TU Clausthal

Clausthal University of Technology

Multi-Agent Programming Contest 2011 Edition Evaluation and Team Descriptions

Tristan Behrens, Jürgen Dix, Michael Köster, Federico Schlesinger

IfI Technical Report Series IfI-12-02

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Federico Schlesinger
Contact: federico.schlesinger@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Multi-Agent Programming Contest
2011 Edition
Evaluation and Team Descriptions

Tristan Behrens, Jürgen Dix, Michael Köster, Federico Schlesinger

Abstract

The Multi-Agent Programming Contest is an annual competition in agent-based artificial intelligence. The year 2011 marked the beginning of a new phase with the introduction of the Agents-on-Mars-scenario. The focus was shifted towards heterogeneous multi-agent systems with both competitive and cooperative agent interaction. On top of that, new means for evaluating the performance of individual agents and whole agent teams were designed and established. In this document we will provide a systematic, statistical evaluation of the 2011 tournament. In a second part we will also present in-depth descriptions of the participating teams.

Part I
Contest Evaluation

1 Introduction

In this Technical Report, we give a comprehensive evaluation of the results of the Multi-Agent Programming Contest[1] 2011 edition. The Contest is an annual international event that started in 2005. In 2011 the competition was organized and held for the seventh time. The Contest is an attempt to stimulate research in the field of programming multi-agent system by

1. identifying key problems,
2. collecting suitable benchmarks, and
3. gathering test cases which require and enforce coordinated action that can serve as milestones for testing multi-agent programming languages, platforms and tools.

[1] `http://multi-agentcontest.org`

Research communities benefit from competitions that (1) attempt to evaluate different aspects of the systems under consideration, (2) allow for comparing state of the art systems, (3) act as a driver and catalyst for developments, and (4) define challenging research problems.

In this report we extend the work presented in [Behrens et al., 2012b], by focusing on the outcomes of the Contest. The document is organized as follows: We firstly mention some related work in Section 2, and then briefly describe the Contest in section 2. In Section 4 a short description of each of the participating teams is provided. Section 5 presents the main contributions of this paper, with an in-depth analysis of the results of the Contest. Finally we present a conclusion and future work in section 6.

Part II of this document presents the in-depth team descriptions provided by team developers themselves. These descriptions follow a template provided by the Multi-Agent Programming Contest organization (a requirement for the participation).

2 Related Work

The Multi-Agent Programming Contest has generated several publications in the recent years [Dastani et al., 2005, Dastani et al., 2006b, Dastani et al., 2008a, Dastani et al., 2008b, Behrens et al., 2009, Behrens et al., 2010]. Similar contests, competitions and challenges are *Google's AI challenge*[2], the *AI-MAS Winter Olympics*[3], the *Starcraft AI Competition*[4], the *Mario AI Championship*[5], the *ORTS competition*[6], and the *Planning Competition*[7]. All these competition are defined in their own research niches. Our Contest has been designed for problem solving approaches that are based on formal approaches and computational logics, thus distinguishing it from the other competitions.

3 The Multi-Agent Programming Contest

The Multi-Agent Programming Contest was initiated in 2005 and since then went through three distinct phases. The first phase began in 2005 with the "food-gatherers"-scenario, where a pre-specified multi-agent system had to be implemented. These MASs were later examined in order to determine the winner. From 2006 - 2007 the "goldminers"-scenario was used. This time it

[2] http://aichallenge.org/
[3] http://www.aiolympics.ro/
[4] http://eis.ucsc.edu/StarCraftAICompetition
[5] http://www.marioai.org/
[6] http://skatgame.net/mburo/orts/
[7] http://ipc.icaps-conference.org/

is provided an environment by means of an online-architecture, and automatically determined the winner. Then from 2008 - 2010 we ran the "cows and cowboys"-scenario, again on the same online-architecture.

We noticed that most approaches used in the agent contest in the last years were *centralized*, contrary to the philosophy of multi-agent programming (MAP). Even the accumulated knowledge of the agents was maintained centrally and shared by internal communication. This aspect has motivated the definition of a new scenario, which is described next.

