nodes in the right side are only next to one directly occupied node each, however the two nodes are correctly colored as they are part of a zone. It can also be seen that there are several nodes that are not in a colored zone, as both green and red agents can reach them without crossing a node dominated by the other team first.

6.3.2 Other tests

The user can enter invalid values in the simulation settings. If so, the simulation shouldn't start. In figure 6.4, the result of invalid settings can be seen. The program checks to see if there are more nodes than possible in the given grid size, and whether the minimum value is smaller than the maximum value for

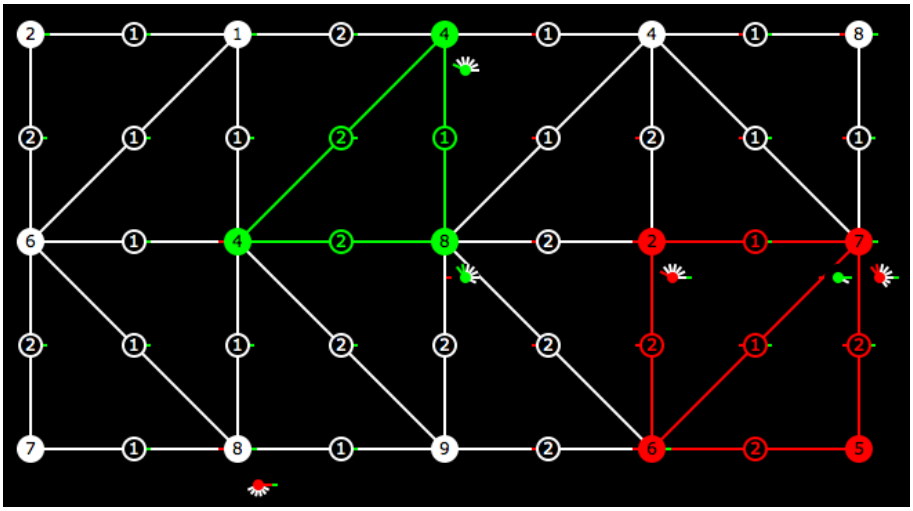

Figure 6.2: A tie in the indirect coloring

node and edge weights.

If a single team reached a milestone first, it should get the proper reward for being the first team, and all subsequent teams should receive the reward for not being the first team. In figure 6.5, an example of this can be seen. All three teams has performed 4 inspections, and thus triggered a milestone, with a reward of 20 for the first team and 10 for the other teams.

If several teams reach the same milestone at the same time, they should all receive the reward for being the first team. An example of this can be seen in figure 6.6, where all three teams has reached an arbitrary milestone of 0 inspection, and thus all three teams has received the reward of 20 money.

In figure 6.7, an example of a shared perception can be seen. The three green agents are in the same zone, and thus all three agents receive the same perception. Note that the two agents to the left has a visibility range of 1, and thus shouldn't be able to see the nodes in the right side of the graph.

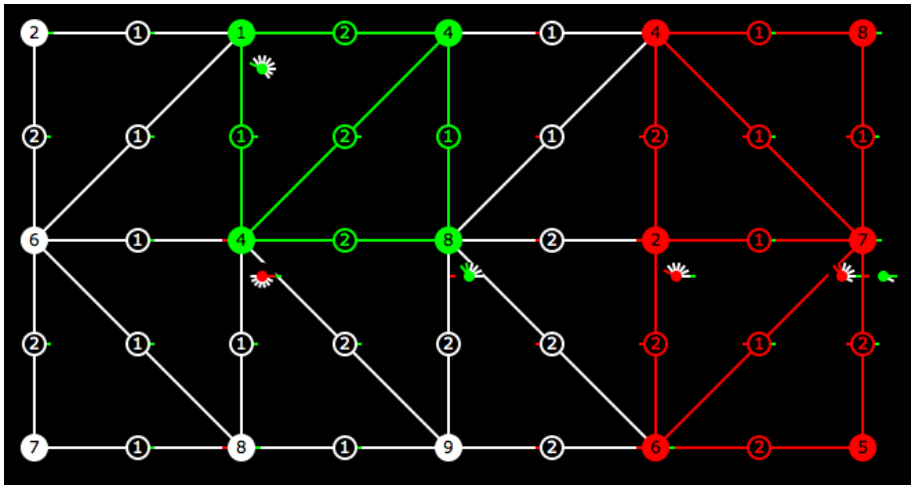

Figure 6.3: Coloring of zones

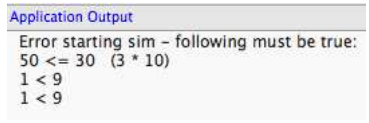

Figure 6.4: Result from invalid simulation settings

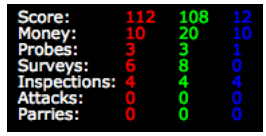

Figure 6.5: Only one team got the high reward for a milestone

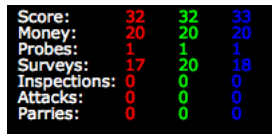

Figure 6.6: Several team reached the same milestone at the same time

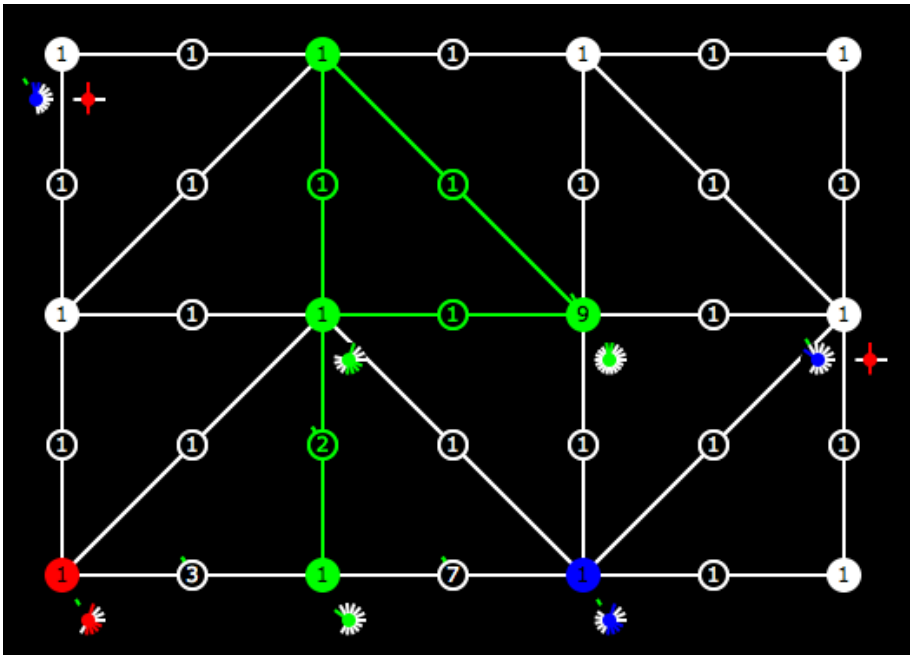

Figure 6.7: An example of a shared perception for the green team

Chapter 7

Artificial intelligence

Having created a simulator, it naturally needs something to simulate. It contains an extremely simple agent, known as the *DummyAgent*, but simulating using only this agent isn't very interesting. This chapter focuses on the development of an artificial intelligence that will be able to perform better than the dummy agent, but still remain very simple, i.e. it won't perform any complex calculations, regarding such things as the intentions of others. The created AI has been named *Aggressive Information Seeker*, or AIS for short.

7.1 Analysis and strategy

A simple agent using a constant strategy has been created. This section analyses and describes the strategy. One of the dangers when using a constant strategy, is that the opponents may be able to figure out the strategy and use this to their advantage. This danger is known and accepted for the developed agent.

7.1.1 Money and the Buy action

The objective for the agents is to maximize the team score over the course of the entire simulation. As mentioned earlier in this document, the score in each step consists of the zone scores for the various teams, plus their amount of money. As such, the more money spent, the lower the potential score will be. Even more so if the money is spent early in the simulation. This means that in order to spend money, the agents need to ensure that they will gain more than they lose. This is a potentially very complex calculation. Ignoring this calculation and thus ignoring the *Buy* action, will simplify the calculations for the agents, while the consequences will remain unknown and possible positive. In this implementation of a rather simple agent, the *Buy* action will thus be completely ignored.

7.1.2 Information sharing

In order for the agents to perform the best decisions, they must have a complete view of the world. There are two ways to receive information about the world: See it for your self, or hear-say. Some extent of first hand knowledge is given to all agents in each step. Second hand knowledge will have to be distributed via messages. Using the message system implemented in the simulation, there will however be a delay of one simulation step when sending messages.

Certain aspects of the simulation remain static, and certain aspects are dynamic. The static aspects of interest are the node and edge weights, and the stats/actions for enemy agents. The only dynamic aspect of interest is the position for all other agents. As the position of all inspected agents is known, via the perception given in each step, the information that need to be shared is that of nodes and edges. There are two cases in which information about a node or edge should be shared: Either the node/edge hasn't been seen before, in which case its completely new to the agents; or its been recently probed/surveyed, in which case information about it is new. With this in mind, it is possible to restrict the amount of messages the agents need to send (bearing in mind that in the first scenario description, there was a limit on the amount of messages the agents could send in each simulation step).

7.1.3 Information is power

For the agents, the following is true: The better information about the world they possess, the better decisions can be expected from them. In this simulation, they can know the state of the world, but not the intentions of other agents, which they can only guess.

As mentioned above, certain aspects of the simulation are dynamic and certain aspects are static. The static aspects need to be investigated, via the *Probe*, *Survey* and *Inspect* actions. As this information is static, the agents should remember this at all times.

The dynamic aspect, being the position of the other agents, will be automatically given to the agents of a team, for all agents that team has inspected.

To ensure the best decisions throughout the simulation, all information should be gathered as early as possible. Another advantage of gathering information early is that the full value of a node isn't given until its been probed.

7.1.4 Aggression

Some agents have the ability to sabotage (attack) enemy agents, and thus potentially disable them, which in turn will make sure there are certain actions they can't perform, and that they can't help their team by dominating nodes. To determine whether or not an attack actually pays off can be a very complex calculation. One might instead assume that on average, attacks pay off, and thus agents should attack as much as possible. The most important question might be which enemy to attack, as this is likely to have a bigger effect than the choice of attack or not. However, this is likely to be a complex calculation as well. One might instead assume a static priority, or simply attack at random. The AI developed here will attack a random target if possible.

7.1.5 Movement in the graph

Movement in the graph, and pathfinding in it, can't accurately be done using the weights of the nodes. These can be used as an approximation of the shortest path from one node to another, but a more accurate path can be found if assuming that the agent should have the same energy level when ending the path as when starting the path.

The amount of energy each agent can receive in each step is assumed to be known (and static, as per the rules of the simulation), denoted here as $E_{perstep}$. When traveling from one node to a neighbor node, the time taken, assuming the same energy level in the end as in the beginning, is determined by the formula:

$$Traveltime(a, b) = \frac{Weight(a, b)}{E_{perstep}} + 1 \tag{7.1}$$

Using this formula, the length of the paths in the graph can be calculated more precisely.

7.2 Implementation

The above reasoning has been implemented into a rather simple agent, with a priority of actions to perform.

7.2.1 MakeStep algorithm

The *MakeStep* function is the main function in the agent interface. It receives a perception, and answers with the requested action.

The implemented algorithm augments the agent's memory with the information received in the perception, and then calls another function to determine the requested action based on the (believed) state of the graph.

The *MakeStep* algorithm proceeds as follows:

1. Augment the believed state of the world with the information of nodes and edges sent from friendly agents.
2. Reset the coloring of the graph.
3. Augment the believed state of the world with the nodes and edges the agent is able to see, store the new information.
4. Augment the believed state of the world with the state of all inspected agents.
5. Calculate all possible and meaningful actions in the current state.

6. Determine the requested action using the calculated actions from step 5, using the *PrioritizedAction* algorithm as described below.
7. Draw the belief-state to the GUI.
8. Return the requested action from step 6, and the new information from step 3 in the form of messages.

7.2.2 PrioritizedAction algorithm

To determine which action the agent should perform, a very simple priority is used. It requires an *Agent* object and a list of possible actions, which it will use to determine whether or not an action is possible and meaningful to perform.

The algorithm (and priority) proceeds as follows:

1. Recharge if the agent has less energy than required to perform any of the actions in steps 2-7.
2. Probe.
3. Survey.
4. Inspect.
5. Attack.
6. Parry.
7. Repair.
8. Determine the best position to be at in the graph, using the *NextNode* algorithm, and move towards it.

7.2.3 NextNode algorithm

The *NextNode* algorithm will determine the estimated best position to be at in the graph, while also considering the amount of time it takes to reach that position. The travel-time T to a node will result in a delay factor of 0.99^T.

The value for the nodes is calculated using an ad-hoc method, considering several different aspects such as the proximity of friendly and enemy agents, whether

the node is unprobed and the agent can probe, and more. Note that when the agent is disabled, only nodes with a friendly agent that can repair will have a positive value.

The *NextNode* algorithm requires an *Agent* object and a list of possible actions, and will return an action, which is always either a *Goto* action or the *Recharge* action. It proceeds as follows:

1. Calculate all shortest paths from the current node, to be used when calculating the delay factor.
2. Calculate node values and apply the delay factor.
3. Select the target node, i.e. the node with the highest value when also considering the delay factor.
4. Find the first node in the shortest path to the target node.
5. If the target node is the node the agent is currently at, or if the agent doesn't have enough energy to move towards it, then return the *Recharge* action. Else, return the *Goto* action, with the node from step 4 as parameter.

7.3 Testing/results

To show how the AIS agents perform, this section describes two simulations in which the AIS agents has to fight against either three teams of dummy agents or three other teams of AIS agents. Primarily their ability to gather information and sabotage enemy agents is analyzed.

7.3.1 Simulation 1

In this simulation, one team of AIS agents are up against three teams of dummy agents. Each team has 10 agents, 2 of each of the predefined roles. There are 50 nodes in a 12x6 grid. The generated graph has 125 edges, and the length of the simulation is 1000 steps. Every 100th step, starting from step 0, can be seen in appendix A.2 on page 70. The milestones used can be seen in appendix A.1 on page 68, as well as in table 7.3.6.

	Simulation 1	Simulation 2
All nodes probed:	151/-/-/- steps	312/252/168/285 steps
All edges surveyed:	44/-/-/- steps	79/71/76/88 steps
All enemy agents inspected:	83/-/-/- steps	69/63/68/75 steps

Table 7.1: Gathering information in the two simulations

7.3.2 Simulation 2

This simulation has the same basic setup as simulation 1, except that there are 4 teams with AIS agents and no teams with dummy agents. The generated graph has 125 edges.

Every 100th step, starting from step 0, can be seen in appendix A.3 on page 82. Note: A lot of agents from the red, green and yellow teams gather at a single node rather fast (before the 100th step), and for the remainder of the simulation, this node is very populated.

7.3.3 Gathering information

One of the main targets with the AIS agent is that it should prioritize gathering information over all other actions.

The most important piece of information is the probing of nodes, as this will increase the potential score for the team. The figures in table 7.3.3 gives an idea of how fast this information can be gathered, against both passive (simulation 1) and aggressive (simulation 2) enemies. The best case is from simulation 1, in which it took 151 steps to probe 50 nodes, with 2 agents able to probe. This gives approximately one probe per 6 steps per agent, which means an average travel/recharge time of 5 steps between each probe for each agent. The worst case is in simulation 2, in which it took the red team 312 steps to probe all nodes. This gives approximately one probe per 12 steps per agent, which means an average travel/recharge time of 11 steps between each probe for each agent. This is however against aggressive opponents, which means that the two red agents that are able to probe, might have been disabled some of the time. In both simulations however, the probing agents may also have spent time surveying edges, as they are able to perform that action as well, and it is prioritized higher than movement.

Surveying of edges shows the same tendencies as probing nodes: Aggressive opponents increase the amount of time taken by at most a factor 2. In the best case (simulation 1) it takes 44 steps to survey all 125 edges, with 10 agents able to survey. This means that there is approximately 3.5 steps between each survey for each agent in the best case, and 7 steps in the worst case. This is faster than probing because the agents can survey multiple edges at a time, but only probe one node at a time.

Inspecting enemy agents take approximately the same amount of time with or without aggressive opponents. With two agents able to inspect, and 30 enemies to inspect, there is approximately 4.2 steps between each inspection in the best case, and 5.5 in the worst case. As with surveying, this is faster than probing because multiple opponents can be inspected at the same time.

Information about the layout of the graph is needed as well; however, this information is automatically gathered when the agents attempt to probe and survey everything. Once all nodes has been probed, or all edges surveyed, all nodes has with 100% certainty been visited, and all information about the graph has been available at one time or another to at least one agent on the team. Assuming perfect sharing of information, all agents on the team is expected to know the layout of the entire graph once all nodes has been probed or all edges surveyed.

7.3.4 Attacking/repairing

The AIS agents are supposed to be aggressive and thus attack enemies very often.

In simulation 1, all enemies of the single AIS team are dummy agents, and thus not able to attack or parry. The result of this, and the aggression from the AIS agents, is that all but three enemy agents are disabled in the end of the simulation. As can be seen in table 7.3.4, 37 attacks successful attacks has been performed. As two agents was able to attack, this gives one attack per 54 simulation steps per agent. This figure suggests that the agents aren't as aggressive as they could be.

In simulation 2, a lot of agents from team 1, 2 and 4 are gathered at a single node through most of the simulation. The cause of this is the priority of actions, such

	Simulation 1	Simulation 2
Total number of attacks:	37/0/0/0 steps	357/394/80/328 steps
Total number of parries:	0/0/0/0 steps	11/126/1/234 steps

Table 7.2: Total number of attacks and parries for the various teams in the two simulations

	Simulation 1	Simulation 2
Total score	140.677/11.263/5.337/ 2.931	61.678/58.289/111.381/ 48.474
Money in last step	24/4/0/0	22/28/24/26
AvgMin	11.67/0.73/0.53/0.29	3.97/3.03/8.74/2.25

Table 7.3: Scores over the course of an entire simulation, i.e. 1000 steps

that attacking and repairing is prioritized above moving away from the lump of agents. This means that the agents that can repair, will be caught on the node as long as there are agents to repair, and agents that can attack will likewise be caught, as long as there are enemies to attack. This yields an infinite loop given the right circumstances. The effect of this is seen in the number of successful attacks and parries for those teams, which are rather large compared to team 3, with the exception of the number of parries for team 1. In this case, aggression might be too high, due to the simple priority of actions.

7.3.5 Forming groups/zones

When moving around, the agents should attempt to form groups, and thus increase the team score, while trying to not stand at the same node as other active agents.

Table 7.3.5 shows the ending score and amount of money for all teams in both simulations. The last row of the table displays the minimum possible average zone-score per agent per step, which is calculated using the following formula:

$$Avg_{Min,i} = \frac{Score_{i,end} - Money_{i,end} \cdot Steps}{Steps \cdot Agents_i} \tag{7.2}$$

In simulation 1, the AIS agents each scored at least 11.67 points in zone-score per step, which is higher than the uncooperative maximum of 9 (the highest possible

	Simulation 1	Simulation 2
Zone score: 50	40/-/-/- steps	112/-/58/514 steps
Probed vertices: 25	59/-/-/- steps	102/87/88/105 steps
Surveyed edges: 100	12/215/-/- steps	18/14/27/26 steps
Inspected vehicles: 20	44/-/-/- steps	21/17/16/37 steps
Successful attacks: 40	-/-/-/- steps	136/182/194/151 steps
Successful parries: 30	-/-/-/- steps	-/216/-/125 steps

Table 7.4: Time taken for completion of milestones

node-weight). This number can't be compared to the other teams in simulation 1 though, as the amount of active agents differ throughout the simulation, but it does suggest some cooperation.

In simulation 2, the three teams that went into partial deadlock scored well below the team that didn't. The agents that wasn't in deadlock did however cooperate to some extent, which is evident from both the images from simulation 2 as well as the data in table 7.3.5.

7.3.6 Achieving milestones

An unintended, but positive, side-effect of gathering information and attacking enemy agents, is that the milestones (if any such are defined) can be achieved. This will result in money, which in turn will return in a higher score for the team.

Table 7.3.6 shows the amount of time take for the various teams to achieve the various milestones.

In simulation 1, the AIS team was very fast to achieve the first four milestones, while only a single dummy team achieved a single milestone.

In simulation 2, all teams took almost the same time to achieve the 2nd, 3rd, 4th and 5th milestones, while the other two (zone score and parries) were up to chance. This suggests that information-seeking and aggression will yield two thirds of the milestones consistently.

Chapter 8

Discussion

The previous chapters has had a rather objective view on the development process and product. In this chapter, a more subjective view on the process and product is given, along with a few comments on the future potential of multi-agent systems.

8.1 The competition

In the beginning of this project, there was no official simulator available. According to the time-schedule for the competition, the simulator should have been released prior to the start of this project though, and as such it wasn't possible to tell when and in what state an official simulator would be released. This enforced the creation of a simulator in this project. Looking back, the creation of this took far too much time and removed time from the development of an artificial intelligence.

Along with the release of the official simulator, a changed scenario description was given. The new scenario was changed on some key points to simplify the development of AI's, but as the development of the simulator in this report had already begun, some of the old requirements was kept.

One of the changes in the new scenario was the removal of enforced communication through the simulation server, and the removal of mixed teams. This would enable agents to communicate with friendly agents without a 1-tick latency. In this project, the agents aren't strictly required to communicate through the simulation server, but it is possible and suggested as the teams can consist of several different types of AI's. This reduces flexibility in the development of the AI's, but highlights another interesting problem: multi-agent systems with delayed communication, which will be further discussed in section 8.4.

Time in the simulation is discrete, which causes both the complexity of solutions and realism of the simulation to be reduced. However, if the simulation ran in realtime it would increase the complexity of not only the AI development, but also the simulation process as the agents might be located on remote machines.

8.2 Functional programming

Using functional programming made the development process much faster than for instance using C#. One reason for this is that the code written is simply more succinct, and type definitions are rarely used, yielding shorter lines. Another reason is that many otherwise recurring errors, such as null-pointers, just doesn't happen in a functional language. The task of memory management is also removed.

Comparing F# to C#, F# is faster to write, able to do nearly everything C# can, plus more. F# is however not able to exploit the .NET 4.0+ features as of writing, as it only exists to .NET 3.5-. It should run exactly as fast, as both languages are compiled to the same byte-code, which also means that they can work together once compiled. Both languages are able to run on Mono, which means they can run on multiple platforms. Up to Mono 2.8, F# needs to be installed separately, but in Mono 2.10+, F# is bundled. However, the F# plugin for MonoDevelop 2.4 doesn't work with Mono 2.10 as of writing.

As it is possible to use all existing .NET and Mono frameworks, creating a GUI was easy, but it also required a lot of the code to be object oriented. A clear distinction between functional code and object oriented code has been sought, making most functions outside of classes purely functional with no side-effects (With the functions in the *Graph* module being a large exception).

8.3 AI performance

The AIS AI uses only a single strategy throughout the simulation, which opponents can exploit to their advantage. Furthermore, the strategy used doesn't involve any form of planning or communication with friendly agents about plans, which means that the team will only get the optimum score by chance. The AIS AI also doesn't try to determine the strategy of opponents. This means that it isn't a truly multi-agent AI, but merely reacts to the current state of the world. It does however perform better than the dummy agent by a large margin, and would thus serve as a great replacement for it, for other AI's to compete against.

8.4 Perspectives

Multi-agent systems has many uses in the real world. Robotic cars for instance can be considered agents in a multi-agent system, where all other agents have varying characteristics.

The development of a true multi-agent theory, considering both friendly and enemy agents, will likely have huge impacts on areas such as financial markets, robotic sports and cars, and to some extent the analysis of human behavior.

Multi-agent systems with delayed communication can for instance be used to improve the performance of high-speed trading, where several systems can be made to cooperate, and thus exploit the strategies of other high-speed trading systems. Due to the speed at which trading occurs, communication might not be possible in real-time though, which is why the notion of delayed communication can be used to improve this area. Furthermore, any multi-agent system with insecure communication might have to perform delayed communication: imagining that there is some state with (relatively) secure communication, the strategy until the next state with secure communication should be agreed upon before entering a state with insecure communication. For example, in (american) football, the players of one team agrees upon a strategy before each play, but changing strategy mid-play will reveal the strategy to the opponents.

Chapter 9

Conclusion

During this project, a simulator and an artificial intelligence has been created. The scenario description [2] [3] that this project built upon changed during the project period, which has caused the scenario used to be a mix of the two different versions of the scenario description. In detail, mainly the newest version of the scenario was used, with an added requirement from the old version, which increased the complexity of the AI's to be used. This in turn opened up another interesting problem, that of delayed communication in multi-agent systems. This however is in slight conflict with an assumption in the simulator, namely that of implied secure communication, whereas delayed communication is mostly relevant for systems with insecure communication.

The simulator is flexible in that it allows the different AI's to display their world model in the GUI. Ensuring that AI's can display whatever they want has however meant that saving a simulation to the disk will have to save the graphics, instead of simply the simulation state. This means that a saved simulation will take up much more disk space than would be necessary if only the actual simulation state was saved. One might assume that this will also have an influence on how much memory a running simulation will consume.

The language F# has been used to create the simulator and the AI. This has in both cases meant a mix of purely functional programming and object oriented programming. To truly be able to take advantage of the potentials of functional

programming, a clear distinction between functions with and without side effects has been kept, while trying to keep the amount of functions with side effects to a minimum.

The GUI was created using existing object oriented frameworks, but this proved to be no problem as F# is able to handle that with grace, and possible even easier than its object oriented counterpart, C#. As F# is able to do nearly everything C# can and more, one might ask: Why use C# at all? Multi-paradigm languages gives the programmer more flexibility, but at what cost?

The created AI, named Aggressive Information Seeker, uses a single strategy, which makes it vulnerable to opponents that can predict behavior – but one is always vulnerable to smarter opponents. The strategy used is more than adequate to beat the built-in dummy agent, which is the only other AI is has been tested against. Against it self it does have a flaw though, in that several agents from several teams might lump together at the same one node, and become deadlocked in perpetual attacks and repairs.

The AIS agents cooperate to some extent, but the amount of cooperation is limited in that they don't communicate about plans at all.

Bibliography

[1] Multi-Agent Contest, 2011 version, http://www.multiagentcontest.org/2011

[2] Tristan Behrens, Michael Köster, Federico Schlesinger, Jürgen Dix, Jomi Hübner, Multi-Agent Programming Contest Scenario Proposal 2011 Edition, November 20th, 2010

[3] Tristan Behrens, Michael Köster, Federico Schlesinger, Jürgen Dix, Jomi Hübner, Multi-Agent Programming Contest Scenario Description 2011 Edition, February 16th, 2011

[4] Mono, http://www.mono-project.com

[5] Mono documentation, http://www.go-mono.com/docs/

[6] GTK+ Reference Manual, Widget Gallery, http://www.gtk.org/api/2.6/gtk/ch02.html (gtk.org)

[7] F# programming language, http://en.wikipedia.org/wiki/F_Sharp_%28programming_la (wikipedia.org)

[8] F# Programming, http://en.wikibooks.org/wiki/F_Sharp_Programming (wikibooks.org)

[9] Adjacency matrix, http://en.wikipedia.org/wiki/Adjacency_matrix (wikipedia.org)

[10] Lambda calculus, http://en.wikipedia.org/wiki/Lambda_calculus (wikipedia.org)

[11] Floyd-Warshall algorithm, http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_alg (wikipedia.org)

[12] Delaunay triangulation, http://en.wikipedia.org/wiki/Delaunay_triangulation (wikipedia.org)

[13] S-hull: a fast sweep-hull routine for Delaunay triangulation http://www.s-hull.org/

[14] Managed Extensibility Framework (MEF), Microsoft, http://mef.codeplex.com/

[15] José M. Vidal, Multiagent Systems, 2009, http://multiagent.com/

Appendix A

Tests and results

This section contains the screenshots from two simulations. Only every 100th step is kept here.

The first section displays the settings used. The next two sections display the two simulations, the first of which consists of one team of AIS agents and three teams of dummy agents, the second of which consists of four teams of AIS agents.

