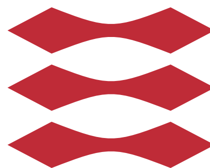
Multi-Agent Programming in Jason

Rasmus Jensen and Bastian Buch



Kongens Lyngby 2014
Compute-B.Sc.-2014

Technical University of Denmark
DTU Compute, Technical University of Denmark, Matematiktorvet, building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk Compute-B.Sc.-2014

Summary (English)

The goal of the thesis is to learn the Jason programming language and use our learned knowledge to successfully analyse and evaluate the strategies used by the UFSCTeam2013 Jason code in relation to the Multi-agent contest scenario in 2013. Once we have analysed both the UFSCTeam2013's program and the scenario we will discus possible additions to the program that would increase its performance. These improvements will then be implemented in our own version which we call JABARA.

To either prove or disprove the effectiveness of our improvements we will run a series of tests to determine whether or not our implementations has improved the UFSCTeam2013's program. These tests will be simulations where our version competes against the UFSCTeam2013's version.

Our results show that we have indeed managed to improve the program, albeit only slightly. Our improved version deals with some scenarios better than UFSCTeam2013. Specifically, in maps with a high number of edges our agents survey more edges and gain more achievement points faster as a result.

Summary (Danish)

Målet med dette projekt er at lære Jason programmeringssproget og bruge den erhvervede viden til succesfuldt at analysere og evaluere strategierne som bliver brugt af UFSCTeam2013 i Jason-koden i relation til multi-agent konkurrencen i 2013. Når vi har analyseret både UFSCTeam2013 programmet og scenariet så vil vi diskutere mulige tilføjelser som vil øge effektiviteten. Disse tilføjelser vil så blive implementeret i vores egen version som vi har valgt at kalde JABARA

For enten at bevise eller afvise effektiviteten af vores forbedringer så vil vi lave en serie af tests for at kunne afklare om vores implementation har forbedret programmet af UFSCTeam2013. Testene wil bestå af simulationer hvor JABARA konkurerer mod UFSCTeam2013.

Vores resultater viser at det har lykkedes os at forbedre programmet, dog kun en smule. Vores forbedrede version håndterer nogle scenarier bedre en UFSCTeam2013. Specielt i baner med et højt nummer af veje, vores agenter får søgt flere edges og får derfor som resultat af dette hurtigere nået milepæle.

Preface

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

This project was started at February 3th 2014 lasting until July 1st 2014 under the supervision of Jørgen Villadsen.

The thesis deals with multi-agent systems, the Jason agent-oriented programming language and the multi-agent programming contest from 2013.

Lyngby, 1-Juli-2014

Bastian Buch

Rasmus Jensen and Bastian Buch

Acknowledgements

We would like to thank our supervisor Jørgen Villadsen for great supervising and always being able to answer his E-mails should need be.

We would also like the thank the MAPC-team for their involvement in this project.

Contents

Summary (English) i

Summary (Danish) iii

Preface v

Acknowledgements vii

1 Introduction 1
- 1.1 Project . . . 1
- 1.2 Learning objectives . . . 2
- 1.3 Motivation and goals . . . 3
- 1.4 Areas of responsibility . . . 3

2 Multi Agent Systems 5
- 2.1 Agents . . . 5
- 2.2 Multi-agent systems . . . 6

3 BDI and The Jason Programming Language 7
- 3.1 BDI - Belief, desire and intention . . . 7
- 3.2 Jason programming . . . 8
  - 3.2.1 Belief base . . . 8
  - 3.2.2 Goals . . . 9
  - 3.2.3 Plans . . . 9

4 The multi-agent contest 11
- 4.1 The scenario . . . 11
  - 4.1.1 The ATPV's . . . 12
- 4.2 The MAPC system . . . 14

5 UFSCTeam2013 Analysis **17**
5.1 Start of simulation 17
5.2 Step beginning 18
5.3 Agent actions 18
5.4 Tokens 19
5.5 End of step 19
5.6 Agent classes 20
5.6.1 Saboteur 20
5.6.2 Repairer 20
5.6.3 Sentinel 20
5.6.4 Explorer 21
5.6.5 Inspector 21
5.7 Description of non-agent classes 21
5.8 Internal actions 23

6 Analysis of the scenario and MAPC system **27**
6.1 Agent improvements 27
6.1.1 Saboteurs 27
6.1.2 Explorers 28
6.1.3 Inspectors 28
6.1.4 Sentinels 29
6.1.5 Region segregation 29
6.2 Strategies 29
6.2.1 Buy strategy 30
6.2.2 Offensive vs. defensive saboteurs 31

7 The implementation **33**
7.1 Edge selection 33
7.1.1 Java additions 34
7.1.2 Internal actions 36
7.1.3 Common survey rules 39
7.1.4 Surveyall 40
7.1.5 Edge selection implementation 41
7.2 Buy strategy 43
7.2.1 Java additions 43
7.2.2 Internal actions 43
7.2.3 Buy strategy implementation 44

8 Testing **45**
8.1 Sentinel improvements 45
8.2 Saboteur improvements 46

9 Results **47**

10 Discussion **51**
10.1 The project . . . 51
10.2 The jason language . . . 52
10.3 Strategy . . . 52
10.4 Possible extensions . . . 52

11 Conclusion **53**
11.1 The project . . . 53
11.2 The multi-agent system . . . 54

A Test run 1 **55**

B Test run 2 **61**

C Test run 3 **67**

D Test run 4 **73**

E Test run 5 **79**

F Test run 6 **85**

Bibliography **91**

Chapter 1

Introduction

This report was written by group BaR and consists of the following group members: s113432 Bastian Buch, s113416 Rasmus Jensen. The thesis consists of theory about multi agent systems, Jason and the multi-agent programming contest. We discuss the implementation of the UFSCTeam2013 and how to improve it. Our own implementation of the improvements will then be tested to see if it was improved.

1.1 Project

Project 118: Multi-Agent Programming in Jason. In this project we have to create a multi agent system which is able to carry out assignments in the environment described at http://multiagentcontest.org/ (the 2013 version). In the beginning we hope to create a clean rebuild of the smadasUFSC but, despite what we hoped to accomplish in the beginning of the project, we had to reduce our goals. The reason for this was making an entirely new program with new agents seemed too much to do in the time we had available when we achieved a greater understanding of the program and challenge. So we chose to try and improve the Jason multi agent system that the MAPC team competed with in 2013 and see how much better we could make their agents and program. The main improvements that we'll try to make will be improvements that the MAPC

team has suggested to take a look at. The MAPC team came with the following suggestions:

- Improving the sentinels to look for non-surveyed edges in a clever way during the exploration phase and in that way might help the explorers to walk faster through the vertices.
- Making 1 or more inspectors inspect enemies that are far away to gain achievement points earlier. Enemies that are far away are enemies that can be seen by other agents and therefore might not be in the area of the inspector.
- Improving the explorers such that they will explore the map faster and can therefore join in the other agents earlier in forming the areas and as a consequence gain more points.

When these improvements has been implemented we will make a series of tests such that we can hopefully show an AI, which has an increased performance in some of the jobs that the agents are given. Hence making the improved AI better at the given scenario. We have chosen to name our program JABARA.

1.2 Learning objectives

The point of this report is to demonstrate that we have understood and are able to carry out the things specified in the learning objectives. We have to show that:

- we individually are capable of structuring a bigger project, keep a timed schedule and organise and plan the work which is to come.
- are able to summarize and understand technical information and can utilize technical problem solving through project work
- are capable to work with all phases of a project,including the drafting of a suggestion, solution and documentation.
- are capable of acquiring new knowledge in a way that is relevant, while maintaining a critical mind set towards the newly acquired knowledge. with this we should show that we can bring light on the present problem.
- are capable of conveying technical information, theory and results in a written format, visually, graphical and vocally.

1.3 Motivation and goals

We thought this project sounded interesting from the beginning and working with a new language "Jason" also sounded like a challenging way to increase our programming experience. The language also works together with Java which we have experience with, so this gives a more soft transition in contrary to if we had to use a new language to the entire agent system. We hope to improve the AI which has been given to us by implementing the suggestions given to us by the MAPC team mentioned earlier. If we spot any other improvements this will naturally be discussed and possibly implemented and become a part of the report.

1.4 Areas of responsibility

In this project the work has almost constantly been done together and in cooperation with each other so we can't say that one has a specific area of responsibility since we have both participated in every part of the project. But if we were to give ourselves some main areas we have been responsible for it would be Rasmus for the report and for Bastian it would be the code. We both did equal work analyzing and finding implementations/strategies that we could use to improve UFSCTeam2013's system.

Chapter 2

Multi Agent Systems

In this chapter we will present the concept of a rational agent and the basic theory behind agents. The idea of a multi agent system and how the systems work will also be presented.

2.1 Agents

To understand agents we have to understand their properties. An agent needs a goal and a plan to get there. Furthermore it must have a belief base, this is obtained by sensors. These sensors has to be able to sense the environment in which it is present. The agent should also have a list of different actions that can be performed in the environment to change its current state, these actions are performed through effectors or actuators. Creating good agents depend on how well an agent determines what to do, these choices of actions are made from what it believes and knows from the sensor input. This is illustrated on the figure 2.1.

Actions made by the Agent is determined by the available actions it has been given. To prevent random actions to be executed, each action has a set of preconditions that has to be met before the agent can execute the action. And

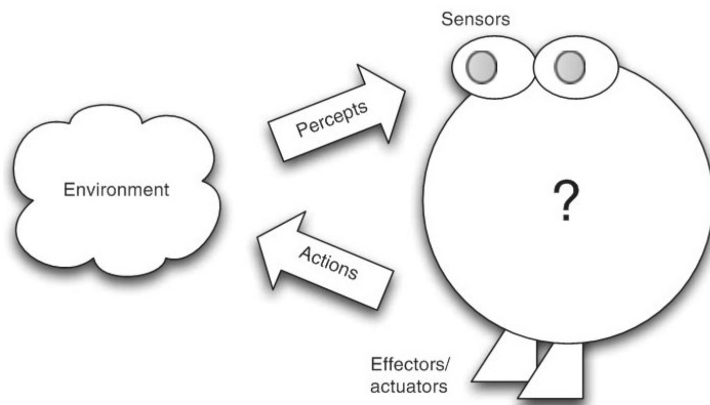


Figure 2.1: Agent in an environment [1], page 3

postconditions to determine what the belief base should changed to after the execution of the specific action. These are the basic needs for an agent to be functional.

To create what's called a rational agent more properties will have to be added. An rational agent will need the following properties: autonomy, proactiveness, reactivity and social ability [1]. Autonomy is needed so that the agent can act by itself and is not controlled by something else, acting to achieve its designated goals on its own. Proactiveness is needed to avoid passive agents - we expect that if an agent has been given a goal it will try to achieve it with its actions. Reactiveness is needed such that if the environment changes the agent will consider the changes and alter its actions accordingly. Lastly a rational agent will need social ability which makes it able to communicate its beliefs, goals and plans to other agents, so that they may cooperate and coordinate actions, benefiting from the enhanced perceptions of the environment.

2.2 Multi-agent systems

A multi-agent system (MAS) is a system where a group of different agents, who each has their own beliefs, plans and goals act towards a solution to their own goals. This means that the agents can either cooperate or compete to achieve their goals.

[1] bibliography item [1] page 4-5

Chapter 3

BDI and The Jason Programming Language

In this chapter we'll explain the BDI model and how it is used in the Jason programming language. Furthermore we'll also explain the basics of the Jason programming language and how the programs are structured.

3.1 BDI - Belief, desire and intention

The BDI model is a simple but useful model to use when programming agents to a multi agent system. The BDI model is based on a model of human behaviour which has been developed philosophers. The model is based on the fact that each agent has a belief, desire and an intention.

The Belief is what the agent knows about its current situation. This knowledge can of course be both correct and incorrect since it only knows what it is told and percieves. If an agent was in charge of a vending machine and had 5 cola's left we could think is the amount of cola's to be a part of the vending machines belief.
The Desire is what the agent wishes to do. However if an agent has a desire,

then it isn't obligated to do anything about it. The desire can therefore be looked upon as a series of options which the agent can choose to pursue.
The Intention is what the agent aspires towards. These intentions can come from given goals or as a consequence of a chosen option(desire). The agent will then work towards its intention in hope of reaching it. it might be necessary for the agent to go through several sub-intentions before it reaches its goal-intention.

If we were to apply this model to the needed functions of an agent we can look at figure 2.1. Here, the belief would be the input from the sensors, the desire would be the legs representing the effectors/actors and the actions on the environment would serve the purpose of realizing its intentions.

3.2 Jason programming

The Jason programming language is an extension to the agent language called AgentSpeak for Java and is based upon the BDI model. The language has 3 main factors, its belief base, plans and goals, which is representing the Belief, the intention and the desire, in the same order. Furthermore the language has been made in Java, which makes it easy to make small functions to aid the agents in their functioning. This section gives a brief explanation of the terms. A more detailed explanation can be found in the book Programming Multi-Agent Systems in AgentSpeak using Jason [1].

3.2.1 Belief base

The Belief base is composed of literals, these are used to show an agents belief. A very simple literal could be *TV(on)*, which means the TV is turned on. Now lets say the TV should not be powered after 8 pm then one might write *TV_until(on,Eight_pm)* in the belief base. In Jason however we use annotations which is represented by square brackets after the literal such that the previous example would look like *TV(on)[expires(Eight_pm)]*. There are many different annotations in Jason - one of the most important and useful ones is the source annotation. This annotation lets us know where the information which the agent believes comes from, whether it is the environment (*perceptual information*), other agents (*communication*) or itself (*mental notes*). Finally nested annotations is also a possibility. This may happen if an agent

[1] [1]

concludes something about a belief obtained by an agent which got the information from another agent. Agent Lis might believe the ball is blue because Max told her, but Max was told by Jonas. The literal would then look like this *Lis(Ball,blue)[source(max)[source(Jonas)]]*.[2] Another part of AgentSpeak is rules, these rules can be set up in such a way that if an agent is told 2 different things it will know which one to believe, usually you would set up rules so that if knowledge is granted via perception, but contradictory knowledge is in the knowledge base, the agent will always trust its perception and believe what it sees. But if an agent does not know, for example, the color of a box, and is told the color of a box by 2 different agents, it will trust the agent which the rules say it should trust. This could be implemented by adding trust to agents such that the agent could chose to acquire knowledge from the agent with the most credibility.

3.2.2 Goals

For an agent to have a direction, it needs goals. If an agent has a goal it will work towards a state where the agent believes the goal is fulfilled. In AgentSpeak and also in Jason there are 2 different kinds of goals *achivements goals* and *test goals*. The *achievements goals* is denoted by the '!' operator and is used to set the goal for the agent. If we set a goal for the agent *!drive(car)* it will work towards a state in which it is driving a car. The *test goals*, denoted by the '?' operator, is use more to retrieve information from the belief base. Say the agent wants to archive its previous goal (driving a car) then it might use the test goal *?owns(car)* to test if a car is available before executing a command to tell the agent to go drive it.

3.2.3 Plans

Another essential part of the AgentSpeak language is plans. Plans are what the agents use to react to the circumstance that they find themselves in while trying to achieve their given goals. Every plan consists of 3 parts *triggering event*, *context* and *body*. For a plan to be followed by an agent it has to be able to evaluate a trigger event as true. An agent might be able to evaluate several trigger events to true at the same time, this is where the context comes into play. The context is used to determine which plan fits the situation the best, several plans contexts might be evaluated and determined to be proper for the given situation, in that case its either a random or whatever comes first the

[2] [1], page 35-39

agent chooses. Next is the body, this is simply just a sequence of actions which should be performed to archive the goal of the plan, thus introducing sub-goals. The sub-goals are needed to ensure that an agent handles an event correctly.

CHAPTER 4

The multi-agent contest

The contest scenario which our multi-agent system is competing in is presented and explained in this chapter. Furthermore we will have a brief look at some of the systems in the MAPC system such as the map colouring and how you control water sections. A more detailed description can be found in the original description [1]

4.1 The scenario

In our given scenario we are a group of colonists situated on Mars. Water was found on the planet and the colonists developed the so-called "All Terrain PLanetary Vehicles" (ATPV) to search for the water. Water has become a precious resource and colonists started sabotaging other colonists. This resulted in 2 factions battling each other for the best water sources on the planet.
Our agents are supposed to take control of these ATPV's roaming the surface of Mars trying to find the best water sources. In some situations they should sabotage and in other situations they should defend. both teams has acquired a specific amount of vehicles of different types, each different type has a special job and therefore has a unique action. This will force the agents to work together

[1] Bibliography item [4]

to get the best performance possible.
As the scenario progress each team will earn achievement points, which acts as a currency, these are given to a team every time they reach a mile stone of some sort. As example when you have successfully attacked 5 times you earn 2 achievement pts. and after that you have to had successfully attacked 10 and then 20 and in this way to increases up until the upper limit of 640 is reached. There are other achievements these can be seen in the multi agent programming contest scenario description [2] page 22. When a faction have these the factions agents will have the option to upgrade themselves.

4.1.1 The ATPV's

There are 5 different ATVP's Repairers, Sentinels, Inspectors, Saboteurs and Explorers, both factions will be given 6 of each except the saboteurs of which they get 4. Common to all of them is that they can all perform some standard actions and has the attributes energy, health, strength and visibility range. Energy is used to perform actions which has a specified cost. Health is used to determine when an agent has been sabotaged and needs repairs. Strength is applied to the saboteur and determines how much of an impact the sabotage effect has. finally the viability range determines how far an agent can see. The common actions to all agents are skip, goto, survey, buy and recharge. These actions allow each ATVP to do the basics. Skip and goto are for movement purposes, survey checks the cost of moving across an edge, buy allows a ATVP to upgrade itself and if a unit runs out of energy it can recharge. Every action except Skip and recharge costs energy to perform. Each ATVP also has its own unique ability which determines what role it will have in the fight for water.

Repairers
This unit had its unique ability revealed in the name it is able to repair damaged units such that they remain able to stay in the battle. should a unit be totally destroyed it cant perform any actions apart from move and recharge. Repairers can also parry if it believes its going be attacked.

Sentinels
Sentinels is the scouting unit of this game. it can also parry, but other then that its greatest power is the increased viability range that allows it to spot enemies from afar.

Inspectors
Inspectors has the ability to inspect and will reveal the information about the

[2]Bibliography item [4]

enemy ATVP which it has inspected, such that the agents know which ATVP to keep an eye out for.

Saboteurs
Saboteurs are the attacking unit of this scenario and has the unique ability to sabotage. This ability can be used to reduce the enemy's vehicles health to 0 and with that reduce the efficiency of the enemy agents. The saboteurs can also parry.

Explorers
The Explorers can perform the action Probe this tells the agents what the value of the targeted node is and only when this is done can the team get the full value of the controlled nodes.

A chart it here presented to give a quick overview of the different Roles, actions and attributes.

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

Table 4.1: Roles, actions and attributes [4], page 14

The next table 4.2 show the actions and the costs of the different actions in the scenario.

4.2 The MAPC system

The MAPC system is run on a server to which you connect the agents. On this server the match is played. When playing in the match the agents are positioned on a map consisting of a lot of nodes connected by edges. The nodes has a value which is the value of water you can collect there and the edges also has a value which symbolises the cost of transporting a unit across. An example of the map can be seen below.