3.1 The 2011 Scenario

In this year's Contest (for a detailed description see [Behrens et al., 2012a]) the participants have to compete in an environment that is constituted by a graph where the vertices have an unique identifier and also a number that determines the value of that vertex. The weights of the edges on the other hand denotes the costs of traversing the edge.

A *zone* is a subgraph (with at least two nodes) whose vertices are colored by a specific graph coloring algorithm. If the vertices of a zone are colored with a certain team color it is said that this team occupies this area. The value of a zone is determined by the sum of its vertices' values. Since the agents do not know a priori the values of the vertices, only probed vertices contribute with their full value to the zone-value, unprobed ones only contribute one point.

The goal of the game is to maximize the score. The score is computed by summing up the values of the zones and the current money for each simulation step:

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

Here `steps` is the number of simulation steps, and $\texttt{zones}_s$ and $\texttt{money}_s$ are the current sum of all zone values and the current amount of money respectively.

Figure 1 shows such a scenario. The numbers depicted in the vertices describe the values of the water wells while the distance of two water wells is labeled with travel costs. The green team controls the green zone while the blue team has the smaller blue zone. The value of the blue zone, assuming that all vertices have been probed by the blue team, is 24.

4 Brief Team-Descriptions

A total of nine teams from all around the world took part in the 2011 edition of the tournament (see Table 1). In the following, a brief description of each of those teams is given. The full descriptions provided by the teams themselves can be found in part II of this document.

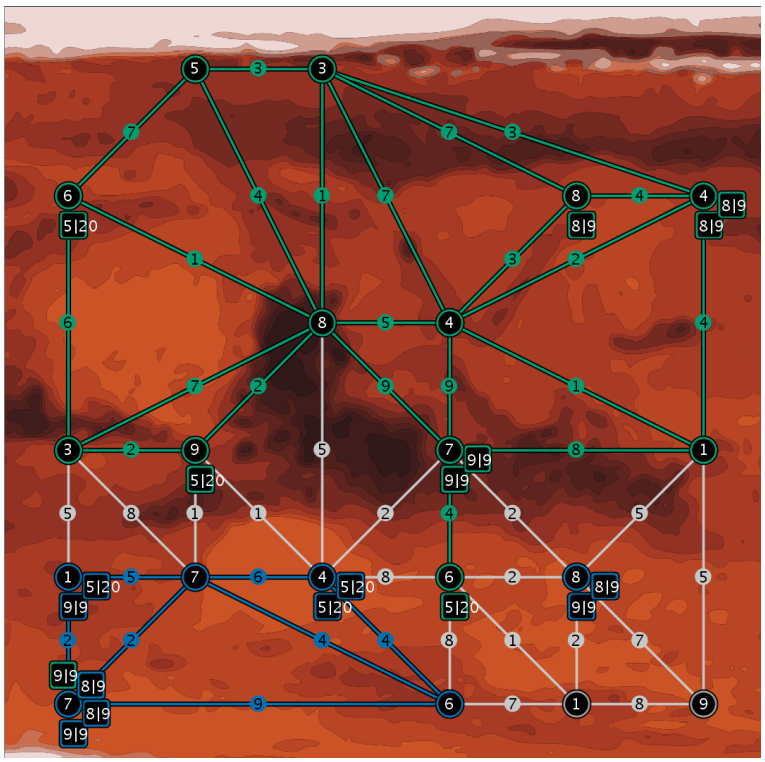


Figure 1: A screenshot.

The *d3lp0r* team from Universidad Nacional del Sur, Argentina, was implemented to show that argumentation via defeasible logic programming can be applied in a multi-agent gaming situation. It has been implemented using Python, Prolog and DeLP. The solution is a decentralized architecture, where each agent runs as an individual process and percepts are shared via a broadcasting mechanism with a minimal complexity. This coordination mechanism is facilitated by a perception server that gathers and distributes all relevant percepts. Decision making takes place on an individual agent level and has no centralized characteristics. The team's main strategy is to detect profitable zones based on the data collected about explored vertices and position the agents correctly to maintain, defend and expand the zones.