A.1 Settings for simulations 1 and 2

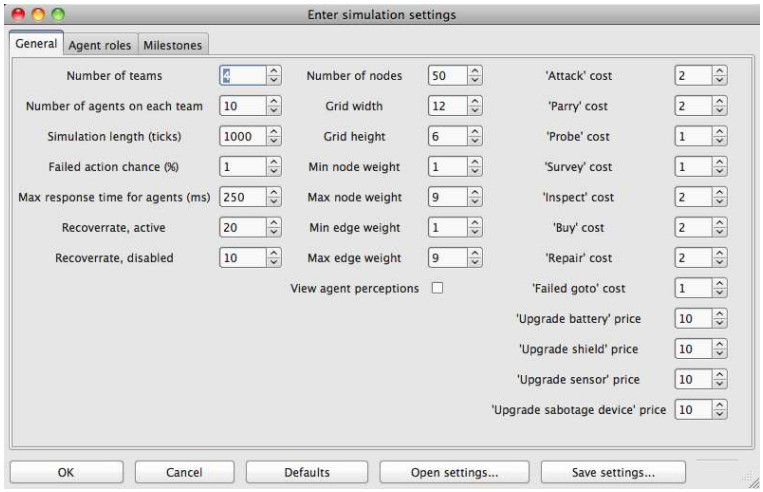

Figure A.1: General settings used in simulations 1 and 2

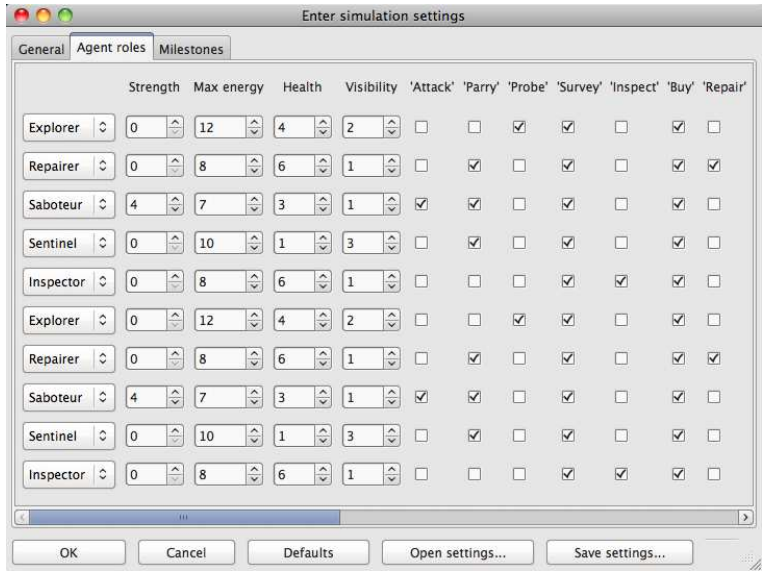

Figure A.2: Agent settings used in simulations 1 and 2

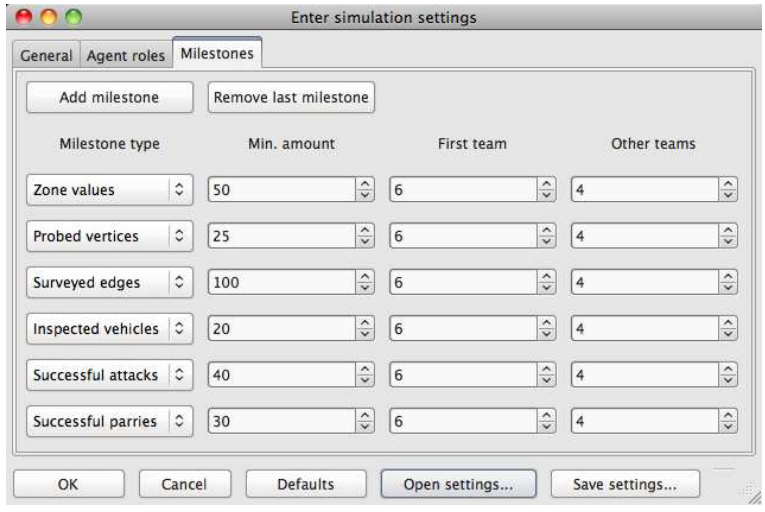

Figure A.3: Milestone settings used in simulations 1 and 2

A.2 Simulation 1

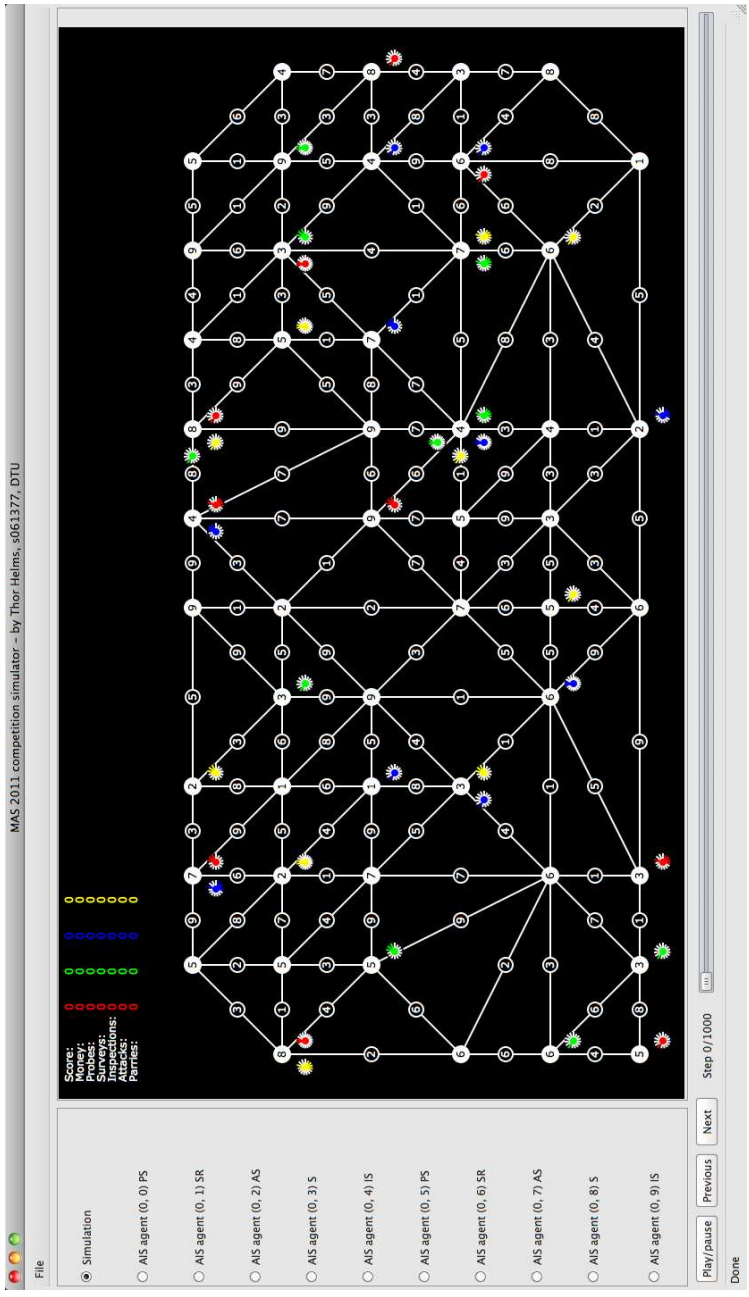

Figure A.4: Simulation 1, step 0

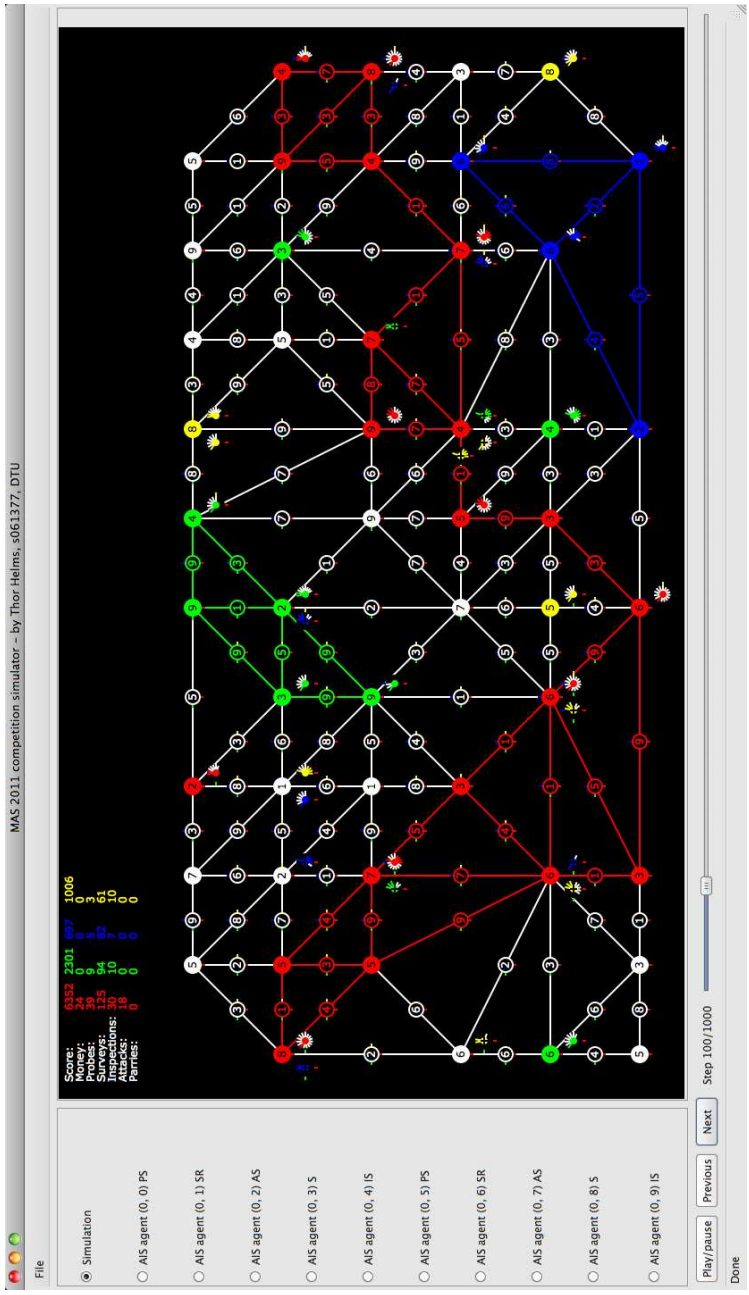

Figure A.5: Simulation 1, step 100

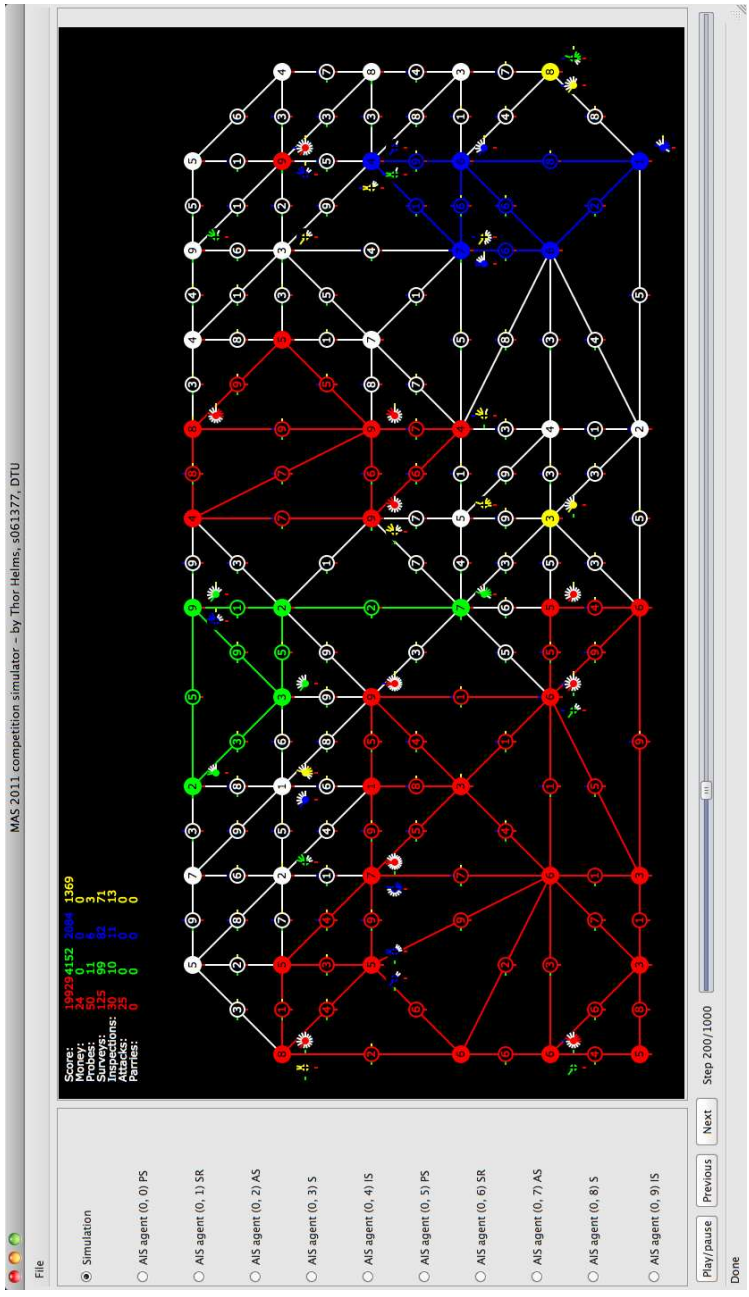

Figure A.6: Simulation 1, step 200

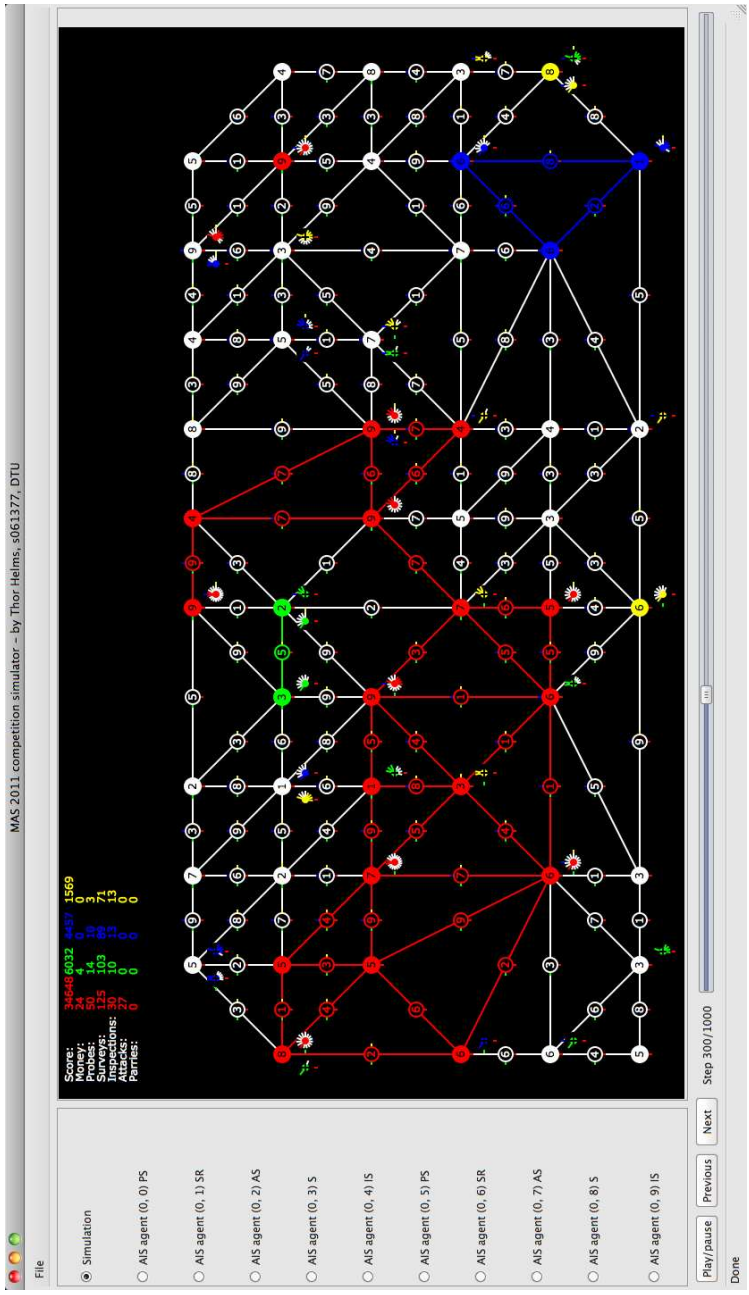

Figure A.7: Simulation 1, step 300

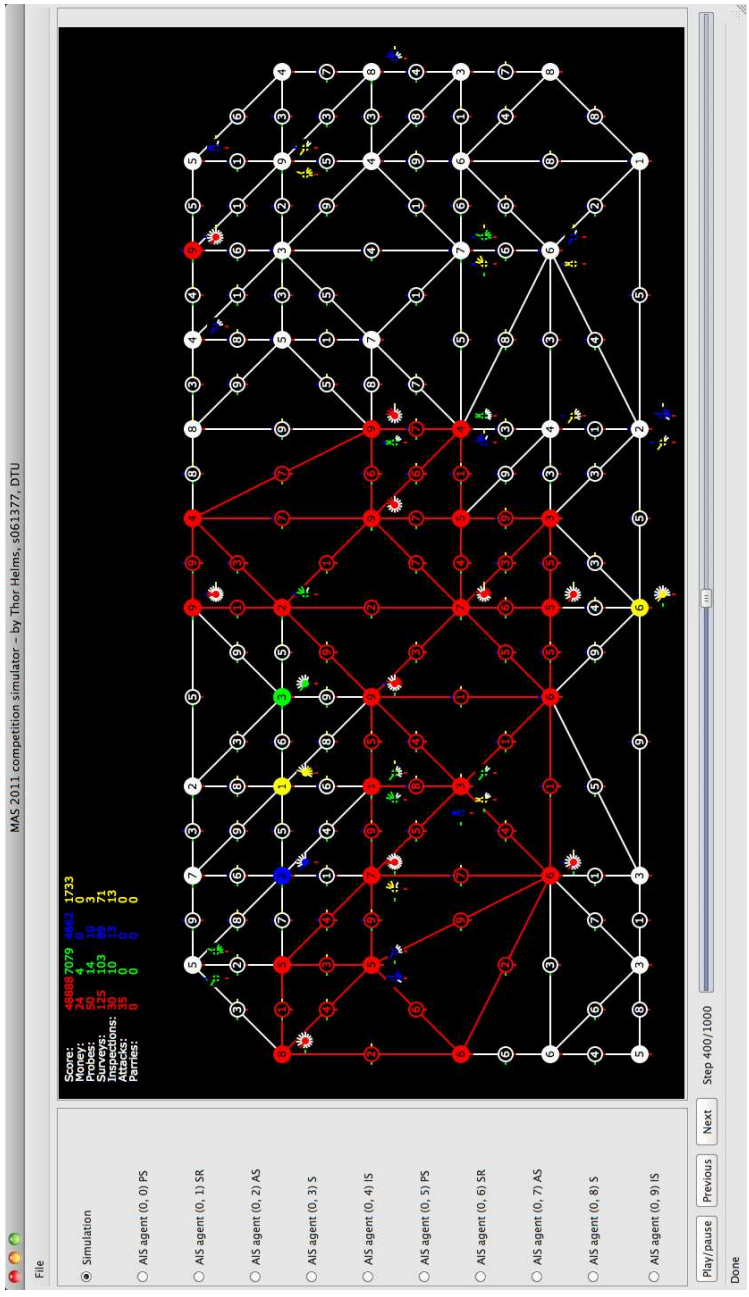

Figure A.8: Simulation 1, step 400

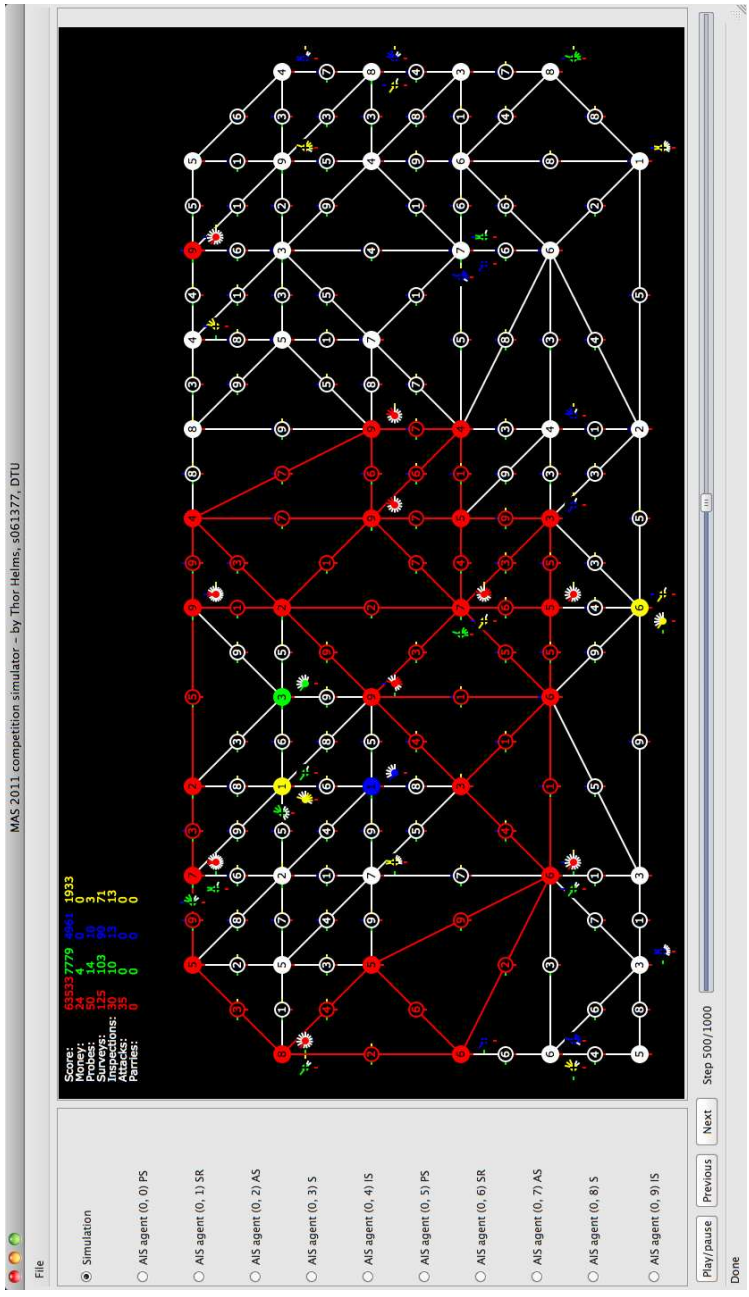

Figure A.9: Simulation 1, step 500

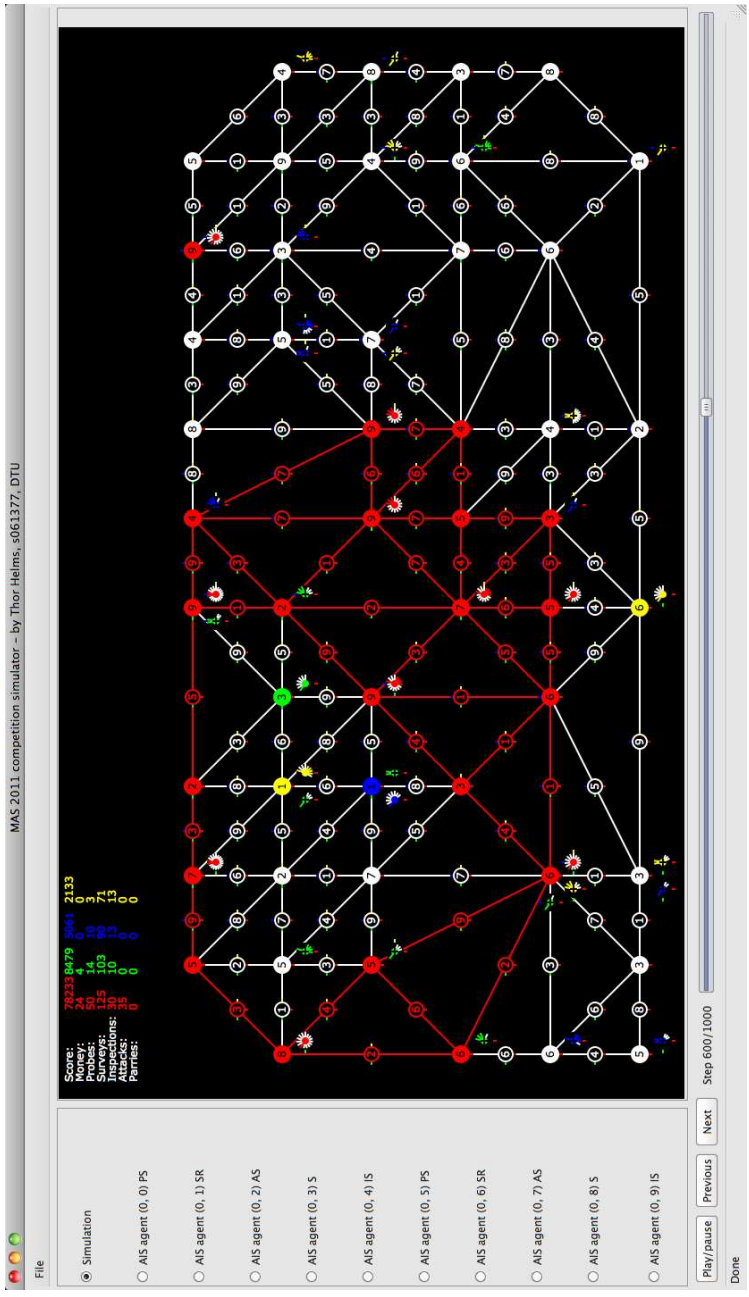

Figure A.10: Simulation 1, step 600

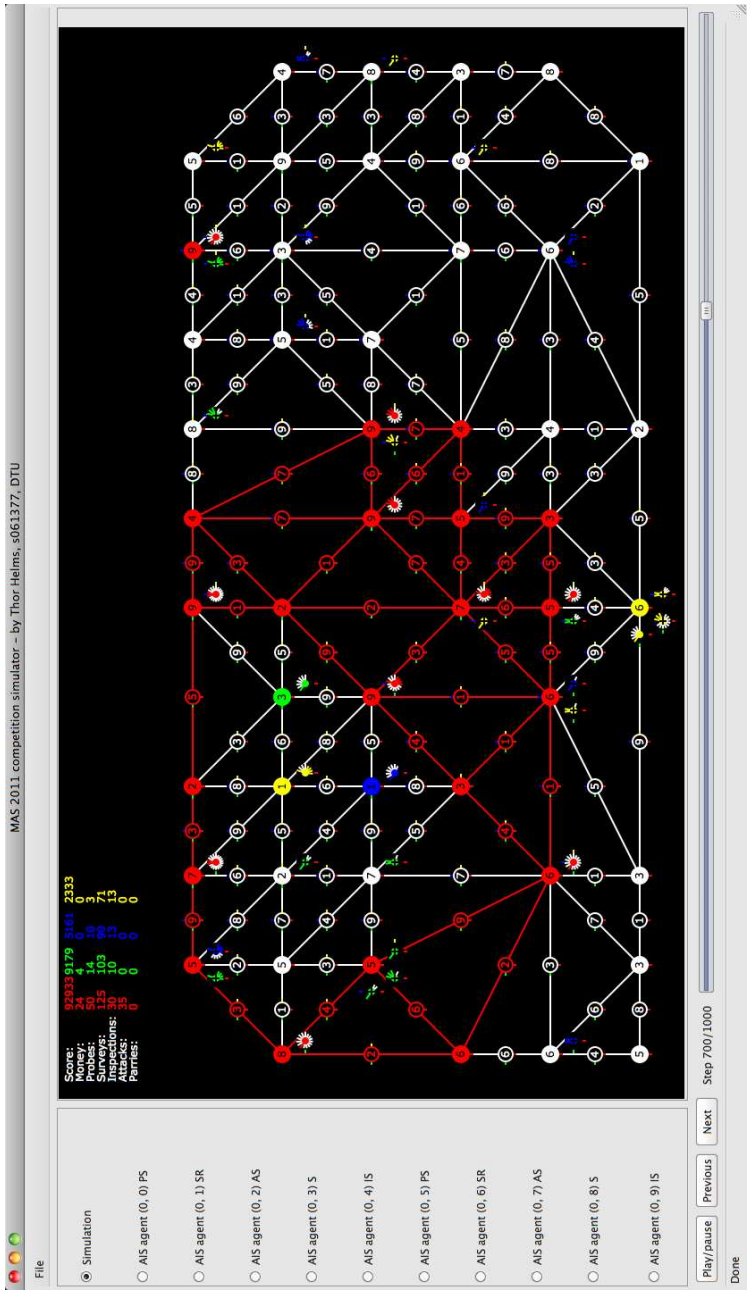

Figure A.11: Simulation 1, step 700

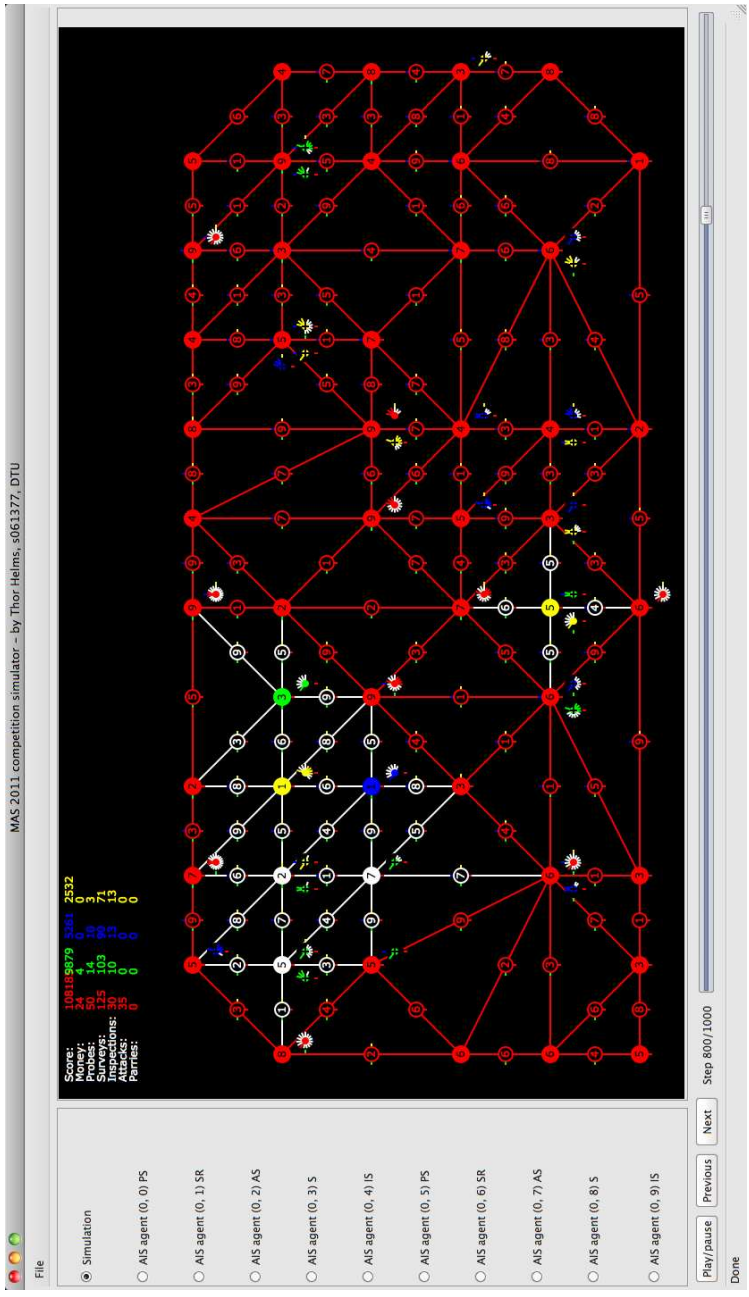

Figure A.12: Simulation 1, step 800

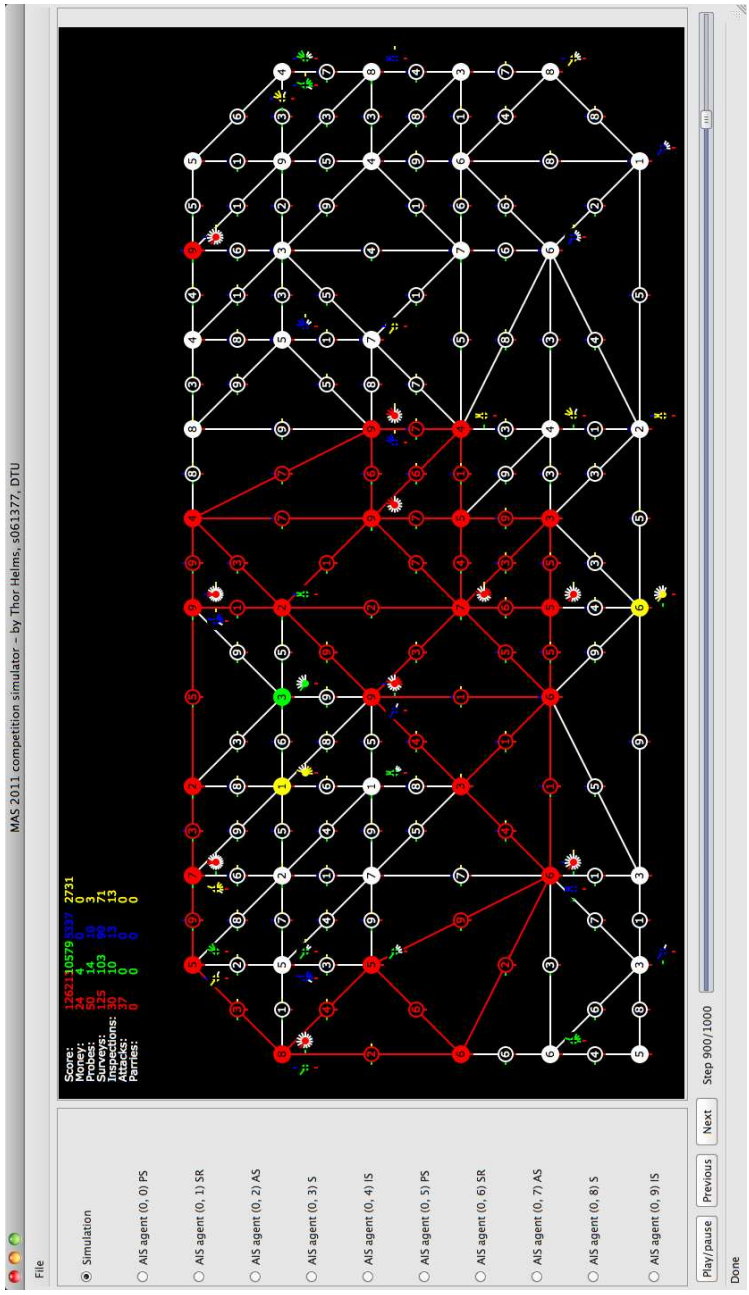

Figure A.13: Simulation 1, step 900

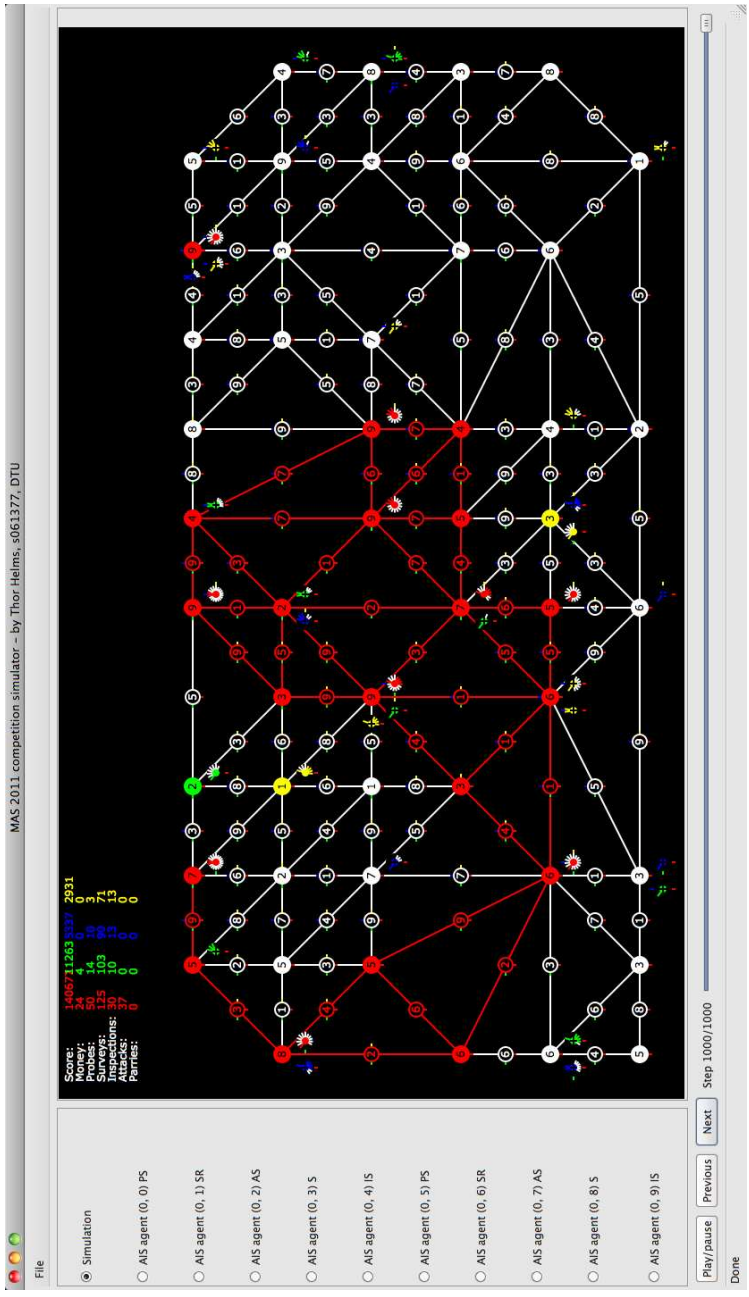

Figure A.14: Simulation 1, step 1000

A.3 Simulation 2

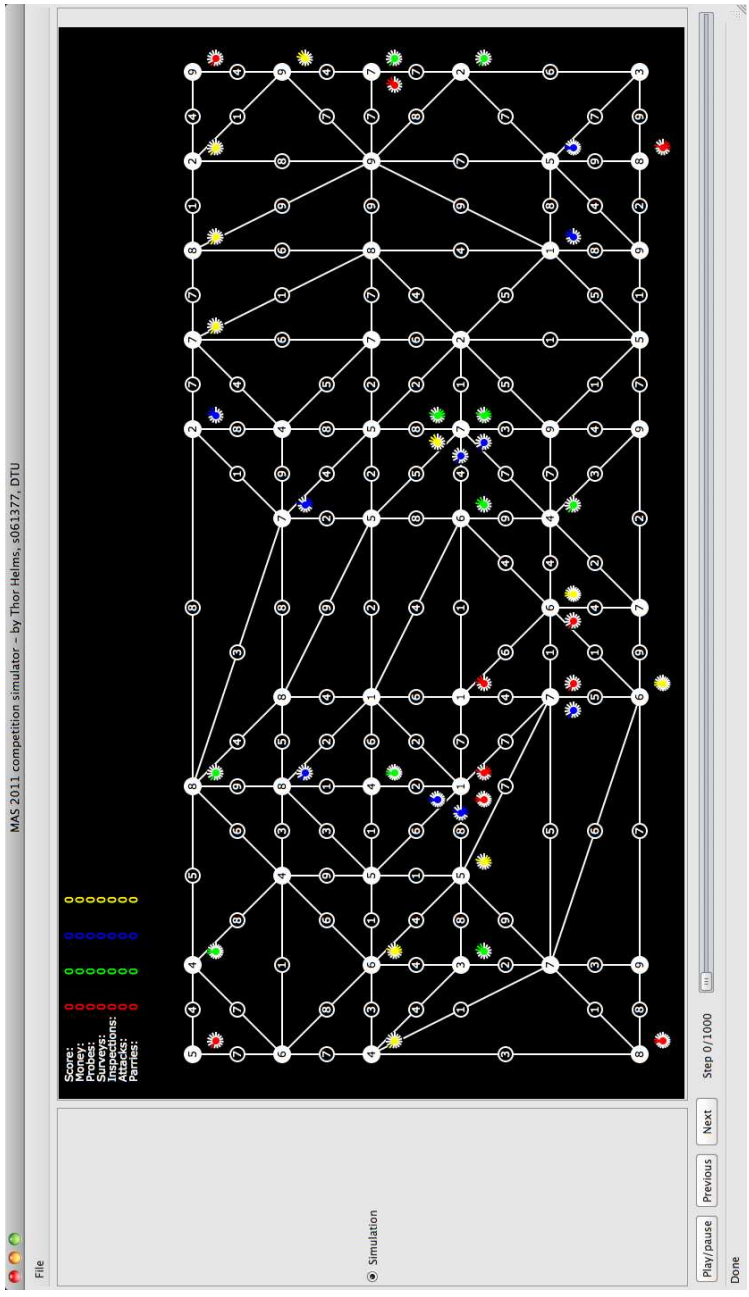

Figure A.15: Simulation 2, step 0

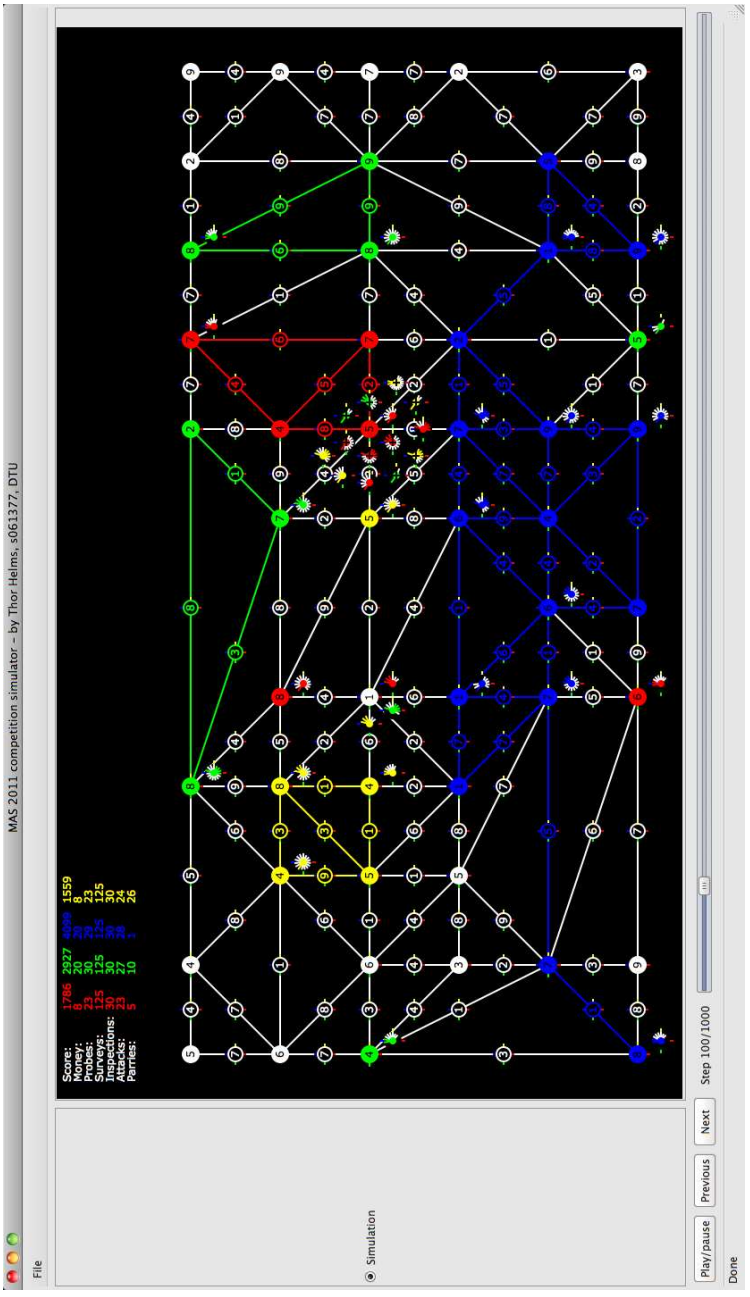

Figure A.16: Simulation 2, step 100

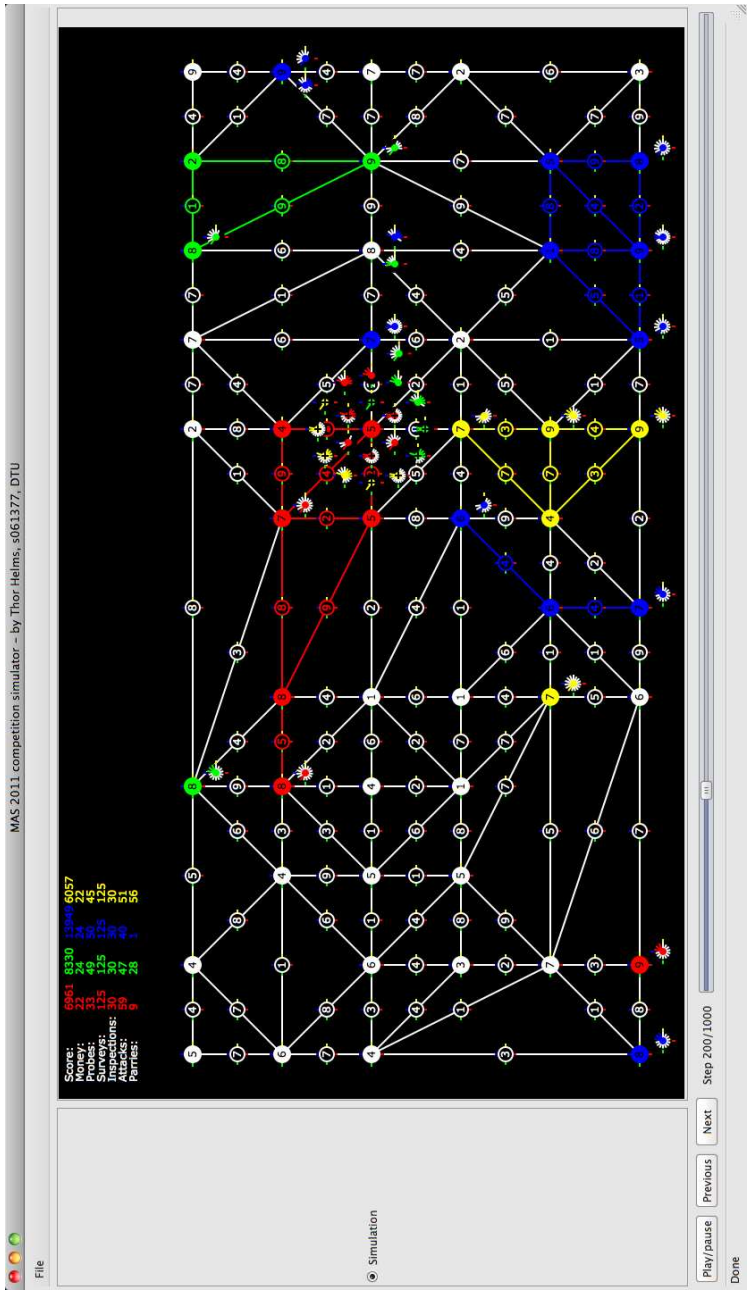

Figure A.17: Simulation 2, step 200

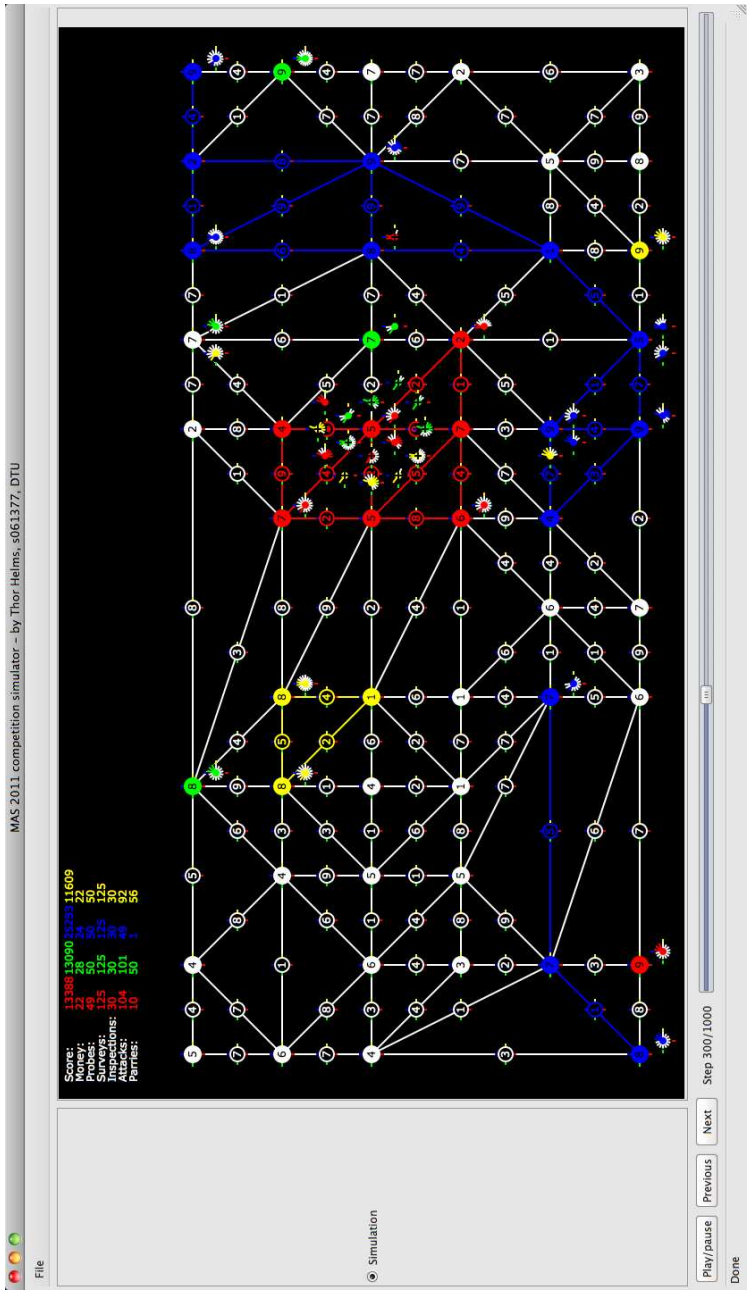

Figure A.18: Simulation 2, step 300

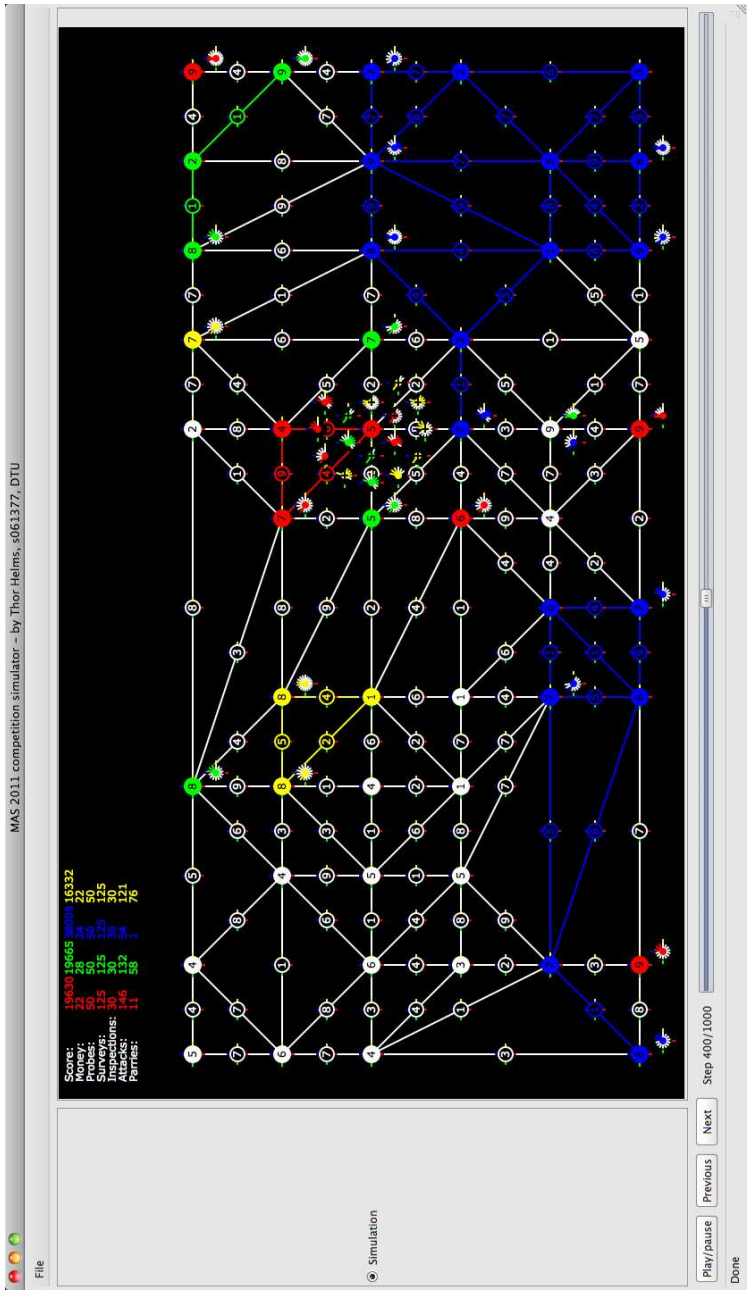

Figure A.19: Simulation 2, step 400

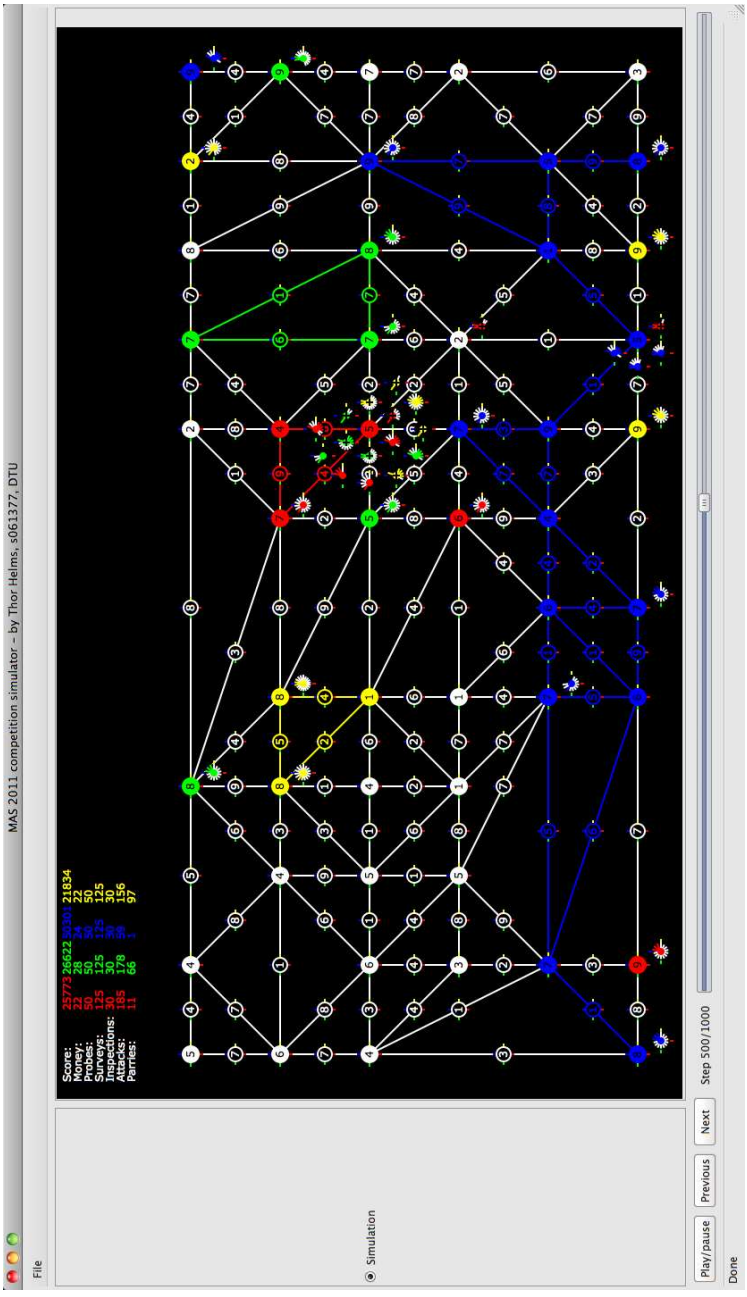

Figure A.20: Simulation 2, step 500

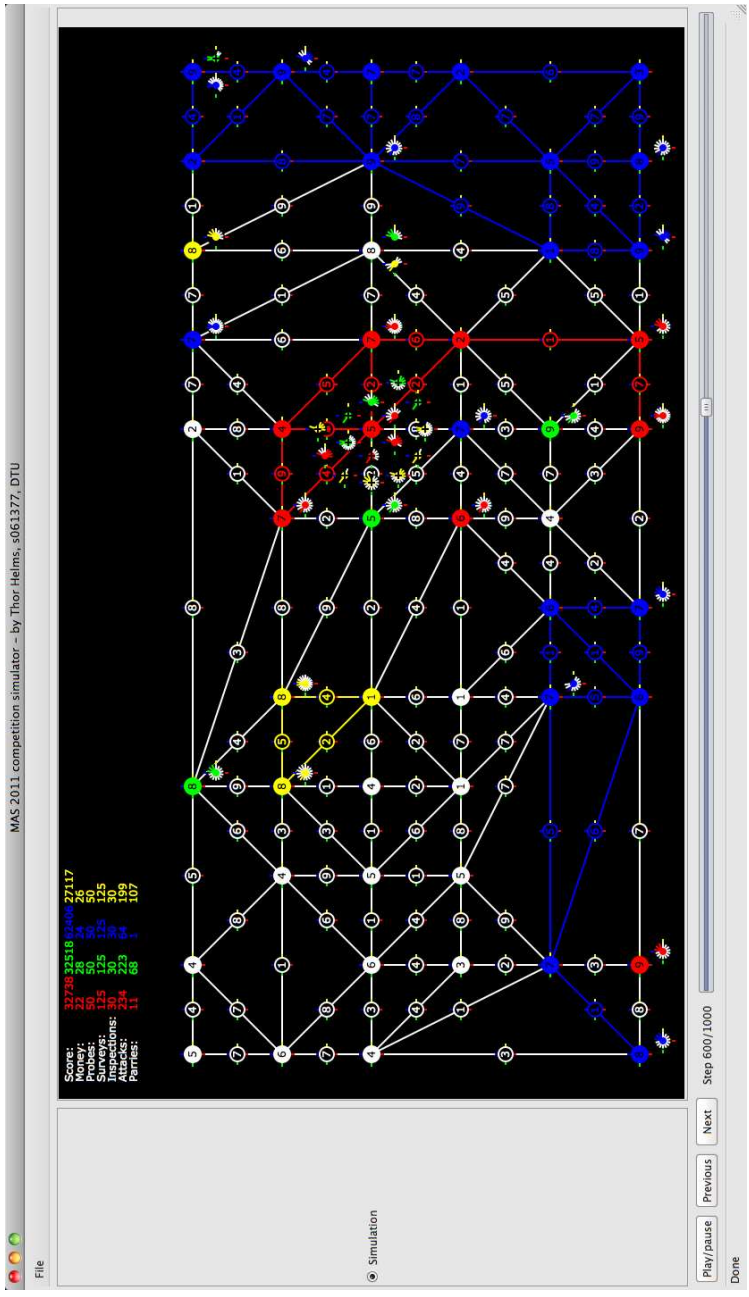

Figure A.21: Simulation 2, step 600

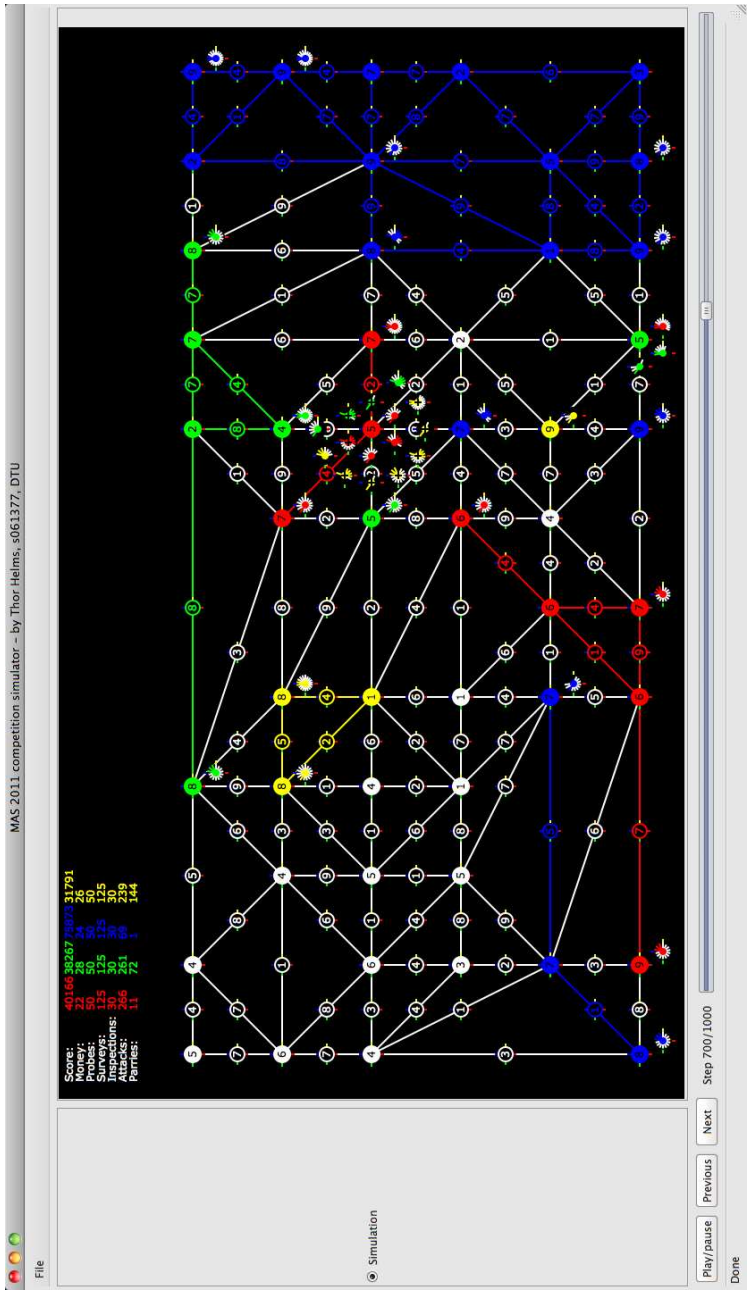

Figure A.22: Simulation 2, step 700

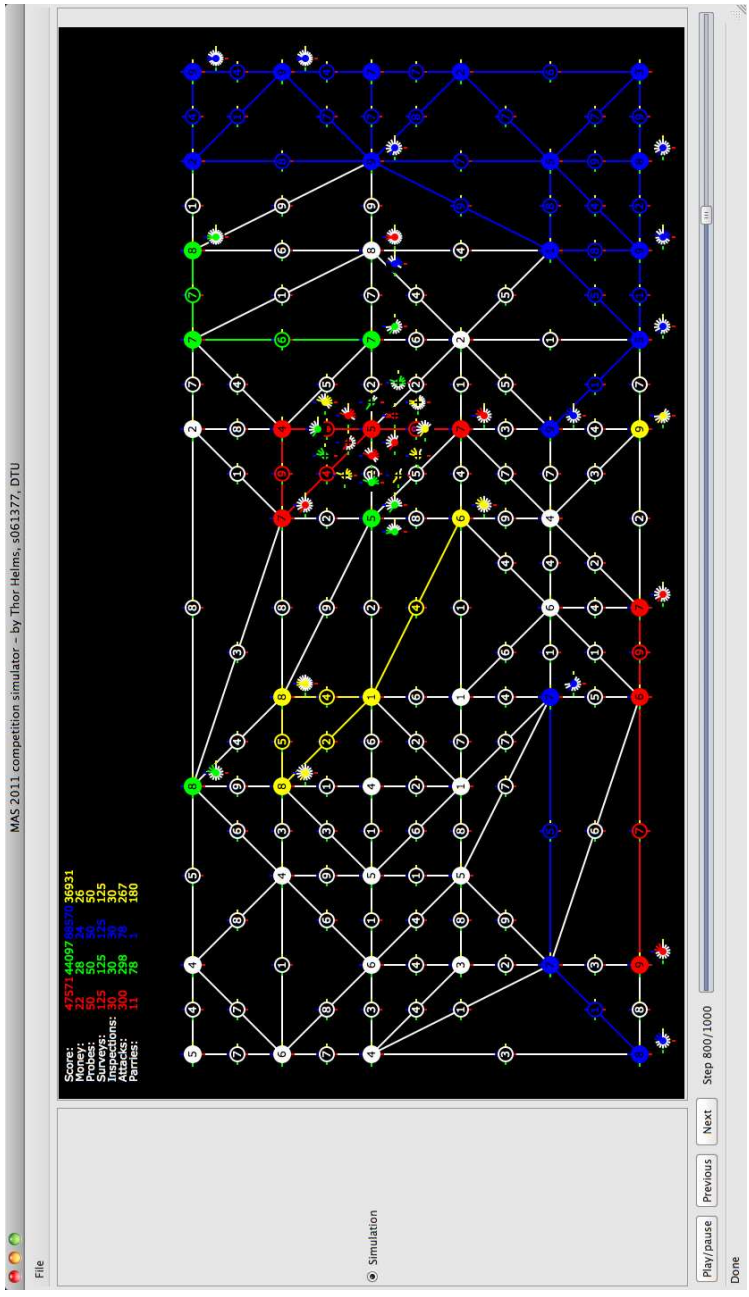

Figure A.23: Simulation 2, step 800

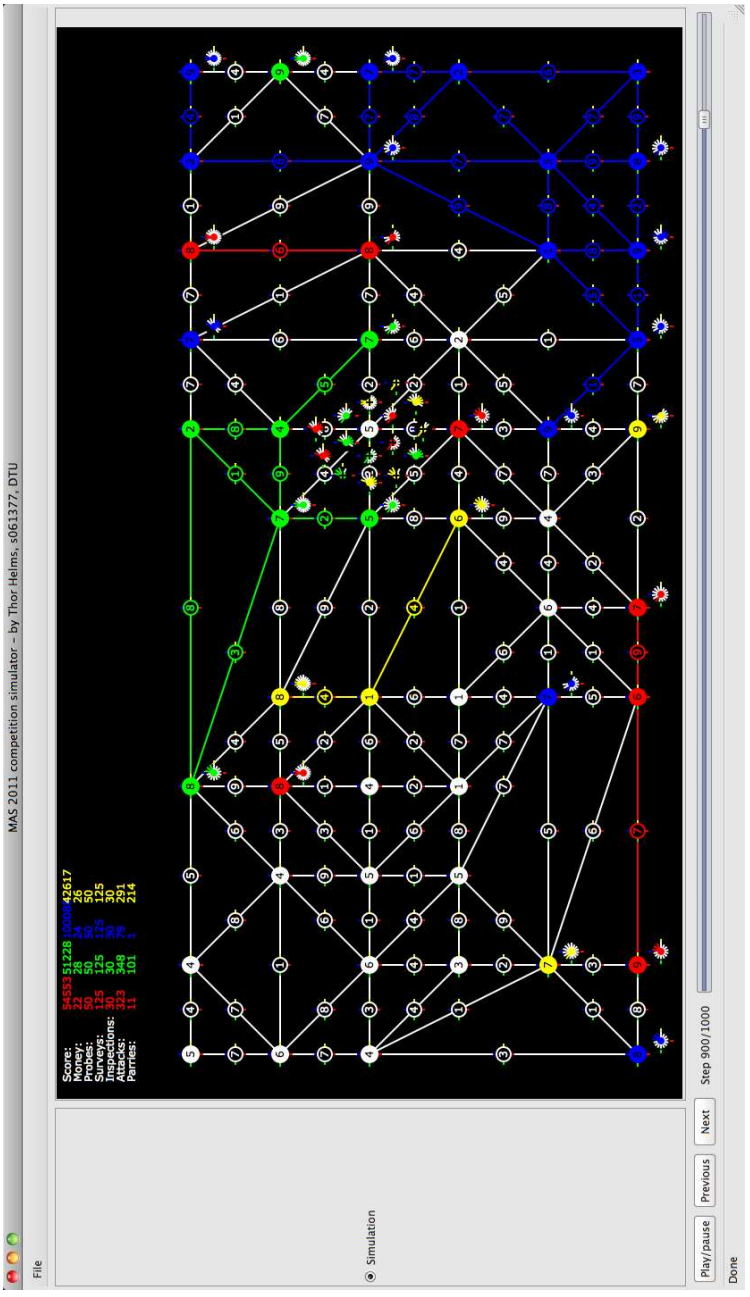

Figure A.24: Simulation 2, step 900

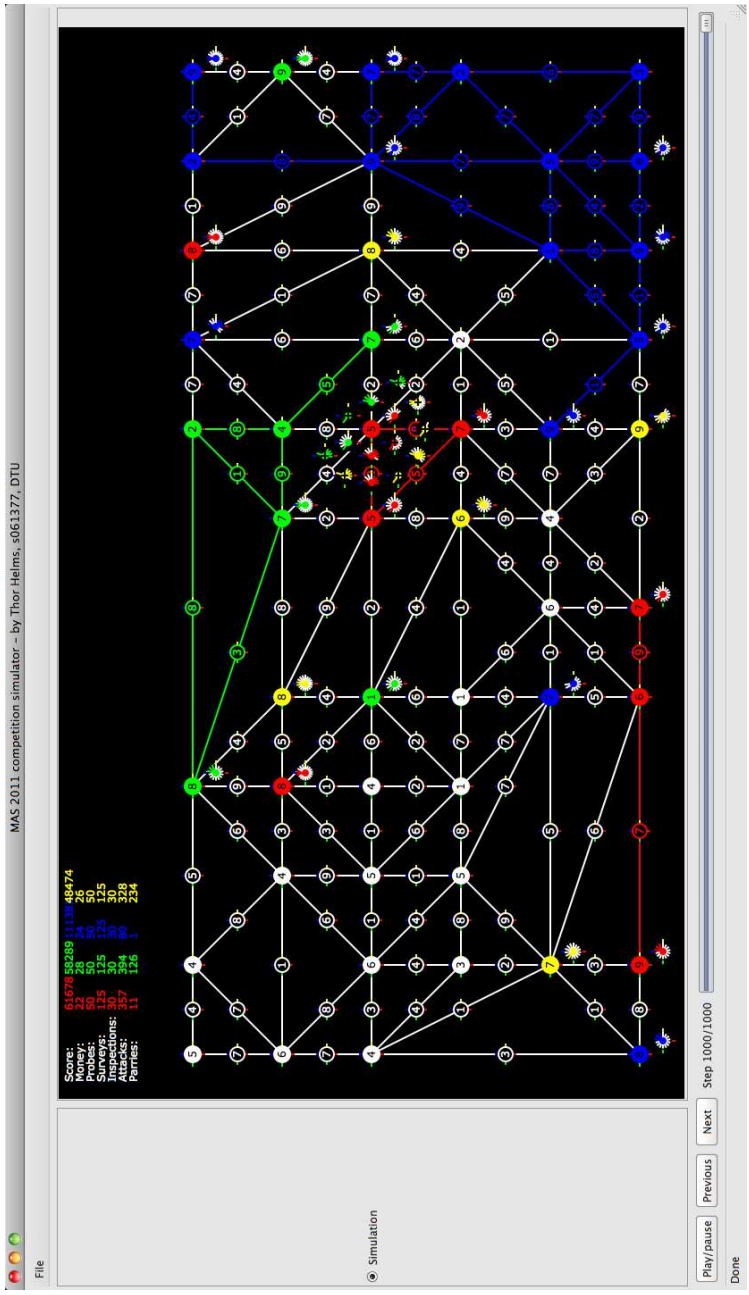

Figure A.25: Simulation 2, step 1000

Appendix B

Source code

Tekst

B.1 Makefile

```
1  # F# compiler:
2  FSC = fsc
3
4  # Targets:
5  PATH = mybin
6  EXE = Simulator
7  DLL = MAS2011
8  AIS = AISAgent
9
10 # Dependencies:
11 GTK = ../gtk-sharp-2.0/gtk-sharp.dll
12 ATK = ../gtk-sharp-2.0/atk-sharp.dll
13 GDK = ../gtk-sharp-2.0/gdk-sharp.dll
14 GLIB = ../gtk-sharp-2.0/glib-sharp.dll
15 PANGO = ../gtk-sharp-2.0/pango-sharp.dll
16 CAIRO = Mono.Cairo.dll
17 MEF = $(PATH)/System.ComponentModel.Composition.dll
18
19 # Files:
```

```
20 SIMFILES = GtkHelpers.fs DummyAgent.fs Agents.fs
       SimSettingsRoles.fs SimSettingsGeneral.fs
       SimSettingsMilestones.fs Simulation.fs SimSettingsWindow.fs
       SimulationView.fs SimulationWindow.fs Initial.fs
21 LIBFILES = Generics.fs ShapePrimitives.fs SimTypes.fs
       SimulationSteps.fs Agent.fs Edge.fs Node.fs Graph.fs IAgent.fs
       TS.fs SimulationSteps.fs SimTypesDrawing.fs
22
23 all: sim agents
24
25 sim: library
26         $(FSC) -r:$(MEF) -r:$(PATH)/$(DLL).dll -r:$(GTK) -r:$(ATK)
               -r:$(GDK) -r:$(GLIB) -r:$(PANGO) -r:$(CAIRO)
               $(SIMFILES) -o:$(PATH)/$(EXE).exe
27
28 library:
29         $(FSC) -r:$(PATH)/Triangulator.dll -r:$(GDK) $(LIBFILES)
               --target:library -o:$(PATH)/$(DLL).dll
30
31 agents: ais
32
33 ais:
34         $(FSC) AgentHelpers.fs $(AIS).fs -r:$(PATH)/$(DLL).dll
               -r:$(MEF) --target:library -o:$(PATH)/$(AIS).dll
```

B.2 Agent.fs

```
1  module Agent
2
3  (*
4  Module containing functions for manipulation of the Agent type.
5  *)
6
7  open MAS2011.Shared.SimTypes
8  open MAS2011.Shared.Generics
9
10 // Determine whether a given agent has been inspected by
11 // an agent on the given team.
12 // int * 'a -> Agent -> bool
13 let InspectedBy (team,_) agent =
14   let otherteam,_ = agent.ID
15   team = otherteam
16     || List.exists (fun (t,_) -> team = t) agent.AgentsInspected
17
18 // Add, that an agent with id has inspected the given agent
19 // Agent -> int * int -> Agent
20 let AddAgentInspected id agent =
21   if InspectedBy id agent
22   then agent
23   else
24     { agent with AgentsInspected = id::agent.AgentsInspected }
25
26 // Reduce the energy of an agent by n
27 // int -> Agent -> Agent
28 let ReduceEnergy n agent =
29   let newen = Max 0 (agent.Energy - n)
30   { agent with Energy = newen }
31
32 // Completely repair an agent
33 // Agent -> Agent
34 let RepairA (a : Agent) =
35   { a with Health = a.MaxHealth }
36
37 // Determine whether a given agent is disabled or not
38 // Agent -> bool
39 let IsDisabled (a : Agent) =
40   a.Health < 1
41
42 // Determine whether two given agents are opponents
43 let IsOpponent a1 a2 =
44   let team1,_ = a1.ID
45   let team2,_ = a2.ID
46   team1 <> team2
47
48 // Recharge an agent using the values in the given
49 // SimulationSettings
50 // SimulationSettings -> Agent -> Agent
51 let RechargeA (settings : SimulationSettings) (a : Agent) =
52   let recoveragent recover =
```

```
53      let newen =
54        (float a.MaxEnergy)*recover/100.0
55        |> int
56        |> Max 1
57        |> (+) a.Energy
58        |> Min a.MaxEnergy
59      { a with Energy = newen }
60    if (IsDisabled a)
61    then
62      recoveragent (float settings.RecoverDisabled)
63    else
64      recoveragent (float settings.RecoverNormal)
65
66  // Determine whether an agent can perform a given action.
67  // This function looks only at whether the action is in the
68  // given agents list of possible actions, and not its energy
69  // level.
70  // Agent -> Action -> bool
71  let rec CanPerform agent action =
72    let isgoto ac =
73      match ac with
74      | Goto(_) -> true
75      | _ -> false
76    let isattack ac =
77      match ac with
78      | Attack(_,_) -> true
79      | _ -> false
80    let isrepair ac =
81      match ac with
82      | Repair(_) -> true
83      | _ -> false
84    let isbuy ac =
85      match ac with
86      | Buy(_) -> true
87      | _ -> false
88    let has f = List.exists f agent.Actions
89    match action with
90    | Goto(_) -> has isgoto
91    | Attack(_,_) -> has isattack
92    | Repair(_) -> has isrepair
93    | Buy(up) ->
94      match up with
95      | SabotageDevice ->
96        has isbuy && CanPerform agent (Attack(0,0))
97      | _ -> has isbuy
98    | _ -> lmem action agent.Actions
99
100 // Anonymize a given agent, depending on a given id. If the
101 // given agent has been inspected by the given id, then
102 // nothing is changed (Except the list of inspected agents).
103 // If the agent has not been inspected, all of its values
104 // are hidden.
105 // int * 'a -> Agent -> Agent
106 let Anonymize (team,a) agent =
107   if InspectedBy (team,a) agent
```

```
108     then { agent with AgentsInspected = [team,a] }
109     else
110       { agent with
111           Health = Min 1 agent.Health
112           MaxHealth = 1
113           Energy = 1
114           MaxEnergy = 1
115           Strength = 1
116           Visibility = 1
117           Actions = []
118           AgentsInspected = [] }
119
120 // Upgrade an agent, using a given upgrade.
121 // Upgrade -> Agent -> Agent
122 let Upgrade up a =
123   match up with
124   | Battery ->
125     { a with
126         Energy = a.Energy + 1
127         MaxEnergy = a.MaxEnergy + 1 }
128   | Sensor ->