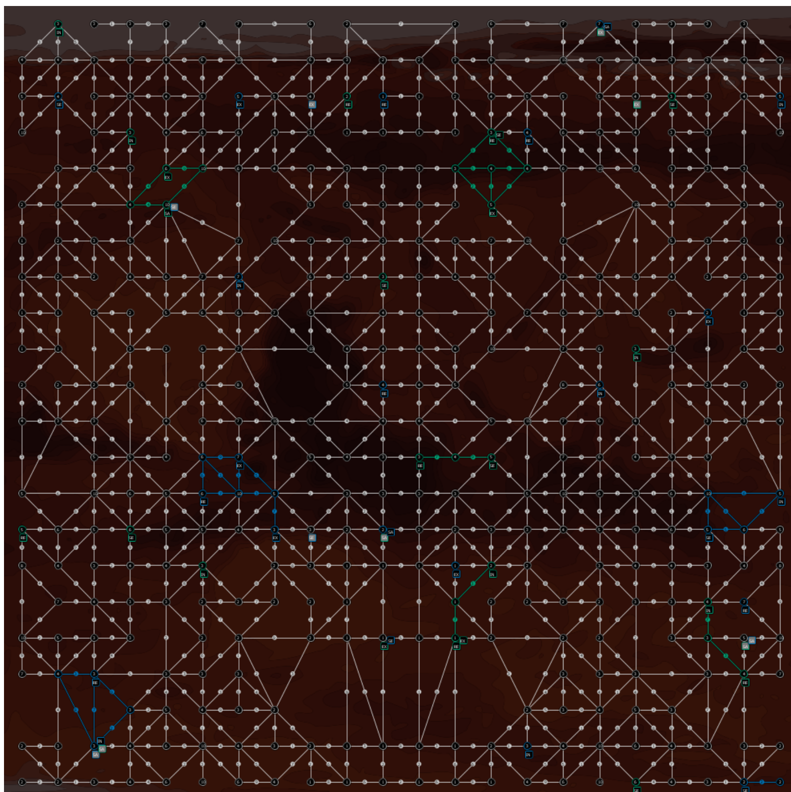


Figure 4.1: Example of the map

The graph colouring algorithm is also run on the server an is sent back to the agents each turn such that they know how they impact the environment by their

movement. When the server enters the colouring phase, it ventures through 4 steps that can be quickly explained like this. (A more detailed explanation can be found in bibliography item [4], page 11-13).

1. colouring all the nodes at which is dominated my only one team. A node is dominated if the team had the most agents on it.
2. Colouring the neighbours of dominated nodes. This will only happen if the neighbour is empty and the team wishing to control it, dominates 2 neighbouring nodes to the empty one.
3. If a team manages to create an closed of area in the form of a chain connecting one end to the other, in which there are no enemy agents, then it is perceived as a protected area and the team that has closed this area off are then controlling it.
4. Finally the fourth step is to not colour any remaining nodes.

examples of all the 4 different cases can be seen in figure 4.1.

Action	Property	Value
skip	Parameter:	-
	Status:	any
	Cost:	-
	Potential Failure causes	-
recharge	Parameter:	-
	Status:	any
	Cost:	-
	Potential Failure causes	-
goto	Parameter:	Vertex (required)
	Status:	any
	Cost:	edge value
	Potential Failure causes	energy shortage, attacked, unreachable, wrong param
probe	Parameter:	Vertex (optional)
	Status:	enabled
	Cost:	1 energy pt.
	Potential Failure causes	energy shortage, attacked, out of range, in range, wrong param, role, status
survey	Parameter:	-
	Status:	enabled
	Cost:	1 energy pt.
	Potential Failure causes	energy shortage, attacked, status
inspect	Parameter:	Agent (opponent - optional)
	Status:	enabled
	Cost:	2 energy pts.
	Potential Failure causes	energy shortage, attacked, out of range, in range, wrong param, role, status
attack	Parameter:	Agent (opponent - optional)
	Status:	enabled
	Cost:	2 energy pts.
	Potential Failure causes	energy shortage, parried, out of range, in range, wrong param, role, status
parry	Parameter:	-
	Status:	enabled
	Cost:	2 energy pts.
	Potential Failure causes	energy shortage, role, status
repair	Parameter:	Agent (teammate - required)
	Status:	any
	Cost:	2 energy pts. if enabled, 3 energy pts. if disabled
	Potential Failure causes	energy shortage, out of range, in range, wrong param, role
Buy	Parameter:	Attribute (Battery, sensor, shield or sabotageDevice)
	Status:	enabled
	Cost:	2 energy pts. 2 achievement pts.
	Potential Failure causes	energy shortage, wrong param, status, limit

Table 4.2: Action properties [4], page 17

Chapter 5

UFSCTeam2013 Analysis

In this chapter we will look on the relevant code which we believe can be improved to perform better in the scenario presented in 2013. The overall structure of the program is based on the JaCaMo platform, using Jason for agents, Cartago for artifacts and Moise for organizations. Since Cartago and Moise are beyond the scope of this project we will be primarily focusing on analysis of the Jason code.

5.1 Start of simulation

The UFSCteam2013 strategy starts off by splitting their agents into subgroups while assigning a leader for each agent type (a saboteur leader, explorer leader, and so on), as well as initializing beliefs and goals. Some additional groups are also defined, namely Main for leaders, and Alpha and Beta containing half of all agents each, as well as Special Exploration and Special Operations. Each group receives their own artifact, and an artifact is defined for the environment. An artifact in this case is part of the CArtAgO code which defines artifacts *"as resources and tools dynamically constructed, used, manipulated by agents to support/ realise their individual and collective activities (like artifacts in human*

contexts)." [1]. All rules apply to values within the artifacts or knowledge received from the contest server, thus the agents have both a knowledge base and a visual perception of their environment. Beliefs are also initialized for hills, pivots and islands. A hill is defined as "a zone formed by several vertices that have a good value and are in the same region of the map." [2]. A pivot is defined as "regions of the map that can be conquered by just two agents."[3]. An islands is defined as "a region of a map that can be conquered by a single agent."[4].

5.2 Step beginning

In the beginning of each step of the simulation, each agent has a chance to process some information before the step proper in the processBeforeStep(S) goal. Repairers use this to check for disabled friendly agents nearby. After this, a test is run to check if there are enemies inside the constructed islands and pivots. (testCleanIsland and testCleanPivot), and if there are the saboteur leader receives a message with the enemy position, and a belief is established that a saboteur has been called to the vertex, so that no duplicate calls are issued. If this is not the case the island/pivot is reported as clean and a eventual call is abolished, as per the Jason internal action.

5.3 Agent actions

After this, each agent runs the wait_and_select_goal plan, which regulates when agents act in order to enforce a order of priority among agents. In saboteurs, explorers and repairers a check is made to see if all saboteurs with higher priority have taken their actions before the acting agent selects a goal. This is used along with beliefs to ensure that no duplicate calls are made (for example, calledSaboteur(V), where V is a vertex). Inspectors and sentinels do not have this implemented, as they select their actions randomly from a list of options and there is less harm in assigning duplicates than with repairers, saboteurs and explorers. In either case, the wait_and_select_goal executes its body, which is the select_goal plan.

The select_goal plan examines the belief base to select the goal for each agent to execute. The plans for each goal are listed with individual contexts, so as to

[1]Bibliography item [3]
[2]Bibliography item [5]
[3]Bibliography item [5]
[4]Bibliography item [5]

best evaluate which action is appopriate to pursue. It then runs a goal called "Init_Goal(G)", which prints information to the console about the state of the agent and simulation, and then adds the goal G to the list of goals for the agent. While the action that happens next is often dependent on a plan for the particular goal, it results in a call to the "do(Act)" goal, the plan for which prints to console and runs the !commitAction goal, marking the step as done for the agent and then uses the act method of the CentralizedEnvrionment package in Jason to execute the specified action.

5.4 Tokens

The next part of the step consists of sending a token for the step S+1 to the next agent with highest priority. Possession of a token (the belief token(s)) is used to regulate some beliefs where several agents of different classes works together, so as to ensure atomic actions among these agents. The token contains the step (used to ensure the agent acts only once per turn) and is incremented once all agents that will act (have not passed the token) have acted. An agent can pass the token if they are disabled. Saboteurs always pass the token as they act outside of the swarm. Repairers that are in the process of attending to an agent also always pass the token. The forward_token goal is recursively added by the plan, and as such each agent that receives the token will forward it as long as an applicable agent can receive it (as this is a condition for the plan). Forwarding of the token happens while in the same step as the token, therefore once the token S+1 is sent the token is no longer forwarded among the agents until the next step begins.

5.5 End of step

A belief is then added called lastStep(S) with S being the current step. a goal is then added called processAfterStep(S), which similarly to processBeforeStep(S) allows the agents to process some information after all steps have been taken.

- Explorer leader uses this step to calculate the total sum of held vertices with an internal action and update the belief (calculateTotalSumVertices) and then builds the area of the hill for the step using the internal actions buildBestCoverage and buildBestCoverageTwo(buildArea(S)).
- Sentinel Leader uses this step to determine pivots and islands with internal actions depending on position and best coverage (buildPivots(S) and

buildIslands(S)).

- Saboteur leader and helper updates the most desirable known island excluding the one currently held and forwards it to the helper saboteur (updateEnemyIsland).
- Repairers ask if the agent they are scheduled to repair have already been repaired (testAppointmentWithDisabledAgent).

Finally, goals are run to recover the system (if the system has lost held information on edges, vertices or steps) from the coach, the current step is added to the log and the graph system is updated.

5.6 Agent classes

5.6.1 Saboteur

The Saboteur class contains a large set of beliefs regarding what agents to attack, as well as beliefs for obtaining paths to valuable islands with, whether or not a given saboteur is the chaser, and paths to attack enemy agents. The saboteur has goals for recharging and scheduling repairs with a repairer, attacking a target and going to and holding a vertex or hill.

5.6.2 Repairer

The Repairer class contains a set of beliefs regarding when to repair a nearby or adjacent (same vertex) ally for each class of ally. It also has a goal for going to a vertex where such an appointment might be scheduled (so that if an ally calls a repairer it will know how to go there.) Note that the repairers will cease to repair allied Explorer agents after step 200. The repairer has goals for recharging and scheduling repairs with another repairer, as well as going to and holding a vertex or hill.

5.6.3 Sentinel

The Sentinel class contains goals for recharging and scheduling repairs with another repairer, as well as going to and holding a vertex, island or hill.

5.6.4 Explorer

The explorer class contains beliefs on whether or not to expand a vertex it is standing on (a condition of which is that it is unprobed) and whether or not an adjacent vertex is unprobed. It also contains beliefs for special explorers that probe only inside the designated hill. Aside from this, the class contains goals for recharging and scheduling repairs with another repairer, as well as going to and holding a vertex, island or hill.

5.6.5 Inspector

The Inspector class contains beliefs and plans on when to inspect a nearby enemy. It also contains goals for recharging and scheduling repairs with another repairer, as well as going to and holding a vertex, island or hill.

5.7 Description of non-agent classes

A short description of each .asl class that is not an agent class in the program follows. These classes contain core goals, beliefs and plans as well as most of the common beliefs held by agents. Note that the program does not halt on exception throws.

- mod.aimVertex - Contains rules and goals for distributing orders among agents to take pivots and islands.
- mod.common - Contains common beliefs and plans for agents that causes internal actions and adds goals if implemented, typically by an attribute change (such as position and HP). For example, when position changes, a goal is added to evaluate enemy positions with the new knowledge gained from visibility. Also contains rules for what to do when health is lost or at zero, which the agent perceives from its environment.
- mod.commonRules - Contains a list of common rules for agents.
- mod.commonWalkRules - Contains a list of common rules for movement. If the destination is good then the agent random-walks there otherwise it recharges. When heading for a specific path, it will walk through the path using tail recursion.
- mod.EISactions - Contains goals which execute EIS actions.

- mod.environment - Contains goals for switching focus among artifacts.
- mod.expanding - Contains rules related to expansion of own vertexes and entering uncontrolled vertexes.
- mod.finishRound - Contains rules and goals for resetting and terminating the program.
- mod.hill - Contains the rules and goals for defining a hill and the optimal coverage of that hill. This is mostly done using internal actions regarding the environment, which use CArtAgO.
- mod.initialization - Contains goals for initializing the program.
- mod.islands - Contains rules for defending and determining the state of islands, as well as goals for determining and moving to islands. The path to get to the islands is composed by an internal action usingDjikstra's algorithm to find the path with the lowest cost.
- mod.loadAgents - Contains the goal to load agents names, details, role and priority (used in mod.Token). This is changed depending on whether the agent is used in slot A or slot B.
- mod.newStep - Contains the plans and actions taken for each step(turn) of the competition. Functions as a BDI interpreter for the program.
- mod.organization - Contains the rules and plans for organizing agents into groups. Also contains all the overall goals for the program during the competition, and the rules and plans determine the mechanics as to which groups receive the individual goals.
- mod.pivot - Contains plans and beliefs for determining pivots and asserting the ownerships of pivots.
- mod.repair - Contains plans for evaluating health and sending repair requests to the repair leader, as well as plans for if the agent can come to the repairer.
- mod.startRound - Contains plans for handling initialization of edges, vertices, steps and the simulation start in the first round.
- mod.swarm - Contains rules for handling movement inside the hill for the swarm, that is to say all non-saboteurs.
- mod.swarm.expanding - Contains rules for expanding the swarm from within the hill, in order to maximize the frontier area beyond the defined hill area.

- mod.token - Contains rules and plans for handling, generating and passing tokens among the swarm.
- mod.walking - Contains plans for random walking and planned walking.

5.8 Internal actions

The Java code can generally be divided into the following categories:

- classes governing implementation of the CaRTagO artifacts (artifacts package),
- the agent architecture class implementation, logging events and agent listeners(env package)
- classes that handle translation of Environment Interface Standard to Jason for communication between Jason, Moise and Cartago(jason.eis package)
- classes that contain the graph implementation and interaction functions and algorithms used for pathfinding (Djikstra, BFS) and determining pivots, islands and hills (graphLib package)
- classes that contain internal actions to be executed by the Java code as a condition in plans in the Jason code (ia package)

Of these packages, the internal actions (ia) package is of particular interest since it is the primary way in which the Jason code interacts with the graph library, pathfinding functions and logging systems. A list of the internal actions follow:

- addEdge - takes atomic arguments vertex U, vertex V and integer weight. Executes the addEdge function of the Graph class, adding an edge between the two vertices with the specified weight. Returns true.
- addEnemyPosition - takes arguments integer id, and strings enemy and vertex. Executes the addEnemyPosition function of the Graph class, adding the enemy to the position vertex in the graph, correcting the older position if such exists. Returns true.
- addLastAction - takes arguments integer agentId and step, and string arguments action and result. Adds an entry of the action to the active instance of ContestLogger. Returns true.

- bestCoverage - takes arguments integer depth and variable terms bestVertex and bestValue. Executes the internal action getBestCoverage of the Graph class, getting the best possible coverage for the listed depth, unifying the bestVertex argument with the best found vertex and the bestValue argument with the value of the vertex. Returns true if a result is found, otherwise throws exception.

- bestCoverageIgnoringVertex - Same as bestCoverage, except adds a string argument ignoreVertex. Returns same as bestCoverage except it will not return the ignored vertex.

- cleanBeliefBase - Gets the belief system of an agent in the transition system as an iterator, then iterates through it and removes a belief if it is a rule. Returns true.

- cleanPostion - takes argument integer id. Gets the active instance of the graph then removes the listed position of the entity with the listed id. Returns true.

- copyOfVisibleEnemy - takes argument terms termEnemy and termVertex. Iterates through the active instance of the graph then unifies all visible enemies with termEnemy and their vertices with termVertex. The result is kept as a unifier and returns true if the unifier is not empty.

- edge - Same as copyOfVisibleEnemy except takes atomic argument vertexU and terms vertexV and weight. Unifies all connecting vertices and the weight of their connecting edges.

- getAgentPosition - takes argument integer id and an unspecified second argument. Uses the getPosition function of the Graph class to get the vertex assigned to the id and unifies it with the second argument as a new atomic term named vertexTerm. Returns false if no such vertex exists.

- getDistance - takes string arguments vertexS and vertexD, and variable term lenght[sic]. Uses the getDistance function from the Graph class to calculate the distance in vertices between vertexS and vertexD and unifies it with lenght. Returns true.

- getIslands - takes arguments integer amount and variable term islandsTerm. Uses the getAllIslands function from the Graph class to get the list of known islands, then adds them as vertices to a list and binds the list with islandsTerm. Returns true.

- getPivots - Same as getIslands, but gets all pivots instead using the getAllPivots function of the Graph class. Returns true.

- getPivotsIgnoringVertices - Same as getPivots, except ignores vertices in the list term verticesIgnore. Returns true.

- getPivotsJustSomeVertices - Same as getPivots, except only returns pivots connected to vertices in the list term vertices. Returns true.

- getVertexGrade - Same as getAgentPosition, except returns vertex grade instead.

- isIsland - takes string argument vertexV. Returns true if the getIsland function of the graph class does not return null with vertexV as argument.

- neighborhood - takes string argument vertexS, integer depth and list term listOfNeighbors as arguments. Uses the getNeighborhood function of the Graph class to return the list of neighbors connected to vertexS in the given depth, bound to the listOfNeighbors term. Returns true if the list of neighbors is not null, throws exception otherwise.

- probedVertex - takes argument term vertex and number term implementation result. If the vertex is an atom, the function binds the result of the getVertexValue function of the Graph class to result and returns false. Otherwise, the second term is renamed probedValue and the function getVertexByValue of the Graph class is called with the value as argument. It returns a list of vertices with the probed value then iterates through it and if a vertice unifies with the vertex term the iterator adds it. Returns true if the result of the iterator is not null.

- remEnemyPosition - same as getEnemyPosition, except removes it. Returns true.

- resetGrapth - Creates a new graph and resets the current graph and logger instance. Returns true.

- setAgentPosition - takes integer argument id and string argument vertex. Uses the setPosition function of the Graph class to create a position in the graph for the given id on the given vertex. Returns true.

- setMaxEdges - takes unspecified integer argument. Sets this argument as max edges using the setMaxEdges function of the Graph class. Returns true.

- setMaxVertices - Same as setMaxEdges, only vertices.

- setVertexValue - Same as setMaxEdges, only sets a value for a vertex that is given as an atomic term.

- setVertexVisited - Same as setVertexVisite, only sets the visited boolean property in the graph instance to true.

- shortestPath - takes string arguments vertexS and vertexD and variable term arguments path and lenght. Uses the getShortestPath function of the Graph class to get the shortest path between vertexS and vertexD as a list of vertices, binding the path to list and unifying lenght with the size of the shortest path list. Returns true if a path exists that is not null and has size >0.
- shortestPathBFSTwo - same as shortestPath, except returns the first result from a list of possible destination vertices in vertexD using the BFS getShortestPathBFSComplete function from the Graph class. Returns true under same conditions.
- getShortesPathDjikstraComplete - same as shortestPath except using the Djikstra complete path. Returns true under same conditions.
- getShortesPathDjikstraComplete - same as shortestPathBFSTwo except using the Djikstra complete path. Returns true under same conditions.
- sumVertices - Returns the sum of vertices in the unspecified term. Returns true.
- synchronizeGraph - Synchronizes the number of edges between the graphs in the agent architecture and the global graph. Returns true.
- thereIsUnprobedVertex - returns the result of the boolean function thereIsUnprobedVertex in the Graph class.
- vertex - same as ProbedVertex, except the first condition returns the team controlling the vertex instead of the vertex value.
- visibleEnemy - same as probedVertex, except the first condition returns the specific position if an enemy term and vertex is specified. If enemy term is specified but no vertex, all vertices with the enemy term is returned (by iterator) in termEnemy and termVertex lists. If neither are specified, all known enemy positions are returned(by iterator) in termEnemy and termVertex lists.
- vertexVisited - takes term argument vertex. Returns true if the vertex is recorded visited by the getVertexVisited function of the Graph class, returns false otherwise.

Chapter 6

Analysis of the scenario and MAPC system

In this chapter we will reflect upon the scenario and the MAPC system described in the previous chapter and from that discuss the possible ways of improving the agents in the MAPC system to preform better in this scenario and what strategies that might be useful following to archive the highest score.

6.1 Agent improvements

The different ideas of improvements for our agents are listed here.

6.1.1 Saboteurs

There is not much room for improvement as we can see in the saboteurs. We thought that a small but useful addition could be made, which is if an inspector sees the opposite team buying health for more than 1 of their saboteurs, then our saboteurs should upgrade their attack such that they would still be able to sabotage them in 1 shot instead of 2 and from there on of whenever they

buy health we buy damage. The key value in this strategy is that if the enemy starts upgrading their health they will have to spend 4 achievement points to upgrade it above the one shot health limit. But we wont have to spend more than 2 achievement points to make ourselves able to one shot it again. this will give us a little advantage in the beginning if the opponent chooses to upgrade its health. If the upgrades has been made within turn 100 then there is 700 turns where we may get 5600 victory points more than the other team, alone on this improvement, since they spend an overall 8 points more upgrading their saboteurs.