The *HactarV2* team from TU Delft, Netherlands, was implemented using the GOAL agent-oriented programming language with Prolog as the knowledge representation language. The team follows a decentralized strategy based on an implicit coordination mechanism, where agents predict the ac-

Team	Affiliation	Language
d3lp0r	Universidad Nacional del Sur, Argentina	Prolog/Python
HactarV2	TU Delft, Netherlands	GOAL
HempelsSofa	Universität Göttingen, Germany	Java
Nargel	Arak University, Iran	Java/JADE
Python-DTU	Technical University of Denmark	Python
Simurgh	Arak University, Iran	Java/JADE
Sorena	Arak University, Iran	JACK
TUB	TU Berlin, Germany	JIAC V
UCDBogtrotters	University College Dublin, Ireland	AgentFactory

Table 1: Participants overview.

tions that other agents perform and base their own choice of actions on that prediction. The agents share all data about the map and opponents with each other, while neither using a centralized information store nor a central coordination manager. The team's main strategy is to firstly compute the zone with the highest value and secondly building and maintaining a swarm of agents around the node with the highest value.

HempelsSofa[8] has been developed at Göttingen University. The team is based on a solution that has been implemented during a course on multi-agent programming, held at Clausthal University of Technology. The agents were developed in Java using a simplified architecture that allows for an explicit mental state and inter-agent communication. All agents are executed in a single process and each agent has access to a shared world model that is updated every time and agent perceives something.

Python-DTU from Technical University of Denmark is based on an auction based agreement approach and has been implemented in Python. The solution is decentralized, allowing agents to share percepts through shared data structures and coordinate their actions via distributed algorithms. Agents share all new percepts in order to keep the agents' internal world models identical. In the first ten percent of each simulation the team explores the map and inspects the opponents. After that, a valuable zone is conquered and maintained while letting saboteurs attack and repairers repair. Valuable zone detection is facilitated by firstly selecting the most valuable known vertex and then focusing on vertices around the selected one. Communication and coordination involves placing bids on different goals and then executing an auction-based agreement algorithm.

Team *Nargel* from Arak University, Iran, is a true multi-agent system developed using Java for agent behaviors and JADE for agent communication. The performance of the agents was on a level that did not require any distribution

[8]The team *HempelsSofa* took part in the Contest out of competition.

of agents on different machines. The team strategy is based on the intent to conquer zones, while also disturbing the opponent's ones. The disturbing behavior is more successful than the zone making strategy. Agents share their acquired knowledge by means of inter-agent communication only and thus do not have a centralized pool if information. On top of that, intentions are also on an individual agent level, thus resulting in a decentralized coordination approach.

The *Simurgh* team, also from Arak University, made use of Java as an agent implementation language, while using the Gaia methodology for analysis and design. The team uses a decentralized coordination and cooperation mechanism, in which agents share their percepts via a shared communication channel. Agents autonomously generate goals based on their knowledge. Goals in conflict are resolved. Each agent is executed in its own thread and has its own world model. The agents are divided into three groups. The zone holders are responsible for creating and maintaining zones using a scattering algorithm. The world explorers intent to complete their world model quickly. And the repairers strive to repair disabled agents as soon as possible.

Sorena is the third team from Arak University. The developers used the Prometheus methodology for the system specification and the JACK agent platform for actually implementing and executing the agents.

The *TUB* team comes from Berlin Technical University. The team's development has been done by roughly following the JIAC methodology. The team is completely decentralized, while each agent is perfectly capable of performing each role. Usually one agent is responsible for zoning and agents position themselves on the map using a simple voting protocol. Agents share all their percepts, which, although having a high complexity, worked perfectly for the small team. The team makes use of both implicit and explicit coordination. Implicit coordination is considered to be achieved by sharing intentions. Explicit coordination, however, is only used for the collaboration of inspectors and saboteurs. From the beginning each agent follows its own achievement collections strategy. The zone score on the other hand is locally optimized by letting agents move to the next node that improves the zone.