129     { a with Visibility = a.Visibility + 1 }
130   | Shield ->
131     { a with
132         Health = a.Health + 1
133         MaxHealth = a.MaxHealth + 1 }
134   | SabotageDevice ->
135     { a with Strength = a.Strength + 1 }
136
137 // Make an Agent using a list of stats and actions, and with a
138 // given ID. Will only make an Agent when the list of stats
139 // and actions have the correct lengths.
140 // int * int -> int lis -> bool list -> Agent Option
141 let Create (id,statlist,aclist) =
142   match statlist,List.length aclist with
143   | str::en::he::vis::[],7 ->
144     let actions =
145       [Attack(0,0);
146       Parry;
147       Probe;
148       Survey;
149       Inspect;
150       Buy(Battery);
151       Repair(0)]
152       |> List.zip aclist
153       |> List.filter (fun (a,_) -> a)
154       |> List.map (fun (_,a) -> a)
155       |> List.append [Skip;Recharge;Goto(0)]
156     let agent =
157       { ID = id
158         Node = 0
159         Health = he
160         MaxHealth = he
161         Energy = en
162         MaxEnergy = en
```

```
163             Strength = str
164             Visibility = vis
165             Actions = actions
166             AgentsInspected = [] }
167         Some agent
168     | _,_ -> None
```

B.3 AgentHelpers.fs

```
1  module MAS2011.Agents.AgentHelpers
2
3  (*
4  Module containing some helper functions the AI's can use.
5  *)
6
7  open MAS2011.Shared.SimTypes
8  open MAS2011.Shared.Generics
9  open MAS2011.Shared.IAgent
10 open MAS2011.Shared.SimTypesDrawing
11 open MAS2011.Shared.SimulationSteps
12 open MAS2011.Shared.ShapePrimitives
13 open System.ComponentModel.Composition
14 open System
15
16 // Retrieve the amount of uninspected enemy agents from a given
17 // node. It will count using the node and its neighbours.
18 // Graph -> Node -> Agent -> float
19 let NumUninspected ((nodes,_) as graph) node agent =
20   if Agent.CanPerform agent Inspect
21     && not (Agent.IsDisabled agent)
22   then
23     Graph.NeighbourNodes graph node.NodeID
24       |> List.append [node.NodeID]
25       |> List.map (fun n -> nodes.[n].Agents)
26       |> List.concat
27       |> List.filter
28          (fun a -> not (Agent.InspectedBy agent.ID a))
29       |> List.length
30       |> float
31   else 0.0
32
33 // Get the amount of unsurveyed edges connected to a given node.
34 // Graph -> Node -> Agent -> float
35 let NumUnsurveyed ((_,edges) : Graph) node agent =
36   if Agent.CanPerform agent Survey
37     && not (Agent.IsDisabled agent)
38   then
39     edges.[node.NodeID]
40       |> Array.toList
41       |> List.choose (fun a -> a)
42       |> List.filter (fun e -> not (Edge.SurveyedBy agent.ID e))
43       |> List.length
44       |> float
45   else 0.0
46
47 // Determine whether a given node has been probed by a given
48 // agent or not. Will only ever return true if the agent can
49 // actually perform the probe-action and isn't disabled.
50 // Agent -> Node -> bool
51 let IsUnprobed agent node =
52   not (Node.ProbedBy agent.ID node)
```

```
53      && Agent.CanPerform agent Probe
54      && not (Agent.IsDisabled agent)
55
56  // Helper function to determine whether one agent can attack
57  // another. Will only return true if neither of the two agents
58  // are disabled, the attacking agent can actually attack and
59  // the two agents are opponents.
60  // Agent -> Agent -> bool
61  let internal CanAttack agent a =
62    not (Agent.IsDisabled a)
63      && not (Agent.IsDisabled agent)
64      && Agent.CanPerform agent (Attack(0,0))
65      && Agent.IsOpponent agent a
66
67  // Helper function to determine whether one agent can be
68  // attacked by another agent, using the above function.
69  // Agent -> Agent -> bool
70  let internal CanBeAttacked a b = CanAttack b a
71
72  // Determine whether a given node has a target that can be
73  // attacked by a given agent.
74  // Agent -> Node -> bool
75  let HasTarget agent node =
76    List.exists (CanAttack agent) node.Agents
77
78  // Determine whether a given node has an enemy agent that can
79  // attack a given agent, while the given agent can perform the
80  // parry action.
81  // Agent -> Node -> bool
82  let HasAttackingCanParry agent node =
83    Agent.CanPerform agent Parry
84      && List.exists (CanBeAttacked agent) node.Agents
85
86  // Determine whether a given node has an enemy agent that can
87  // attack a given agent, while the given agent can't perform the
88  // parry action.
89  // Agent -> Node -> bool
90  let HasAttackingCantParry agent node =
91    not (Agent.CanPerform agent Parry)
92      && List.exists (CanBeAttacked agent) node.Agents
93
94  // Determine whether a given node has an enemy agent, compared
95  // to a given agent.
96  // Agent -> Node -> bool
97  let HasEnemy agent node =
98    List.exists
99      (fun a ->
100     Agent.IsOpponent agent a
101     && not (Agent.IsDisabled a))
102     node.Agents
103
104 // Determine whether a given node has a friendly agent, compared
105 // to a given agent.
106 // Agent -> Node -> bool
107 let HasFriendly agent node =
```

```
108   List.exists
109     (fun a ->
110     not (Agent.IsOpponent agent a)
111     && a.ID <> agent.ID
112     && not (Agent.IsDisabled a))
113     node.Agents
114
115 // Determine whether a given node has an agent that can repair
116 // the given agent. Will only return true if the given agent is
117 // disabled, and the repairing agent is friendly but not the
118 // same agent, and can perform the repair action.
119 // Agent -> Node -> bool
120 let HasRepairingAgent agent node =
121   let CanRepair a =
122     Agent.CanPerform a (Repair(0))
123     && not (Agent.IsOpponent a agent)
124     && a.ID <> agent.ID
125   Agent.IsDisabled agent
126     && List.exists CanRepair node.Agents
127
128 // Calculate all possible moves in a graph, using a given agent
129 // and a single action. For instance, if given the Goto action,
130 // the algorithm will determine all the possible Goto-actions
131 // for the given agent.
132 // Skip and Buy actions are ignored. The Skip action can always
133 // be replaced by the Recharge action, and the (so far) created
134 // agents all ignore the Buy action, as the amount of money
135 // in each step counts towards the score.
136 // SimulationSettings -> Graph -> Agent -> Action -> Action list
137 let AllPossibleMoves settings ((nodes,edges) as g : Graph)
138     agent action =
139   let en = agent.Energy
140   match action with
141   | Recharge -> [Recharge]
142   | Goto(_) ->
143     edges.[agent.Node]
144       |> Array.toList
145       |> List.choose (fun a -> a)
146       |> List.choose
147          (fun edge ->
148          if edge.Weight > agent.Energy
149          then None
150          else Some(Goto(nodes.[edge.To].NodeID)))
151   | Attack(_,_) when
152     en >= settings.AttackCost
153     && not (Agent.IsDisabled agent) ->
154     nodes.[agent.Node].Agents
155       |> List.choose
156          (fun a ->
157          if Agent.IsOpponent a agent && not (Agent.IsDisabled a)
158          then
159            let tid,aid = a.ID
160            Some(Attack(tid,aid))
161          else None)
162   | Parry when
```

```
163       en >= settings.ParryCost
164       && not (Agent.IsDisabled agent) ->
165       let HasAttackingEnemy node =
166         node.Agents
167           |> List.exists
168              (fun a ->
169              Agent.IsOpponent a agent