6.1.2 Explorers

Sometimes the explorers are gathered in one or more sections of the map with a part of the map without any explorers. This is an inefficient way to explore all the nodes. Instead the map should be divided into regions where each region would get an explorer associated with it and in that way remove movement time for the explorers since the regions would be of approximately same size. This could gain us an advantage in the following ways. We would probe the map earlier and therefore have better knowledge of the best water sources, This will reduce the time we spent finding the best place to gather water. Additionally the finished explorers can join up with the others in forming the control area earlier which also reduced the time spend. To appoint the explorers to the given regions of the map the program would have to calculate the best way of appointing the explorers to have the shortest way in total for all the explorers to the region they would be given.

6.1.3 Inspectors

One of the suggestions we got from the MAPC team was the make the inspectors inspect far enemies, maybe not all of them but just the leader inspector. where far enemies are enemies that could be seen by other agents, but this wont work since an inspector can inspect them. The only way an inspection will work is if the effective range is grater or equal to the range at which the target is from the inspector and because the $effectiveRange = VisRange * rand^2$ we can never succeed in inspecting enemy agents that is beyond the Vision range of the inspector. Since this vision range is 1 and improving it will cost achievement points which will reduce the overall points gained throughout the match we have concluded that this strategy is suboptimal and should not be implemented in JABARA.

6.1.4 Sentinels

A way to improve the sentinels - Instead of them walking randomly around surveying edges that haven't been surveyed, we would make the sentinels move to the node with the most unsurveyed edges, and implement prioitized goal selection like the explorers, so that no two sentinels act at the same time and end up surveying the same node. This way we would gain maximum knowledge about the map each time a sentinel used its ability survey. Once we reach turn 130 enough knowledge of the map has been found and then the main force of the sentinels joins in controlling an area. We leave one sentinel in charge of walking around surveying the rest of the map to hopefully find more optimal routes and once that is done it will join the rest in controlling a big part of the map.

6.1.5 Region segregation

A way to implement segregation of the map into regions would be using the sentinels to map a series of interconnected vertices that would be saved as a set. A Sentinel and an Explorer could then go through the regions vertices one by one. Whenever a sentinel then moves to another vertex to survey its edges it would in most cases increase its known part of the map by discovering new vertices because of its vision range. These vertices should then be added to the region belonging to the sentinel that were able to see them if and only if the vertex hasn't already been given a region or the path to the island is blocked by another region. Should it happen that a sentinel surveys its entire region we categorize it as region completed and the sentinel should then go and survey in another region. Should it discover new islands while surveying the new region it will then expand the region which it has been assigned to and not the original one.

6.2 Strategies

This section will discuss different strategies that has been considered to be viable as a strategy that might have influence on the outcome of who the winner is going to be.

6.2.1 Buy strategy

The idea of this strategy is to use the achievement points early on and hence lose score early on, but in the overall match archive the achievement points earlier. In this way it might increase the achievement points pool earlier than usually and then ending up with more points in the end than what you lost on using them in the early stages of the game. This was mainly an idea we had for the sentinels. The idea was to increase their visual range than with that increase the number of edges it could survey and then reach the achievements goals for amount of surveyed edges faster. We later realized that this was not a strategy that could work, since the needed amount of edges you have to survey between each achievement milestone is doubled for every time you get one with the exception of the jump at 320. For surveyed edges the achievement milestones are 10, 20, 40, 80, 160, 320, 320, 640 and finally 1280. An upgrade to the sentinel can therefore only gain more victory points than it loses, if the upgrade more than quadruples edges searched per turn and we do not gain all achievement points for surveying. The effective range in which we survey edges is $EffectiveRange = (Visualrange - 1) * rand^2 + 1$, this means that the numerical value of the effective range to increase by upgrading from 3 to 4 gives a 0.11 extra chance of surveying nodes within 2 range instead of 1. Assuming that the sentinel surveys constantly on the map, recharges once a while and doesn't move then average number of surveys for one Sentinel would be 533 surveys which results in $533 * 4$ edges surveyed assuming that there are approximately 4 edges for each node. Of course you cannot survey the same edges constantly, but this is a best-case example. With a range at 3 we have for our four adjacent edges a 0.18 probability of surveying the edges on the other side of an edge (which will total out to 12 edges), we can therefore multiply by $12 * 0.18$, for a total of $533 * 4 + 533 * (12 * 0.18) + 1$ edges surveyed. With a range at 4 we have for our four adjacent edges a 0.29 probability of surveying the edges on the other side of an edge (which will total out to 12 edges), we can therefore multiply by $12 * 0.29$, for a total of $533 * 4 + 533 * (12 * 0.29) + 1$ edges surveyed. Since we have six sentinels, we will gain all achievement points for surveying edges. as even assuming the sentinels have other things to do, we would have to lose more than 90 percent of our potential surveys at range 3 to not get the last two achievement points, which will definitely happen if the sentinels keep moving around The only other possibility for a gain by improving range is if the upgrade quadruples the number of edges seen per survey, which it does not do as the math above demonstrates. Even though the smadasUFSC implementation effectively stops surveying after turn 133, the gain per survey is still too small to justify such an upgrade.

Buying other advantages to hurry the achievement unlocking are not present so the buying strategy is therefore not a viable strategy to follow in this scenario.

6.2.2 Offensive vs. defensive saboteurs

We also considered what would be most effective, to use an offensive or defensive strategy for the saboteurs. Our conclusion was that defensive will never be as good as offensive since if you position yourself defensively you will simply just allow the enemy to come to you and then they can do what they want while you stay somewhere at the map. This means it will never be profitable to try and hold your ground you should always attack and try to destroy the opposing faction by making the front lines be positioned at their position and then by that, ruin the amount of water sources they can control.

These are the ideas of improvement and strategies that we thought of. Regarding the implementation of the ones that we believe will improve the UFSCTeam2013 system. We do not need to implement an aggressive play-style since the system is already programmed aggressively. We have implemented the improved saboteurs, such that we can counter players that might upgrade health. We implemented the improved sentinels such that we survey faster. This can be seen in the next chapter.

Chapter 7

The implementation

The implementation of the improvements of the agents and how they work is presented in this chapter.

7.1 Edge selection

The original implementation of the smadasUFSC multi-agent system did not assign any one agent type or instance the task of surveying the map. Instead, if inspectors, sentinels and repairers had no other assignments, they would check if surveying was a good idea by checking if an unsurveyed edge is connected to their position. Sentinels, since they had few other assignments, would usually survey their position. If left with no immediate goals to accomplish, the agents would all random walk, thus implementing an entirely randomized surveying strategy. We believe that a more coordinated, less randomized way of surveying the edges will improve the strategy, as the agents will be more capable of finding the optimal path to a given vertex and the achievement points for surveying edges will be given earlier, resulting in more points overall. Since the original implementation of sentinels left them with no particular role except a higher priority on the surveying goal, we decided to implement edge selection on the sentinels.

7.1.1 Java additions

We have added new global variables to the Graph java class to allow the use of our functions. These are:

- short unsurveyedEdges[] - The number of unsurveyed edges are stored in an array of shorts, each index corresponding to a vertex, incremetned for both connected vertices when an unsurveyed edge is added and decreased when an edge is surveyed.
- boolean allEdgesSurveyed - Used to stop the thereIsUnsurveyedEdge function early if it has already returned false once.
- int lastCheckForUnsurveyedEdges - Used to store the last vertex that had an unsurveyed edge, so as to prevent unnecessary repetition of the edge scanning of thereIsUnsurveyedEdge.

We have added several functions to the Graph java class in order to facilitate the discovery and handling of unsurveyed edges. These functions are:

- thereIsUnsurveyedEdge - Returns true if an edge on the graph has been recorded as having more than or equal to the max observed weight assigned, and not infinite weight. Any observed edge that has not yet been surveyed will fulfil these conditions as although maxWeight may decrease as a result of agent actions it will never increase. This function have a worst-case runtime of i*(i-1) but will only run through the entire set of edges once in total and it will start from the last vertex that had an unsurveyed edge. Returns true if an unsurveyed edge is found.
- getUnsurveyedEdgesOfVertex - Returns the number of unsurveyed edges of the vertex as a short.
- getVertexWithMostUnsurveyedEdges(List<String> vertexList) - Returns a list of vertices with the highest observed number of unsurveyed edges from among the vertex list. Uses a for- loop to iterate through the vertexList, examining the index in unsurveyedEdges. If a vertex with higher edge count than stored is found, the list is cleaned and the new entry is added. If it has the same edge count, it is added to the list. Returns the list.
- getVerticesWithUnsurveyedEdges - Returns a list of all vertices with unsurveyed edges.

```
// Added by BaR

        // If an isolated vertex exists in the graph database then
        // clearly an edge connecting to this vertex has not been
             surveyed yet. This
        // assumes that all vertices have at least one edge.

        @SuppressWarnings("unused")
        public boolean thereIsUnsurveyedEdge() {
                if (allEdgesSurveyed)
                        return true;
                if (getSize() <= 1)
                        return true;
                // Assuming no isolated vertices this is the bare minimum
                     of edges
                if (MAXVERTICES > (edgeCounter - 1))
                        return true;

                for (int i = lastCheckForUnsurveyedEdges; i < getSize();
                     i++) {
                        for (int j = MAXVERTICES; j < getSize(); j++) {
                                // We know this vertex exists, so at least
                                     one edge connects
                                // here.
                                if (w[i][j] <= MAXWEIGHT && w[i][j] != INF)
                                     {
                                        // Since the number of vertices and
                                             position of vertices in
                                        // the array never change, we can
                                             continue from where we
                                        // left off when we check again
                                        lastCheckForUnsurveyedEdges = i;
                                        return true;
                                }
                        }
                }
                allEdgesSurveyed = true;
                return false;
        }

        //Returns the amount of unsurveyed edges at the vertex
        public short getUnsurveyedEdgesOfVertex(String vertexV){
                return unsurveyedEdges[getVertexValue(vertexV)];
        }

        //Returns the vertices with the most unsurveyed edges
```

```
public List<String>
    getVerticesWithMostUnsurveyedEdges(List<String> vertexList){
        short maxUnsurveyedEdges = 0;
        List <String> targetList = new ArrayList<String>();
        for (int i = 0; i < vertexList.size(); i++){
                if
                    (unsurveyedEdges[vertex2Integer(vertexList.get(i))]>maxUnsurveyedEdg
                        maxUnsurveyedEdges =
                            unsurveyedEdges[vertex2Integer(vertexList.get(i))];
                        targetList.clear();
                } else if (unsurveyedEdges[i]==maxUnsurveyedEdges){
                        targetList.add(vertexList.get(i));
                }

        }
        return targetList;
}
//Returns all unsurveyed vertices
public List<String> getVerticesWithUnsurveyedEdges(){
        List <String> targetList = new ArrayList<String>();
        for (int i = 0; i < MAXVERTICES; i++){
                if (unsurveyedEdges[i]>0){
                        targetList.add(integer2vertex[i]);
                }
        }
        return targetList;
}
```

7.1.2 Internal actions

We have added three internal actions that use our newly implemented functions:

- thereIsUnsurveyedEdge - Returns the result of graph.thereIsUnsurveyedEdge.

```
package ia;

import env.MixedAgentArch;
import graphLib.Graph;
import jason.asSemantics.*;
import jason.asSyntax.*;

//Added by BaR. Internal action handler for check for unsurveyed edges.
    Functionally identical to ia.thereIsUnprobedVertex.
```

```
public class thereIsUnsurveyedEdge extends DefaultInternalAction {

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[] args)
         throws Exception {
        MixedAgentArch arch = (MixedAgentArch)ts.getUserAgArch();
        Graph graph = arch.getGraph();

        return graph.thereIsUnsurveyedEdge();
    }

}
```

- getUnsurveyedEdges - Binds the result of graph.getVerticesWithUnsurveyedEdges as a list in a variable term nodes and the size of the list in a variable term nodeCount.

```
package ia;

import java.util.List;
import env.MixedAgentArch;
import graphLib.Graph;
import jason.asSemantics.*;
import jason.asSyntax.*;


//Added by BaR: Returns a list in bar notation with all known nodes that
    have unsurveyed edges, and the length of that list
public class getUnsurveyedEdges extends DefaultInternalAction {

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[] terms)
         throws Exception {
        MixedAgentArch arch = (MixedAgentArch)ts.getUserAgArch();
        Graph graph = arch.getGraph();

        VarTerm nodes = ((VarTerm) terms[0]);
        VarTerm nodeCount = ((VarTerm) terms[1]);

        List<String> nodeList = graph.getVerticesWithUnsurveyedEdges();

        if (nodeList != null && nodeList.size() > 0) {
            ListTerm list = new ListTermImpl();
            ListTerm tail = list;
            for (String s : nodeList) {
```

```
            tail = tail.append(new Atom(s));
        }

        un.bind(nodes, list);
        un.unifiesNoUndo(nodeCount, new
            NumberTermImpl(nodeList.size()));

        return true;
    } else {
        return false;
        //throw new JasonException("No nodes with unsurveyed edges");
    }
  }
}
```

- getUnsurveyedEdgesTwo - Takes argument list term startNodes. Binds a list of the nodes with the highest amount of edges as a variable term highestNodes.

```
package ia;

import java.util.List;
import env.MixedAgentArch;
import graphLib.Graph;
import jason.asSemantics.*;
import jason.asSyntax.*;


//Added by BaR: Returns a list in bar notation with all known nodes that
    have unsurveyed edges, and the length of that list
public class getUnsurveyedEdges extends DefaultInternalAction {

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[] terms)
        throws Exception {
        MixedAgentArch arch = (MixedAgentArch)ts.getUserAgArch();
        Graph graph = arch.getGraph();

        VarTerm nodes = ((VarTerm) terms[0]);
        VarTerm nodeCount = ((VarTerm) terms[1]);

        List<String> nodeList = graph.getVerticesWithUnsurveyedEdges();

        if (nodeList != null && nodeList.size() > 0) {
            ListTerm list = new ListTermImpl();
```

```
            ListTerm tail = list;
            for (String s : nodeList) {
                tail = tail.append(new Atom(s));
            }

            un.bind(nodes, list);
            un.unifiesNoUndo(nodeCount, new
                NumberTermImpl(nodeList.size()));

            return true;
        } else {
            return false;
            //throw new JasonException("No nodes with unsurveyed edges");
        }
    }
}
```

7.1.3 Common survey rules

We have added a list of beliefs called mod.commonSurveyRules to the Jason implementation. These beliefs are called is_good_survey_destination(Op) and is_good_survey_destination(D, Path) and can contain either a single vertex or a destination vertex and a list of connecting vertices, as the gotoPath(D, path) plan. These beliefs return true for their respective terms under the condition that there is more than one unsurveyed node. This is deduced by the getUnsurveyedEdges and getUnsurveyedEdgesTwo internal actions, the latter being used to find the vertices or vertex with the most unsurveyed edges. The beliefs are prioritized towards the closest vertices, then vertices at a distance of two, three, and afterwards any distance. If the target vertex is directly connected to the vertex, the belief is for a single vertex. If there is distance between the vertices, the belief contains the destination vertex and the path to the destination vertex from the agents current vertex, using the shortestPath internal action.

```
/*
 * Added by BaR. Contains common rules for determining survey locations.
 */
 //Choose the nearest vertex with the most unsurveyed edges within range
     1, 2 or 3. After that we don't care about max
 //path length and we just go for the best edge we can get to.

 is_good_survey_destination(Op):-   position(MyV) &
     ia.getUnsurveyedEdges(TargetNodes, NodeNumber) &
 .setOf(TargetNodes, ia.shortestPath(MyV,TargetNodes,_,Length) & Length
```

```
    = 1, EdgesInRange) &
ia.getUnsurveyedEdgesTwo(EdgesInRange,Options) &
.length(Options, TotalOptions) & TotalOptions > 0 &
.nth(math.random(TotalOptions), Options, Op).

is_good_survey_destination(D, Path):- position(MyV) &
    ia.getUnsurveyedEdges(TargetNodes, NodeNumber) &
.setOf(TargetNodes, ia.shortestPath(MyV,TargetNodes,_,Length)& Length =
    2, EdgesInRange) &
ia.getUnsurveyedEdgesTwo(EdgesInRange,Options) &
.length(Options, TotalOptions) & TotalOptions > 0 &
.nth(math.random(TotalOptions), Options, D) &
    ia.shortestPath(MyV,D,Path,_).

is_good_survey_destination(D, Path):- position(MyV) &
    ia.getUnsurveyedEdges(TargetNodes, NodeNumber) &
.setOf(TargetNodes, ia.shortestPath(MyV,TargetNodes,_,Length)& Length =
    3, EdgesInRange) &
ia.getUnsurveyedEdgesTwo(EdgesInRange,Options) &
.length(Options, TotalOptions) & TotalOptions > 0 &
.nth(math.random(TotalOptions), Options, D) &
    ia.shortestPath(MyV,D,Path,_).

is_good_survey_destination(D, Path):- position(MyV) &
    ia.getUnsurveyedEdges(TargetNodes, NodeNumber) &
.setOf(TargetNodes, ia.shortestPath(MyV,TargetNodes,_,Length)& 3 <
    Length, EdgesInRange) &
ia.getUnsurveyedEdgesTwo(EdgesInRange,Options) &
.length(Options, TotalOptions) & TotalOptions > 0 &
.nth(math.random(TotalOptions), Options, D) &
    ia.shortestPath(MyV,D,Path,_).
```

7.1.4 Surveyall

In order to search all edges effectively we must know when to stop looking for more edges, and thus we have implemented a goal called surveyAll in mod.organization. This goal is an obligation to the Sentinel Leader. The goal is fulfilled when the added internal action notAllEdgesSurveyed returns false. A belief is also added to the sentinel agent class called is_survey_goal_active, defined as the result of the internal action notAllEdgesSurveyed. This belief is used as a condition in the implementation for edge selection.

```
/*
 * Added by BaR: Goal: surveyAll
```

```
 */

+!surveyAll[artifact_name(Scheme)]:
        .my_name(MyName) & play(MyName,sentinelLeader,"grMain") &
            ia.thereIsUnsurveyedEdge
    <-
    .wait({+step(_)}, 1000);
    !!surveyAll[artifact_name(Scheme)].
-!surveyAll[artifact_name(Scheme)]
    <-
    !!surveyAll[artifact_name(Scheme)].
+!surveyAll[artifact_name(Scheme)]:
    .my_name(MyName) & play(MyName,sentinelLeader,"grMain")
<-
    goalAchieved(surveyAll)[artifact_name(Scheme)];
    .print("All edges surveyed!").
+!surveyAll[artifact_name(Scheme)].
```

7.1.5 Edge selection implementation

Several plans are added to the Sentinel agent class with select_goal as a triggering event, all having is is_survey_goal_active as a condition. Another belief, is_first_phase_acitve is also used, this belief returns true when the step number is less than or equal to 133. While both these conditions are true, along with one of the is_good_survey_destination beliefs, the sentinel will execute a goto(Op) or gotoPath(D, path) plan. These plans are prioritized below survey and parry plans as well as repair scheduling plans and recharge plans. After is_first_phase_active becomes false, a second set of plans with select_goal as triggering event can be triggered, but with the condition that the agent must be sentinel one. This is so that all remaining sentinels will focus on defending the hills and vertices.

```
//Added by BaR:
//Try using the survey rules to determine a good place to survey if
    survey goal and first phase are still active
+!select_goal: is_survey_goal_active & is_first_phase_active &
    is_good_survey_destination(Op)
<-
!init_goal(goto(Op)).

+!select_goal: is_survey_goal_active & is_first_phase_active &
    is_good_survey_destination(D, Path)
<-
```

```
!init_goal(gotoPath(Path)).

//Go survey an unvisited vertex if you are Sentinel One and the survey
    goal is still active
+!select_goal: .my_Name(sentinel1) & friend(sentinel1, _, sentinel, _) &
    is_survey_goal_active & is_good_survey_destination(Op)
<-
!init_goal(goto(Op)).

+!select_goal: .my_Name(sentinel1) & friend(sentinel1, _, sentinel, _) &
    is_survey_goal_active & is_good_survey_destination(D, Path)
<-
!init_goal(gotoPath(Path)).
```

We have also implemented a code snippet in mod.token so that sentinel one will pass the token when there are still edges to survey.

```
/*
 * If I'm disabled I don't need to token.
 * If I'm a explorer and there is still some vertices to probe, I also
     don't need the token
 * If I'm a repairer and I have an appointment with some agent, I also
     don't need the token
 * Added by BaR: If I'm Sentinel one and there are edges left to survey,
     I don't need the token
 */
+token(S):
    (
        is_disabled
    |
        .my_name(MyName) & friend(MyName, _, explorer, _) & not
            noMoreVertexToProbe
    |
        .my_name(MyName) & friend(MyName, _, repairer, _) & busy(_)
    |
        .my_Name(sentinel1) & friend(sentinel1, _, sentinel, _) & not
            noMoreEdgesToSurvey
    ) & not lastToken(S)
<-
    .abolish(lastToken(_));
    +lastToken(S);
    !forwardToken(S);
    .print("Received token to step ", S, " but forwarding it because I'm
        disabled").
```

7.2 Buy strategy

The original implementation of smadasUFSC did not have a buy strategy. We have implemented a reactive buy strategy for the saboteurs that will buy sabotageDevices if the enemy saboteurs are noticed to have more health than our saboteurs have in attack strength.