The *UCDBogtrotters* team from University College Dublin, Ireland, has been implemented using the AF-TeleoReactive and AF-AgentSpeak multi-agent programming languages running on the AgentFactory platform. The overall team strategy involves a leader agent, which assigns tasks to other agents, and platform services for information sharing. Finding zones is facilitated by a simple clustering algorithm. The team combines a set of role dependent strategies and the overall zone creation strategy.

Pos.	Team	Score			Difference	Points
1	HactarV2	2,979,591	:	734,185	2,245,406	72
2	Python-DTU	3,468,448	:	745,940	2,722,508	60
3	TUB	2,835,401	:	914,883	1,920,518	57
4	UCDBogtrotters	2,379,663	:	1,459,391	920,272	45
5	HempelsSofa	1,243,262	:	2,185,634	-942,372	36
6	Simurgh	928,893	:	2,219,281	-1,290,388	18
7	Sorena	888,631	:	2,366,805	-1,478,174	15
8	d3lp0r	888,837	:	3,821,088	-2,932,251	15
9	Nargel	765,296	:	1,930,815	-1,165,519	6

Table 2: Final Ranking.

5 Contest Organization and Results

5.1 Tournament Organization

The rules of the tournament indicated that every team had to compete in a match against each of the other eight teams. The winner of the tournament was the team that earned the most tournament-points in total (as opposed to in-simulation points). Matches consisted of 3 independent simulations, played in randomly generated maps of 3 different predefined sizes. Tournament-points were awarded in a per-simulation basis: winning a simulation was worth 3 tournament points, whereas no points where given to the losing team. In the unlikely event of a tie, both teams would be given 1 point.

The tournament took place from 5th to 9th of September 2011. For each day of the contest, the teams where divided in 3 groups of 3 teams, and played against each other within the group. Each group was assigned to a different server, so matches from different groups were held in parallel. Groups where resorted every day to make sure every combination was covered and every team played against each other exactly once.

5.2 Tournament Results

The four days of competition allowed team *HactarV2* to stand out as the clear winner, after defeating their opponents in every single simulation they took place in. Team *Python-DTU* achieved a distinguished second place, being the team that collected the biggest simulation-score sum throughout the tournament. A close third was *TUB*, only 3 points below. The complete final ranking is depicted in Table 2.

We present the result of each simulation of each match in Section 5.3. Matches are presented in chronological order; this is relevant to some of the results because some teams experienced bugs and connection problems that were not detected in the two-week connection-testing period before the tournament, and which in most cases were corrected during the competition.

Further information in this regard is given in Section 5.4, which is also concerned with the agent teams' quality and stability. Next, in Section 5.5, we analyze simulations individually to a deeper extend by looking at the evolution and composition of the score. Finally, in Section 5.6, we observe how actions were selected by each role of each team, and relate this numbers to that team's strategy.

5.3 Simulation Results

Match	**PythonDTU** vs. **TUB**		**HempelsS.** vs. **TUB**		**HempelsS.** vs. **PythonDTU**	
Sim 1	84,824	62,588	5,953	131,215	4,644	314,353
Sim 2	82,078	65,277	3,140	119,445	5,967	303,220
Sim 3	66,032	68,509	8,846	99,295	4,356	244,999

Match	**UCD** vs. **HactarV2**		**d3lp0r** vs. **HactarV2**		**d3lp0r** vs. **UCD**	
Sim 1	3,409	238,653	5,253	134,638	16,546	142,091
Sim 2	2,073	147,552	10,863	395,720	3,170	280,481
Sim 3	1,772	186,471	7,888	297,740	264	370,728

Match	**Simurgh** vs. **Nargel**		**Sorena** vs. **Nargel**		**Sorena** vs. **Simurgh**	
Sim 1	43,739	29,361	35,393	23,938	41,617	57,245
Sim 2	13,380	41,848	38,270	33,400	53,603	37,215
Sim 3	9,404	66,468	37,603	23,786	51,246	52,054

Table 3: Matches Day 1.