170              && Agent.CanPerform a (Attack(0,0))
171              && a.Energy >= settings.AttackCost)
172       if HasAttackingEnemy nodes.[agent.Node]
173       then [Parry]
174       else []
175     | Probe when
176       en >= settings.ProbeCost
177       && not (Agent.IsDisabled agent) ->
178       if Node.ProbedBy agent.ID nodes.[agent.Node]
179       then []
180       else [Probe]
181     | Survey when
182       en >= settings.SurveyCost
183       && not (Agent.IsDisabled agent) ->
184       let AnyUnsurveyed =
185         edges.[agent.Node]
186           |> Array.toList
187           |> List.choose (fun a -> a)
188           |> List.exists
189              (fun edge -> not (Edge.SurveyedBy agent.ID edge))
190       if AnyUnsurveyed
191       then [Survey]
192       else []
193     | Inspect when
194       en >= settings.InspectCost
195       && not (Agent.IsDisabled agent) ->
196       let agents =
197         Graph.NeighbourNodes g agent.Node
198         |> List.append [agent.Node]
199         |> List.map (fun n -> nodes.[n].Agents)
200         |> List.concat
201       if List.exists
202         (fun a -> not (Agent.InspectedBy agent.ID a)) agents
203       then [Inspect]
204       else []
205     | Repair(_) when en >= settings.RepairCost ->
206       let agents =
207         nodes.[agent.Node].Agents
208         |> List.filter
209            (fun a ->
210            not (Agent.IsOpponent a agent)
211            && a.ID <> agent.ID
212            && a.Health < a.MaxHealth)
213       let getaid a =
214         let _,aid = a.ID
215         aid
216       agents
217         |> List.map (fun a -> Repair(getaid a))
```

```
218   | _ -> [] // Ignore Skip and Buy actions
219
220 // Create an unknown node with a given ID. Unknown nodes are
221 // identified by their (-1,-1) coordinate.
222 // int -> Node
223 let UnknownNode i =
224   { NodeID = i
225     Weight = 0
226     Agents = []
227     AgentsProbed = []
228     Coordinate = (-1,-1) // To identify an unknown node
229     DominatingTeam = None }
230
231 // Update a graph with the information from a given node. Will
232 // return a Node option, depending on whether the given node
233 // holds any relevant new information or not. If so, this
234 // information should be shared with all friendly agents, but to
235 // keep the information flow on a minimum, only certain new
236 // information will be shared.
237 // Has sideeffects.
238 // Graph -> Node -> Node option
239 let UpdateGraphWithNode ((nodes,_) : Graph) node =
240   let oldnode = nodes.[node.NodeID]
241   let ignoreupdate =
242     oldnode.AgentsProbed <> [] && node.AgentsProbed = []
243   if not ignoreupdate
244   then nodes.[node.NodeID] <- node
245   if oldnode.Coordinate = (-1,-1)
246     || (oldnode.AgentsProbed = [] && node.AgentsProbed <> [])
247   then Some node
248   else None
249
250 // Update a graph with the information from a given edge. Will
251 // return an EdgeInfo option, depending on whether the given
252 // edge holds any relevant new information or not, for the same
253 // reasons as above.
254 // Has sideeffects.
255 // Graph -> EdgeInfo -> EdgeInfo option
256 let UpdateGraphWithEdge ((_,edges) as g : Graph) edge =
257   let oldedge = edges.[edge.From].[edge.To]
258   let ignoreupdate =
259     match oldedge with
260     | Some e ->
261       e.AgentsSurveyed <> [] && edge.AgentsSurveyed = []
262     | _ -> false
263   if not ignoreupdate
264   then
265     let otheredge =
266       { edge with
267           From = edge.To
268           To = edge.From}
269     Graph.ReplaceEdge g edge
270     Graph.ReplaceEdge g otheredge
271   match oldedge with
272   | None -> Some edge
```

```
273     | Some e when
274       e.AgentsSurveyed = []
275       && edge.AgentsSurveyed <> [] ->
276       Some edge
277     | _ -> None
278
279   // Update a graph with an agent. Will remove the agent from all
280   // other nodes, to eliminate obsolete information.
281   // Has sideeffects.
282   // Graph -> Agent -> unit
283   let UpdateGraphWithAgent ((nodes,_) as g : Graph) agent =
284     nodes
285       |> Array.iteri
286           (fun i n -> nodes.[i] <- Node.RemoveAgent n agent)
287     Graph.AddAgent g agent.Node agent
288
289   // Default simulation settings for agents.
290   // SimulationSettings
291   let StandardSettings =
292     { FailChance = 1
293       Length = 10000
294       MaxAgentResponse = 100
295       RecoverNormal = 1
296       RecoverDisabled = 1
297       AttackCost = 1
298       ParryCost = 1
299       ProbeCost = 1
300       SurveyCost = 1
301       InspectCost = 1
302       BuyCost = 1
303       RepairCost = 1
304       FailedGotoCost = 1
305       UpgradeBatteryPrice = 1
306       UpgradeSensorPrice = 1
307       UpgradeShieldPrice = 1
308       UpgradeSabotageDevicePrice = 1 }
309
310   // Convert an action to string format.
311   // Action -> string
312   let ActionToString action =
313     match action with
314     | Skip -> "Skip"
315     | Recharge -> "Recharge"
316     | Goto(n) -> sprintf "Goto %d" n
317     | Attack(t,a) -> sprintf "Attack (%d,%d)" t a
318     | Parry -> "Parry"
319     | Probe -> "Probe"
320     | Survey -> "Survey"
321     | Inspect -> "Inspect"
322     | Repair(n) -> sprintf "Repair %d" n
323     | Buy(_) -> "Upgrade something"
324
325   // Determine whether an agent lacks energy. Will return true
326   // if the agent would be unable to perform the probe, inspect,
327   // attack, survey, repair or parry actions, or if the agent
```

```
328 // doesn't have enough energy for a failed goto-action.
329 // SimulationSettings -> Agent -> bool
330 let LacksEnergy settings agent =
331   let en = agent.Energy
332   en < settings.AttackCost
333     || en < settings.ProbeCost
334     || en < settings.InspectCost
335     || en < settings.SurveyCost
336     || en < settings.RepairCost
337     || en < settings.ParryCost
338     || en < settings.FailedGotoCost
```

B.4 Agents.fs

```
module MAS2011.Agents

(*
Module for loading and creating agent objects.
Loads agents via MEF, the agents must implement the IAgent
interface and define a name by using the IAgentMetadata
*)

open MAS2011.Shared.IAgent
open MAS2011.DummyAgent
open System.ComponentModel.Composition
open System.ComponentModel.Composition.Hosting

// Class that MEF can use to load agents
type internal AgentHolder () =
  [<ImportMany>]
  let agents : ExportFactory<IAgent,IAgentMetadata> [] = [||]

  member __.GetAgent s =
    let find (ec : ExportFactory<IAgent,IAgentMetadata>) =
      ec.Metadata.Name = s
    match Array.tryFind find agents with
    | Some ec -> ec.CreateExport().Value
    | None -> new DummyAgent() :> IAgent

  member __.GetAllNames () =
    let agentnames =
      agents
      |> Array.map (fun ec -> ec.Metadata.Name)
      |> Array.toList
    "Dummy agent"::agentnames

// Set up MEF, load agents etc.
let internal catalog = new AggregateCatalog()
let internal directoryCatalog =
  new DirectoryCatalog(@"./","*.dll")
let internal container = new CompositionContainer(catalog)
catalog.Catalogs.Add(directoryCatalog)
let internal agentholder = new AgentHolder()
container.ComposeParts(agentholder)

// Get the correct agent for a given string
// Will return a DummyAgent when the string is unknown
let GetAgent s =
  agentholder.GetAgent s

// Gets the names of all agents loaded
let GetAllNames () =
  agentholder.GetAllNames ()
```