7.2.1 Java additions

We added a single global variable and a function to the Java code in the inspectArtifact class, the variable higestObservedHealth, which is a static byte containing the highest observed health of a saboteur, and an addition to the addEntity function that checks if the observed health of a saboteur entity is higher than higestObservedHealth, if so highestObservedHealth is set to that value. The function added is getHighestObservedHealth() which returns the highestObservedHealth variable.

```
// Added by BaR
    public static byte highestObservedHealth = 4;

    @OPERATION
    void addEntity(String entity, String type, int maxHealth, int
         strength, int visRange) {
        // Added by BaR
        if (maxHealth > highestObservedHealth &&
             type.matches("(.*)aboteu(.*)")) highestObservedHealth =
             (byte) maxHealth;
        if (entityToObs.containsKey(entity)) return;

        [...]

// Added by BaR
    public static byte getHighestObservedHealth(){
        return highestObservedHealth;
    }
```

7.2.2 Internal actions

A single internal action has been added, getHighestObservedHealth, which takes the number term implementaton argument health and unifies it with the result

of the getHighestObservedHealth() function.

```
package ia;

import jason.asSemantics.*;
import jason.asSyntax.*;
import artifacts.InspectArtifact;

//Added by BaR. Internal action handler for check for unsurveyed edges.
    Functionally identical to ia.thereIsUnprobedVertex.
public class getHighestObservedHealth extends DefaultInternalAction {

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[] terms)
        throws Exception {

        NumberTermImpl health = ((NumberTermImpl) terms[0]);
        un.unifiesNoUndo(health, new
            NumberTermImpl(InspectArtifact.getHighestObservedHealth()));
        return true;
    }
}
```

7.2.3 Buy strategy implementation

We simply added a plan for executing the buy action, using as conditions that the health from getHighestObservedHealth is equal to or higher than the saboteurs own attack strength.

```
// Added by BaR: Observe own strength and only buy if it is less than
    highest observed health among enemy Saboteurs
is_buy_goal(sabotageDevice) :- not is_disabled & money(M) & M >= 8 &
                              strength(Str) &
                                  ia.getHigestObservedHealth(Required) &
                                  Str < Required.
```

Chapter 8

Testing

In this chapter we go through the different tests that we have put our multi-agent system through to test the improvements.

8.1 Sentinel improvements

For testing the improvements on the sentinels we ran around 30 simulations. From these a pattern will begin to show and we can evaluate on the results of these simulations. In the simulations or program will fight the original such that we will see the actual improvement compared to the old one. Every time a simulation has been executed a folder with statistics is created filled with data which we can use to analyse the simulation. Examples of these statistics can be seen in the appendix with test runs. In reality we should run 1000, maybe 10000 simulations to make any real judgement on our system, but these are time consuming and we could see the pattern early on and it was consistent.

8.2 Saboteur improvements

For the inspector improvements we have no way of testing their efficiency since we have no program that actually utilizes a buy strategy. If we where to test it we should have had access to an implementation using an offensive buy strategy, for example HactarV2 [1], as it had a buy strategy for which we could have tested our strategy's effectiveness.

[1] Bibliography item [2]

Chapter 9

Results

The results of our improvements does not show a substantial difference in the programs overall performance since the improvements doesn't improve the program in all ways but only increases the efficiency of the sentinels. Another improvement that we made was for the saboteurs but since our added buy strategy for those depends on the enemy to buy health then it wont ave any effect because the UFSCTeam2013 doesn't buy health.

The improved sentinels in JABARA are able to survey much faster than the sentinels in UFSCTeam2013 - this can be seen in the two examples below and on the test runs in the appendix. The improvement show itself much more if there are a lot of edges, more than 600, if the map only consists of around 300 there are few way to improve the surveying since the map will be mostly connected by single connections and dead ends. An example of our improvement on a map with more than 600 edges can be seen in test run number 1 on figure 9.1.

Here we can see that our improvement make it possible for JABARA to reach the 640 edges surveyed achievement around 100 turns earlier than the UFSCTeam2013. Also in the beginning we get a small advantage since JABARA get to some of the earlier achievement positions earlier. Now if the map has a few edges we wont see such a great improvement since our algorithm wont have a lot of edges to choose from and will therefore have a hard time improving itself

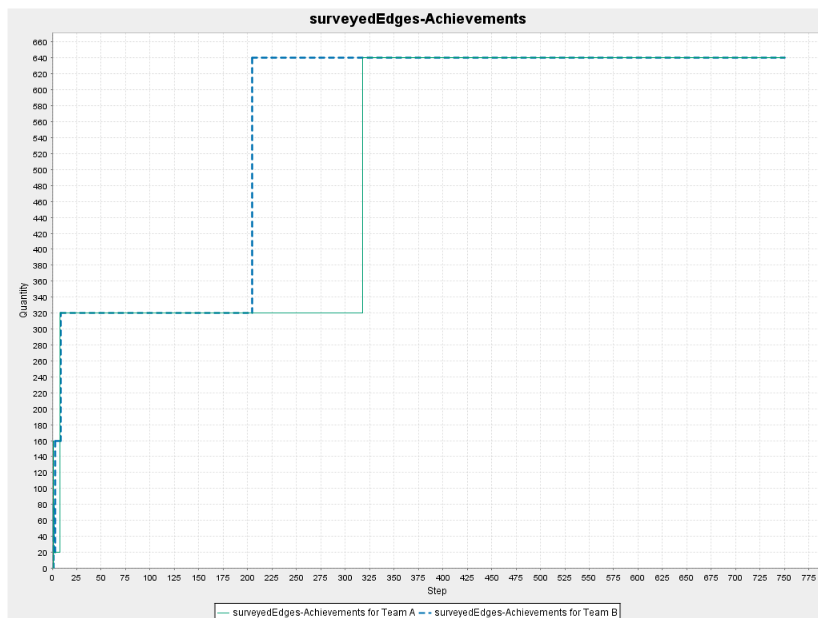


Figure 9.1: Survey Achievement goals

since it is more or less forced to go in one direction. an example of this can be seen in test run 4 on figure 9.2.

In these cases it it more or less luck that determines which way will be the best for the agents to go. we see that we are very equal in those instances, so in case of a few edges our algorithm still works as well as the one used by UFSCTeam2013. Some maps JABARA also makes it survey more of the map than UFSCTeam2013 before turn 133 which is where it start giving other orders than surveying. an example of this is test run 2 in figure 9.3. In this figure we see that UFSCTeam2013 never reaches the 640 edges surveyed goal while our JABARA does.

Finally as one can see in the test runs if there is a few edges it is equally distributed who wins (random), but at the maps with many edges, the gained advantage by our increased knowledge about of edges makes JABARA win almost all the maps. There are still situations where UFSCTeam2013 win, but there is a consistency that allows us to say that our algorithm has increased the performance of the program.

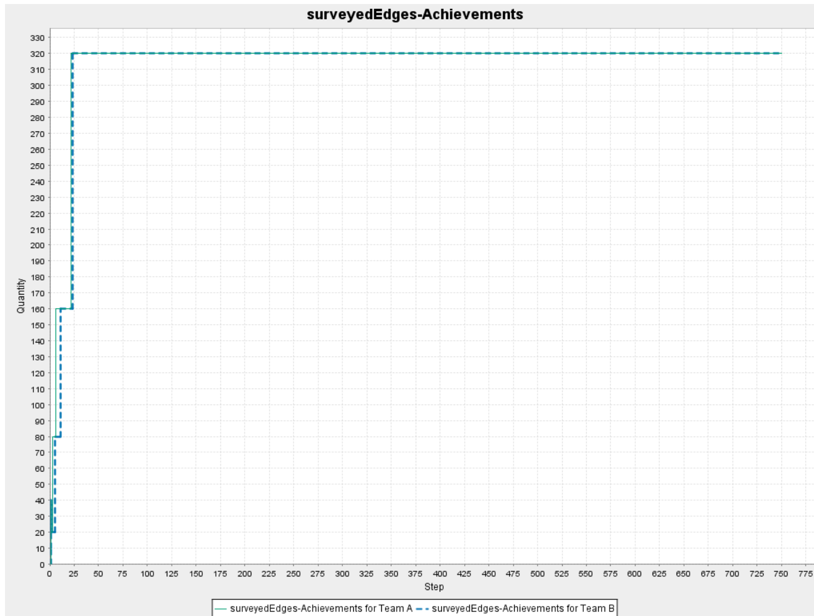


Figure 9.2: Survey Achievement goals

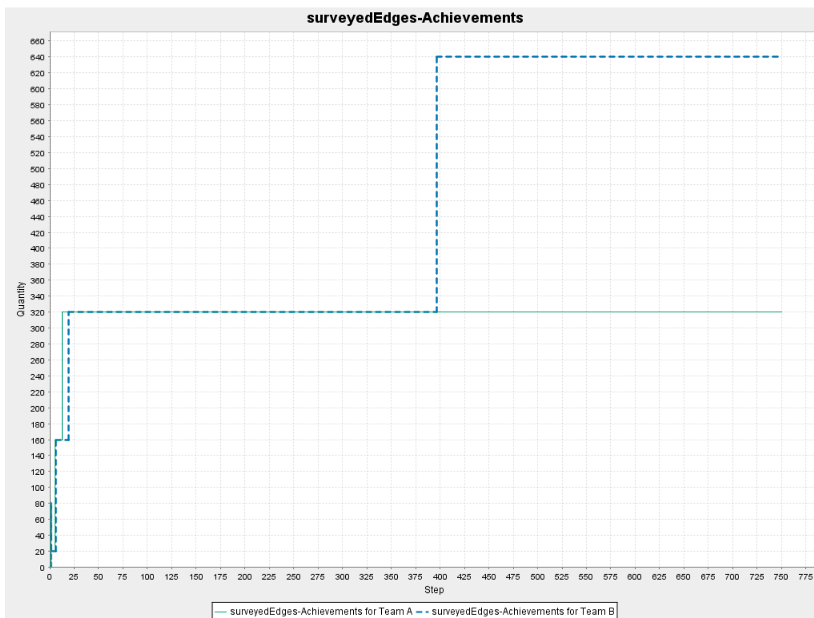


Figure 9.3: Survey Achievement goals

Chapter 10

Discussion

In this chapter we'll discuss the solutions and difficulties we've encountered during this project.

10.1 The project

In the beginning of this project we wanted to make our own clean build of smadasUFSC for the given scenario, but this quickly changed when we saw the extent of knowledge we would need about Jason and time necessary to do that, as well as knowledge of CArtAgO and Moise which is outside the scope of this project. Therefore we chose to improve the already made program smadasUFSC. With that decision came some other difficulties. Except from the suggestions that were given by the MAPC-team there weren't many improvements that we where able to find, and those we did find was not improvements that would increase its general performance but rather make it able to fend off different tactics. Because it was difficult to find ways of improving it we used a lot of time analysing and looking at the code trying to understand what it did and when. With the understanding that it gave us, we came up with the improvements mentioned in our Scenario analysis.

10.2 The jason language

The Jason language was actually fairly simple, reminding us a lot about Prolog which we worked with last semester. The biggest problem we had was that there wasn't that many exercises to get familiar with Jason and we couldn't find much in the internet most of the time it would just refer to the Jason book which we had already. So with the little knowledge we had to started to analyse the smadasUFSC Jason code. The only major problem we had with it was the initial understanding of how it was connected together and how data was transferred between them since it was a big program with a lot of methods entangled into each other from different files. The use of Moise and CArtAgO made it a little difficult to know how they did some of the things in their code, but thanks to their description we knew what they did.

10.3 Strategy

We believe that we have found the best ways of improving the strategy since the scenario rewards the player that waits other that the player that initiates when it comes to buying upgrades for your achievement points since every turn with achievements points spend is a turn with less victory points. Our strategy can be countered, but for it to happen the enemy would need a very specific strategy (upgrading only one saboteur with health). But that is the case with every strategy there is no strategy that cant be exploited with another, but one can make a strategy that can beat the majority of the strategies and that is what we believe we have made.

10.4 Possible extensions

If we where to further increase the performance of our program it would obviously be the the segregation of regions on the map. If this was implemented we would not only be able to use it for our explorers but also enhance our current solution for our improved sentinels, so that the sentinels would also be given a region to survey. We believe that the region segregation would be beneficial for the sentinels because it might run into the same problem as the dummy sentinels does. They may be surveying up into a corner and then have to move all the way down to another section of the map before they can survey a new part of the map, as well as remaining at a distance in which they will not interfere with each other.

Chapter 11

Conclusion

Here we make a conclusion on our multi-agent system and on our project.

11.1 The project

In this project we believe that we have done an overall good job. We have managed to structure our work hours in such a way that the workload was spread out in the best way possible, though we did get a little stressed at the end of it. Additionally we have learned a new language Jason and explained the key features and used it to implement improvements into the smadasUFSC program. This shows that we are able to summarize, understand and utilize technical information. This would never have been accomplished if we where not able to work with every phase of a project and where capable of acquiring new knowledge in a relevant way to shine light upon problems that we met along the way. Furthermore this report is a proof of our capability of conveying technical information, theory and results in a written format as well as presenting it visually and graphically.

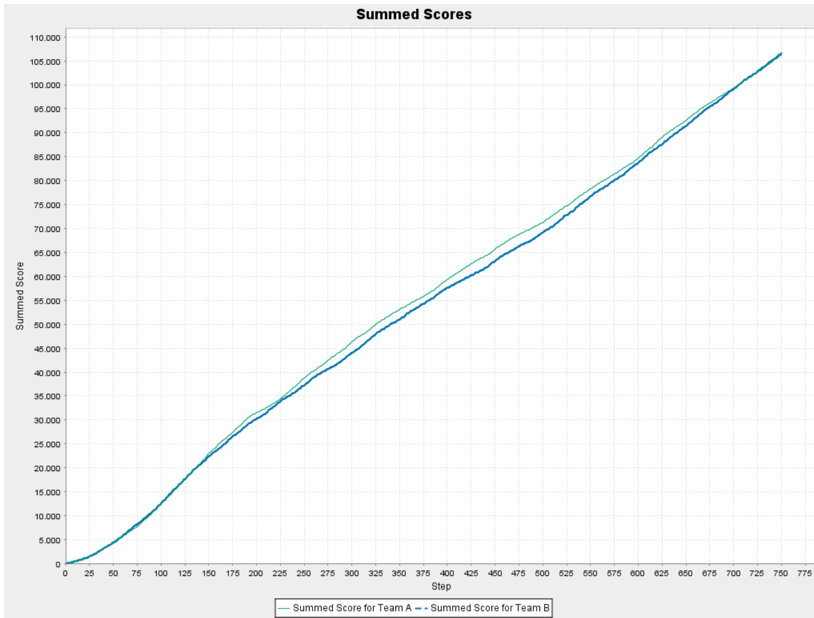
11.2 The multi-agent system

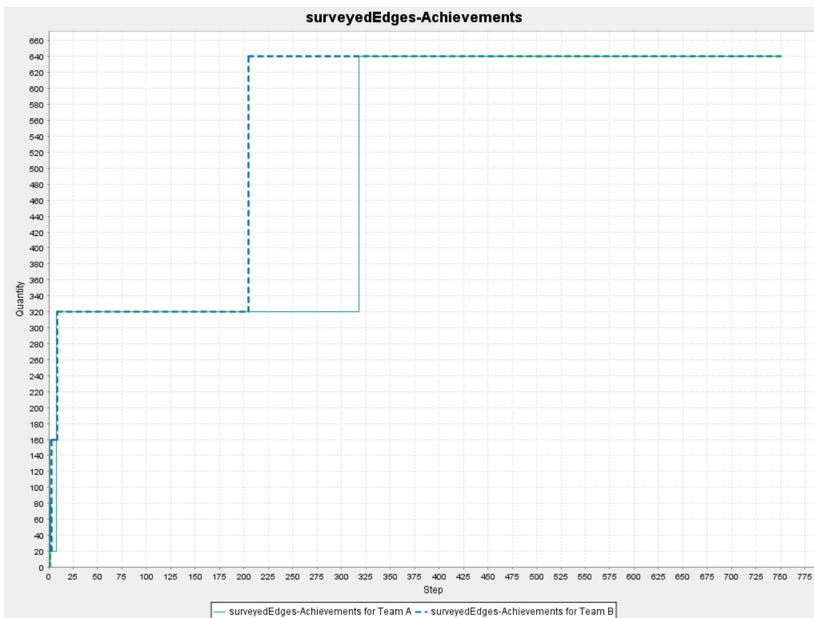
Our improved smadasUFSC system which we named JABARA works. The implemented new strategies for sentinels makes it a better system then the previous since it is able to win in scenarios with a high number of edges and make it a 1:1 win ratio in maps with low edge count. The improved saboteurs should also work, but we didn't get to test them, since we did not have a program to test it against in a series of simulations. We are a little sad that we didn't have sufficient understanding of the program and language to make the improved explorers and the region segregation, since we could have applied this to the sentinels also to further improve the speed of the surveying being done on the map.