Match	**TUB** vs. **Nargel**		**HactarV2** vs. **Nargel**		**HactarV2** vs. **TUB**	
Sim 1	112,946	22,462	103,529	23,335	60,001	44,850
Sim 2	109,550	33,086	105,908	27,732	61,203	45,194
Sim 3	91,486	26,849	90,834	28,605	61,787	47,857

Match	**HempelsS.** vs. **Sorena**		**d3lp0r** vs. **Sorena**		**d3lp0r** vs. **HempelsS.**	
Sim 1	86,991	44,331	93,362	52,703	49,344	91,276
Sim 2	78,432	45,745	53,852	59,861	29,211	85,869
Sim 3	89,411	45,757	77,687	69,807	48,556	86,796

Match	**Python-DTU** vs. **Simurgh**		**UCD** vs. **Simurgh**		**UCD** vs. **Python-DTU**	
Sim 1	109,959	37,986	358,679	6,974	26,147	103,163
Sim 2	229,456	5,060	127,301	20,703	11,559	106,436
Sim 3	103,894	1,243	118,793	36,083	12,699	93,466

Table 4: Matches Day 2.

Match	**TUB** vs. **Simurgh**		**d3lp0r** vs. **Simurgh**		**d3lp0r** vs. **TUB**	
Sim 1	101,839	47,316	32,167	73,716	4,461	401,122
Sim 2	94,264	47,735	46,291	79,094	4,232	336,559
Sim 3	96,152	48,901	64,020	72,134	19,301	169,868

Match	**HempelsS.** vs. **Nargel**		**UCD** vs. **Nargel**		**UCD** vs. **HempelsS.**	
Sim 1	96,558	16,638	66,205	28,930	65,272	35,967
Sim 2	77,142	26,696	171,492	16,168	81,933	39,802
Sim 3	76,768	39,299	67,462	32,653	75,197	45,101

Match	**Sorena** vs. **HactarV2**		**Py.-DTU** vs. **HactarV2**		**Sorena** vs. **Py.-DTU**	
Sim 1	35,040	102,450	43,881	57,652	13,906	313,396
Sim 2	35,243	111,989	46,670	101,998	14,126	240,445
Sim 3	39,717	104,925	40,514	73,955	17,558	123,332

Table 5: Matches Day 3.

Match	**UCD** vs. **Sorena**		**Sorena** vs. **TUB**		**UCD** vs. **TUB**	
Sim 1	88,638	29,875	31,773	133,157	51,278	67,730
Sim 2	80,336	28,439	23,482	148,223	28,962	74,957
Sim 3	74,924	24,587	18,949	137,617	72,232	75,701

Match	**Simurgh** vs. **HactarV2**		**HempS.** vs. **HactarV2**		**HempS.** vs. **Simurgh**	
Sim 1	37,267	85,366	58,134	87,063	66,330	47,881
Sim 2	37,741	92,349	31,167	99,575	63,318	38,998
Sim 3	33,020	90,626	46,960	87,607	54,334	44,004

Match	**Python-DTU** vs. **Nargel**		**d3lp0r** vs. **Nargel**		**d3lp0r** vs. **Python-DTU**	
Sim 1	131,224	23,661	99,676	52,094	34,385	100,726
Sim 2	104,713	20,869	92,530	55,810	1,449	308,442
Sim 3	97,123	43,897	57,880	27,711	36,449	76,102

Table 6: Matches Day 4.