B.5 AISAgent.fs

```
1  module MAS2011.Agents.AISAgent
2
3  (*
4  AIS agent: Aggressive Information-Seeker
5
6  An agent with prioritized actions. At the beginning of a
7  simulation, this agent will seek out information and possible
8  attack enemy agents. When all information has been gathered, it
9  attempts to stay near friendly agents, but not on the same node
10 as friendly or enemy agents. When an agent has been disabled, it
11 should seek out a repairing agent.
12
13 The priority for actions are as follows:
14 1. Recharge if energy is too low to perform one of the other
15    actions
16 2. Probe
17 3. Survey
18 4. Inspect
19 5. Attack
20 6. Parry
21 7. Repair
22 8. Find the optimal node position - either move towards it, or
23    recharge
24
25 *)
26
27 open MAS2011.Shared.SimTypes
28 open MAS2011.Shared.Generics
29 open MAS2011.Shared.IAgent
30 open MAS2011.Shared.SimTypesDrawing
31 open MAS2011.Shared.SimulationSteps
32 open MAS2011.Shared.ShapePrimitives
33 open MAS2011.Agents.AgentHelpers
34 open System.ComponentModel.Composition
35 open System
36
37 // Weights used for finding the optimal node
38 let UninspectedWeight = 10000.0
39 let UnsurveyedWeight = 12000.0
40 let UnprobedWeight = 50000.0
41 let AttackTargetWeight = 8000.0
42 let AttackingParryWeight = -1000.0
43 let AttackingNoParryWeight = -10000.0
44 let EnemyWeight = -5000.0
45 let FriendlyWeight = 2000.0
46 let RepaireeWeight = 10000.0
47 let NodeWeight = 500.0
48
49 // Precalculated list of delay factors.
50 // The delay factor will enable the agents to prioritize nearby
51 // nodes.
52 // Float array
```

```
53  let DelayFactor =
54    Array.init 1000 (fun n -> pow 0.99 n)
55
56  // Retrieve the estimated node value for a given node in a given
57  // graph.
58  // SimulationSettings -> Graph -> Agent -> Node -> float
59  let NodeValue settings ((nodes,_) as graph : Graph) agent node =
60    if Agent.IsDisabled agent
61    then
62      if HasRepairingAgent agent node
63      then RepaireeWeight
64      else 0.0
65    else
66      let tempval = ref 0.0
67      let add n = tempval := !tempval + n
68      add ((NumUninspected graph node agent)*UninspectedWeight)
69      add ((NumUnsurveyed graph node agent)*UnsurveyedWeight)
70      if IsUnprobed agent node
71      then add UnprobedWeight
72      if HasTarget agent node
73      then add AttackTargetWeight
74      if HasAttackingCanParry agent node
75      then add AttackingParryWeight
76      if HasAttackingCantParry agent node
77      then add AttackingNoParryWeight
78      if HasEnemy agent node
79      then add EnemyWeight
80      if HasFriendly agent node
81      then add (-5.0*FriendlyWeight)
82      add (node.Weight |> float |> (*) NodeWeight)
83      let numfriendly l =
84        l |> List.map (fun n -> nodes.[n])
85          |> List.filter (HasFriendly agent)
86          |> List.length
87          |> float
88      let neighbours = Graph.NeighbourNodes graph agent.Node
89      neighbours
90        |> numfriendly
91        |> (*) FriendlyWeight
92        |> add
93      neighbours
94        |> List.map (Graph.NeighbourNodes graph)
95        |> List.concat
96        |> List.filter
97           (fun n -> n <> agent.Node && not (lmem n neighbours))
98        |> numfriendly
99        |> (*) (5.0*FriendlyWeight)
100       |> add
101     !tempval
102
103 // Agent class
104 [<ExportMetadata("Name", "Aggressive Information-Seeker")>]
105 [<Export(typeof<IAgent>)>]
106 type AISAgent () =
107   let mutable graph : Graph = ([||],[||])
```

```
108   let mutable settings = StandardSettings
109   let mutable lastnodevalues = [||]
110
111   // Algorithm from:
112   // http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm
113   // Will find all shortest paths in the graph. Runtime of
114   // O(V^3)
115   // Agent -> int array array * int array array
116   member __.FloydWarshall agent =
117     let nodes,edges = graph
118     let agent2 =
119       Agent.RechargeA settings { agent with Energy = 0 }
120     let enperstep = agent2.Energy
121     let initialweight i j =
122       if i = j then 0
123       else
124         match edges.[i].[j] with
125         | Some edge when edge.Weight <= agent.MaxEnergy ->
126           edge.Weight / enperstep + 1
127         | _ -> 1000000
128     let numnodes = Array.length nodes
129     let path =
130       Array.init numnodes
131         (fun i ->
132         Array.init numnodes
133           (fun j -> initialweight i j))
134     let next =
135       Array.init numnodes
136         (fun _ -> Array.init numnodes (fun _ -> -1))
137     let maxn = numnodes - 1
138     for k in 0..maxn do
139       for i in 0..maxn do
140         for j in 0..maxn do
141           let newpath = path.[i].[k] + path.[k].[j]
142           if newpath < path.[i].[j]
143           then
144             path.[i].[j] <- newpath
145             next.[i].[j] <- k
146     (path,next)
147
148   // Draw the graph as the agent sees/remembers it. Will also
149   // draw the last estimated values for all nodes (which is only
150   // updated when the agent has to move, in priority 8 as above)
151   // Action -> Agent -> int -> unit
152   member __.DrawGraph action agent stepnum =
153     let team,_ = agent.ID
154     let colors =
155       List.init 10 (fun i -> if i = team then Green else Red)
156     let (nodes,edges) = graph
157     let (cx,cy) =
158       nodes.[agent.Node].Coordinate
159       |> ScaleInts scaling
160     let ac = action |> ActionToString
161     let (nodes,_) = graph
162     let makenodevaluetext nodeid value =
```

```
163         let x,y = nodes.[nodeid].Coordinate |> ScaleInts scaling
164         let text = value |> sprintf "%d: %.1f" nodeid
165         Text((x-10.0,y-20.0),text,White)
166       let nodevalues =
167         lastnodevalues
168         |> Array.toList
169         |> List.mapi makenodevaluetext
170       let shapes =
171         GraphToShapes scaling colors graph
172         |> List.append [Circle((cx,cy),15.0,Filled(Yellow))]
173         |> List.append nodevalues
174         |> List.map
175           (DisplaceShape (scaling/2.0,scaling/2.0+100.0))
176         |> List.append [Text((30.0,30.0),ac,White)]
177       let actions =
178         [Attack(0,0),"A";Probe,"P";Inspect,"I";Survey,"S";
179         Repair(0),"R"]
180         |> List.fold
181           (fun olds (a,s) ->
182           if Agent.CanPerform agent a then olds + s else olds) ""
183       let name = sprintf "AIS agent %A %s" agent.ID actions
184       AddSimStep name (stepnum-1) ((1500,1500),shapes)
185
186     // Retrieve the prioritezed action from a list of possible
187     // actions.
188     // Uses the priority mentioned in the top of this file.
189     // Agent -> Action list -> Action
190     member this.PrioritizedAction agent actions =
191       let IsAttack a =
192         match a with
193         | Attack(_,_) -> true
194         | _ -> false
195       let IsRepair a =
196         match a with
197         | Repair(_) -> true
198         | _ -> false
199       if LacksEnergy settings agent
200       then Recharge
201       else if lmem Probe actions
202       then Probe
203       else if lmem Survey actions
204       then Survey
205       else if lmem Inspect actions
206       then Inspect
207       else if List.exists IsAttack actions
208       then List.find IsAttack actions
209       else if lmem Parry actions
210       then Parry
211       else if List.exists IsRepair actions
212       then List.find IsRepair actions
213       else this.NextNode agent actions
214
215     // Retrieve the wanted next node. It does so by calculating
216     // the estimated value for all nodes, applying a delay factor,
217     // and selecting the maximum value.
```

```
218   // Agent -> Action list -> Action
219   member this.NextNode agent actions =
220     let (nodes,edges) = graph
221     let nodevalues = nodes |> Array.map (fun _ -> -1000000.0)
222     let (path,next) = this.FloydWarshall agent
223     let UpdateNode node =
224       let tt = path.[agent.Node].[node.NodeID]
225       if tt < 1000
226       then
227         nodevalues.[node.NodeID] <-
228          (NodeValue settings graph agent node)*DelayFactor.[tt]
229     Array.iter UpdateNode nodes
230     lastnodevalues <- nodevalues
231     let targetnode =
232       nodevalues
233       |> Array.toList
234       |> List.mapi (fun i v -> (i,v))
235       |> List.sortBy (fun (_,v) -> -v)
236       |> List.head
237       |> fun (i,_) -> i
238     let rec getnext target =
239       if next.[agent.Node].[target] = -1
240       then target
241       else getnext next.[agent.Node].[target]
242     let nextnode = getnext targetnode
243     if targetnode = agent.Node
244       || not (lmem (Goto(nextnode)) actions)
245     then Recharge
246     else Goto(nextnode)
247
248   interface IAgent with
249     // Algorithm for updating the information the agent has and
250     // calculating the action the agent should perform.
251     // Perception -> Action
252     member this.MakeStep perc =
253       let nodemsgs =
254         perc.Messages
255         |> List.choose
256            (fun (_,msg) ->
257            match msg with
258            | InfoNode n -> Some n
259            | _ -> None)
260       let edgemsgs =
261         perc.Messages
262         |> List.choose
263            (fun (_,msg) ->
264            match msg with
265            | InfoEdge e -> Some e
266            | _ -> None)
267       List.map (UpdateGraphWithNode graph) nodemsgs |> ignore
268       List.map (UpdateGraphWithEdge graph) edgemsgs |> ignore
269       Graph.ResetColoring graph // old coloring irrelevent
270       let newnodes =
271         List.choose (UpdateGraphWithNode graph) perc.Nodes
272         |> List.map (fun n -> Broadcast(InfoNode(n)))
```

```
273         let newedges =
274           List.choose (UpdateGraphWithEdge graph) perc.Edges
275           |> List.map (fun e -> Broadcast(InfoEdge(e)))
276         perc.OtherAgents |> List.iter (UpdateGraphWithAgent graph)
277         let actions =
278           perc.Agent.Actions
279           |> List.map (AllPossibleMoves settings graph perc.Agent)
280           |> List.concat
281         let action = this.PrioritizedAction perc.Agent actions
282         this.DrawGraph action perc.Agent perc.StepNum
283         (action,newnodes @ newedges)
284
285       // SimulationSettings -> unit
286       member __.AddSettings s =
287         settings <- s
288
289       // This function is exptected to be called before the first
290       // simulation step is executed. It will set up the agent's
291       // internal structures.
292       // int * int * int -> unit
293       member __.SetNumNodesEdgesSteps (n,e,s) =
294         let nodes =
295           Array.init n (fun i -> UnknownNode i)
296         let edges =
297           Array.init n (fun _ -> Array.init n (fun _ -> None))
298         graph <- (nodes,edges)
```

B.6 DummyAgent.fs

```
1  module MAS2011.DummyAgent
2
3  (*
4  Dummy agent used by the simulation. This is the default agent.
5  When asked for a move, the agent will randomly go to a
6  neighbour node with 50% probability, and recharge with 50%
7  probability.
8  *)
9
10 open MAS2011.Shared.IAgent
11 open MAS2011.Shared.SimTypes
12 open MAS2011.Shared.Generics
13
14 type DummyAgent () =
15   interface IAgent with
16     member __.MakeStep perc =
17       let mypos = perc.Agent.Node
18       let neighbours1 =
19         perc.Edges
20         |> List.filter (fun x -> x.From = mypos)
21         |> List.map (fun x -> x.To)
22       let neighbours2 =
23         perc.Edges
24         |> List.filter (fun x -> x.To = mypos)
25         |> List.map (fun x -> x.From)
26       let neighbours =
27         neighbours1 @ neighbours2
28         |> RandomizeList
29       match neighbours with
30       | hd::_ ->
31         let action =
32           [Probe;Survey;Inspect;Recharge;Goto(hd)]
33           |> List.filter (Agent.CanPerform perc.Agent)
34           |> RandomizeList
35           |> List.head
36         (action,[])
37       | _ -> (Recharge,[])
38
39     member __.AddSettings _ = ()
40     member __.SetNumNodesEdgesSteps (_,_,_) = ()
```

B.7 Edge.fs

```
1  module Edge
2
3  (*
4  Module containing functions for manipulation of the EdgeInfo
5  type.
6  *)
7
8  open MAS2011.Shared.SimTypes
9  open MAS2011.Shared.Generics
10
11 // Add that an agent with given id has surveyed the edge
12 // EdgeInfo -> int * int -> EdgeInfo
13 let AddAgentSurveyed edge id =
14   if lmem id edge.AgentsSurveyed
15   then edge
16   else { edge with AgentsSurveyed = id::edge.AgentsSurveyed }
17
18 // Remove the dominating team.
19 // EdgeInfo -> EdgeInfo
20 let RemoveDominatingTeam edge : Edge =
21   match edge with
22   | Some e -> Some { e with DominatingTeam = None }
23   | None -> None
24
25 // Color the edge. The edge will be colored if the two nodes
26 // it is connecting, are both colored with the same color.
27 // The resulting color of the edge will be the same as the two
28 // nodes it connects.
29 // Graph -> Edge -> Edge
30 let Color ((nodes,_) : Graph) (edge : Edge) =
31   match edge with
32   | Some e ->
33     let domfrom = nodes.[e.From].DominatingTeam
34     let domto = nodes.[e.To].DominatingTeam
35     if domfrom = domto
36     then Some { e with DominatingTeam = domfrom }
37     else edge
38   | _ -> edge
39
40 // Determine whether a given edge has been surveyed by an agent
41 // on a given team.
42 // int * 'a -> EdgeInfo -> bool
43 let SurveyedBy (team,_) edge =
44   List.exists (fun (t,_) -> team = t) edge.AgentsSurveyed
45
46 // Anonymize a given edge, depending on whether an agent on a
47 // given team has surveyed the edge. If not, its weight will
48 // be set to 1.
49 // int * int -> EdgeInfo -> EdgeInfo
50 let Anonymize (team,a) edge =
51   if SurveyedBy (team,a) edge
52   then { edge with AgentsSurveyed = [team,a] }
```

```
53      else
54        { edge with
55            Weight = 1
56            AgentsSurveyed = [] }
57
58  // Make an EdgeInfo with a certain weight, between two given
59  // nodes.
60  // int -> int -> int -> EdgeInfo
61  let Create (w,f,t) =
62    { Weight = w
63      From = f
64      To = t
65      AgentsSurveyed = []
66      DominatingTeam = None }
```

B.8 Generics.fs

```
module MAS2011.Shared.Generics

(*
General functions used in the project.
*)

open System

// Randomizes the list l
// 'a list -> 'a list
let RandomizeList l =
  let r = new Random(DateTime.Now.Ticks |> int)
  l |> List.map (fun a -> (a, r.Next()))
    |> List.sortBy (fun (_,a) -> a)
    |> List.map (fun (a,_) -> a)

// Converts an F# list to a generic list
// 'a list -> List<'a>
let ToGenericList (l : _ list) =
  Collections.Generic.List(l)

// Converts a generic list to an F# list
// Seq<'a> -> 'a list
let OfGenericList l =
  List.ofSeq(l)

// Scales two integers by a given float value
// float -> int * int -> float * float
let ScaleInts s (i1,i2) = (float i1) * s, (float i2) * s

// Creates n lines arranged in a circle, all going from center
// to the perimiter of a circle (with radius 1)
// int -> (float * float) list
let CreateCircularLines n =
  [ for a in 1..n do
    let nf = float n
    let af = float a
    let angle = af*2.0*Math.PI/nf
    yield (Math.Cos(angle),Math.Sin(angle)) ]

// Calculate the minimum//maximum of two given values
// Requires comparison
// 'a -> 'a -> 'a
let Min a b = if a < b then a else b
let Max a b = if a > b then a else b

// Determine whether a given element is a member of a given list
// 'a -> 'a list -> bool
let lmem a = List.exists (fun b -> a = b)

// Counts all occurences of all values in a list
// 'a list -> Map<'a,int>
```

```
53  let CountAll l =
54    let rec CountAllHelper l map =
55      match l with
56      | hd::tl ->
57        match Map.tryFind hd map with
58        | Some num -> CountAllHelper tl (map.Add(hd,num+1))
59        | _ -> CountAllHelper tl (map.Add(hd,1))
60      | _ -> map
61    CountAllHelper l Map.empty
62
63  // Splits a list in two lists, by using a given filter
64  // The same could be achieved by using List.filter twice, once
65  // for the filter, and once for the negated filter
66  // ('a -> bool) -> 'a list -> 'a list * 'a list
67  let ListSplit f l =
68    let rec ListSplitHelper f l (ret1,ret2) =
69      match l with
70      | hd::tl ->
71        if f hd
72        then ListSplitHelper f tl (hd::ret1,ret2)
73        else ListSplitHelper f tl (ret1,hd::ret2)
74      | _ -> (ret1,ret2)
75    ListSplitHelper f l ([],[])
76
77  // Replace the nth occurence in a list, with the same element
78  // after f has been applied to it
79  // int -> ('a -> 'a) -> 'a list -> 'a list
80  let ListReplace n f l =
81    if 0 <= n && n < List.length l
82    then
83      let ar = l |> List.toArray
84      ar.[n] <- f ar.[n]
85      ar |> Array.toList
86    else l
87
88  // Converts a list of characters to a string
89  // char list -> string
90  let StringFromChars cl =
91    new System.String(cl |> Array.ofList) |> string
92
93  // Split a string by a given character, into a list of strings
94  // char -> string -> string list
95  let StringToList del (s : string) =
96    let rec StringToListHelper curlist curstr remchars =
97      match remchars,curstr with
98      | [],[] -> curlist
99      | [],_ -> (StringFromChars (List.rev curstr))::curlist
100     | hd::tl,[] when hd = del -> StringToListHelper curlist [] tl
101     | hd::tl,_ when hd = del ->
102       let newlist = (StringFromChars (List.rev curstr))::curlist
103       StringToListHelper newlist [] tl
104     | hd::tl,_ -> StringToListHelper curlist (hd::curstr) tl
105   let chars = List.ofSeq s
106   List.rev (StringToListHelper [] [] chars)
107
```

```
108 // Converts a list of strings to one string, with the
109 // string 'glue' between
110 // string -> string list -> string
111 let StringGlue glue sl =
112   match sl with
113   | hd::tl ->
114     let rest = List.fold (fun s t -> s + glue + t) "" tl
115     hd + rest
116   | _ -> ""
117
118 // Tries to find the index of the first element in a list,
119 // that satisfies a given function
120 // ('a -> bool) -> 'a list -> int Option
121 let rec ListTryFindIndex f l =
122   match l with
123   | [] -> None
124   | hd::tl ->
125     if f hd
126     then Some 0
127     else
128       match ListTryFindIndex f tl with
129       | Some n -> Some (n + 1)
130       | _ -> None
131
132 // Splits a list to a list of lists, each sub-list containing
133 // 3 elements
134 // 'a list -> 'a list list
135 let rec ListDivide3 l =
136   match l with
137   | a::b::c::tl -> [a;b;c]::ListDivide3 tl
138   | _ -> []
139
140 // If the key n exists in the given map, the list l will
141 // be appended to the end of the existing list in the map.
142 // Otherwise, the list l will be added to the map with key n.
143 // Map<'a,'b list> -> 'a -> 'b list -> Map<'a,'b list>
144 let MapAppend map n l =
145   match Map.tryFind n map with
146   | Some el -> map.Add(n,List.append el l)
147   | _ -> map.Add(n,l)
148
149 // Determines the dominating element of a list, with a certain
150 // minimum value. In case of a draw, it returns None.
151 // int -> int list -> int Option
152 let Dominator min (l : int list) =
153   let counted =
154     CountAll l
155     |> Map.toList
156     |> List.sortBy (fun (_,a) -> -a)
157     |> List.filter (fun (_,a) -> a >= min)
158   match counted with
159   | (team1,num1)::(team2,num2)::_ when num1 > num2 ->
160     Some team1
161   | (team1,_)::[] -> Some team1
162   | _ -> None
```

```
163
164 // Calculate a, lifted to the power of b (a^b)
165 // float -> int -> float
166 let pow a b =
167   let rec powhelp c d =
168     match d with
169     | 0 -> 1.0
170     | 1 -> c
171     | _ -> powhelp (c*a) (d-1)
172   powhelp a b
```

B.9 Graph.fs

```
1  module Graph
2
3  (*
4  Module with functions for manipulation of the Graph type.
5  *)
6
7  open MAS2011.Shared.SimTypes
8  open MAS2011.Shared.Generics
9  open DelaunayTriangulator
10 open System
11
12 // Add a given agent to the node with a given id.
13 // Has sideeffects.
14 // Graph -> int -> Agent -> unit
15 let AddAgent ((nodes,_) : Graph) n a =
16   if Array.length nodes > n
17   then nodes.[n] <- Node.AddAgent nodes.[n] a
18
19 // Get the neighbour nodes for a given id.
20 // Graph -> int -> int list
21 let NeighbourNodes ((_,edges) : Graph) n =
22   edges.[n]
23     |> Array.choose (fun a -> a)
24     |> Array.toList
25     |> List.map (fun a -> a.To)
26
27 // Get all agents on a given graph
28 // Graph -> Agent list
29 let GetAllAgents ((nodes,_) : Graph) =
30   nodes
31     |> Array.map (fun n -> n.Agents)
32     |> Array.toList
33     |> List.concat
34
35 // Replace an agent on a graph.
36 // Has sideeffects.
37 // Graph -> Agent -> unit
38 let ReplaceAgent ((nodes,_) : Graph) (agent : Agent) =
39   let n = agent.Node
40   if Node.HasAgent nodes.[n] agent
41   then
42     nodes.[n] <- Node.ReplaceAgent nodes.[n] agent
43
44 // Replace an edge on a graph.
45 // Has sideeffects.
46 // Graph -> EdgeInfo -> unit
47 let ReplaceEdge ((_,edges) : Graph) edge =
48   edges.[edge.From].[edge.To] <- Some edge
49
50 // Move an agent in a graph, the node with a given id.
51 // If the agent doesn't have enough energy, its energy should
52 // be reduced by an amount specified in the given settings.
```

```
53  // Else, its energy should be reduced by the weight of the edge
54  // connecting the two nodes and it should be moved.
55  // Has sideeffects.
56  // SimulationSettings -> Graph -> Agent -> int -> bool
57  let MoveAgent settings ((nodes,edges) as g : Graph)
58      (agent : Agent) n =
59    let from = agent.Node
60    match edges.[from].[n] with
61    | Some e when e.Weight <= agent.Energy ->
62      nodes.[from] <- Node.RemoveAgent nodes.[from] agent
63      nodes.[n] <-
64        Agent.ReduceEnergy e.Weight agent
65        |> Node.AddAgent nodes.[n]
66      true
67    | Some e ->
68      Agent.ReduceEnergy  settings.FailedGotoCost agent
69        |> ReplaceAgent g
70      false
71    | _ -> false
72
73  // Color the dominated nodes in a graph.
74  // Has sideeffects.
75  // Graph -> unit
76  let ColorDominatedNodes ((nodes,_) : Graph) =
77    nodes
78      |> Array.iteri (fun n node -> nodes.[n] <- Node.Color node)
79
80  // Color nodes that have at least 2 neighbour nodes of the same
81  // color, using the color of the dominating neighbouring team.
82  // In case of a draw, it will not be colored.
83  // Has sideeffects.
84  // Graph -> unit
85  let ColorNeighbours ((nodes,edges) as g : Graph) =
86    let undominated =
87      nodes
88      |> Array.toList
89      |> List.filter
90         (fun n ->
91         n.DominatingTeam = None
92         && not (Node.HasActiveAgent n))
93      |> List.map (fun n -> n.NodeID)
94    let domination =
95      undominated
96      |> List.map
97         (fun n ->
98           NeighbourNodes g n
99           |> List.choose (fun i -> nodes.[i].DominatingTeam))
100     |> List.map (Dominator 2)
101   let setdomination n dom =
102     if dom <> None
103     then nodes.[n] <- { nodes.[n] with DominatingTeam = dom }
104     else ()
105   List.iter2 setdomination undominated domination
106
107 // Find a zone given a starting node number. Uses two given
```

```
108 // to determine whether to accept a node into the zone, or
109 // whether the zone is invalid if a certain node is given.
110 // The accept function must return true for nodes that are to
111 // be accepted. The fail function must return true for nodes
112 // that invalidate the zone.
113 // (Node -> bool) -> (Node -> bool) -> Graph -> int
114 //   -> int list Option
115 let GetZone accept fail ((nodes,_) as g : Graph) n =
116   let rec GetZoneHelper zonelist trylist =
117     match trylist with
118     | [] when zonelist <> [] -> Some zonelist
119     | hd::tl when accept nodes.[hd] ->
120       let notmem n = not (lmem n (zonelist @ trylist))
121       let neighbours =
122         NeighbourNodes g hd
123         |> List.filter notmem
124       GetZoneHelper (hd::zonelist) (tl @ neighbours)
125     | hd::tl when not (fail nodes.[hd]) ->
126       GetZoneHelper zonelist tl
127     | _ -> None
128   GetZoneHelper [] [n]
129
130 // Color the zones in a graph.
131 // Has sideeffects.
132 // Graph -> unit
133 let ColorZones ((nodes,edges) as g : Graph) =
134   let rec colorzone (dom,zoneoption) =
135     match zoneoption with
136     | None | Some ([]) -> ()
137     | Some (hd::tl) ->
138       nodes.[hd] <- { nodes.[hd] with DominatingTeam = dom }
139       colorzone (dom,(Some (tl)))
140   let accept col (node : Node) =
141     node.DominatingTeam = None && not (Node.HasActiveAgent node)
142   let fail col (node : Node) =
143     let dom = node.DominatingTeam
144     (dom = None && Node.HasActiveAgent node)
145       || (dom <> None && dom <> col)
146   let getuncoloredneighbours (node : Node) =
147     let dom = node.DominatingTeam
148     NeighbourNodes g node.NodeID
149       |> List.map (fun n -> nodes.[n])
150       |> List.filter (fun n -> n.DominatingTeam = None)
151       |> List.map (fun n -> (dom,n))
152   nodes
153     |> Array.toList
154     |> List.filter (fun n -> n.DominatingTeam <> None)
155     |> List.map getuncoloredneighbours
156     |> List.concat
157     |> List.map
158        (fun (col,n) ->
159        (col, GetZone (accept col) (fail col) g n.NodeID))
160     |> List.iter colorzone
161
162 // Color the edges in a graph.
```

```
163 // Has sideeffects.
164 // Graph -> unit
165 let ColorEdges ((nodes,edges) as g : Graph) =
166   edges
167     |> Array.iteri
168        (fun x ->
169        Array.iteri
170          (fun y edge ->
171          edges.[x].[y] <- Edge.Color g edge))
172
173 // Reset the coloring of a graph (Remove all coloring).
174 // Has sideeffects.
175 // Graph -> unit
176 let ResetColoring ((nodes,edges) : Graph) =
177   nodes
178     |> Array.iteri
179        (fun n node ->
180        nodes.[n] <- Node.RemoveDominatingTeam node)
181   edges
182     |> Array.iteri
183        (fun x ->
184        Array.iteri
185          (fun y edge ->
186          edges.[x].[y] <- Edge.RemoveDominatingTeam edge))
187
188 // Survey the edges of a graph, starting at a certain node.
189 // It will survey all edges connected to the given node.
190 // Returns the amount of edges surveyed.
191 // Has sideeffects.
192 // Graph -> int -> (int * int) -> int
193 let SurveyEdges ((_,edges) as g : Graph) n id =
194   let initialedges =
195     edges.[n]
196     |> Array.choose (fun a -> a)
197     |> Array.toList
198     |> List.filter (fun e -> not (Edge.SurveyedBy id e))
199   initialedges
200     |> List.choose (fun a -> edges.[a.To].[a.From])
201     |> List.append initialedges
202     |> List.map (fun e -> Edge.AddAgentSurveyed e id)
203     |> List.iter (ReplaceEdge g)
204   List.length initialedges
205
206 // Inspect the nearby agents on a graph, given a node number.
207 // It will inspect all agents on the given node and on
208 // neighbour nodes, that haven't already been inspected by
209 // any agent on the same team as the given agent id.
210 // Returns the amount of agents inspected.
211 // Has sideeffects.
212 // Graph -> int -> int * int -> int
213 let InspectAgents ((nodes,_) as g : Graph) n id =
214   let agents =
215     n::(NeighbourNodes g n)
216     |> List.map (fun x -> nodes.[x].Agents)
217     |> List.concat
```

```
218     |> List.filter (fun x -> not (Agent.InspectedBy id x))
219     |> List.map (Agent.AddAgentInspected id)
220   List.iter (ReplaceAgent g) agents
221   List.length agents
222
223 // Create an edge between two nodes in a graph, with a given
224 // weight. Will create two edges, as the graph is to be
225 // bi-directional.
226 // Has sideeffects.
227 // Graph -> int * int * int -> unit
228 let AddEdge graph (n1,n2,w) =
229   [Edge.Create(w,n1,n2); Edge.Create(w,n2,n1)]
230     |> List.iter (ReplaceEdge graph)
231
232 // Create a graph using given values for number of nodes,
233 // width and height of the graph (when drawing it), and
234 // minimum and maximum values for edges and nodes.
235 // The algorithm will make a Delaunay triangulation on the
236 // generated nodes, and use this for the edges.
237 // Delauney triangulation module from:
238 // http://www.s-hull.org/     (Phil Atkin's C# code)
239 // int * int * int * int * int * int -> Graph
240 let Create (v,w,h,minn,maxn,mine,maxe) : Graph =
241   let rand = new Random(DateTime.Now.Ticks |> int)
242   let coords =
243     [ for x in 0..w do for y in 0..h do yield (x,y) ]
244     |> RandomizeList |> Array.ofList
245   let nodes = Array.init v (fun n ->
246     let weight = rand.Next(minn,maxn)
247     Node.Create(weight,coords.[n],n))
248   let edges =
249     Array.init v (fun _ -> Array.init v (fun _ -> None))
250   let points =
251     nodes
252     |> List.ofArray
253     |> List.map (fun a -> a.Coordinate)
254     |> List.map (fun (a,b) -> new Vertex(float32 a,float32 b))
255   let angulator = new Triangulator()
256   let triangles =
257     angulator.Triangulation(ToGenericList points)
258     |> OfGenericList
259   let addtriad (t : Triad) =
260     let e1,e2,e3 =
261       rand.Next(mine,maxe),
262       rand.Next(mine,maxe),
263       rand.Next(mine,maxe)
264     [t.a,t.b,e1; t.a,t.c,e2; t.b,t.c,e3]
265       |> List.iter (AddEdge (nodes,edges))
266   triangles |> List.iter addtriad
267   (nodes,edges)
268
269 // Get the amount of edges in the graph, assuming all edges
270 // are bidirectional. A bidirectional edge will count as one
271 // edge.
272 // Graph -> int
```

```
273 let NumEdges ((_,edges) : Graph) =
274   edges
275     |> Array.toList
276     |> List.map (Array.toList)
277     |> List.concat
278     |> List.choose (fun a -> a)
279     |> List.length
280     |> fun n -> n / 2
281
282 // Given an agent and a graph, get the version of the agent
283 // the graph knows. Assuming same position of old and updated
284 // agents.
285 // Graph -> Agent -> Agent
286 let GetUpdatedAgent ((nodes,_) : Graph) agent =
287   let agents = nodes.[agent.Node].Agents
288   match List.tryFind (fun a -> a.ID = agent.ID) agents with
289   | Some a -> a
290   | _ -> agent
```