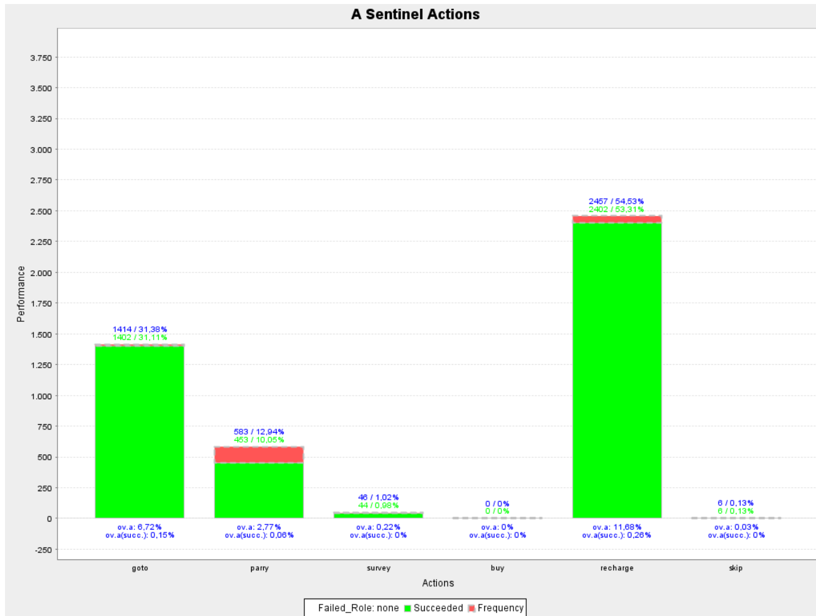
Appendix A

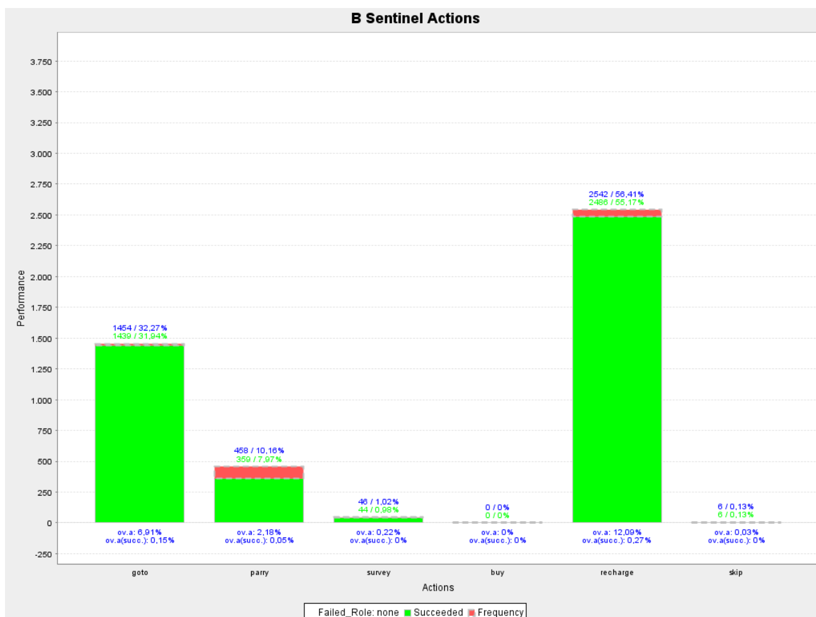
Test run 1

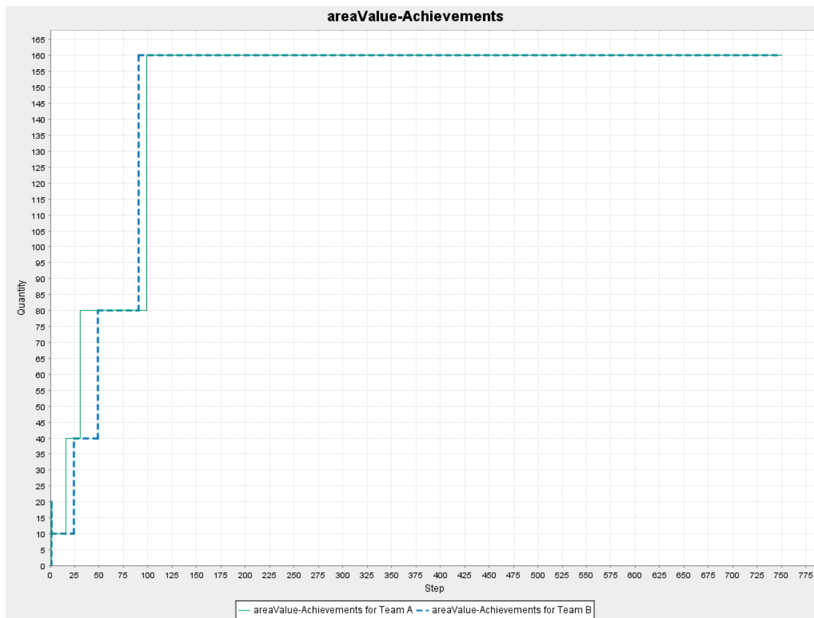
In these test runs the original program is displayed by A (green) and our program by B (blue)

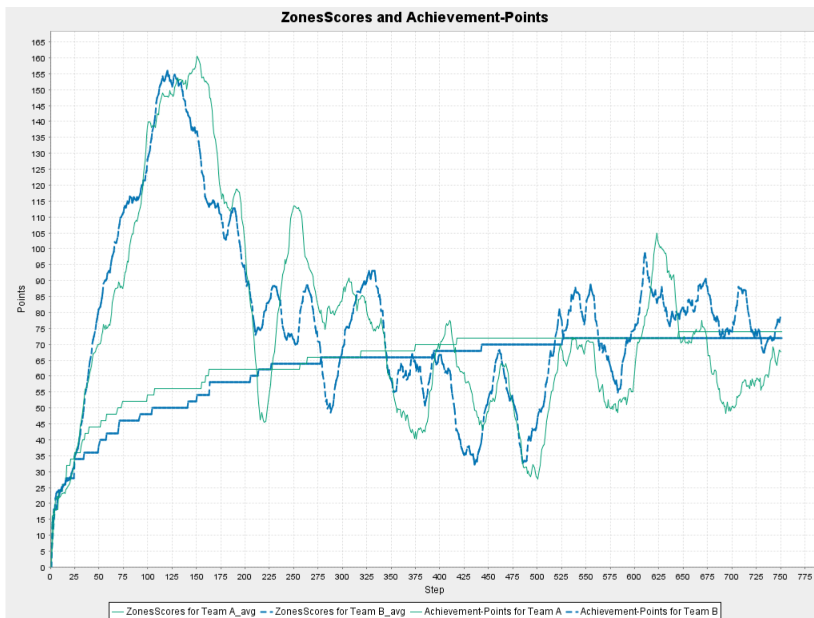

Summed Scores
Summed Score
110.000
105.000
100.000
95.000
90.000
85.000
80.000
75.000
70.000
65.000
60.000
55.000
50.000
45.000
40.000
35.000
30.000
25.000
20.000
15.000
10.000
5.000
0
0 25 50 75 100 125 150 175 200 225 250 275 300 325 350 375 400 425 450 475 500 525 550 575 600 625 650 675 700 725 750 775
Step
Summed Score for Team A
Summed Score for Team B

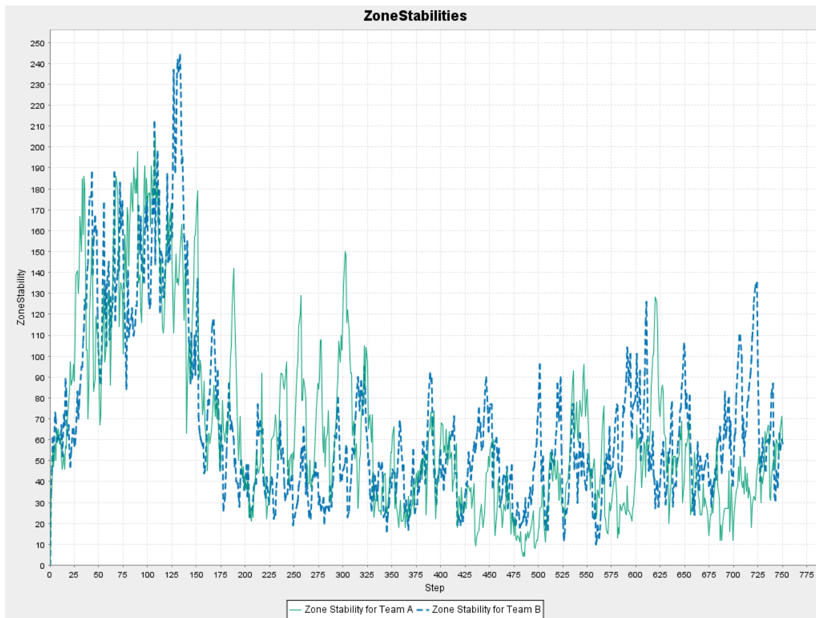

surveyedEdges-Achievements
Quantity
660
640
620
600
580
560
540
520
500
480
460
440
420
400
380
360
340
320
300
280
260
240
220
200
180
160
140
120
100
80
60
40
20
0
0 25 50 75 100 125 150 175 200 225 250 275 300 325 350 375 400 425 450 475 500 525 550 575 600 625 650 675 700 725 750 775
Step
surveyedEdges-Achievements for Team A
surveyedEdges-Achievements for Team B


A Sentinel Actions
Performance
3.750
3.500
3.250
3.000
2.750
2.500
2.250
2.000
1.750
1.500
1.250
1.000
750
500
250
0
-250
1414 / 31,38%
1402 / 31,11%
583 / 12,94%
453 / 10,05%
46 / 1,02%
44 / 0,98%
0 / 0%
0 / 0%
2457 / 54,53%
2402 / 53,31%
6 / 0,13%
6 / 0,13%
ov.a: 6,72%
ov.a(succ.): 0,15%
ov.a: 2,77%
ov.a(succ.): 0,06%
ov.a: 0,22%
ov.a(succ.): 0%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 11,68%
ov.a(succ.): 0,26%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none Succeeded Frequency


B Sentinel Actions
Performance
3.750
3.500
3.250
3.000
2.750
2.500
2.250
2.000
1.750
1.500
1.250
1.000
750
500
250
0
-250
1454 / 32,27%
1439 / 31,94%
458 / 10,16%
359 / 7,97%
46 / 1,02%
44 / 0,98%
0 / 0%
0 / 0%
2542 / 56,41%
2486 / 55,17%
6 / 0,13%
6 / 0,13%
ov.a: 6,91%
ov.a(succ.): 0,15%
ov.a: 2,18%
ov.a(succ.): 0,05%
ov.a: 0,22%
ov.a(succ.): 0%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 12,09%
ov.a(succ.): 0,27%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none Succeeded Frequency


areaValue-Achievements
Quantity
Step
areaValue-Achievements for Team A
areaValue-Achievements for Team B

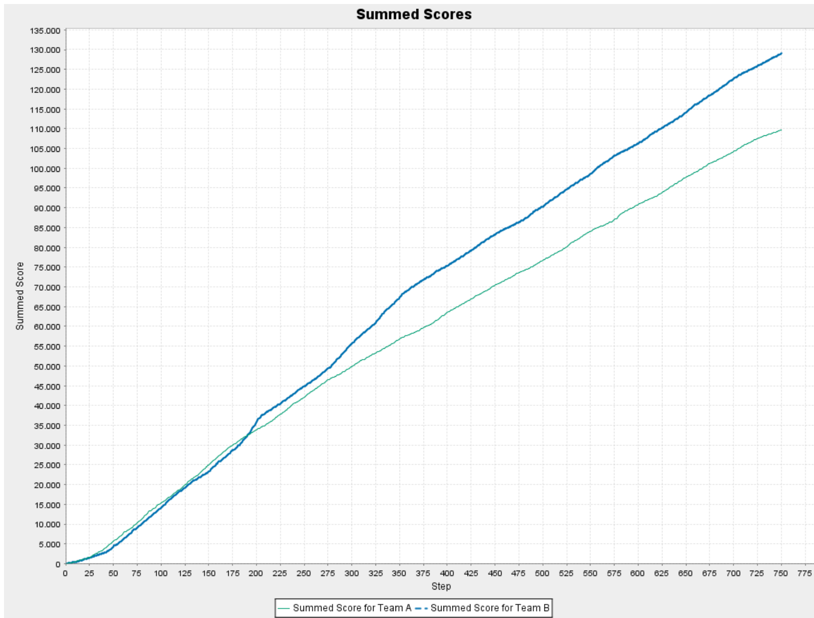

ZonesScores and Achievement-Points
Points
Step
ZonesScores for Team A_avg
ZonesScores for Team B_avg
Achievement-Points for Team A
Achievement-Points for Team B

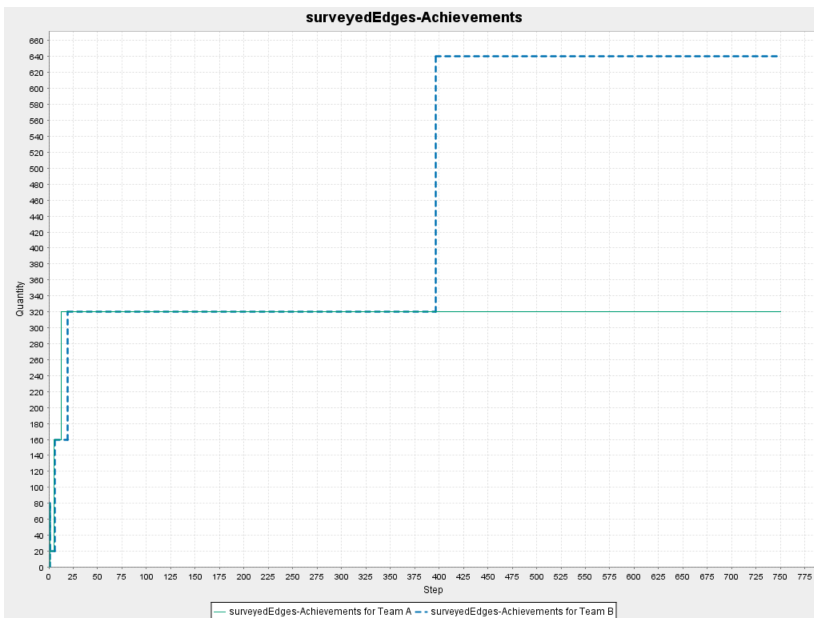

ZoneStabilities
ZoneStability
Step
Zone Stability for Team A
Zone Stability for Team B

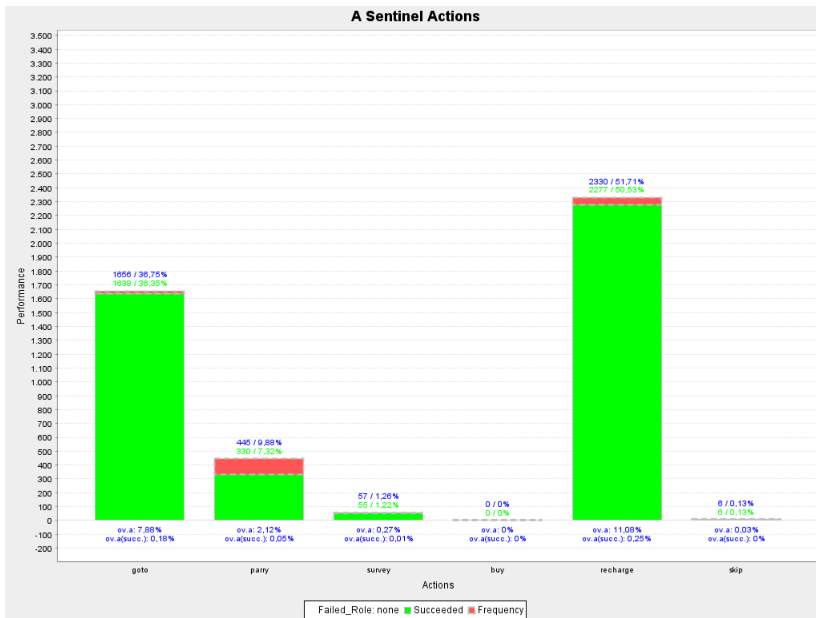
Appendix B

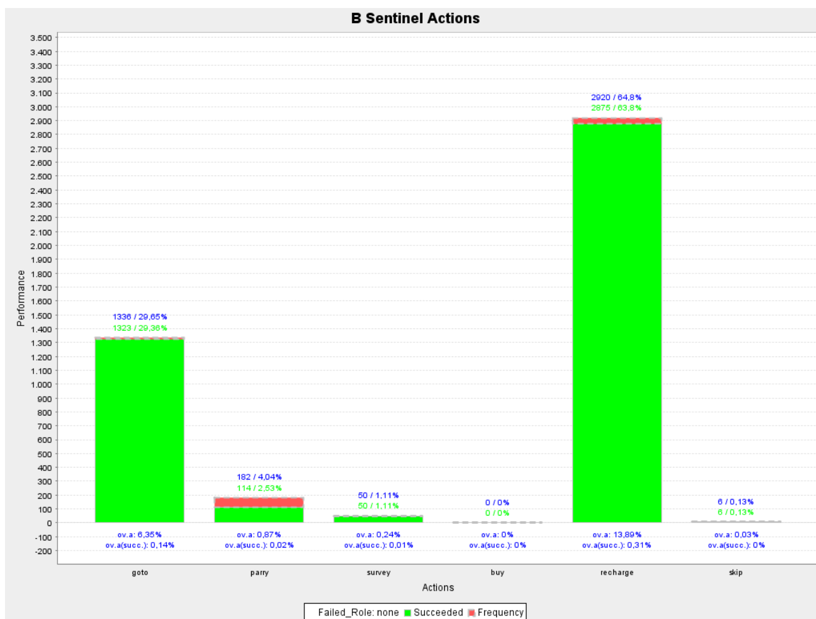
Test run 2

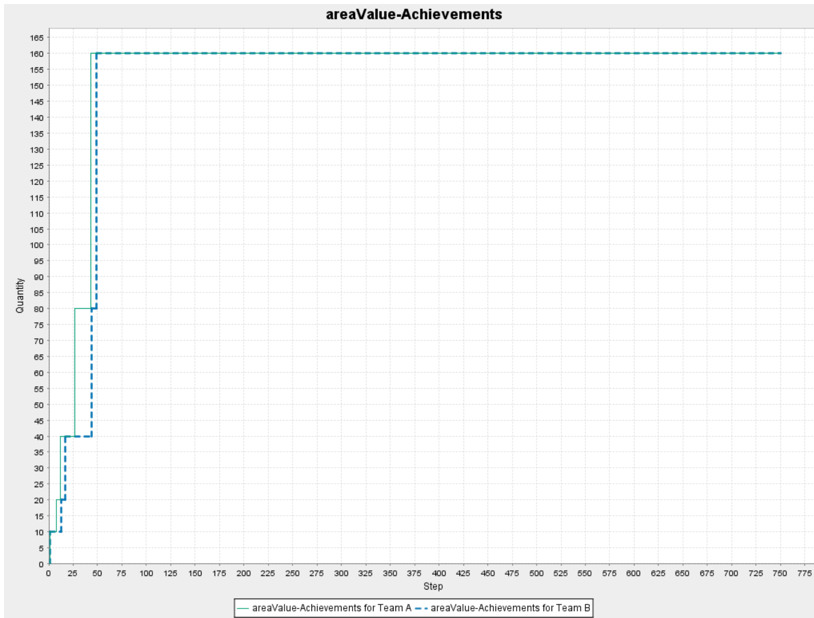
In these test runs the original program is displayed by A (green) and our program by B (blue)

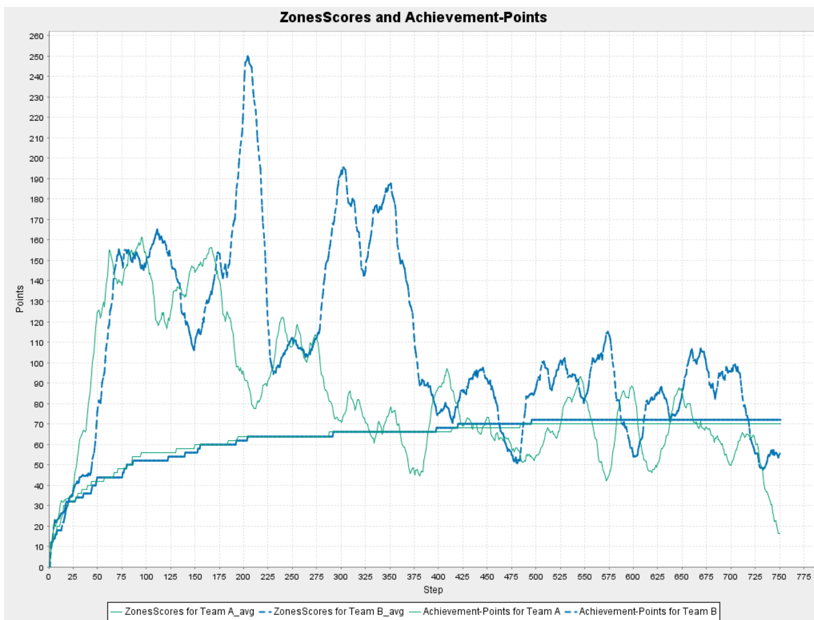

Summed Scores
Summed Score
Step
Summed Score for Team A
Summed Score for Team B

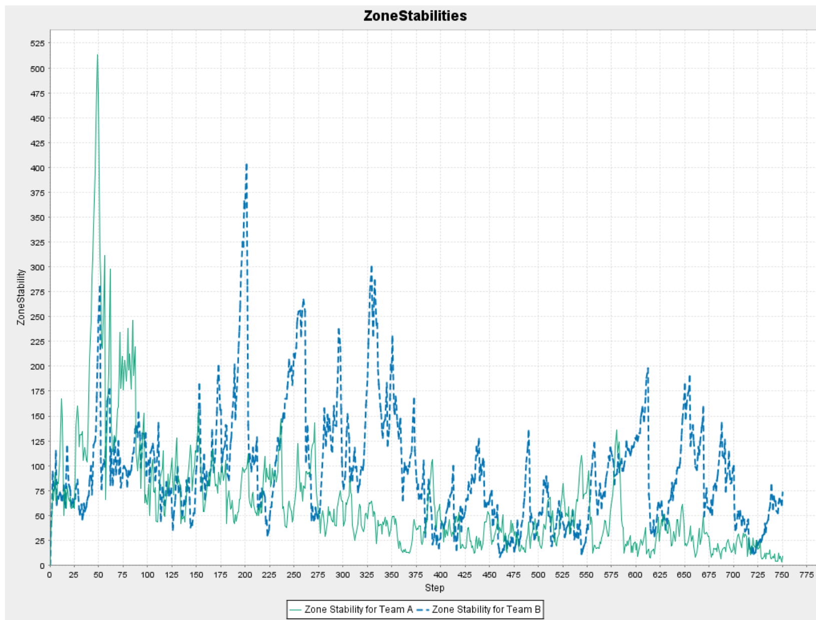

surveyedEdges-Achievements
Quantity
Step
surveyedEdges-Achievements for Team A
surveyedEdges-Achievements for Team B


A Sentinel Actions
Performance
Actions
1656 / 36,75%
1638 / 36,35%
445 / 9,88%
330 / 7,32%
57 / 1,26%
55 / 1,22%
0 / 0%
0 / 0%
2330 / 51,71%
2277 / 50,53%
6 / 0,13%
6 / 0,13%
ov.a: 7,88%
ov.a(succ.): 0,18%
ov.a: 2,12%
ov.a(succ.): 0,05%
ov.a: 0,27%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 11,08%
ov.a(succ.): 0,25%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Failed_Role: none
Succeeded
Frequency


B Sentinel Actions
Performance
Actions
1336 / 29,65%
1323 / 29,36%
182 / 4,04%
114 / 2,53%
50 / 1,11%
50 / 1,11%
0 / 0%
0 / 0%
2920 / 64,8%
2875 / 63,8%
6 / 0,13%
6 / 0,13%
ov.a: 6,35%
ov.a(succ.): 0,14%
ov.a: 0,87%
ov.a(succ.): 0,02%
ov.a: 0,24%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 13,89%
ov.a(succ.): 0,31%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Failed_Role: none
Succeeded
Frequency


areaValue-Achievements
Quantity
Step
areaValue-Achievements for Team A
areaValue-Achievements for Team B

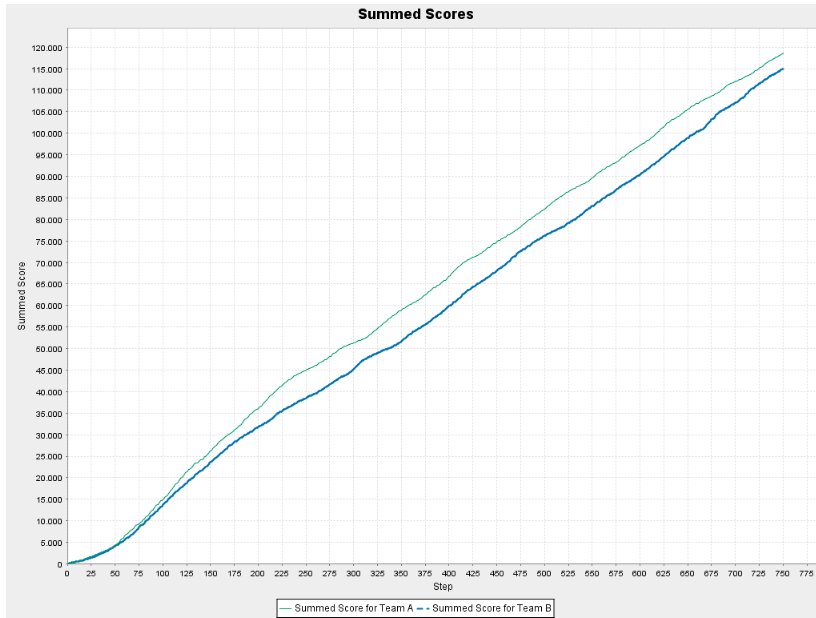

ZonesScores and Achievement-Points
Points
Step
ZonesScores for Team A_avg
ZonesScores for Team B_avg
Achievement-Points for Team A
Achievement-Points for Team B

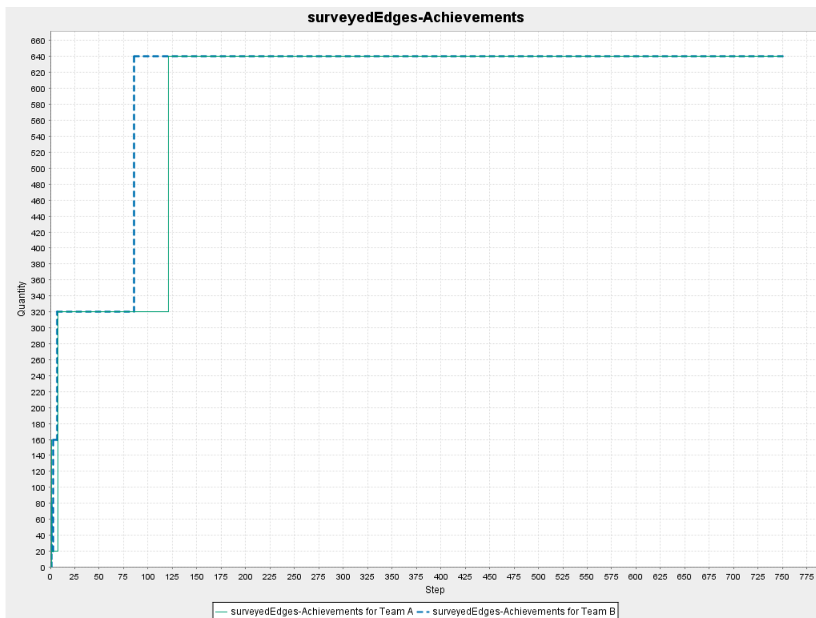

ZoneStabilities
ZoneStability
Step
Zone Stability for Team A
Zone Stability for Team B

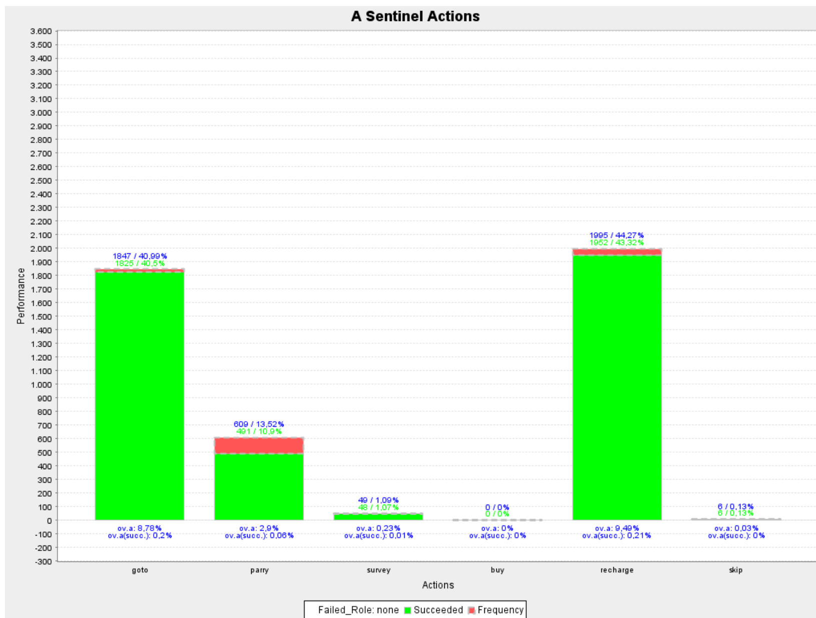
Appendix C

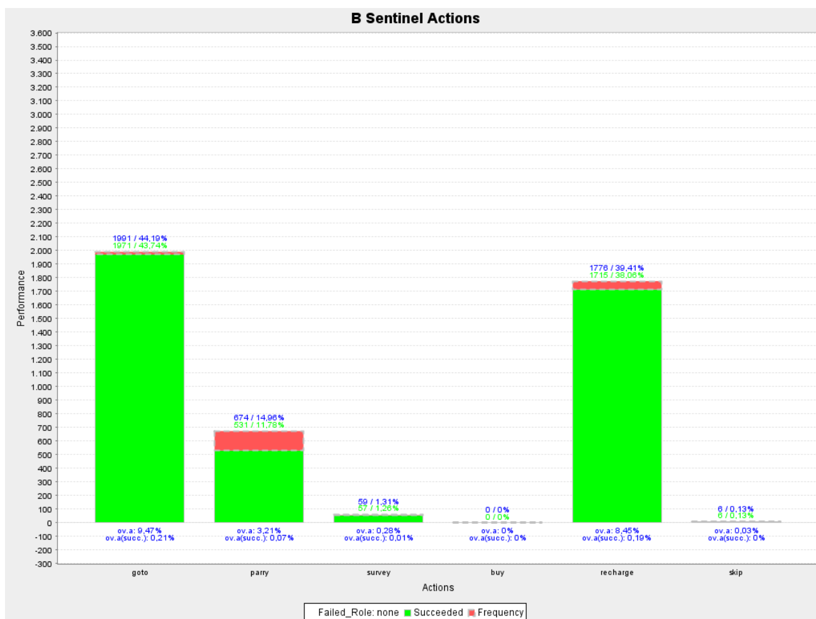
Test run 3

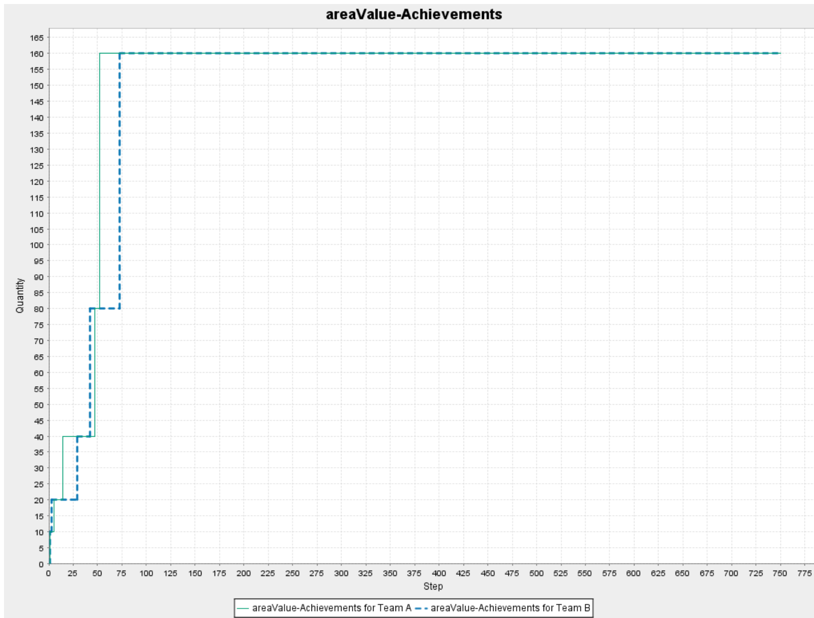
In these test runs the original program is displayed by A (green) and our program by B (blue)

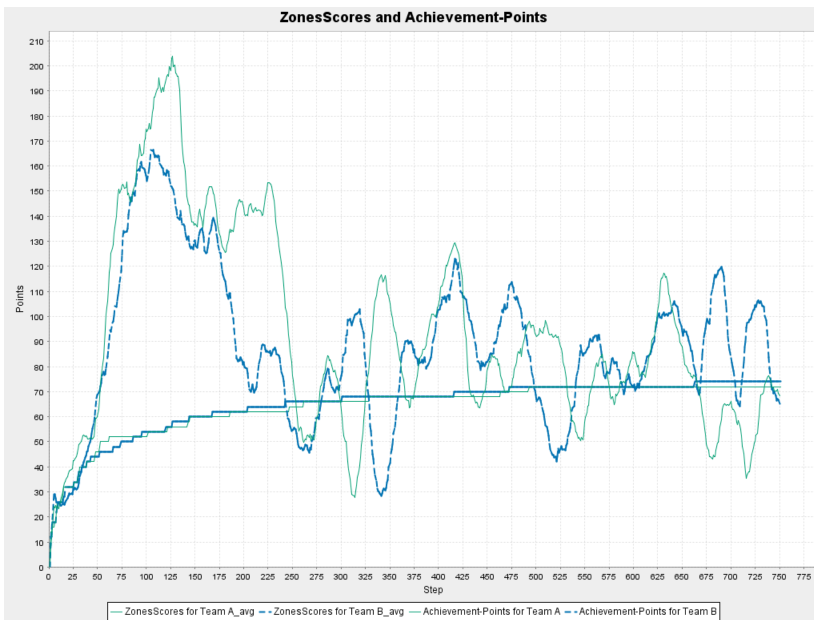

Summed Scores
Summed Score
Step
Summed Score for Team A
Summed Score for Team B

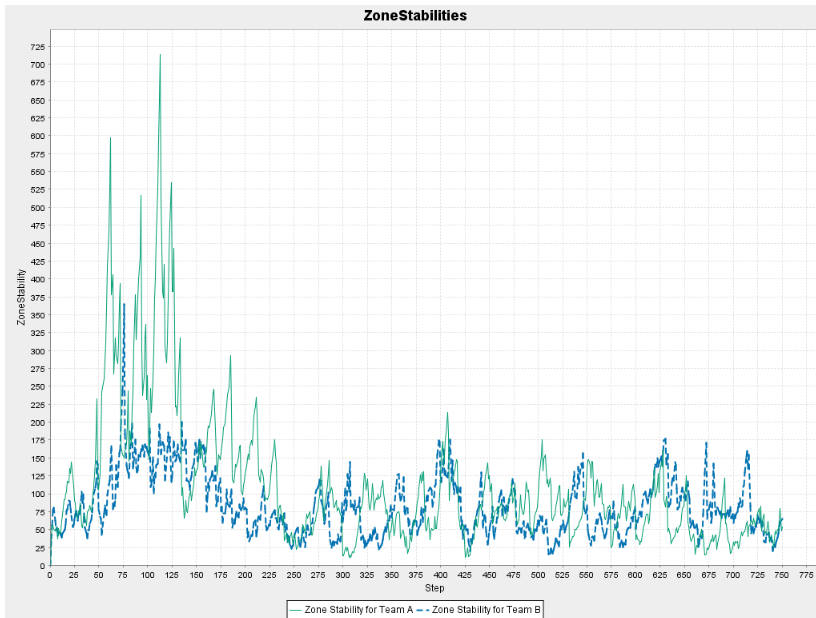

surveyedEdges-Achievements
Quantity
Step
surveyedEdges-Achievements for Team A
surveyedEdges-Achievements for Team B


A Sentinel Actions
Performance
3.600
3.500
3.400
3.300
3.200
3.100
3.000
2.900
2.800
2.700
2.600
2.500
2.400
2.300
2.200
2.100
2.000
1.900
1.800
1.700
1.600
1.500
1.400
1.300
1.200
1.100
1.000
900
800
700
600
500
400
300
200
100
0
-100
-200
-300
1847 / 40,99%
1825 / 40,5%
609 / 13,52%
491 / 10,9%
49 / 1,09%
48 / 1,07%
0 / 0%
0 / 0%
1995 / 44,27%
1952 / 43,32%
6 / 0,13%
6 / 0,13%
ov.a: 8,78%
ov.a(succ.): 0,2%
ov.a: 2,9%
ov.a(succ.): 0,06%
ov.a: 0,23%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 9,49%
ov.a(succ.): 0,21%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


B Sentinel Actions
Performance
3.600
3.500
3.400
3.300
3.200
3.100
3.000
2.900
2.800
2.700
2.600
2.500
2.400
2.300
2.200
2.100
2.000
1.900
1.800
1.700
1.600
1.500
1.400
1.300
1.200
1.100
1.000
900
800
700
600
500
400
300
200
100
0
-100
-200
-300
1991 / 44,19%
1971 / 43,74%
674 / 14,96%
531 / 11,78%
59 / 1,31%
57 / 1,26%
0 / 0%
0 / 0%
1776 / 39,41%
1715 / 38,06%
6 / 0,13%
6 / 0,13%
ov.a: 9,47%
ov.a(succ.): 0,21%
ov.a: 3,21%
ov.a(succ.): 0,07%
ov.a: 0,28%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 8,45%
ov.a(succ.): 0,19%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


areaValue-Achievements
Quantity
Step
areaValue-Achievements for Team A
areaValue-Achievements for Team B

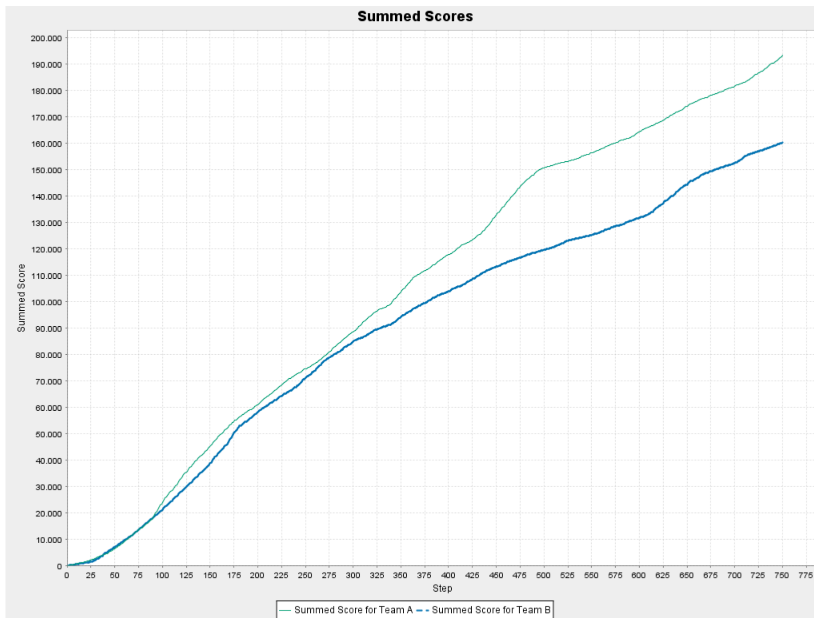

ZonesScores and Achievement-Points
Points
Step
ZonesScores for Team A_avg
ZonesScores for Team B_avg
Achievement-Points for Team A
Achievement-Points for Team B

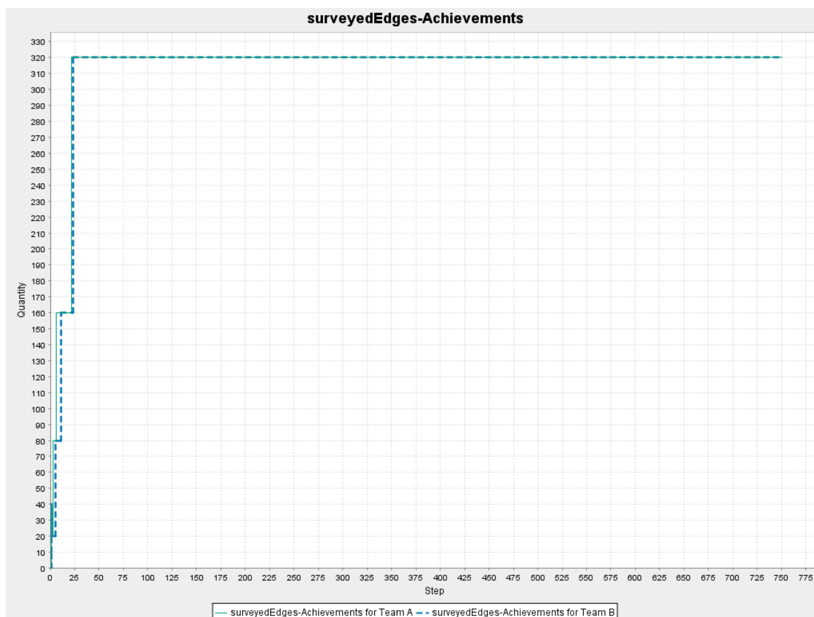

ZoneStabilities
ZoneStability
Step
Zone Stability for Team A
Zone Stability for Team B