B.10 GtkHelpers.fs

```
1  module MAS2011.GtkHelpers
2
3  (*
4  A collection of helper functions for the GTK# library
5  Most of the functions are 'convenience' functions
6  *)
7
8  open Gtk
9
10 // Attach options with short names
11 let fill = AttachOptions.Fill
12 let expand = AttachOptions.Expand
13 let fillex = fill ||| expand
14
15 // Creates a new SpinButton with the 3 given values
16 // int * int * int -> SpinButton
17 let sb (a,b,c) = new SpinButton(float a, float b, float c)
18
19 // Set the value of the given SpinButton to n.
20 // Has sideeffects.
21 // SpinButton -> int -> unit
22 let sbLoadVal (s : SpinButton) n =
23   s.Value <- (float n)
24
25 // Set the selected value of a given ComboBox to n.
26 // Has sideeffects.
27 // ComboBox -> int -> unit
28 let cbLoadVal (c : ComboBox) n =
29   c.Active <- n
30
31 // Set the 'active' value of a given CheckButton according to n.
32 // Has sideeffects.
33 // CheckButton -> bool -> unit
34 let chLoadVal (c : CheckButton) b =
35   c.Active <- b
36
37 // Set the color value of a given CollorButton according to
38 // the values in cl. Cl must be of length 3.
39 // Has sideeffects.
40 // ColorButton -> byte list -> unit
41 let colLoadVal (c : ColorButton) cl =
42   match cl with
43   | r::g::b::[] ->
44     let newcol = new Gdk.Color(r,g,b)
45     c.Color <- newcol
46   | _ -> ()
47
48 // Creates a label from a given string.
49 // string -> Label
50 let l (s : string) = new Label(s)
51
52 // Gets the value from a SpinButton.
```

```
53 // SpinButton -> int
54 let spinval (s : SpinButton) = s.ValueAsInt
55
56 // Gets the 'active' value from a CheckButton.
57 // CheckButton -> bool
58 let checkval (c : CheckButton) = c.Active
59
60 // Gets the selected value from a ComboBox.
61 // BomboBox -> string
62 let comboval (c : ComboBox) = c.ActiveText
63
64 // Gets the color value from a ColorButton
65 // ColorButton -> Gdk.Color
66 let colorval (c : ColorButton) = c.Color
67
68 // Gets the color value from a ColorButton, as RGB values
69 // ColorButton -> uint16 list
70 let colorvalrgb (c : ColorButton) =
71   [ c.Color.Red;c.Color.Green;c.Color.Blue ]
```

B.11 IAgent.fs

```
module MAS2011.Shared.IAgent

(*
Interface for agents. Agents should also define a name.


Usage:

open System.ComponentModel.Composition

[<ExportMetadata("Name", "Some name for the agent here")>]
[<Export(typeof<IAgent>)>]
type SomeAgent() = ...  // class definition
  ... // Variables and functions
  interface IAgent with
    ... // Interface functions

*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.ShapePrimitives

type IAgent =
  abstract MakeStep : Percept -> (Action * Message list)
  abstract AddSettings : SimulationSettings -> unit
  abstract SetNumNodesEdgesSteps : int * int * int -> unit

type IAgentMetadata =
  abstract Name : string
```

B.12 Initial.fs

```
1  module MAS2011.Initial
2
3  (*
4  Initial file for the simulator. Creates a SimulationWindow,
5  shows it and starts GTK's run-loop.
6  *)
7
8  open Monitor.SimulationWindow
9  open Gtk
10 open MAS2011.Simulation
11
12 Application.Init()
13 let sw = new SimulationWindow()
14 sw.Destroyed.Add(fun _ -> Application.Quit())
15 sw.Destroyed.Add(fun _ -> KillSim())
16 sw.ShowAll()
17 sw.Maximize()
18 Application.Run()
```

B.13 Node.fs

```
1  module Node
2
3  (*
4  Module containing functions for manipulation of the Node type.
5  *)
6
7  open MAS2011.Shared.SimTypes
8  open MAS2011.Shared.Generics
9
10 // Determine whether a given node contains an agent with a
11 // given id.
12 // Node -> int * int -> bool
13 let HasAgent node a =
14   List.exists (fun x -> a.ID = x.ID) node.Agents
15
16 // Add a given agent to a node.
17 // Node -> Agent -> Node
18 let AddAgent node a =
19   if HasAgent node a
20   then node
21   else
22     let agent = { a with Node = node.NodeID }
23     { node with Agents = agent::node.Agents }
24
25 // Remove an agent from a node.
26 // Node -> Agent -> Node
27 let RemoveAgent n a =
28   let agents =
29     n.Agents
30     |> List.filter (fun x -> x.ID <> a.ID)
31   { n with Agents = agents }
32
33 // Replace an agent on a node. If the node doesnt contain the
34 // agent in the first place, nothing is changed.
35 // Node -> Agent -> Node
36 let ReplaceAgent n a =
37   if HasAgent n a
38   then
39     RemoveAgent n a |> fun n2 -> AddAgent n2 a
40   else n
41
42 // Add that an agent probed a given node.
43 // Node -> int * int -> Node
44 let AddAgentProbed node id =
45   if lmem id node.AgentsProbed
46   then node
47   else { node with AgentsProbed = id::node.AgentsProbed }
48
49 // Remove the dominating team from a node.
50 // Node -> Node
51 let RemoveDominatingTeam n : Node =
52   { n with DominatingTeam = None }
```

```
53
54  // Color a given node by the dominating team on it.
55  // Considers only active agents.
56  // Node -> Node
57  let Color (node : Node) =
58    let teams =
59      node.Agents
60      |> List.choose
61         (fun a ->
62         let (teamid,_) = a.ID
63         if Agent.IsDisabled a
64         then None
65         else Some teamid)
66    { node with DominatingTeam = (Dominator 1 teams) }
67
68  // Determine whether a node has any active agents on it.
69  // Node -> bool
70  let HasActiveAgent node =
71    node.Agents
72      |> List.exists (fun a -> not (Agent.IsDisabled a))
73
74  // Determine whether a node has been probed by a given team.
75  // int * 'a -> Node -> bool
76  let ProbedBy (team,_) node =
77    List.exists (fun (t,_) -> team = t) node.AgentsProbed
78
79  // Anonymize a node, according to whether it has been probed
80  // by a certain team. If not, its weight is set to 1.
81  // int * int -> Node -> Node
82  let Anonymize (team,a) node =
83    if ProbedBy (team,a) node
84    then
85      { node with
86          AgentsProbed = [team,a]
87          Agents =
88            node.Agents
89            |> List.map (Agent.Anonymize (team,a)) }
90    else
91      { node with
92          Weight = 1
93          AgentsProbed = []
94          Agents =
95            node.Agents
96            |> List.map (Agent.Anonymize (team,a)) }
97
98  // Make a Node with a certain weight, ID and coordinate.
99  // The coordinate only has meaning when drawing the graph.
100 // int -> int * int -> int -> Node
101 let Create (w,c,n) =
102   { NodeID = n
103     Weight = w
104     Agents = []
105     AgentsProbed = []
106     Coordinate = c
107     DominatingTeam = None }
```

B.14 ShapePrimitives.fs

```
module MAS2011.Shared.ShapePrimitives

(*
Shape primitives used for drawing. The simulation can draw
anything, as long as it consists of these costum types.
*)

type Point = float * float

type Color =
  | Hex of byte * byte * byte
  | Red
  | Blue
  | Black
  | Yellow
  | Green
  | White

type ColorType =
  | Stroked       of Color
  | Filled        of Color
  | StrokedFilled of Color * Color

// Shapes:
// Circle: coordinate (center) * radius * ColorType
// Rectangle: coordinate (upper left) * size * ColorType
// Line: Start coordinate * end coordinate * Color
// Text: Coordinate * text * Color
type Shape =
  | Circle    of Point * float * ColorType
  | Rectangle of Point * Point * ColorType
  | Line      of Point * Point * Color
  | Text      of Point * string * Color

// Convert a given Gdk.Color to the costrum Color-type
// Gdk.Color -> Color
let GdkColor (gc : Gdk.Color) =
  Hex (byte (gc.Red/256us),
       byte (gc.Green/256us),
       byte (gc.Blue/256us))

// Displace a shape by (dx,dy)
// float * float -> Shape -> Shape
let DisplaceShape (dx,dy) s =
  match s with
  | Circle ((x,y),r,ct) -> Circle ((x+dx,y+dy),r,ct)
  | Rectangle ((x,y),dim,ct) -> Rectangle ((x+dx,y+dy),dim,ct)
  | Line ((x1,y1),(x2,y2),c) ->
    Line ((x1+dx,y1+dy),(x2+dx,y2+dy),c)
  | Text ((x,y),s,c) -> Text ((x+dx,y+dy),s,c)
```

B.15 SimSettingsGeneral.fs

```
1  module MAS2011.Monitor.SimSettingsGeneral
2
3  (*
4  Module containing the 'general' pane of the simulation settings
5  window, in a class called SimSettingsGeneral.
6  *)
7
8  open Gtk
9  open MAS2011.GtkHelpers
10 open MAS2011.Shared.Generics
11
12 // Class SimSettingsGeneral - inherits from Gtk.ScrolledWindow
13 type SimSettingsGeneral () as this = class
14   inherit ScrolledWindow()
15
16   let x = (0,100,1)
17   let spinOptions1 =
18     [1,10,1; 1,100,1; 50,10000,50; x; 50,5000,50; x; x]
19     |> List.map sb
20   let spinOptions2 =
21     [5,1000,1; 3,100,1; 3,100,1; x; 1,100,1; x; 1,100,1]
22     |> List.map sb
23   let spinOptions3 =
24     [x; x; x; x; x; x; x; x; x; x; x; x]
25     |> List.map sb
26   let viewpercept = new CheckButton()
27
28   let contenttable = new Table(7u,6u,false)
29
30   do
31     this.AddWithViewport(contenttable)
32     let ca a n o =
33       let un = uint32 n
34       contenttable.Attach(o,a,a+1u,un,un+1u,fill,fill,4u,4u)
35
36     List.iteri (ca 1u) spinOptions1
37     List.iteri (ca 3u) spinOptions2
38     List.iteri (ca 5u) spinOptions3
39     ca 3u 7 viewpercept
40
41     ["Number of teams";
42      "Number of agents on each team";
43      "Simulation length (ticks)";
44      "Failed action chance (%)";
45      "Max response time for agents (ms)";
46      "Recoverrate, active";
47      "Recoverrate, disabled"]
48      |> List.map l |> List.iteri (ca 0u)
49     ["Number of nodes"; "Grid width"; "Grid height";
50      "Min node weight"; "Max node weight";
51      "Min edge weight"; "Max edge weight";
52      "View agent perceptions"]
```

```
53        |> List.map l |> List.iteri (ca 2u)
54      ["'Attack' cost"; "'Parry' cost"; "'Probe' cost";
55       "'Survey' cost"; "'Inspect' cost";"'Buy' cost";
56       "'Repair' cost"; "'Failed goto' cost";
57       "'Upgrade battery' price"; "'Upgrade shield' price";
58       "'Upgrade sensor' price";"'Upgrade sabotage device' price"]
59       |> List.map l |> List.iteri (ca 4u)
60
61    // Add a function to be called when the number of teams are
62    // changed. The function must be of type:
63    // int -> unit
64    // (int -> unit) -> unit
65    member __.AddTeamsChangedHandler f =
66      let n () = spinOptions1.[0].ValueAsInt
67      spinOptions1.[0].ValueChanged.Add(fun _ -> f (n()))
68
69    // Add a function to be called when the number of agents are
70    // changed. The function must be of type:
71    // int -> unit
72    // (int -> unit) -> unit
73    member __.AddAgentsChangedHandler f =
74      let n () = spinOptions1.[1].ValueAsInt
75      spinOptions1.[1].ValueChanged.Add(fun _ -> f (n()))
76
77    // Returns the information from the various boxes.
78    // unit -> int list
79    member __.GetInformation () =
80      spinOptions1 @ spinOptions2 @ spinOptions3
81      |> List.map spinval
82
83    // Convert this to a string, for saving the settings.
84    // unit -> string
85    override this.ToString () =
86      this.GetInformation() |> List.map string |> StringGlue ","
87
88    // Load settings from a string. The values are to be
89    // seperated by a comma, and there must be 26 values.
90    // string -> unit
91    member __.LoadFromString s =
92      try
93        let vals =
94          s |> StringToList ',' |> List.map int |> Array.ofList
95        if Array.length vals = 26
96        then
97          List.iter2 sbLoadVal spinOptions1
98            (vals.[..6] |> Array.toList)
99          List.iter2 sbLoadVal spinOptions2
100           (vals.[7..13] |> Array.toList)
101         List.iter2 sbLoadVal spinOptions3
102           (vals.[14..] |> Array.toList)
103       else
104         printfn "List of wrong length: %d (should be 26)"
105           (Array.length vals)
106     with
107       | :? System.FormatException ->
```

```
108             printfn "String list couldn't be cast to integers"
109         ()
110
111     // Whether the viewpercept CheckButton is active or not
112     // unit -> bool
113     member __.ViewPercept () = viewpercept.Active
114 end
```

B.16 SimSettingsMilestones.fs

```
module MAS2011.Monitor.SimSettingsMilestones

(*
Module containing the 'milestones' pane of the simulation
settings window, in the form of a class.
*)

open Gtk
open MAS2011.GtkHelpers
open MAS2011.Shared.Generics

type Milestone =
  { Type : ComboBox
    Values : SpinButton list }

// Class SimSettingsMilestones - inherits from Gtk.ScrolledWindow
type SimSettingsMilestones () as this = class
  inherit ScrolledWindow()

  let maxms = 25
  let mutable curms = 0

  let types =
    [|"Zone values";"Probed vertices";
      "Surveyed edges";"Inspected vehicles";
      "Successful attacks";"Successful parries"|]
  let makems _ =
    let x = 0,1000,1
    { Type = new ComboBox(types,Active=0)
      Values = [x; x; x] |> List.map sb }
  let Milestones = Array.init maxms makems

  let contenttable = new Table(4u,2u,true)
  let ca r n o =
    let un = uint32 n
    contenttable.Attach(o,un,un+1u,r,r+1u,fill,fill,5u,5u)

  do
    this.AddWithViewport(contenttable)

    let addbutton = new Button("Add milestone")
    let rembutton = new Button("Remove last milestone")
    [addbutton;rembutton] |> List.iteri (ca 0u)

    addbutton.Clicked.Add(fun _ -> this.AddMilestone())
    rembutton.Clicked.Add(fun _ -> this.RemoveMilestone())

    ["Milestone type";"Min. amount";"First team";"Other teams"]
    |> List.map l |> List.iteri (ca 1u)

  // Add a milestone.
  // unit -> unit
```

```
53    member __.AddMilestone () =
54      if curms < maxms
55      then
56        contenttable.Resize(uint32 (curms+3),4u)
57        let ms = Milestones.[curms]
58        let row = uint32 (curms + 2)
59        ms.Type |> ca row 0
60        ms.Values |> List.iteri (fun n -> ca row (n+1))
61        curms <- curms + 1
62        this.ShowAll()
63
64    // Remove the last milestone.
65    // unit -> unit
66    member __.RemoveMilestone () =
67      if curms > 0
68      then
69        let rem w = contenttable.Remove(w)
70        let ms = Milestones.[curms-1]
71        ms.Type |> rem
72        ms.Values |> List.iter rem
73        curms <- curms - 1
74        contenttable.Resize(uint32 (curms+2),4u)
75
76    // Get the information the user has entered.
77    // unit -> (string * int list) list
78    member __.GetInformation () =
79      let getval ms =
80        (ms.Type |> comboval, ms.Values |> List.map spinval)
81      Milestones |> Seq.take curms |> Seq.map getval
82      |> Seq.toList
83
84    // Convert the information to a string.
85    // unit -> string
86    override __.ToString () =
87      let mstostr ms =
88        let tx = ms.Type.ActiveText
89        let ss = ms.Values |> List.map spinval |> List.map string
90        tx::ss |> StringGlue ","
91      if curms = 0
92      then ";"
93      else
94        Milestones.[..(curms-1)]
95          |> Array.toList
96          |> List.map mstostr
97          |> StringGlue ";"
98
99    // Load values from a string. The different milestones
100   // must be seperated by a semicolon, and the values of a
101   // milestone must be seperated by a comma.
102   // string -> unit
103   member this.LoadFromString (s : string) =
104     let ts = types |> Array.toList
105     let getnumcb s =
106       match ListTryFindIndex (fun x -> x = s) ts with
107       | Some x -> x
```

```
108         | _ ->
109           printfn "Invalid milestone-name: %s" s
110           0
111       let addsingle n sl =
112         let vals = StringToList ',' sl |> Array.ofList
113         let ms = Milestones.[n]
114         if Array.length vals = 4
115         then
116           try
117             let vallist =
118               vals.[1..] |> Array.toList |> List.map int
119             ms.Type.Active <- getnumcb vals.[0]
120             List.iter2 sbLoadVal ms.Values vallist
121             this.AddMilestone()
122           with
123             | :? System.FormatException ->
124               printfn "Invalid integer for milestone"
125         else printfn "Wrong length list for milestones"
126       while curms > 0 do this.RemoveMilestone()
127       s |> StringToList ';' |> List.iteri addsingle
128   end
```

B.17 SimSettingsRoles.fs

```
1  module MAS2011.Monitor.SimSettingsRoles
2
3  (*
4  Module containing the 'roles' pane of the simulation settings
5  window, in the form of a class.
6  *)
7
8  open Gtk
9  open MAS2011.GtkHelpers
10 open MAS2011.Shared.Generics
11
12 type AgentRole =
13   { Name : ComboBox
14     Stats : SpinButton list
15     Actions : CheckButton list
16     Teams : ComboBox list }
17
18 // Set the values of a given AgentRole (see above), to the
19 // values given in two lists.
20 // Has sideeffects.
21 // AgentRole -> int list * bool list -> unit
22 let SetValues a (stats,actions) =
23   if List.length a.Stats = List.length stats
24      && List.length a.Actions = List.length actions
25   then
26     List.iter2 sbLoadVal a.Stats stats
27     let cbLoadVal (c : CheckButton) b =
28       c.Active <- b
29     List.iter2 cbLoadVal a.Actions actions
30
31 // Set the proper roles of a given AgentRole. The role depends
32 // on the selected name in the ComboBox.
33 // The roles are defined in the scenario description,
34 // describing the scenario to be simulated.
35 // Has sideeffects.
36 // AgentRole -> unit
37 let SetRole a =
38   let t,f = true,false
39   match a.Name.ActiveText with
40   | "Explorer"  ->
41     SetValues a ([0;12;4;2], [f;f;t;t;f;t;f])
42   | "Repairer"  ->
43     SetValues a ([0;8;6;1], [f;t;f;t;f;t;t])
44   | "Saboteur"  ->
45     SetValues a ([4;7;3;1], [t;t;f;t;f;t;f])
46   | "Sentinel"  ->
47     SetValues a ([0;10;1;3], [f;t;f;t;f;t;f])
48   | "Inspector" ->
49     SetValues a ([0;8;6;1], [f;f;f;t;t;t;f])
50   | _ -> ()
51
52 // Get the ComboBox associated with a certain team in a given
```

```
53  // AgentRole. The ComboBox represents the AI to be used.
54  // int -> AgentRole -> ComboBox
55  let getteam n a = a.Teams.[n]
56
57  // SimSettingsRoles class - inherits Gtk.ScrolledWindow
58  // Instantiated with a list of available AI's, in the form
59  // of a string list
60  type SimSettingsRoles (agentlist) as this = class
61    inherit ScrolledWindow()
62
63    let agents = List.toArray (agentlist)
64    let mutable numAgents = 0
65    let mutable numTeams = 4
66    let mutable RoleChangeDisabled = false
67
68    let ainames = agentlist
69
70    let names =
71      [|"Custom";"Explorer";"Repairer";
72        "Saboteur";"Sentinel";"Inspector"|]
73    let CreateRow n =
74      { Name = new ComboBox(names, Active=0);
75        Stats = [ for _ in 1..4 do
76                  yield sb(0,20,1) ];
77        Actions = [ for _ in 1..7 do
78                    yield new CheckButton() ];
79        Teams = [ for _ in 1..10 do
80                  yield new ComboBox(agents, Active=0) ] }
81    let AgentRoles = Array.init 100 CreateRow
82
83    let contenttable = new Table(1u,13u,false)
84    let ca r c o =
85      contenttable.Attach(o,c,c+1u,r,r+1u,fill,fill,4u,4u)
86
87    let teamcolors = List.init 10 (fun n -> new ColorButton())
88
89    do
90      this.AddWithViewport(contenttable)
91
92      let ca2 n = ca 0u (uint32 (n+1))
93      ["Strength"; "Max energy"; "Health"; "Visibility";
94       "'Attack'"; "'Parry'"; "'Probe'"; "'Survey'";
95       "'Inspect'"; "'Buy'"; "'Repair'"]
96       |> List.map l |> List.iteri ca2
97
98      teamcolors |> Seq.take numTeams
99                 |> Seq.iteri (fun n -> ca2 (n+11))
100
101     let SetHandlers a =
102       let sr _ =
103         RoleChangeDisabled <- true
104         SetRole a
105         RoleChangeDisabled <- false
106       a.Name.Changed.Add sr
107       let SetCustom _ =
```

```
108             if not RoleChangeDisabled
109             then a.Name.Active <- 0
110           a.Stats |> List.iter
111                      (fun sb -> sb.ValueChanged.Add (SetCustom))
112           a.Actions |> List.iter
113                        (fun cb -> cb.Toggled.Add (SetCustom))
114         AgentRoles |> Array.iter SetHandlers
115
116       // Display row of roles number n
117       // int -> unit
118       member internal __.AddRoleRow n =
119         let un = uint32 (n+1)
120         let ca2 d i = ca un (uint32 (i+d))
121         let role = AgentRoles.[n]
122         ca2 0 0 role.Name
123         role.Stats |> List.iteri (ca2 1)
124         role.Actions |> List.iteri (ca2 5)
125         role.Teams |> Seq.take numTeams |> Seq.iteri (ca2 12)
126
127       // Hide row of roles number n
128       // int -> unit
129       member internal __.RemoveRoleRow n =
130         let role = AgentRoles.[n]
131         let rem w = contenttable.Remove(w)
132         role.Name |> rem
133         role.Stats |> List.iter rem
134         role.Actions |> List.iter rem
135         role.Teams |> Seq.take numTeams |> Seq.iter rem
136
137       // Display column of teams number n
138       // int -> unit
139       member internal __.AddTeamColumn n =
140         let un = uint32 (n+12)
141         let ca2 d i = ca (uint32 (i+d)) un
142         ca2 0 0 teamcolors.[n]
143         AgentRoles |> Seq.map (getteam n) |> Seq.take numAgents
144                    |> Seq.iteri (ca2 1)
145
146       // Hide column of teams number n
147       // int -> unit
148       member internal __.RemoveTeamColumn n =
149         contenttable.Remove(teamcolors.[n])
150         let rem w = contenttable.Remove(w)
151         Array.map (getteam n) AgentRoles.[..numAgents-1]
152         |> Array.iter rem
153
154       // Number of agent roles updated to n
155       // int -> unit
156       member this.UpdateNumAgents n =
157         contenttable.Resize(uint32 (n+1),uint32 (numTeams+12))
158         let nOld = numAgents
159         if nOld < n // Increased number of agents
160         then [nOld..(n-1)] |> List.iter this.AddRoleRow
161         if nOld > n // Decreased number of agents
162         then [n..(nOld-1)] |> List.iter this.RemoveRoleRow
```

```
163     numAgents <- n
164     this.ShowAll()
165
166   // Number of teams updated to t
167   // int -> unit
168   member this.UpdateNumTeams t =
169     contenttable.Resize(uint32 (numAgents+1),uint32 (t+12))
170     let tOld = numTeams
171     if tOld < t // Increased number of teams
172     then [tOld..(t-1)] |> List.iter this.AddTeamColumn
173     if tOld > t // Decreased number of teams
174     then [t..(tOld-1)] |> List.iter this.RemoveTeamColumn
175     numTeams <- t
176     this.ShowAll()
177
178   // Get the information the user has entered.
179   // unit -> Gdk.Color list * (int list * bool list *
180   //                           string list) list
181   member __.GetInformation () =
182     let getval a =
183       (a.Stats |> List.map spinval,
184        a.Actions |> List.map checkval,
185        a.Teams |> Seq.take numTeams
186        |> Seq.map comboval |> Seq.toList)
187     let roles =
188       AgentRoles |> Seq.take numAgents |> Seq.map getval
189       |> Seq.toList
190     let teams =
191       teamcolors |> Seq.take numTeams |> Seq.map colorval
192       |> Seq.toList
193     (teams,roles)
194
195   // Convert the information the user has entered, to a string.
196   // Information on all roles and all teams will be generated,
197   // not just the displayed ones.
198   // unit -> string
199   override __.ToString () =
200     let btostr b =
201       if b then "true" else "false"
202     let artostr ar =
203       let ct = ar.Name.ActiveText
204       let st =
205         ar.Stats
206         |> List.map spinval
207         |> List.map string
208       let at =
209         ar.Actions
210         |> List.map checkval
211         |> List.map btostr
212       let ait =
213         ar.Teams
214         |> List.map comboval
215       ct::(st@at@ait) |> StringGlue ","
216     let colors =
217       teamcolors
```

```
218         |> List.map colorvalrgb
219         |> List.concat
220         |> List.map string
221         |> StringGlue ","
222       AgentRoles
223         |> Array.toList
224         |> List.map artostr
225         |> List.append [colors]
226         |> StringGlue ";"
227
228     // Load information from a string. The different roles must
229     // be seperated by a semicolon, and the different values
230     // must be seperated by comma.
231     // Furthermore, the first element in the semicolon-seperated
232     // list must be a list of colors. The colors must be
233     // seperated by comma, with the RGB values of each color
234     // all in the list of colors.Chars
235     // string -> unit
236     member __.LoadFromString s =
237       let strtobool s =
238         match s with
239         | "true" -> true
240         | _ -> false
241       let ns = names |> Array.toList
242       let getnumname s =
243         match ListTryFindIndex (fun x -> x = s) ns with
244         | Some x -> x
245         | _ ->
246           printfn "Invalid agent-name: %s" s
247           0
248       let getnumai s =
249         match ListTryFindIndex (fun x -> x = s) ainames with
250         | Some x -> x
251         | _ -> 0
252       let addsingle n sl =
253         let vals = StringToList ',' sl |> Array.ofList
254         let ag = AgentRoles.[n]
255         if Array.length vals = 22
256         then
257           try
258             let role = getnumname vals.[0]
259             ag.Name.Active <- role
260             let ais =
261               vals.[12..] |> Array.toList |> List.map getnumai
262             List.iter2 cbLoadVal ag.Teams ais
263             if role = 0
264             then
265               let statlist =
266                 vals.[1..4] |> Array.toList |> List.map int
267               let aclist =
268                 vals.[5..11] |> Array.toList |> List.map strtobool
269               List.iter2 sbLoadVal ag.Stats statlist
270               List.iter2 chLoadVal ag.Actions aclist
271           with
272             | :? System.FormatException ->
```

```
273             printfn "Invalid integer for agent-role"
274       else printfn "Invalid length of agent-role list"
275     let vals =
276       StringToList ';' s
277     match vals with
278     | color::roles when List.length roles = 100 ->
279       List.iteri addsingle roles
280       try
281         let colors =
282           color
283           |> StringToList ','
284           |> List.map (fun x -> (int x)/256 |> byte)
285           |> ListDivide3
286         List.iter2 colLoadVal teamcolors colors
287       with
288         | :? System.FormatException ->
289           printfn "Invalid number for color"
290     | _ -> printfn "Wrong length of roles-list"
291 end
```

B.18 SimSettingsWindow.fs

```
1  module MAS2011.Monitor.SimSettingsWindow
2
3  (*
4  Module containing the simulation settings window, in the form
5  of a class.
6  *)
7
8  open Gtk
9  open MAS2011.GtkHelpers
10 open MAS2011.Monitor.SimSettingsGeneral
11 open MAS2011.Monitor.SimSettingsRoles
12 open MAS2011.Monitor.SimSettingsMilestones
13 open MAS2011.Simulation
14 open MAS2011.Shared.Generics
15
16 // Class SimSettingsWindow - inherits Gtk.Window
17 // Instantiated with a list of available AI's, in the form
18 // of a string list
19 type SimSettingsWindow (agentlist) as this = class
20   inherit Window("Enter simulation settings")
21
22   let gen = new SimSettingsGeneral()
23   let roles = new SimSettingsRoles(agentlist)
24   let ms = new SimSettingsMilestones()
25   let contenttable = new Table(2u,6u,false)
26   let nb = new Notebook()
27
28   do
29     this.DefaultSize <- new Gdk.Size(950,550)
30     this.Add(contenttable)
31     nb.AppendPage(gen,l "General") |> ignore
32     nb.AppendPage(roles,l "Agent roles") |> ignore
33     nb.AppendPage(ms,l "Milestones") |> ignore
34
35     let okb = new Button("     OK     ")
36     let canb = new Button("Cancel")
37     let defb = new Button("  Defaults  ")
38     let openb = new Button("Open settings...")
39     let saveb = new Button("Save settings...")
40     okb.Clicked.Add(fun _ -> this.Destroy())
41     canb.Clicked.Add(fun _ -> this.Destroy())
42     defb.Clicked.Add(fun _ -> this.SetDefaultValues())
43
44     contenttable.Attach(nb,0u,6u,0u,1u,fillex,fillex,5u,5u)
45     contenttable.Attach(okb,0u,1u,1u,2u,fill,fill,5u,5u)
46     contenttable.Attach(canb,1u,2u,1u,2u,fill,fill,5u,5u)
47     contenttable.Attach(defb,2u,3u,1u,2u,fill,fill,5u,5u)
48     contenttable.Attach(openb,3u,4u,1u,2u,fill,fill,5u,5u)
49     contenttable.Attach(saveb,4u,5u,1u,2u,fill,fill,5u,5u)
50     contenttable.Attach(
51       new Statusbar(),5u,6u,1u,2u,fill,fill,5u,5u)
52
```

```
53      gen.AddTeamsChangedHandler roles.UpdateNumTeams
54      gen.AddAgentsChangedHandler roles.UpdateNumAgents
55      this.SetDefaultValues()
56
57      let startsim _ =
58        let s = gen.GetInformation()
59        let r = roles.GetInformation()
60        let m = ms.GetInformation()
61        if List.length s = 26 && s.[7] <= s.[8] * s.[9]
62          && s.[10] < s.[11] && s.[12] < s.[13]
63        then
64          StartSim s r m (gen.ViewPercept())
65        else
66          printfn "Error starting sim - following must be true:"
67          printfn "%d <= %d   (%d * %d)" s.[7] (s.[8]*s.[9])
68            s.[8] s.[9]
69          printfn "%d < %d" s.[10] s.[11]
70          printfn "%d < %d" s.[12] s.[13]
71      okb.Clicked.Add(startsim)
72
73      let openfile _ =
74        let fc =
75          new FileChooserDialog
76            ("Choose the map to open",
77            this,FileChooserAction.Open,
78            "Cancel",ResponseType.Cancel,
79            "Open",ResponseType.Accept)
80        if fc.Run() = int(ResponseType.Accept)
81        then this.OpenFile fc.Filename
82        fc.Destroy() |> ignore
83      openb.Clicked.Add(openfile)
84
85      let savefile _ =
86        let fc =
87          new FileChooserDialog
88            ("Choose the map to open",
89            this,FileChooserAction.Save,
90            "Cancel",ResponseType.Cancel,
91            "Save",ResponseType.Accept)
92        if fc.Run() = int(ResponseType.Accept)
93        then
94          let contents =
95            [ gen.ToString(); roles.ToString(); ms.ToString() ]
96            |> StringGlue "@"
97          let file = fc.Filename
98          System.IO.File.WriteAllText(file,contents)
99        fc.Destroy() |> ignore
100     saveb.Clicked.Add(savefile)
101
102     this.ShowAll()
103
104   // Open a given file, and load the simulation setting values
105   // it contains. The file must contain a list of strings, with
106   // each element seperated by @. The list must be of length 3,
107   // with each element adhering to the restrictions given in
```

```
108     // the functions SimSettingsGeneral.LoadFromString,
109     // SimSettingsMilestones.LoadFromString and
110     // SimSettingsRoles.LoadFromString.
111     // string -> unit
112     member __.OpenFile path =
113       if System.IO.File.Exists(path)
114       then
115         let contents =
116           System.IO.File.ReadAllText(path)
117           |> StringToList '@'
118         match contents with
119         | a::b::c::[] ->
120           gen.LoadFromString a
121           roles.LoadFromString b
122           ms.LoadFromString c
123         | _ -> printfn "Wrong length list, unable to open"
124       else printfn "Unable to find file: %s" path
125
126     // Loads the values in the file DefaultSettings.txt if it
127     // exists.
128     // unit -> unit
129     member this.SetDefaultValues () =
130       this.OpenFile "DefaultSettings.txt"
131 end
```