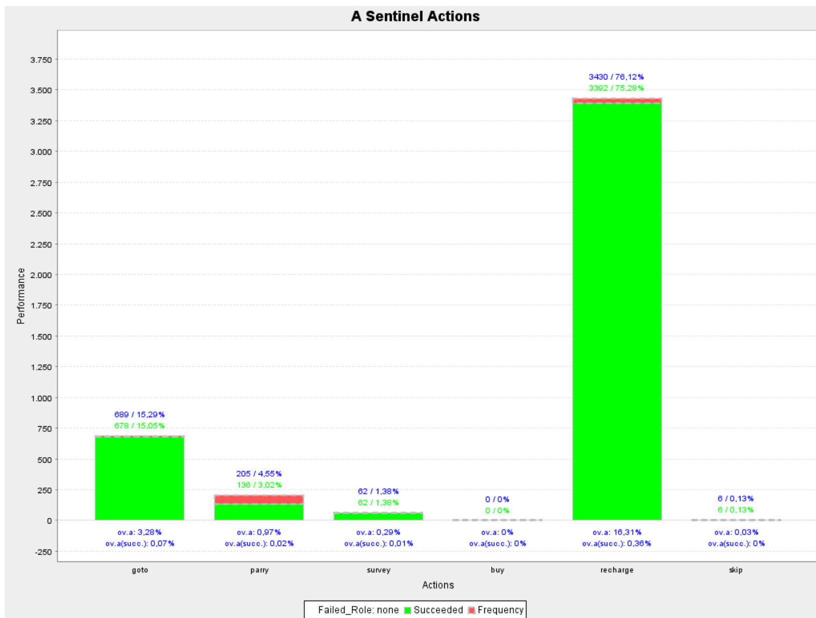
Appendix D

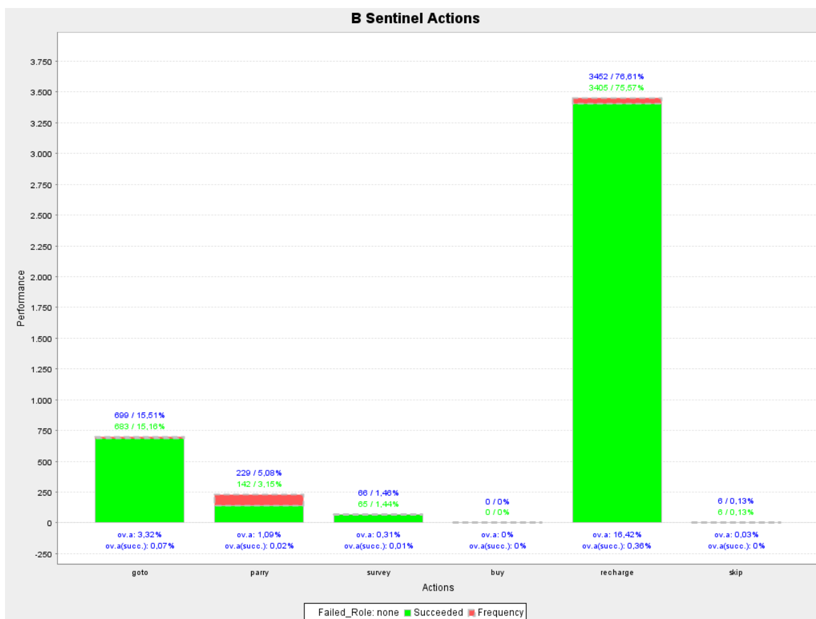
Test run 4

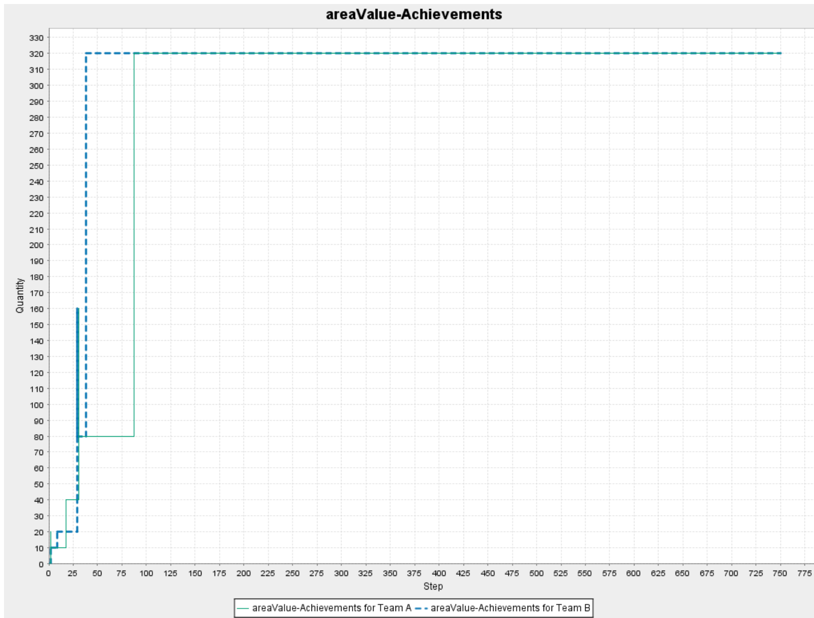
In these test runs the original program is displayed by A (green) and our program by B (blue)

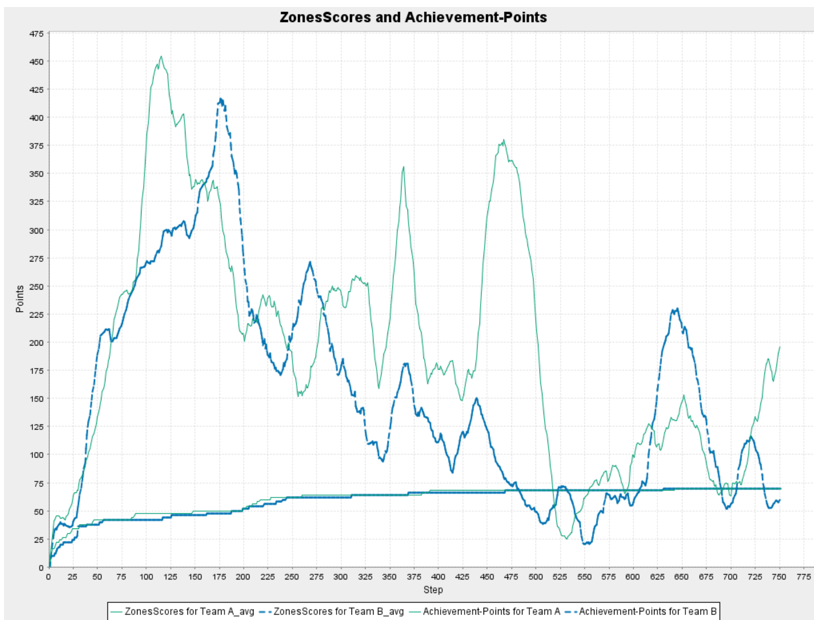

Summed Scores
Summed Score
Step
Summed Score for Team A
Summed Score for Team B

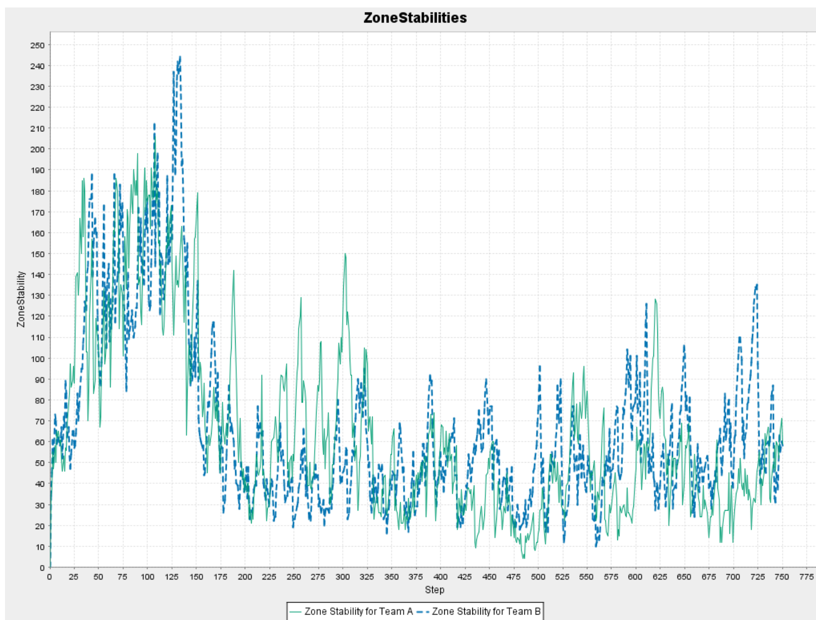

surveyedEdges-Achievements
Quantity
Step
surveyedEdges-Achievements for Team A
surveyedEdges-Achievements for Team B


A Sentinel Actions
Performance
3.750
3.500
3.250
3.000
2.750
2.500
2.250
2.000
1.750
1.500
1.250
1.000
750
500
250
0
-250
3430 / 76,12%
3392 / 75,28%
689 / 15,29%
678 / 15,05%
205 / 4,55%
136 / 3,02%
62 / 1,38%
62 / 1,38%
0 / 0%
0 / 0%
6 / 0,13%
6 / 0,13%
ov.a: 3,28%
ov.a(succ.): 0,07%
ov.a: 0,97%
ov.a(succ.): 0,02%
ov.a: 0,29%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 16,31%
ov.a(succ.): 0,36%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


B Sentinel Actions
Performance
3.750
3.500
3.250
3.000
2.750
2.500
2.250
2.000
1.750
1.500
1.250
1.000
750
500
250
0
-250
3452 / 76,61%
3405 / 75,57%
699 / 15,51%
683 / 15,16%
229 / 5,08%
142 / 3,15%
66 / 1,46%
65 / 1,44%
0 / 0%
0 / 0%
6 / 0,13%
6 / 0,13%
ov.a: 3,32%
ov.a(succ.): 0,07%
ov.a: 1,09%
ov.a(succ.): 0,02%
ov.a: 0,31%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 16,42%
ov.a(succ.): 0,36%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


areaValue-Achievements
Quantity
Step
areaValue-Achievements for Team A
areaValue-Achievements for Team B

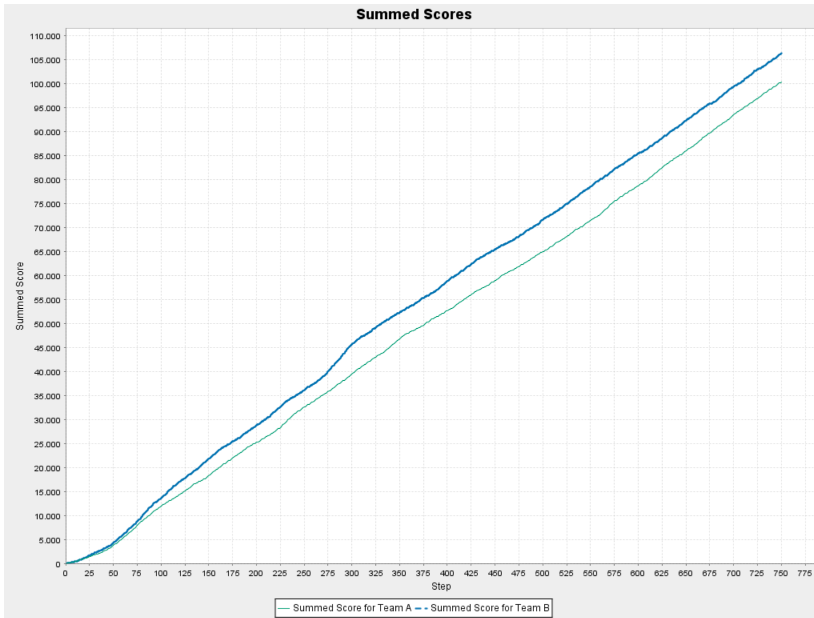

ZonesScores and Achievement-Points
Points
Step
ZonesScores for Team A_avg
ZonesScores for Team B_avg
Achievement-Points for Team A
Achievement-Points for Team B

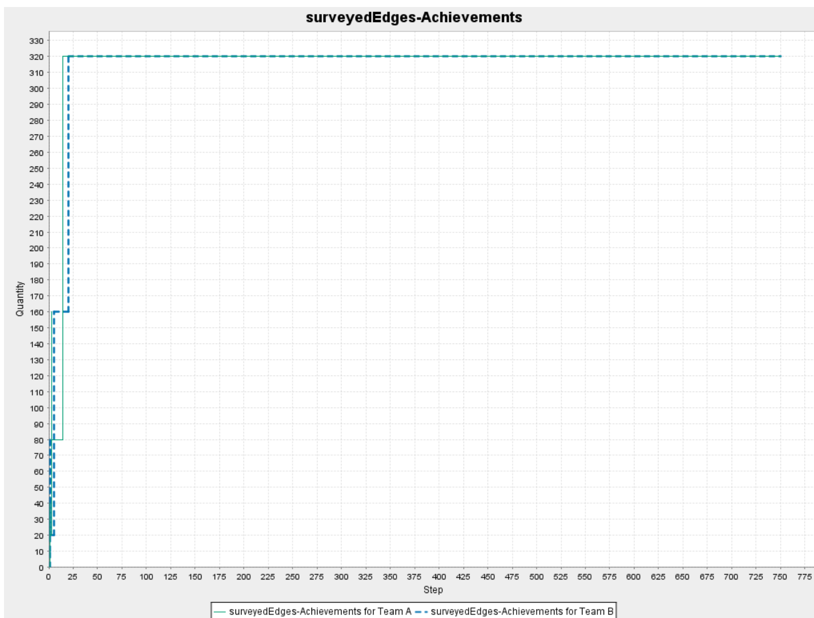

ZoneStabilities
ZoneStability
Step
Zone Stability for Team A
Zone Stability for Team B

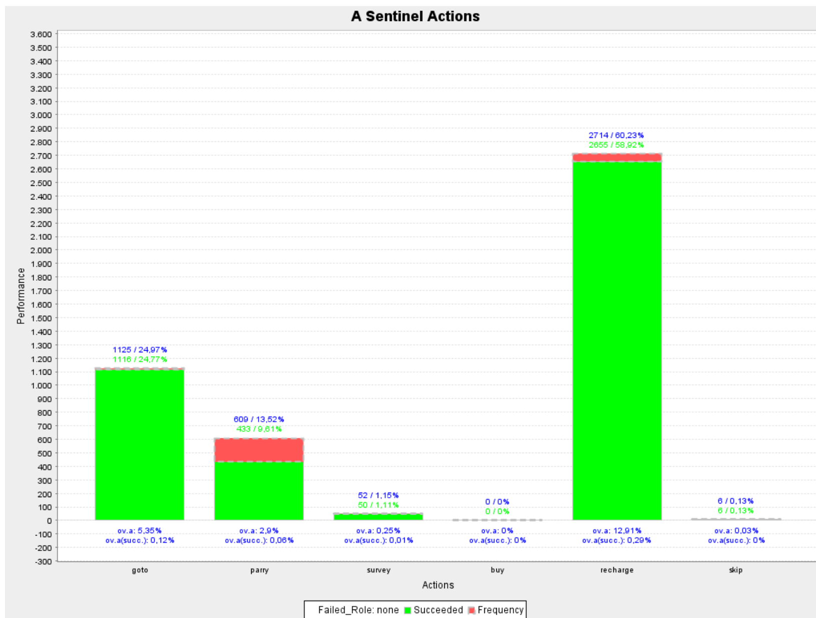
Appendix E

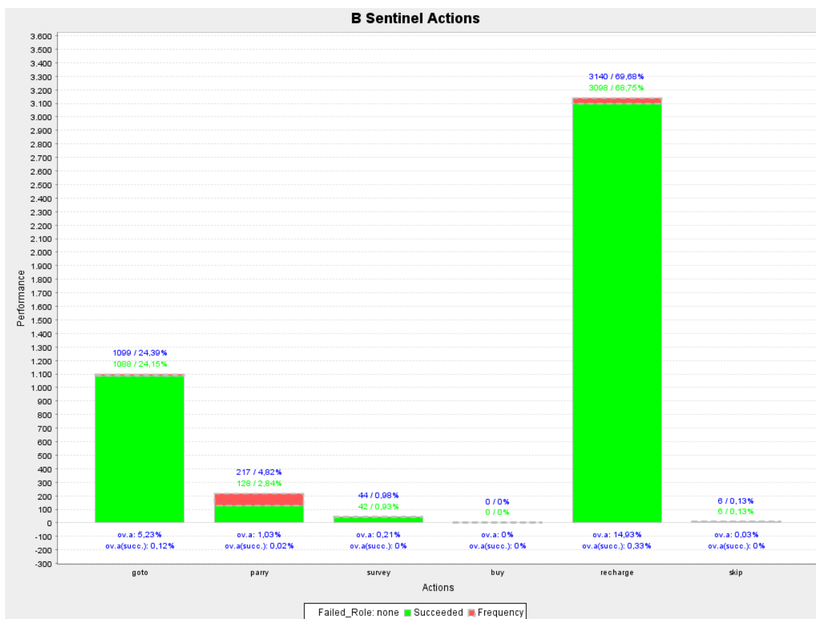
Test run 5

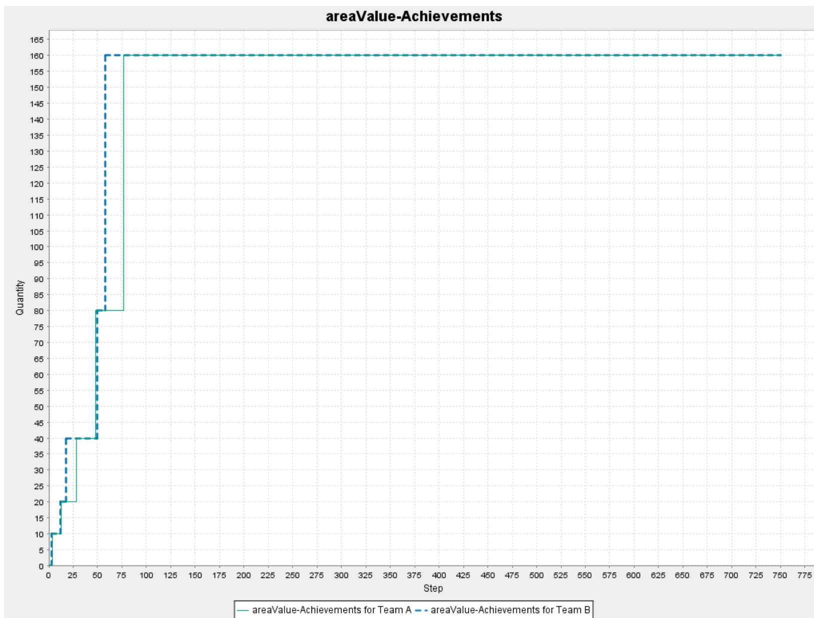
In these test runs the original program is displayed by A (green) and our program by B (blue)