B.19 SimTypes.fs

```
1  module MAS2011.Shared.SimTypes
2
3  (*
4  Module containing the different types used in the simulation
5  calculations, along with some functions to manipulate/create
6  the different types.
7  *)
8
9  open System
10 open MAS2011.Shared.Generics
11
12 // Type of upgrade an agent wants to perform
13 type Upgrade =
14   | Battery
15   | Sensor
16   | Shield
17   | SabotageDevice
18
19 type TeamID = int
20 type AgentID = int
21 type NodeID = int
22
23 // The action an agent wants to perform
24 type Action =
25   | Skip
26   | Recharge
27   | Goto of NodeID
28   | Attack of TeamID * AgentID
29   | Parry
30   | Probe
31   | Survey
32   | Inspect
33   | Repair of AgentID
34   | Buy of Upgrade
35
36 // Stats for an agent
37 type Agent =
38   { ID : TeamID * AgentID
39     Node : NodeID
40     Health : int
41     MaxHealth : int
42     Energy : int
43     MaxEnergy : int
44     Strength : int
45     Visibility : int
46     Actions : Action list
47     AgentsInspected : (TeamID * AgentID) list }
48
49 // Node in a graph, along with some simulation-specific
50 // information
51 type Node =
52   { NodeID : int
```

```
53      Weight : int
54      Agents : Agent list
55      AgentsProbed : (TeamID * AgentID) list
56      Coordinate : int * int
57      DominatingTeam : int Option }
58
59  // Information about an edge - in a graph, edges are
60  // EdgeInfo Option's
61  type EdgeInfo =
62    { Weight : int
63      From : int
64      To : int
65      AgentsSurveyed : (TeamID * AgentID) list
66      DominatingTeam : int Option }
67  type Edge = EdgeInfo option
68
69  // Graph, with edges in an adjancency matrix
70  type Graph = Node array * Edge array array
71
72  // Information about a milestone. Format is:
73  // Amount needed for the milestone to trigger *
74  // Award for the first team(s) to reach the milestone *
75  // Award for all other teams reaching the milestone
76  type Milestone =
77    | ZoneValues of int * int * int
78    | ProbedVertices of int * int * int
79    | SurveyedEdges of int * int * int
80    | InspectedVehicles of int * int * int
81    | SuccessfulAttacks of int * int * int
82    | SuccessfulParries of int * int * int
83
84  // A milestone and a set of teams that has completed it
85  type MilestoneStatus = Milestone * Set<int>
86
87  // Status for a team, in order to keep track of score/money
88  // and status for milestones.
89  type TeamStatus =
90    { Score : int
91      Money : int
92      Probes : int
93      Surveys : int
94      Inspections : int
95      Attacks : int
96      Parries : int }
97
98  // Messages agents can send
99  type MessageContent =
100   | InfoEdge of EdgeInfo
101   | InfoNode of Node
102   | Feromone of NodeID * int
103
104 // Messages need a recipient
105 type Message =
106   | Broadcast of MessageContent
107   | Single of int * MessageContent
```

```
108
109 // Perception given to an agent when a move is requested
110 type Percept =
111   { StepNum : int
112     Score : int
113     Money : int
114     Agent : Agent
115     Nodes : Node list
116     Edges : EdgeInfo list
117     OtherAgents : Agent list
118     Milestones : MilestoneStatus list
119     Messages : (int * MessageContent) list }
120
121 // Various simulation specific settings, defined in the user
122 // interface
123 type SimulationSettings =
124   { FailChance : int
125     Length : int
126     MaxAgentResponse : int
127     RecoverNormal : int
128     RecoverDisabled : int
129     AttackCost : int
130     ParryCost : int
131     ProbeCost : int
132     SurveyCost : int
133     InspectCost : int
134     BuyCost : int
135     RepairCost : int
136     FailedGotoCost : int
137     UpgradeBatteryPrice : int
138     UpgradeSensorPrice : int
139     UpgradeShieldPrice : int
140     UpgradeSabotageDevicePrice : int }
141
142 // Parse a Milestone from a string and a list of integers.
143 // Will only parse a Milestone when the string is one of six
144 // known values, and the integer list is of length 3.
145 // string * int list -> MilestoneStatus
146 let ParseMilestone (s, il) =
147   let es : Set<int> = Set.empty
148   match il with
149   | a::b::c::[] ->
150     match s with
151     | "Zone values" ->
152       Some (ZoneValues(a,b,c),es)
153     | "Probed vertices" ->
154       Some (ProbedVertices(a,b,c),es)
155     | "Surveyed edges" ->
156       Some (SurveyedEdges(a,b,c),es)
157     | "Inspected vehicles" ->
158       Some (InspectedVehicles(a,b,c),es)
159     | "Successful attacks" ->
160       Some (SuccessfulAttacks(a,b,c),es)
161     | "Successful parries" ->
162       Some (SuccessfulParries(a,b,c),es)
```

```
163       | _ -> None
164     | _ -> None
165
166   // Get the price for a certain upgrade, from a given
167   // SimulationSettings.
168   // SimulationSettings -> Upgrade -> int
169   let GetPrice settings upgrade =
170     match upgrade with
171     | Battery -> settings.UpgradeBatteryPrice
172     | Sensor -> settings.UpgradeSensorPrice
173     | Shield -> settings.UpgradeShieldPrice
174     | SabotageDevice -> settings.UpgradeSabotageDevicePrice
```

B.20 SimTypesDrawing.fs

```
module MAS2011.Shared.SimTypesDrawing

(*
This module contains functions to convert edges, nodes, agents
and thus entire graphs to a list of Shape's. It also contains
a few functions for adding these to SimulationSteps, which
keeps track of the different steps in the simulation.
*)

open MAS2011.Shared.SimTypes
open MAS2011.Shared.ShapePrimitives
open MAS2011.Shared.Generics
open MAS2011.Shared.SimulationSteps

// Text alignment in order to be able to write to the center of
// a circle.
let internal tdx, internal tdy = -3.0,3.0

// Scaling-value, ie. distance between nodes when drawing the
// graph.
let scaling = 100.0

// Create a colored line, depending on a given function. This
// method is used to display which teams have probed/surveyed/
// inspected various nodes/edges/agents.
// float * float -> float -> int -> (int -> bool) -> int
//   -> Color -> Shape Option
let internal InfoLine (x,y) scale total f n color =
  if f n
  then
    let lines = CreateCircularLines total
    let dx,dy = lines.[n]
    Some (Line((x,y),(x+scale*dx,y+scale*dy),color))
  else None

// Convert an edge to shapes. Needs the starting and ending
// coordinates. The edge will be drawn white if it is not
// dominated by any team.
// float -> (int * int) * (int * int) -> Color list -> EdgeInfo
//   -> Shape list
let EdgeToShape scale (f,t) colors (e : EdgeInfo) =
  let x1,y1 = f |> ScaleInts scale
  let x2,y2 = t |> ScaleInts scale
  let x3,y3 = (x1+x2)/2.0, (y1+y2)/2.0
  let mutable color = White
  match e.DominatingTeam with
  | Some z when List.length colors > z ->
    color <- colors.[z]
  | _ -> ()
  let line = Line((x1,y1),(x2,y2),color)
  let circle = Circle((x3,y3),8.0,StrokedFilled(color,Black))
  let text = Text((x3+tdx,y3+tdy),(string e.Weight),color)
```

```
53    let f n = Edge.SurveyedBy (n,0) e
54    let infolines =
55      List.mapi (InfoLine (x3,y3) 12.0 (List.length colors) f)
56        colors
57      |> List.choose (fun a -> a)
58    line::infolines @ [circle;text]
59
60  // Convert an agent to a Shape list. Needs the coordinate of
61  // the agent. Some lines will be drawn from the center of the
62  // agent and to the perimeter of its drawing area, displaying
63  // the agents stats.
64  // Strength is short, colored lines.
65  // Energy is short, white lines.
66  // Visibility is long, colored lines.
67  // Health is long, white lines.
68  // float * float -> Color list -> Agent -> Shape list
69  let AgentToShape (x,y) (colors : Color list) (a : Agent) =
70    let maxe,en = a.MaxEnergy,a.Energy
71    let maxh,he = a.MaxHealth,a.Health
72    let str,vis = a.Strength,a.Visibility
73    let team,_ = a.ID
74    let makeline n (dx,dy) =
75      let coord = (x+10.0*dx,y+10.0*dy)
76      let coord2 = (x+8.5*dx,y+8.5*dy)
77      // Draw strength lines
78      if n < str
79      then Some (Line((x,y),coord2,colors.[team]))
80      // Draw energy lines
81      else if n < str + en
82      then Some (Line((x,y),coord2,White))
83      // Don't draw (max energy - energy) lines
84      else if n < str + maxe
85      then None
86      // Draw visibility lines
87      else if n < str + maxe + vis
88      then Some (Line((x,y),coord,colors.[team]))
89      // Draw health lines
90      else if n < str + maxe + vis + he
91      then Some (Line((x,y),coord,White))
92      // Don't draw (max health - health) lines
93      else None
94    let lines =
95      CreateCircularLines (maxe+maxh+str+vis)
96      |> List.mapi makeline
97      |> List.choose (fun a -> a)
98    let circle = Circle((x,y),4.0,Filled(colors.[team]))
99    let bgcircle = Circle((x,y),10.0,Filled(Black))
100   let f n =
101     if n = team then false
102     else Agent.InspectedBy (n,0) a
103   let infolines =
104     List.mapi (InfoLine (x,y) 14.0 (List.length colors) f)
105       colors
106     |> List.choose (fun a -> a)
107   let disabledcross =
```

```
108     if Agent.IsDisabled a
109     then
110       [Line((x-4.0,y),(x+4.0,y),Black);
111         Line((x,y-4.0),(x,y+4.0),Black)]
112     else []
113   infolines @ [bgcircle] @ lines @ [circle] @ disabledcross
114
115 // Array of agent positions, relative to a node. A maximum
116 // amount of agents on each node is expected, and if this
117 // maximum is reached the application could crash. The maximum
118 // is about 90 agents per node.
119 // (float * float) array
120 let internal AgentRelPos =
121   let makecircle n =
122     CreateCircularLines (n*6)
123     |> List.map (fun (x,y) ->
124       let nf = (float n)*30.0
125       (x*nf,y*nf))
126     |> Array.ofList
127   [| for a in 1..5 do yield makecircle a |]
128   |> Array.concat
129
130 // Convert a node to a Shape list. Will draw agents on the node
131 // in a circular fashion around it self.
132 // float -> Color list -> Node -> Shape list
133 let NodeToShape scale colors (node : Node) =
134   let x,y = node.Coordinate |> ScaleInts scale
135   let mutable color = White
136   match node.DominatingTeam with
137   | Some z when List.length colors > z ->
138     color <- colors.[z]
139   | _ -> ()
140   let circle = Circle((x,y),10.0,Filled(color))
141   let text = Text((x+tdx,y+tdy),(string node.Weight),Black)
142   let agentpos i =
143     let xc,yc = AgentRelPos.[i]
144     (xc+x,yc+y)
145   let agents =
146     node.Agents
147     |> List.mapi
148       (fun i -> AgentToShape (agentpos i) colors)
149     |> List.concat
150   let f n = Node.ProbedBy (n,0) node
151   let infolines =
152     List.mapi (InfoLine (x,y) 14.0 (List.length colors) f)
153       colors
154     |> List.choose (fun a -> a)
155   infolines @ [circle;text] @ agents
156
157 // Convert a graph to a Shape list, by converting all nodes
158 // and edges to shapes.
159 // float -> Color list -> Graph -> Shape list
160 let GraphToShapes scale colors (g : Graph) =
161   let nodes,edges = g
162   let getcoords (e : EdgeInfo) =
```

```
163       (nodes.[e.From].Coordinate,nodes.[e.To].Coordinate)
164     let edgeshapes =
165       edges.[1..]
166       |> Array.mapi (fun i a -> a.[0..i])
167       |> Array.concat
168       |> Array.choose (fun a -> a)
169       |> Array.map
170          (fun e -> EdgeToShape scale (getcoords e) colors e)
171       |> List.ofArray
172       |> List.concat
173     let nodeshapes =
174       nodes
175       |> Array.map (NodeToShape scale colors)
176       |> List.ofArray
177       |> List.concat
178     edgeshapes @ nodeshapes
179
180   // Convert an agent's perception to a Shape list by converting
181   // all visible nodes and edges to shape's.
182   // float -> Color list -> Percept -> Shape list
183   let PerceptToShapes scale colors (p : Percept) =
184     let getcoord n =
185       match List.tryFind (fun x -> x.NodeID = n) p.Nodes with
186       | Some x -> x.Coordinate
187       | _ -> (0,0)
188     let getcoords (e : EdgeInfo) =
189       (getcoord e.From,getcoord e.To)
190     let nodes =
191       p.Nodes
192       |> List.map (NodeToShape scale colors)
193       |> List.concat
194     let edges =
195       p.Edges
196       |> List.map
197          (fun e ->EdgeToShape scale (getcoords e) colors e)
198       |> List.concat
199     edges @ nodes
200
201   // Draw a graph by converting it to a Shape list and adding
202   // it to the SimulationSteps module. Will also draw the
203   // TeamStatus for all teams.
204   // Has sideeffects.
205   // Color list -> int * int -> Graph -> TeamStatus list -> int
206   //   -> unit
207   let DrawGraph (colors : Color list) (w,h) graph status stepnum =
208     let strtoshapes dy color x n s =
209       let y = dy + n*12 |> float
210       Text((float x,y),s,color)
211     let strstoshapes a b c = List.mapi (strtoshapes a b c)
212     let text =
213       ["Score:";"Money:";"Probes:";"Surveys:";"Inspections:";
214        "Attacks:";"Parries:"]
215        |> strstoshapes 15 White 20
216     let scorestext =
217       status
```

```
218       |> List.map TS.ToList
219       |> List.map (List.map string)
220       |> List.mapi (fun n -> strstoshapes 15 colors.[n] (n*40+100))
221       |> List.concat
222     let shapes =
223       GraphToShapes scaling colors graph
224       |> List.map (DisplaceShape (scaling/2.0,scaling/2.0+100.0))
225     let finalshapes = [text;shapes;scorestext] |> List.concat
226     let sc = scaling |> int
227     AddSimStep "Simulation" stepnum ((w*sc,h*sc+100),finalshapes)
228
229 // Draw an agents perception by converting it to a Shape list
230 // and ading it to the SimulationSteps module. Its name depends
231 // on the agent's ID.
232 // Has sideeffects.
233 // Color list -> int * int -> int -> Percept -> unit
234 let DrawPercept (colors : Color list) (w,h) n p =
235     let shapes =
236       PerceptToShapes scaling colors p
237       |> List.map (DisplaceShape (scaling/2.0,scaling/2.0+100.0))
238     let team,agent = p.Agent.ID
239     let name = sprintf "Team %d, agent %d" team agent
240     let sc = scaling |> int
241     AddSimStep name (n-1) ((w*sc,h*sc+100),shapes)
```

B.21 Simulation.fs

```
1  module MAS2011.Simulation
2
3  (*
4  This module contains the simulation algorithm, along with a lot
5  of helper functions for it. It also contains a function to
6  start a simulation calculation, which will kill a running
7  simulation if one exists, and start a new thread with a new
8  simulation.
9  *)
10
11 open System
12 open System.Threading
13 open MAS2011.Shared.SimTypes
14 open MAS2011.Shared.SimTypesDrawing
15 open MAS2011.Shared.IAgent
16 open MAS2011.Shared.Generics
17 open MAS2011.Shared.ShapePrimitives
18 open MAS2011.Shared.SimulationSteps
19
20 // Simulation thread.
21 // System.Threading.Thread
22 let mutable internal thread =
23   new Thread(new ThreadStart(fun _ -> ()))
24
25 // Abort the running simulation. Used when loading saved
26 // simulations.
27 // Has sideeffects.
28 // unit -> unit
29 let KillSim () =
30   thread.Abort()
31
32 // Expand the view from a list of nodes, IE. add all the
33 // neighbour nodes to the list. Do this recursively until n is
34 // 1. When n is 1, retrieve the nodes and the edges connecting
35 // the found nodes.
36 // Graph -> int -> int list -> Node list * EdgeInfo list
37 let rec internal ExpandView ((nodes,edges) as g : Graph) n
38     nodelist =
39   if n < 1
40   then
41     let getedges i =
42       edges.[i]
43       |> Array.toList
44       |> List.choose (fun a -> a)
45       |> List.filter (fun e -> lmem e.To nodelist)
46     let newnodes =
47       nodelist
48       |> List.map (fun i -> nodes.[i])
49     let newedges =
50       nodelist
51       |> List.map (fun i -> getedges i)
52       |> List.concat
```

```
53      (newnodes,newedges)
54    else
55      let newnodelist =
56        nodelist
57        |> List.map (Graph.NeighbourNodes g)
58        |> List.concat
59        |> List.append nodelist
60        |> Set.ofList
61        |> Set.toList
62      ExpandView g (n-1) newnodelist
63
64  // Share the perception between agents, based on the zone they
65  // are in.
66  // int list -> int -> Node list * EdgeInfo list -> Agent
67  //   -> Node list * EdgeInfo list -> Node list * EdgeInfo list
68  let internal SharePercept zonelist team (nodeset,edgeset) agent
69      (nodes : Node list,edges : EdgeInfo list) =
70    let t,_ = agent.ID
71    if t = team && List.exists (fun x -> x = agent.Node) zonelist
72    then
73      let newnodeset =
74        nodes
75        |> Set.ofList
76        |> (+) (nodeset |> Set.ofList)
77        |> Set.toList
78      let newedgeset =
79        edges
80        |> Set.ofList
81        |> (+) (edgeset |> Set.ofList)
82        |> Set.toList
83      (newnodeset,newedgeset)
84    else (nodeset,edgeset)
85
86  // Prepare/create the perceptions for all agents.
87  // int -> Graph -> (Agent * Message list) -> TeamStatus list
88  //   -> Percept list
89  let internal PreparePercepts stepnum (g : Graph) ml
90      (status : TeamStatus list) milestones =
91    let getscore (tid,_) = status.[tid].Score
92    let getmoney (tid,_) = status.[tid].Money
93    let allagents =
94      Graph.GetAllAgents g
95      |> List.sortBy (fun a -> a.ID)
96      // Sorted to get the corrrect order of agents in the GUI
97    let getmsgmap map (agent,msgs) =
98      let team,_ = agent.ID
99      msgs
100       |> List.map (fun a -> agent,a)
101       |> MapAppend map team
102   let messages =
103     ml |> List.fold getmsgmap Map.empty
104   let getmsg (team,agent) =
105     match Map.tryFind team messages with
106     | Some msgs ->
107       msgs
```

```
108           |> List.choose
109           (fun (a,msg) ->
110            let _,sender = a.ID
111            match msg with
112            | Broadcast m -> Some(sender,m)
113            | Single (r,m) when r = agent -> Some(sender,m)
114            | _ -> None)
115       | _ -> []
116     let getagents (team,agent) =
117       allagents |> List.filter (Agent.InspectedBy (team,agent))
118     let vis (a : Agent) = Max 1 a.Visibility
119     let visibilities =
120       allagents
121       |> List.map (fun a -> ExpandView g (vis a) [a.Node])
122     let team a =
123       let (t,_) = a.ID
124       t
125     let accept agent (node : Node) =
126       node.DominatingTeam = Some (team agent)
127     let fail _ = false
128     let zones =
129       allagents
130       |> List.map
131          (fun a -> Graph.GetZone (accept a) fail g a.Node)
132       |> List.map
133          (fun zone ->
134          match zone with
135          | Some z -> z
136          | _ -> [])
137     let sharedvis =
138       List.zip3 allagents visibilities zones
139         |> List.map
140            (fun (a,v,z) ->
141            List.fold2 (SharePercept z (team a)) v
142              allagents visibilities)
143     let makepercept (a : Agent) (nodes,edges) =
144       { StepNum = stepnum
145         Score = getscore a.ID
146         Money = getmoney a.ID
147         Agent = a
148         Nodes = nodes |> List.map (Node.Anonymize a.ID)
149         Edges = edges |> List.map (Edge.Anonymize a.ID)
150         OtherAgents = getagents a.ID
151         Milestones = milestones
152         Messages = getmsg a.ID }
153     List.map2 makepercept allagents sharedvis
154
155   // Fail actions by a certain percentage chance.
156   // SimulationSettings -> 'a list -> 'a list
157   let internal FailActions settings actions =
158     let r = new Random(DateTime.Now.Ticks |> int)
159     let addrandom v =
160       (r.Next(100),v)
161     actions
162       |> List.map addrandom
```

```
163     |> List.filter (fun (a,_) -> a >= settings.FailChance)
164     |> List.map (fun (_,a) -> a)
165
166 // Send perceptions to agents, and retrieve their wanted actions
167 // and messages they wish to send.
168 // SimulationSettings -> Map<int * int,IAgent> -> Percept list
169 //   -> (Agent * Action) list * (Agent * Message list) list
170 let internal SendPercepts settings (agents : Map<_,IAgent>)
171     percepts =
172   let answers =
173     percepts
174     |> List.map
175        (fun p -> (p.Agent,agents.[p.Agent.ID].MakeStep(p)))
176   let msgs =
177     answers
178     |> List.map (fun (agent,(_,msgs)) -> (agent,msgs))
179   let actions =
180     answers
181     |> List.map (fun (a,(b,_)) -> (a,b))
182     |> FailActions settings
183   (actions,msgs)
184
185 // Filter for determining whether a given action is either
186 // attack or parry, or neither.
187 // 'a * Action -> bool
188 let internal AttackParryFilter (_,action) =
189   match action with
190     | Parry | Attack(_,_) -> true
191     | _ -> false
192
193 // Determine whether a given agent and the id of another agent,
194 // are opponents.
195 // Agent -> int * 'a -> bool
196 let internal IsOpponent a (tid,_) =
197   let (tid2,_) = a.ID
198   tid <> tid2
199
200 // Get a list of the agents who has the energy to perform the
201 // parry action, can do it and isn't disabled. The agents
202 // energy will be redued before returning them.
203 // SimulationSettings -> (Agent * Action) list -> Agent list
204 let internal Parries settings actions =
205   actions
206   |> List.filter
207      (fun (a,ac) ->
208      a.Energy >= settings.ParryCost
209      && ac = Parry
210      && not (Agent.IsDisabled a)
211      && Agent.CanPerform a ac)
212   |> List.map (fun (a,_) -> a)
213   |> List.map (Agent.ReduceEnergy settings.ParryCost)
214
215 // Execute all attack and parry actions. In reality, only attack
216 // actions will be executed, depending on whether or not the
217 // target is parrying. A successful attacks of course also
```

```
218 // depends on the energy, health and position of the
219 // attacking agent, and whether or not it can perform the
220 // attack action in the first place.
221 // Has sideeffects.
222 // SimulationSettings -> TeamStatus list
223 let internal ExecuteAttackParry settings status
224     ((nodes,_) as g : Graph) actions =
225   let parries = Parries settings actions
226   let ExecuteAttackHelper (status,attacked) (agent,action)
227       (tid,aid) =
228     let isparrying = // Is target parrying?
229       List.exists (fun a -> a.ID = (tid,aid)) parries
230     let target = // Find target agent in graph
231       nodes.[agent.Node].Agents
232       |> List.tryFind (fun a -> a.ID = (tid,aid))
233     let newattacker = // Reduce energy for attacker
234       Agent.ReduceEnergy settings.AttackCost agent
235     match target,isparrying with
236       | Some targetagent,false when
237         not (Agent.IsDisabled targetagent) ->
238         // Successful attack
239         let newhealth =
240           Max 0 (targetagent.Health - agent.Strength)
241         let newag = { targetagent with Health = newhealth }
242         Graph.ReplaceAgent g newag
243         Graph.ReplaceAgent g newattacker
244         let teamid,_ = agent.ID
245         let newstat = ListReplace teamid TS.IncAttacks status
246         (newstat,(tid,aid)::attacked)
247       | Some targetagent,true -> // Successful parry
248         Graph.ReplaceAgent g newattacker
249         (ListReplace tid TS.IncParries status,attacked)
250       | _ -> // Unable to find target agent in graph
251         (status,attacked)
252   let ExecuteAttack (status,attacked) (agent,action) =
253     match action with
254     | Attack(tid,aid) when
255       not (Agent.IsDisabled agent)
256       && IsOpponent agent (tid,aid)
257       && agent.Energy >= settings.AttackCost
258       && Agent.CanPerform agent (Attack(0,0)) ->
259         ExecuteAttackHelper (status,attacked) (agent,action)
260           (tid,aid)
261     | _ -> (status,attacked) // Unsuccessful attack
262   // Reduce energy for all parrying agents
263   List.iter (Graph.ReplaceAgent g) parries
264   List.fold ExecuteAttack (status,[]) actions
265
266 // Execute single action other than attack and parry.
267 // Has sideeffects.
268 // SimulationSettings -> Graph -> (int * int) list
269 //   -> TeamStatus list -> Agent * Action -> TeamStatus list
270 let internal ExecuteOtherAction settings
271     ((nodes,edges) as g : Graph) attacked status (a,action) =
272   let agent = Graph.GetUpdatedAgent g a
```

```
273     if Agent.CanPerform agent action
274     then
275       let (teamid,agentid) = agent.ID
276       let isattacked =
277         List.exists (fun id -> id = agent.ID) attacked
278       let disabled = Agent.IsDisabled agent
279       let reduce n =
280         let newa = Graph.GetUpdatedAgent g a
281         Agent.ReduceEnergy n newa |> Graph.ReplaceAgent g
282       match action,isattacked,disabled with
283       // Recharge - agent hasn't been attacked
284       | Recharge,false,_ ->
285         Agent.RechargeA settings agent
286           |> Graph.ReplaceAgent g
287         status
288       // Move agent
289       | Goto(n),_,_ when
290         agent.Energy >= settings.FailedGotoCost
291         && 0 <= n && n < Array.length nodes ->
292         Graph.MoveAgent settings g agent n |> ignore
293         status
294       // Probe node - agents hasn't been attacked and isn't
295       // disabled
296       | Probe,false,false when
297         agent.Energy >= settings.ProbeCost
298         && not (Node.ProbedBy agent.ID nodes.[agent.Node]) ->
299         let node = nodes.[agent.Node]
300         nodes.[agent.Node] <- Node.AddAgentProbed node agent.ID
301         reduce settings.ProbeCost
302         ListReplace teamid TS.IncProbes status
303       // Survey edges - agent hasn't been attacked and isn't
304       // disabled
305       | Survey,false,false when
306         agent.Energy >= settings.SurveyCost ->
307         reduce settings.SurveyCost
308         let num = Graph.SurveyEdges g agent.Node agent.ID
309         ListReplace teamid (TS.IncSurveys num) status
310       // Inspect an opponent agent - agent hasn't been attacked
311       // and isn't disabled
312       | Inspect,false,false when
313         agent.Energy >= settings.InspectCost ->
314         reduce settings.InspectCost
315         let num = Graph.InspectAgents g agent.Node agent.ID
316         ListReplace teamid (TS.IncInspections num) status
317       // Repair a friendly agent - agent hasn't been attacked
318       | Repair(aid),false,_ when
319         aid <> agentid && agent.Energy >= settings.RepairCost ->
320         let target =
321           List.tryFind
322             (fun a -> a.ID = (teamid,aid))
323             nodes.[agent.Node].Agents
324         match target with
325         | Some a ->
326           Agent.RepairA a |> Graph.ReplaceAgent g
327           reduce settings.RepairCost
```

```
328         | _ -> ()
329         status
330       // Upgrade - agent hasn't been attacked and isn't
331       // disabled
332       | Buy(up),false,false when
333         agent.Energy >= settings.BuyCost ->
334         let money = status.[teamid].Money
335         let price = GetPrice settings up
336         if money >= price
337         then
338           agent
339             |> Agent.ReduceEnergy settings.BuyCost
340             |> Agent.Upgrade up
341             |> Graph.ReplaceAgent g
342           ListReplace teamid (TS.ReduceMoney price) status
343         else status
344       | _ -> status
345     else status
346
347 // Execute all actions other than parry and attack
348 // Has sideeffects
349 // SimulationSettings -> TeamStatus list -> Graph
350 //   -> (int * int) list -> (Agent * Action) list
351 //   -> TeamStatus list
352 let internal ExecuteOtherActions settings status graph attacked
353       actions =
354     List.fold (ExecuteOtherAction settings graph attacked) status
355       actions
356
357 // Color a graph using the grap coloring algorithm.
358 // Has sideeffects.
359 // Graph -> unit
360 let internal ColorGraph graph =
361   Graph.ResetColoring graph
362   Graph.ColorDominatedNodes graph
363   Graph.ColorNeighbours graph
364   Graph.ColorZones graph
365   Graph.ColorEdges graph
366
367 // Update a single milestone and team status'
368 // int lis -> TeamStatus list * MilestoneStatus
369 //   -> TeamStatus list * MilestoneStatus
370 let internal UpdateMilestone zonescores
371       (status,(milestone,compset)) =
372     let teamlist =
373       List.zip zonescores status
374       |> List.mapi (fun n (a,b) -> (n,a,b))
375     let f (team,score,ts) =
376       match milestone with
377       | ZoneValues(_,_,_) -> score
378       | ProbedVertices(_,_,_) -> ts.Probes
379       | SurveyedEdges(_,_,_) -> ts.Surveys
380       | InspectedVehicles(_,_,_) -> ts.Inspections
381       | SuccessfulAttacks(_,_,_) -> ts.Attacks
382       | SuccessfulParries(_,_,_) -> ts.Parries
```

```
383   match milestone with
384   | ZoneValues(num,fst,scn)
385   | ProbedVertices(num,fst,scn)
386   | SurveyedEdges(num,fst,scn)
387   | InspectedVehicles(num,fst,scn)
388   | SuccessfulAttacks(num,fst,scn)
389   | SuccessfulParries(num,fst,scn) ->
390     let completed =
391       teamlist
392       |> List.choose
393          (fun ((team,_,_) as a) ->
394          if f a >= num then Some team else None)
395       |> Set.ofList
396     let newcomp = completed - compset
397     let amount = if compset.IsEmpty then fst else scn
398     let inc (ts : TeamStatus list) n =
399       ListReplace n (TS.IncreaseMoney amount) ts
400     let newstatus =
401       newcomp
402       |> Set.toList
403       |> List.fold inc status
404     (newstatus,(milestone,compset + completed))
405
406 // Update all milestones and team status'
407 // Graph -> MilestoneStatus list -> TeamStatus list
408 //   ->  MilestoneStatus list * TeamStatus list
409 let internal UpdateMilestonesAndStatus ((nodes,_) : Graph)
410     milestones status =
411   let getweight n node =
412     if Node.ProbedBy (n,0) node
413     then node.Weight
414     else 1
415   let getstepscore n =
416     nodes
417     |> Array.toList
418     |> List.filter (fun node -> node.DominatingTeam = Some n)
419     |> List.map (getweight n)
420     |> List.sum
421   let zonescores =
422     [for a in 0..(List.length status - 1) do yield getstepscore a]
423   let tempstatus = ref status
424   let newmilestones =
425     milestones
426     |> List.map
427        (fun ms ->
428        let newts,newms =
429          UpdateMilestone zonescores (!tempstatus,ms)
430        tempstatus := newts
431        newms)
432   let stepscores =
433     zonescores
434     |> List.map2
435        (fun (st : TeamStatus) score -> score + st.Money)
436          !tempstatus
437   let newstatus =
```

```
438     !tempstatus
439     |> List.map2 TS.IncreaseScore stepscores
440   (newmilestones,newstatus)
441
442 // Main algorithm.
443 // Calculate a single simulation step. Do this recursively
444 // until all steps have been calculated.
445 // Has sideeffects.
446 // SimulationSettings
447 //   -> (Graph -> TeamStatus list -> int -> unit)
448 //   -> (int -> Percept -> unit)
449 //   -> bool
450 //   -> Map<int*int,IAgent>
451 //   -> int
452 //   -> Graph
453 //   -> (Agent * Message list) list
454 //   -> MilestoneStatus list
455 //   -> TeamStatus list
456 //   -> unit
457 let rec internal SimulationStep settings drawgraph drawpercept
458     viewp agents stepnum graph messages milestones status =
459   if stepnum > settings.Length
460   then
461     SendMsg "Done" // simulation endeth
462   else
463     SendMsg
464       (sprintf "Calculating step %d of %d" stepnum
465       settings.Length)
466     let percepts =
467       PreparePercepts stepnum graph messages status milestones
468     let (actions,msgs) =
469       SendPercepts settings agents percepts
470     let (apactions,restactions) =
471       ListSplit AttackParryFilter actions
472     let (status2,attacked) =
473       ExecuteAttackParry settings status graph apactions
474     let status3 =
475       ExecuteOtherActions settings status2 graph attacked
476         restactions
477     ColorGraph graph
478     let (newms,newstatus) =
479       UpdateMilestonesAndStatus graph milestones status3
480     drawgraph graph newstatus stepnum
481     if viewp
482     then List.iter (drawpercept stepnum) percepts
483     SimulationStep settings drawgraph drawpercept viewp agents
484       (stepnum+1) graph msgs newms newstatus
485
486 // Add agents to a graph, defined by their roles. The agents
487 // will be positioned randomly on the graph. The agents are
488 // created according to the values in a given int list and bool
489 // list.
490 // Has sideeffects.
491 // Graph -> (int list * bool list * 'a list) list -> unit
492 let internal NodeAddAgents ((nodes,_) as g) roles =
```

```
493   let n = Array.length nodes
494   let r = new Random(DateTime.Now.Ticks |> int)
495   let add a =
496     let randid = r.Next(n)
497     Graph.AddAgent g randid a
498   let makerole agentid (statlist,aclist,namelist) =
499     let addagents teamid _ =
500       Agent.Create((teamid,agentid),statlist,aclist)
501     List.mapi addagents namelist
502     |> List.choose (fun a -> a)
503     |> List.iter add
504   List.iteri makerole roles
505
506 // Create the AI objects according to the defined roles.
507 // Tells the created object the maximum allowed response time.
508 // int -> ('a list * 'b list * string list) list
509 //   Map<int*int,IAgent>
510 let internal CreateAI settings nums roles =
511   let makeagent agentid teamid name =
512     ((teamid,agentid),MAS2011.Agents.GetAgent name)
513   let makeagents agentid (_,_,slist) =
514     List.mapi (makeagent agentid) slist
515   let agents =
516     roles |> List.mapi makeagents |> List.concat
517   List.iter
518     (fun (_,(a : IAgent)) -> a.AddSettings settings)
519     agents
520   List.iter
521     (fun (_,(a : IAgent)) -> a.SetNumNodesEdgesSteps nums)
522     agents
523   agents |> Map.ofList
524
525 // Start a simulation with a given list of values, colors,
526 // roles and milestones, and a boolean value for whether or not
527 // the agents' perceptions are to be drawn.
528 // Creates the simulation in a thread. Aborts the existing
529 // simulation, if any.
530 // Has sideeffects.
531 // int list
532 //   -> Gdk.Color list * (int list * bool list * string list) list
533 //   -> (string * int list) list -> bool -> unit
534 let StartSim s (gdkcolors,roles) m viewp =
535   match s with
536   | nt::na::sl::fp::rt::recn::recd::nn::w::h::minn::maxn::
537     mine::maxe::acost::pacost::prcost::scost::icost::bcost::
538     rcost::fgcost::upbatt::upshie::upsens::upsabo::[]
539     ->
540     thread.Abort()
541     let settings =
542       { FailChance = fp
543         Length = sl
544         MaxAgentResponse = rt
545         RecoverNormal = recn
546         RecoverDisabled = recd
547         AttackCost = acost
```

```
548             ParryCost = pacost
549             ProbeCost = prcost
550             SurveyCost = scost
551             InspectCost = icost
552             BuyCost = bcost
553             RepairCost = rcost
554             FailedGotoCost = fgcost
555             UpgradeBatteryPrice = upbatt
556             UpgradeSensorPrice = upsens
557             UpgradeShieldPrice = upshie
558             UpgradeSabotageDevicePrice = upsabo }
559         let graph =
560           Graph.Create (nn,w-1,h-1,minn,maxn+1,mine,maxe+1)
561         let nums = (nn,Graph.NumEdges graph,sl)
562         NodeAddAgents graph roles
563         let colors = List.map GdkColor gdkcolors
564         let agents = CreateAI settings nums roles
565         let drawg = DrawGraph colors (w,h)
566         let drawp = DrawPercept colors (w,h)
567         let status = List.init nt (fun _ -> TS.StartStatus)
568         let milestones = List.choose ParseMilestone m
569         ResetSteps()
570         drawg graph status 0
571         thread <-
572           new Thread
573             (new ThreadStart
574             (fun _ ->
575             SimulationStep settings drawg drawp viewp agents 1
576               graph [] milestones status))
577         thread.Start()
578       | _ -> printfn "wrong length of s: %d" (List.length s)
```