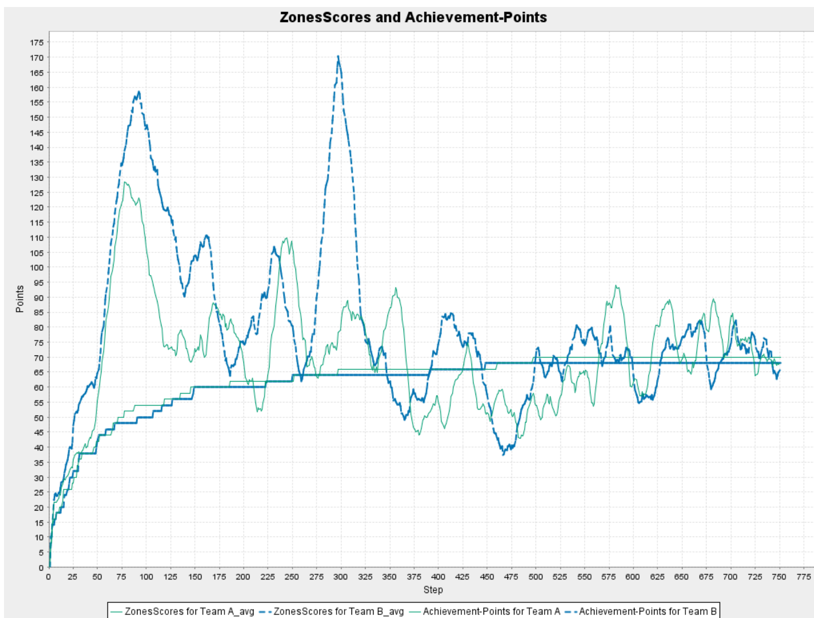

Summed Scores
Summed Score
Step
Summed Score for Team A
Summed Score for Team B

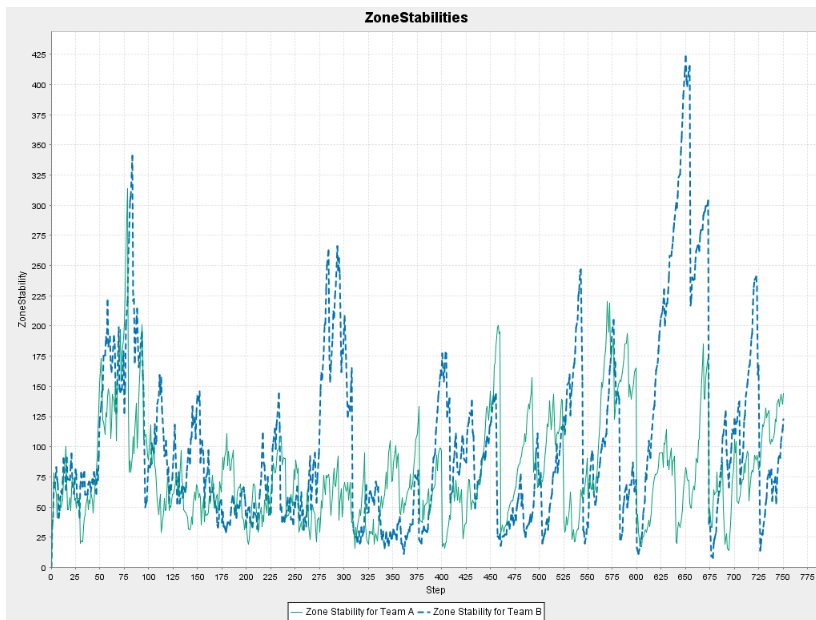

surveyedEdges-Achievements
Quantity
Step
surveyedEdges-Achievements for Team A
surveyedEdges-Achievements for Team B


A Sentinel Actions
Performance
3.600
3.500
3.400
3.300
3.200
3.100
3.000
2.900
2.800
2.700
2.600
2.500
2.400
2.300
2.200
2.100
2.000
1.900
1.800
1.700
1.600
1.500
1.400
1.300
1.200
1.100
1.000
900
800
700
600
500
400
300
200
100
0
-100
-200
-300
1125 / 24,97%
1116 / 24,77%
609 / 13,52%
433 / 9,61%
52 / 1,15%
50 / 1,11%
0 / 0%
0 / 0%
2714 / 60,23%
2655 / 58,92%
6 / 0,13%
6 / 0,13%
ov.a: 5,35%
ov.a(succ.): 0,12%
ov.a: 2,9%
ov.a(succ.): 0,06%
ov.a: 0,25%
ov.a(succ.): 0,01%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 12,91%
ov.a(succ.): 0,29%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


B Sentinel Actions
Performance
3.600
3.500
3.400
3.300
3.200
3.100
3.000
2.900
2.800
2.700
2.600
2.500
2.400
2.300
2.200
2.100
2.000
1.900
1.800
1.700
1.600
1.500
1.400
1.300
1.200
1.100
1.000
900
800
700
600
500
400
300
200
100
0
-100
-200
-300
3140 / 69,68%
3098 / 68,75%
1099 / 24,39%
1088 / 24,15%
217 / 4,82%
128 / 2,84%
44 / 0,98%
42 / 0,93%
0 / 0%
0 / 0%
6 / 0,13%
6 / 0,13%
ov.a: 5,23%
ov.a(succ.): 0,12%
ov.a: 1,03%
ov.a(succ.): 0,02%
ov.a: 0,21%
ov.a(succ.): 0%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 14,93%
ov.a(succ.): 0,33%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


areaValue-Achievements
Quantity
Step
areaValue-Achievements for Team A
areaValue-Achievements for Team B

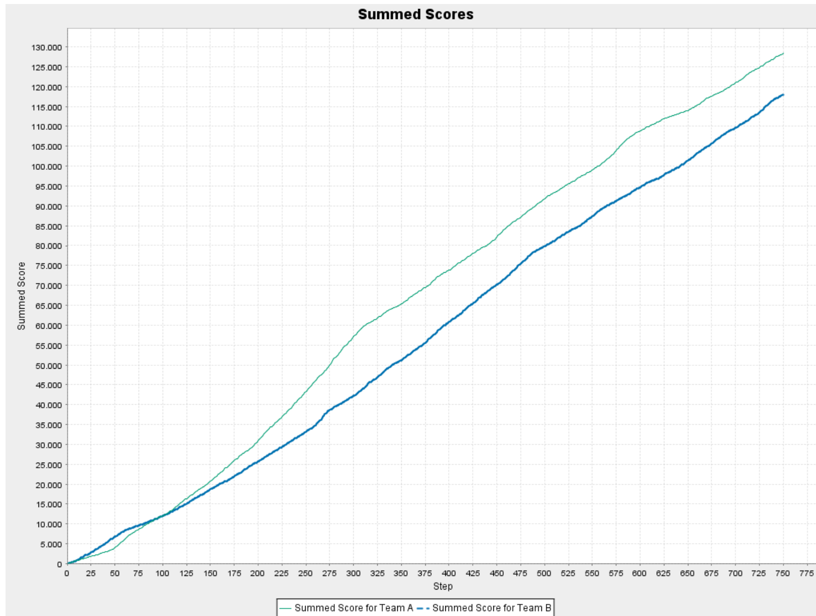

ZonesScores and Achievement-Points
Points
Step
ZonesScores for Team A_avg
ZonesScores for Team B_avg
Achievement-Points for Team A
Achievement-Points for Team B

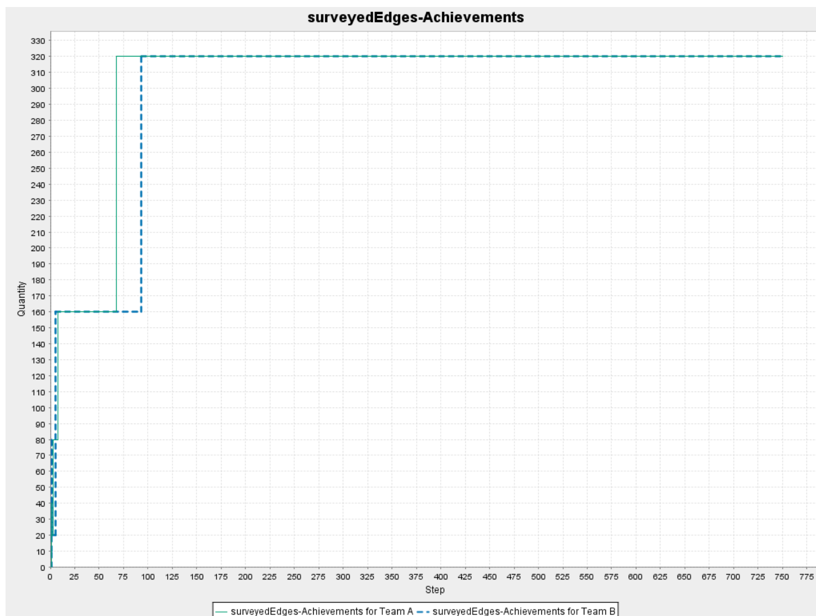

ZoneStabilities
ZoneStability
Step
0
25
50
75
100
125
150
175
200
225
250
275
300
325
350
375
400
425
450
475
500
525
550
575
600
625
650
675
700
725
750
775
Zone Stability for Team A
Zone Stability for Team B

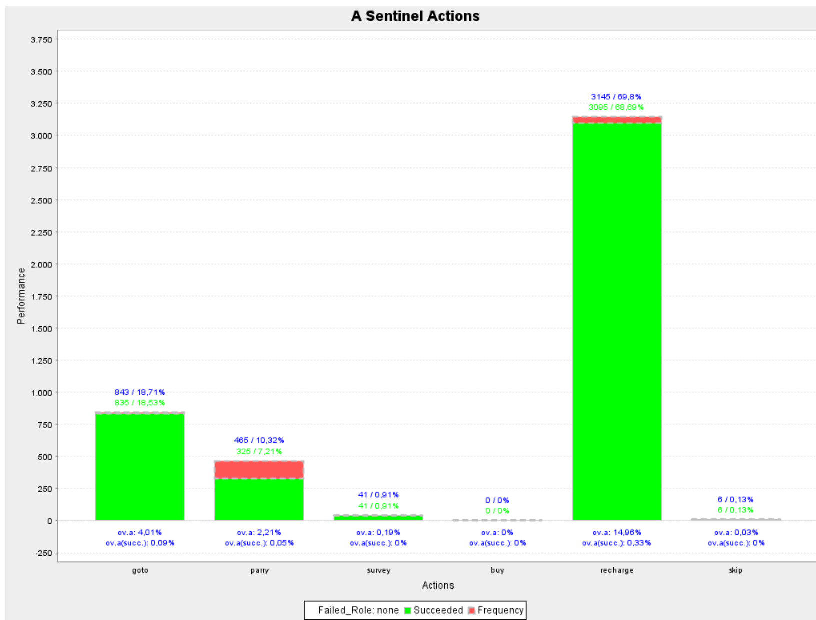
Appendix F

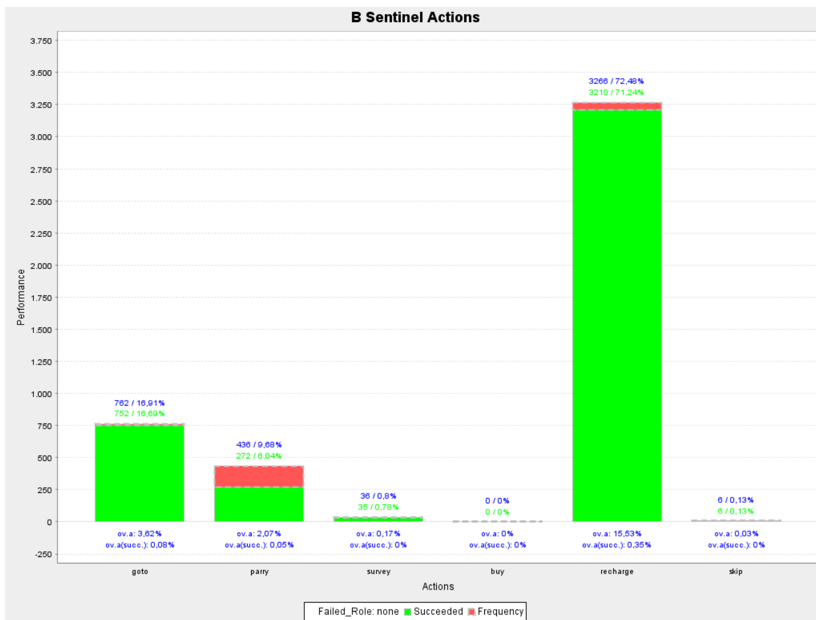
Test run 6

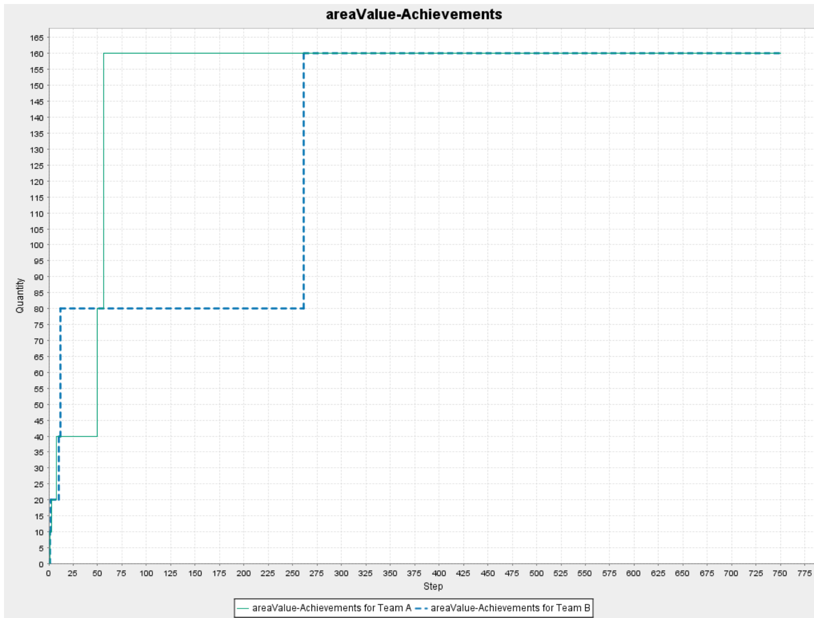
In these test runs the original program is displayed by A (green) and our program by B (blue)

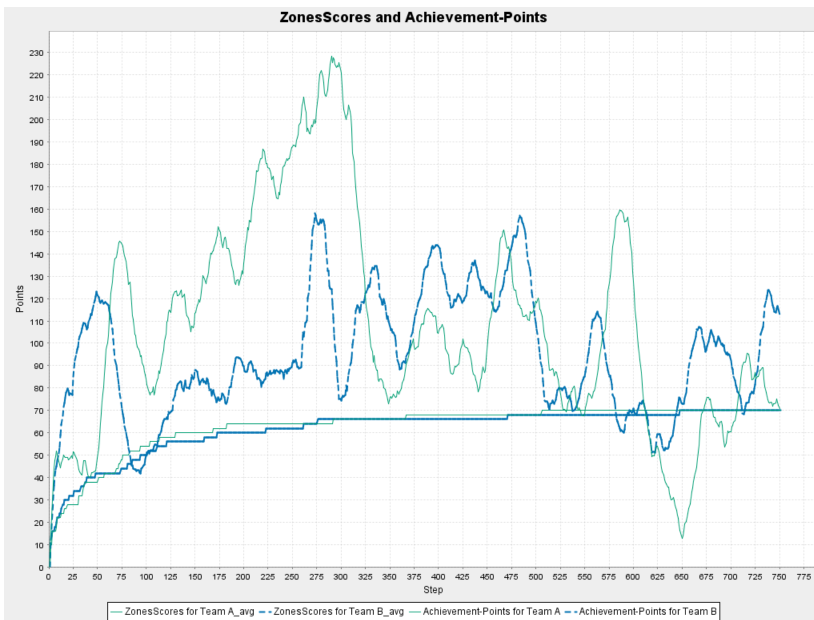

Summed Scores
Summed Score
Step
Summed Score for Team A
Summed Score for Team B

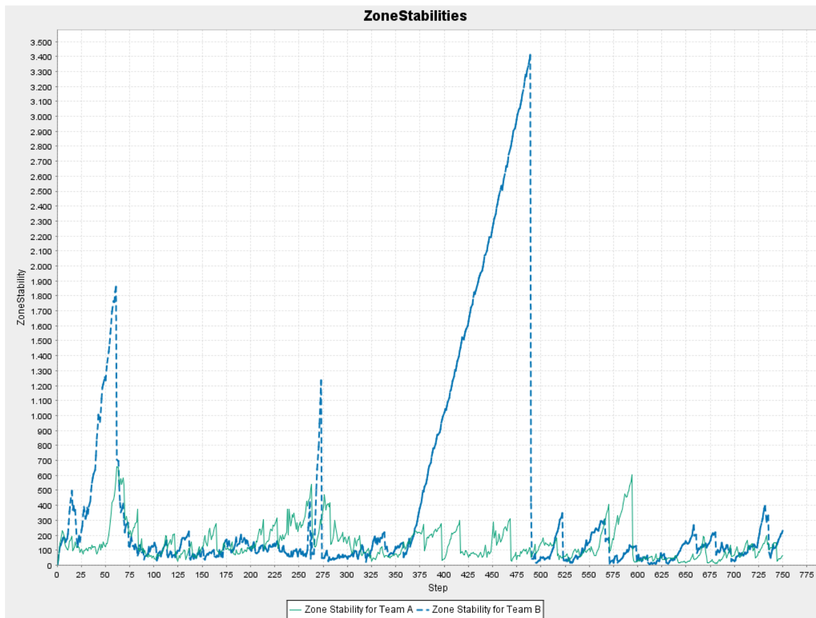

surveyedEdges-Achievements
Quantity
Step
surveyedEdges-Achievements for Team A
surveyedEdges-Achievements for Team B


A Sentinel Actions
Performance
3.750
3.500
3.250
3.000
2.750
2.500
2.250
2.000
1.750
1.500
1.250
1.000
750
500
250
0
-250
843 / 18,71%
835 / 18,53%
465 / 10,32%
325 / 7,21%
41 / 0,91%
41 / 0,91%
0 / 0%
0 / 0%
3145 / 69,8%
3095 / 68,69%
6 / 0,13%
6 / 0,13%
ov.a: 4,01%
ov.a(succ.): 0,09%
ov.a: 2,21%
ov.a(succ.): 0,05%
ov.a: 0,19%
ov.a(succ.): 0%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 14,96%
ov.a(succ.): 0,33%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


B Sentinel Actions
Performance
3.750
3.500
3.250
3.000
2.750
2.500
2.250
2.000
1.750
1.500
1.250
1.000
750
500
250
0
-250
762 / 16,91%
752 / 16,69%
436 / 9,68%
272 / 6,04%
36 / 0,8%
35 / 0,78%
0 / 0%
0 / 0%
3266 / 72,48%
3210 / 71,24%
6 / 0,13%
6 / 0,13%
ov.a: 3,62%
ov.a(succ.): 0,08%
ov.a: 2,07%
ov.a(succ.): 0,05%
ov.a: 0,17%
ov.a(succ.): 0%
ov.a: 0%
ov.a(succ.): 0%
ov.a: 15,53%
ov.a(succ.): 0,35%
ov.a: 0,03%
ov.a(succ.): 0%
goto
parry
survey
buy
recharge
skip
Actions
Failed_Role: none
Succeeded
Frequency


areaValue-Achievements
Quantity
Step
areaValue-Achievements for Team A
areaValue-Achievements for Team B


ZonesScores and Achievement-Points
Points
Step
ZonesScores for Team A_avg
ZonesScores for Team B_avg
Achievement-Points for Team A
Achievement-Points for Team B


ZoneStabilities
ZoneStability
Step
Zone Stability for Team A
Zone Stability for Team B

Bibliography

[1] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[2] Andreas Viktor Hess and Øyvind Grønland Woller. Multi-agent systems and agent-oriented programming, 2013.

[3] Jomi F. Hübner Oliver Boissier, Rafael H. Bordini, Alessandro Ricci, and Andrea Santi. The jacamo project, `http://jacamo.sourceforge.net/`.

[4] Federico Schlesinger Tobias Ahlbrecht, Jürgen Dix. Multi agent programming contest scenario, `https://multiagentcontest.org/downloads/func-startdown/1663/`.

[5] Maicon Rafael Zatelli, Daniela Maria Uez, Maiquel De Brito, Jomi Fred Hübner, Tiago Luiz Schmitz, Kaio Siqueira De Souza, and Marcelo Menezes Morato. Smadas: A team for mapc considering the organization and the environment as first-class abstractions. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8245:319–328, 2013.