B.22 SimulationSteps.fs

```
1  module MAS2011.Shared.SimulationSteps
2
3  (*
4  Module that keeps tracks of the saved simulation steps.
5  *)
6
7  open MAS2011.Shared.ShapePrimitives
8  open MAS2011.Shared.Generics
9  open System
10
11 // TickView: (Width * Height) * Shapes
12 type TickView = (int * int) * Shape list
13
14 let mutable internal SimSteps : Map<string,Map<int,TickView>> =
15   Map.empty
16 let internal mylock = new Object()
17 let mutable internal MsgRecievers = []
18 let mutable internal UpdateRecievers = []
19 let mutable internal NameAddedRecievers = []
20 let mutable internal ResetNamesRecievers = []
21 let mutable internal curpath = "Simulations/Default.txt"
22 let mutable internal min = 1
23 let mutable internal max = 1
24
25
26 // Whether to automatically save the simulation steps, or not.
27 // Steps will be saved to  curpath
28 // Warning: File size is very large, and will slow application!
29 let AutoSaveSteps = false
30
31
32 // Updates the path for autosave files.
33 // Has sideeffects.
34 // unit -> unit
35 let internal UpdatePath () =
36   curpath <- "Simulations/" + DateTime.Now.ToString() + ".txt"
37
38 // Deletes the currently saved steps. Called when a simulation
39 // starts, in order to display only the steps from the new
40 // simulation.
41 // Has sideeffects.
42 // unit -> unit
43 let ResetSteps () =
44   lock mylock
45     (fun _ ->
46       List.iter (fun f -> f()) ResetNamesRecievers
47       UpdatePath()
48       min <- 1
49       max <- 1
50       SimSteps <- Map.empty)
51
52 // Add simulation step n to the given name.
```

```
53  // Has sideeffects.
54  // string -> int -> TickView -> unit
55  let AddSimStepNoWrite name n s =
56    lock mylock
57      (fun _ ->
58        if SimSteps.ContainsKey name
59        then
60          let newmap = SimSteps.[name].Add(n,s)
61          SimSteps <- SimSteps.Add(name,newmap)
62        else
63          List.iter (fun f -> f name) NameAddedRecievers
64          SimSteps <- SimSteps.Add(name,Map.empty.Add(n,s))
65        if n < min then min <- n
66        if n > max then max <- n
67        List.iter (fun f -> f (name,n,min,max)) UpdateRecievers)
68
69  // Value to display when a requested simulation step doesn't
70  // exist.
71  let internal noret =
72    ((200,200),[Text((20.0,20.0),"Nothing here",White)])
73
74  // Get a certain simulation step.
75  // string -> int -> TickView
76  let GetSimStep name n =
77    lock mylock
78      (fun _ ->
79        match Map.tryFind name SimSteps with
80        | Some a ->
81          match Map.tryFind n a with
82          | Some x -> x
83          | _ -> noret
84        | _ -> noret)
85
86  // Convert a simulation step to a string representation, in
87  // a list. Returns [name;number;width;height;shapes]
88  // string -> int * TickView -> string list
89  let StepToString name (n,tv : TickView) =
90    let colortostr c =
91      match c with
92      | Hex(r,g,b) ->
93        sprintf "H,%d,%d,%d" r g b
94      | Red -> "R"
95      | Blue -> "B"
96      | Black -> "Bl"
97      | Yellow -> "Y"
98      | Green -> "G"
99      | White -> "W"
100   let colortypetostr ct =
101     match ct with
102     | Stroked(color) ->
103       "S," + colortostr color
104     | Filled(color) ->
105       "F," + colortostr color
106     | StrokedFilled(color1,color2) ->
107       sprintf "SF,%s,%s"
```

```
108         (colortostr color1) (colortostr color2)
109   let shapetostr s =
110     match s with
111     | Circle((x,y),rad,color) ->
112       sprintf "C,%.1f,%.1f,%.1f,%s" x y rad
113         (colortypetostr color)
114     | Rectangle((x,y),(w,h),color) ->
115       sprintf "R,%.1f,%.1f,%.1f,%.1f,%s" x y w h
116         (colortypetostr color)
117     | Line((x1,y1),(x2,y2),color) ->
118       sprintf "L,%.1f,%.1f,%.1f,%.1f,%s" x1 y1 x2 y2
119         (colortostr color)
120     | Text((x,y),s,color) ->
121       sprintf "T,%.1f,%.1f,%s,%s" x y s (colortostr color)
122   let shapestostr shapes =
123     shapes
124       |> List.map shapetostr
125       |> StringGlue ";"
126   let (w,h),shapes = tv
127   [name;string n;string w;string h;shapestostr shapes]
128
129 // Convert all simulation steps to a string.
130 // unit -> string
131 let StepsToString () =
132   let s =
133     SimSteps
134     |> Map.toList
135     |> List.map
136        (fun (name,map) ->
137        Map.toList map
138          |> List.map (StepToString name))
139     |> List.concat
140     |> List.concat
141   StringGlue "@" s
142
143 // Load simulation steps from a string representation.
144 // Has sideeffects.
145 // string -> unit
146 let LoadStepsFromString s =
147   let rec split5 l =
148     match l with
149     | a::b::c::d::e::tl ->
150       (a,b,c,d,e)::split5 tl
151     | _ -> []
152   let tickviews =
153     s |> StringToList '@'
154       |> split5
155   let strstocolor sl =
156     match sl with
157     | "H"::rs::gs::bs::tl ->
158       let r,g,b = byte rs,byte gs,byte bs
159       (Hex(r,g,b),tl)
160     | "R"::tl -> (Red,tl)
161     | "B"::tl -> (Blue,tl)
162     | "Bl"::tl -> (Black,tl)
```

```
163     | "Y"::tl -> (Yellow,tl)
164     | "G"::tl -> (Green,tl)
165     | "W"::tl -> (White,tl)
166     | _ -> (White,sl)
167   let strstocolortype sl =
168     match sl with
169     | "S"::tl ->
170       let (c,_) = strstocolor tl
171       Stroked(c)
172     | "F"::tl ->
173       let (c,_) = strstocolor tl
174       Filled(c)
175     | "SF"::tl ->
176       let (c,tl2) = strstocolor tl
177       let (c2,_) = strstocolor tl2
178       StrokedFilled(c,c2)
179     | _ -> Filled(White)
180   let strtoshape s =
181     let vals =
182       s |> StringToList ','
183     match vals with
184     | "C"::x::y::rad::colortype ->
185       let ct = strstocolortype colortype
186       Some (Circle((float x,float y),float rad,ct))
187     | "R"::x::y::w::h::colortype ->
188       let ct = strstocolortype colortype
189       Some (Rectangle((float x,float y),(float w,float h),ct))
190     | "L"::x1::y1::x2::y2::color ->
191       let (c,_) = strstocolor color
192       Some (Line((float x1,float y1),(float x2,float y2),c))
193     | "T"::x::y::str::color ->
194       let (c,_) = strstocolor color
195       Some (Text((float x,float y),str,c))
196     | _ -> None
197   let addtickview (name,ns,ws,hs,shapes) =
198     try
199       let (n,w,h) = (int ns,int ws,int hs)
200       let ss =
201         shapes
202         |> StringToList ';'
203         |> List.choose strtoshape
204       AddSimStepNoWrite name n ((w,h),ss)
205     with
206       | :? System.FormatException ->
207         printfn "Unable to format number when adding tickview"
208   tickviews |> List.iter addtickview
209
210 // Add a simulation step. Saves the simulation to a file if
211 // the boolean value AutoSaveSteps is true.
212 // Has sideeffects.
213 // string -> int -> TickView -> unit
214 let AddSimStep name n s =
215   let isempty = SimSteps.IsEmpty
216   AddSimStepNoWrite name n s
217   if AutoSaveSteps
```

```
218   then
219     if isempty
220     then System.IO.File.WriteAllText(curpath,StepsToString())
221     else
222       let str =
223         StepToString name (n,s)
224         |> StringGlue "@"
225       System.IO.File.AppendAllText(curpath,"@"+str)
226
227 // Add a function to be called when a message is sent.
228 // The function must be of type  string -> unit
229 // Has sideeffects.
230 // (string -> unit) -> unit
231 let AddMsgReciever f =
232   lock mylock
233     (fun _ -> MsgRecievers <- f::MsgRecievers)
234
235 // Send a message to all functions in MsgRecievers.
236 // Has potential sideeffects.
237 // string -> unit
238 let SendMsg (s : string) =
239   lock mylock
240     (fun _ -> List.iter (fun f -> f s) MsgRecievers)
241
242 // Add a function to be called when a simulation step is added.
243 // The function must be of type
244 // string * int * int * int -> unit
245 // The format is: name * step number * min * max in steps
246 // Has sideeffects.
247 // (string * int * int * int -> unit) -> unit
248 let AddUpdateReciever f =
249   lock mylock
250     (fun _ -> UpdateRecievers <- f::UpdateRecievers)
251
252 // Add a function to be called when a new name is added to the
253 // simulation steps. The function must be of type
254 // string -> unit
255 // Has sideeffects.
256 // (string -> unit) -> unit
257 let AddNameAddedReciever f =
258   lock mylock
259     (fun _ -> NameAddedRecievers <- f::NameAddedRecievers)
260
261 // Add a function to be called when the names (simulation
262 // steps) are reset. The function must be of type:
263 // unit -> unit
264 // Has sideeffects.
265 // (unit -> unit) -> unit
266 let AddResetNamesReciever f =
267   lock mylock
268     (fun _ -> ResetNamesRecievers <- f::ResetNamesRecievers)
```

B.23 SimulationView.fs

```
1  module MAS2011.Monitor.SimulationView
2
3  (*
4  This module contains a class to display the simulation steps,
5  along with some internal helper functions.
6  *)
7
8  open Gtk
9  open MAS2011.Shared.SimulationSteps
10 open MAS2011.Shared.ShapePrimitives
11 open Mono
12 open System
13
14 // Get the RGB values for a Color type, range 0.0 to 1.0
15 // Color -> float * float * float
16 let internal GetColor c =
17   match c with
18   | Hex(rc,gc,bc) ->
19     let r1 = (float rc) / 255.0
20     let g1 = (float gc) / 255.0
21     let b1 = (float bc) / 255.0
22     (r1, g1, b1)
23   | Red    -> (1.0, 0.0, 0.0)
24   | Blue   -> (0.0, 0.0, 1.0)
25   | Black  -> (0.0, 0.0, 0.0)
26   | Yellow -> (1.0, 1.0, 0.0)
27   | Green  -> (0.0, 1.0, 0.0)
28   | White  -> (1.0, 1.0, 1.0)
29
30 // Set the active color for a Cairo.Context.
31 // Has sideeffects.
32 // Cairo.Context -> Color -> unit
33 let internal SetColor (g : Cairo.Context) c =
34   let (rc,gc,bc) = GetColor c
35   g.Color <- new Cairo.Color(rc,gc,bc)
36
37 // Colors the active path in a Cairo.Context, according to
38 // a given ColorType.
39 // Has sideeffects.
40 // Cairo.Context -> ColorType -> unit
41 let internal ColorShape (g : Cairo.Context) ct =
42   match ct with
43   | Stroked(c) ->
44     SetColor g c
45     g.Stroke()
46   | Filled(c) ->
47     SetColor g c
48     g.Fill()
49   | StrokedFilled(c1,c2) ->
50     SetColor g c2
51     g.FillPreserve()
52     SetColor g c1
```

```
53      g.Stroke()
54
55  // Value used to approximate a circle with Bezier curves.
56  // From:
57  // http://www.whizkidtech.redprince.net/bezier/circle/
58  let internal kappa = 0.5522847498
59
60  // Draws a given shape on a given Cairo.Context.
61  // Has sideeffects.
62  // Cairo.Context -> Shape -> unit
63  let internal MakeShape (g : Cairo.Context) s =
64    match s with
65    | Circle ((x,y),r,ct) ->
66      let k = kappa*r
67      g.MoveTo(x,y-r)
68      g.CurveTo(x+k,y-r, x+r,y-k, x+r,y)
69      g.CurveTo(x+r,y+k, x+k,y+r, x,y+r)
70      g.CurveTo(x-k,y+r, x-r,y+k, x-r,y)
71      g.CurveTo(x-r,y-k, x-k,y-r, x,y-r)
72      g.ClosePath()
73      ColorShape g ct
74    | Rectangle ((x,y),(w,h),ct) ->
75      g.Rectangle(x,y,w,h)
76      ColorShape g ct
77    | Line ((x1,y1),(x2,y2),c) ->
78      g.MoveTo(x1,y1)
79      g.LineTo(x2,y2)
80      SetColor g c
81      g.Stroke()
82    | Text ((x,y),t,c) ->
83      g.MoveTo(x-1.0,y-12.0) // Pango position correction
84      SetColor g c
85      let layout = Pango.CairoHelper.CreateLayout(g)
86      layout.FontDescription <-
87        Pango.FontDescription.FromString("Verdana 12")
88      layout.SetText(t)
89      Pango.CairoHelper.ShowLayout(g,layout)
90
91  // SimulationView class - inherits from Gtk.DrawingArea
92  // The class that displays the simulation steps.
93  type SimulationView () as this = class
94    inherit DrawingArea()
95
96    let mutable activetick = 1
97    let mutable cursize = (20,20)
98    let mutable curname = "Simulation"
99
100   do
101     this.RedrawOnAllocate <- true
102     this.ExposeEvent.Add(this.OnExposed)
103     let (x,y) = cursize
104     this.SetSizeRequest(x,y)
105
106   member this.SetActiveTick n =
107     activetick <- n
```

```
108     this.QueueDraw()
109
110   member this.SetName name =
111     curname <- name
112     this.QueueDraw()
113
114   member __.Name = curname
115
116   member this.OnExposed _ =
117     let ((w,h),shapes) = GetSimStep curname activetick
118     if (w,h) <> cursize
119     then
120       cursize <- (w,h)
121       this.SetSizeRequest(w,h)
122     else
123       let g = Gdk.CairoHelper.Create(this.GdkWindow)
124       let background =
125         Rectangle((0.0,0.0),(float w, float h),Filled(Black))
126       List.iter (MakeShape g) (background::shapes)
127       let g2 = g :> IDisposable
128       g2.Dispose()
129 end
```

B.24 SimulationWindow.fs

```
module MAS2011.Monitor.SimulationWindow

(*
Module containing the class SimulationWindow. This class is
the main window in a simulation.
*)

open Gtk
open MAS2011.Monitor.SimulationView
open MAS2011.Monitor.SimSettingsWindow
open MAS2011.Shared.SimulationSteps

// Class SimulationWindow
// Inherits from Gtk.Window
type SimulationWindow () as this = class
  inherit Window("MAS 2011 competition simulator -"
    + " by Thor Helms, s061377, DTU")

  let contenttable = new Table(4u,2u,false)
  let timelabel = new Label("Step 1/2",WidthRequest=100)
  let mutable displayedtick = 1
  let mutable mintick = 1
  let mutable maxtick = 2
  let scale =
    new HScale(1.0,float maxtick,1.0,WidthRequest=200,
      DrawValue=false)
  let simview = new SimulationView()
  let statusbar = new Statusbar()
  let msgid = 1u
  let mutable playing = false
  let namebox = new VBox()
  let mutable names : RadioButton list = []

  do
    this.DefaultSize <- new Gdk.Size(600,500)

    let menubar = new MenuBar()
    let filemenu = new Menu()
    let newsimitem = new MenuItem("New simulation...")
    let openitem = new MenuItem("Open simulation...")
    let saveitem = new MenuItem("Save simulation...")
    let exititem = new MenuItem("Exit")
    let agentnames = MAS2011.Agents.GetAllNames()
    newsimitem.Activated.Add(
      fun _ -> new SimSettingsWindow(agentnames) |> ignore)
    exititem.Activated.Add(fun _ -> Application.Quit())
    exititem.Activated.Add(
      fun _ -> MAS2011.Simulation.KillSim())
    filemenu.Append(newsimitem)
    filemenu.Append(openitem)
    filemenu.Append(saveitem)
    filemenu.Append(exititem)
```

```
53      let fileitem = new MenuItem("File")
54      fileitem.Submenu <- filemenu
55      menubar.Append(fileitem)
56
57      let openfile _ =
58        let fc =
59          new FileChooserDialog
60            ("Choose the simulation to open",
61            this, FileChooserAction.Open,
62            "Cancel", ResponseType.Cancel,
63            "Open", ResponseType.Accept)
64        if fc.Run() = int(ResponseType.Accept)
65        then
66          let contents = System.IO.File.ReadAllText(fc.Filename)
67          MAS2011.Simulation.KillSim()
68          ResetSteps()
69          LoadStepsFromString contents
70        fc.Destroy() |> ignore
71      openitem.Activated.Add(openfile)
72
73      let savefile _ =
74        let fc =
75          new FileChooserDialog
76            ("Select a name/location for the simulation",
77            this, FileChooserAction.Save,
78            "Cancel", ResponseType.Cancel,
79            "Save", ResponseType.Accept)
80        if fc.Run() = int(ResponseType.Accept)
81        then
82          let contents = StepsToString()
83          let file = fc.Filename
84          System.IO.File.WriteAllText(file, contents)
85        fc.Destroy() |> ignore
86      saveitem.Activated.Add(savefile)
87
88      let timebox = new HBox()
89      let playpausebutton = new Button("Play/pause")
90      let prevbutton = new Button("Previous")
91      prevbutton.Clicked.Add(this.PrevTick)
92      let nextbutton = new Button("  Next  ")
93      nextbutton.Clicked.Add(this.NextTick)
94      scale.ValueChanged.Add(this.ScaleTick)
95      timebox.PackStart(playpausebutton, false, false, 4u)
96      timebox.PackStart(prevbutton, false, false, 4u)
97      timebox.PackStart(nextbutton, false, false, 4u)
98      timebox.PackStart(timelabel, false, false, 4u)
99      timebox.PackStart(scale, true, true, 4u)
100
101     statusbar.Push(msgid, "Starting") |> ignore
102
103     let fill = AttachOptions.Fill
104     let expand = AttachOptions.Expand
105     let fillex = fill ||| expand
106
107     let scrolled = new ScrolledWindow()
```

```
108     scrolled.AddWithViewport(simview)
109     let scrollnames = new ScrolledWindow(WidthRequest = 200)
110     scrollnames.AddWithViewport(namebox)
111
112     this.Add(contenttable)
113     contenttable.Attach(menubar,0u,2u,0u,1u,fillex,fill,4u,4u)
114     contenttable.Attach(scrollnames,0u,1u,1u,2u,fill,fill,4u,4u)
115     contenttable.Attach(scrolled,1u,2u,1u,2u,fillex,fillex,4u,4u)
116     contenttable.Attach(timebox,0u,2u,2u,3u,fillex,fill,4u,4u)
117     contenttable.Attach(statusbar,0u,2u,3u,4u,fillex,fill,4u,4u)
118
119     let playpause _ =
120       playing <- not playing
121       if playing
122       then GLib.Timeout.Add(333u, fun _ -> this.Play()) |> ignore
123
124     playpausebutton.Clicked.Add(playpause)
125
126     AddUpdateReciever this.StepUpdated
127     AddMsgReciever this.SendMsg
128     AddNameAddedReciever this.AddName
129     AddResetNamesReciever this.ResetNames
130
131   member this.Play () =
132     if displayedtick = maxtick
133     then playing <- false
134     else this.DisplayTick (displayedtick + 1)
135     playing
136
137   member this.NextTick _ =
138     this.DisplayTick (displayedtick + 1)
139
140   member this.PrevTick _ =
141     this.DisplayTick (displayedtick - 1)
142
143   member this.ScaleTick _ =
144     this.DisplayTick (int scale.Value)
145
146   member this.DisplayTick n =
147     if mintick <= n && n <= maxtick
148     then
149       if n <> displayedtick
150       then
151         scale.Value <- (float n)
152       timelabel.Text <- (sprintf "Step %d/%d" n maxtick)
153       displayedtick <- n
154       simview.SetActiveTick n
155     else ()
156
157   // When a simulation step is added, this method is called.
158   // It will then update the window and possible display the
159   // newly added simulation step.
160   member this.StepUpdated (name,n,min,max) =
161     let f () =
162       maxtick <- max
```

```
163         mintick <- min
164         if min < max
165         then scale.SetRange(double min,double max)
166         else scale.SetRange(double min, min+1 |> double)
167         if name = simview.Name &&
168           (displayedtick = n || displayedtick+1 = max)
169         then this.DisplayTick n
170         else
171           timelabel.Text <-
172             (sprintf "Step %d/%d" displayedtick maxtick)
173       Gtk.Application.Invoke(fun _ _ -> f())
174
175     // Will display the string s in the statusbar.
176     member this.SendMsg s =
177       let f () =
178         statusbar.Pop(msgid)
179         statusbar.Push(msgid,s) |> ignore
180       Gtk.Application.Invoke(fun _ _ -> f())
181
182     // When a new type of name is added to the simulation
183     // steps, this method is called with the newly added name.
184     member this.AddName s =
185       let makeradiob (s : string) =
186         if names = []
187         then new RadioButton(s)
188         else
189           let group = names |> List.head
190           new RadioButton(group,s)
191       let f () =
192         let r = makeradiob s
193         let toggled _ =
194           if r.Active
195           then simview.SetName s
196         r.Toggled.Add(toggled)
197         namebox.PackStart(r)
198         namebox.ShowAll()
199         names <- r::names
200       Gtk.Application.Invoke(fun _ _ -> f())
201
202     // This method removes all simulation step names from
203     // the window.
204     member this.ResetNames () =
205       let f () =
206         List.iter (fun w ->
207             namebox.Remove(w)
208             w.Destroy()) names
209         names <- []
210       Gtk.Application.Invoke(fun _ _ -> f())
211 end
```

B.25 TS.fs

```
1  module TS
2
3  (*
4  Module for manipulation of TeamStatus
5  *)
6
7  open MAS2011.Shared.SimTypes
8  open MAS2011.Shared.Generics
9
10 // Increase the amount of probes done by 1
11 // TeamStatus -> TeamStatus
12 let IncProbes ts =
13   { ts with Probes = ts.Probes + 1 }
14
15 // Increase the amount of surveys done by n
16 // TeamStatus -> TeamStatus
17 let IncSurveys n ts =
18   { ts with Surveys = ts.Surveys + n }
19
20 // Increase the amount of inspections done by n
21 // TeamStatus -> TeamStatus
22 let IncInspections n ts =
23   { ts with Inspections = ts.Inspections + n }
24
25 // Increase the amount of successful attacks by 1
26 // TeamStatus -> TeamStatus
27 let IncAttacks ts =
28   { ts with Attacks = ts.Attacks + 1 }
29
30 // Increase the amount of successful parries by 1
31 // TeamStatus -> TeamStatus
32 let IncParries ts =
33   { ts with Parries = ts.Parries + 1 }
34
35 // Convert a TeamStatus to an int list
36 // TeamStatus -> int list
37 let ToList (ts : TeamStatus) =
38   [ts.Score;ts.Money;ts.Probes;ts.Surveys;ts.Inspections;
39    ts.Attacks;ts.Parries]
40
41 // Reduce the money of a TeamStatus by a given amount
42 // int -> TeamStatus -> TeamStatus
43 let ReduceMoney amount (ts : TeamStatus) =
44   { ts with Money = ts.Money - amount }
45
46 // Increase the score of a TeamStatus by a given amount
47 // int -> TeamStatus -> TeamStatus
48 let IncreaseScore amount (ts : TeamStatus) =
49   { ts with Score = ts.Score + amount }
50
51 // Increase the money of a TeamStatus by a given amount
52 // int -> TeamStatus -> TeamStatus
```

```
53 let IncreaseMoney amount (ts : TeamStatus) =
54   { ts with Money = ts.Money + amount }
55
56 // TeamStatus starts with all values as 0.
57 // TeamStatus
58 let StartStatus =
59   { Score = 0
60     Money = 0
61     Probes = 0
62     Surveys = 0
63     Inspections = 0
64     Attacks = 0
65     Parries = 0 }
```