5.4 Teams' Quality and Stability

In order to analyze the quality and stability of the teams – and to a certain extent also the stability of the platforms – we summed up the number of actions sent by an agent in time (i.e., in two seconds) and the actions that failed due to lack of time. The results are shown in Table 7. Since each team had the opportunity to test the network connection (and especially the network bandwidth) two weeks before in some test matches, these results cannot only give us some hints regarding the overall performance but also some indications concerning the quality and stability of each agent team. Together with the experiments we made throughout the Contest, i.e., checking the network ping and bandwidth, we can conclude the following:

Team	Day 1	Day 2	Day 3	Day 4	All
HactarV2	0,67%	0,10%	0%	0%	0,19%
Python-DTU	0%	0,19%	0%	0%	0,05%
TUB	0,03%	0,30%	0,11%	1,08%	0,38%
UCDBogtrotters	39,51%	5,47%	1,55%	0,64%	11,79%
HempelsSofa	8,10%	0,03%	0,08%	4,57%	3,20%
Simurgh	41,14%	42,47%	12,63%	3,03%	24,82%
Sorena	1,85%	6,58%	0,55%	0,37%	2,34%
d3lp0r	29,42%	13,32%	16,79%	0,54%	15,02%
Nargel	6,24%	0,45%	0,91%	1,10%	2,18%

Table 7: Actions not sent in time.

The first three teams *HactarV2*, *Python-DTU* and *TUB* did not have any problems sending actions in the two second time interval. Their network connection was good, but more important – as the Contest results (Tab. 2) show - their agents were able to process the percepts sent by our server and to provide an useful answer in time.

The *UCDBogtrotters* had some problems (a bug in the code of the agents) on the first day. After fixing it the agents were sometimes still too slow. Additionally, the explorer agents as well as the *inspector* agents tried to execute the *parry*-action which was not allowed for these rules. For this reason we argue that the stability was quite okay but the code quality was not perfect.

HempelsSofa had a serious bug at day one and the team performed very badly. The bug was not detected in an early phase, because the wrong credentials were used for testing. Afterwards the response time was good. Thus, the quality and stability was okay, but the testing routines failed.

The Iranian teams *Simurgh*, *Nargel* and *Sorena* faced some network bandwidth problems in the test matches. However, they improved there code and/or used some computers from different countries for the real Contest. Nevertheless they did not perform well. The results of *Simurgh* – especially

when taking into account that they were only sending 60 percent of actions in time for the first two days – were still okay but the code had some major flaws: The *explorers* and *inspectors* tried to execute the *parry*-action which was not allowed for these rules. The stability and code quality of *Nargel* and *Sorena* can be classified as medium since on day 1 respectively day 2 the teams had some connection problems.

d3lp0r finally implemented the communication protocol between the server and the agent teams in a wrong way. Their agents were not able to attack other agents if the name of the opponent was starting with an upper case letter. Aside, only at the very last day they were able to send enough actions in time. Thus we infer that both, the code quality as well as the stability was low. The results attest this claim as well.

5.5 Analysis of Individual Matches

In order to further analyze the performance of teams and the effectiveness of the strategies applied, we now analyze some selected simulations in more detail. In order to improve comparability we will focus only on the mid-size map simulations, which in most cases is also representative of the final result of the match. We present graphs that show the evolution of the current score throughout a simulation, distinguishing zones-score and achievement points. This graphs give already by themselves a handful of information, and sometimes directly reflect aspects of the different strategies applied. Nevertheless, it is interesting in some cases to go further and study some particular situations of the simulations that can ultimately help in explaining these graphs.

5.5.1 HactarV2 vs. PythonDTU

The match between HactarV2 vs. PythonDTU (Figure 2) was a decisive one, as it faced the two teams that ended up being the winner and the runner-up of the contest. PythonDTU started the simulation well, collecting more points than HactarV2 during the first steps, both in terms of zones-score and achievement points. The big difference in terms of achievement points at this stage was mainly because of HactarV2's more aggressive buying strategy, which seems to have paid off as both teams achievement points stabilized towards the end.

Around step 90, as both teams had explored an adequate portion of the map and focused on conquering the most valuable nodes at the center of the map, a particular situation arose, that remained until the end of the match and gave a huge advantage to HactarV2: Gathered on a single node where PythonDTU's repairers and HactarV2's saboteurs, among some extra agents. HactarV2's saboteurs were made strong enough (via the *buy* action)

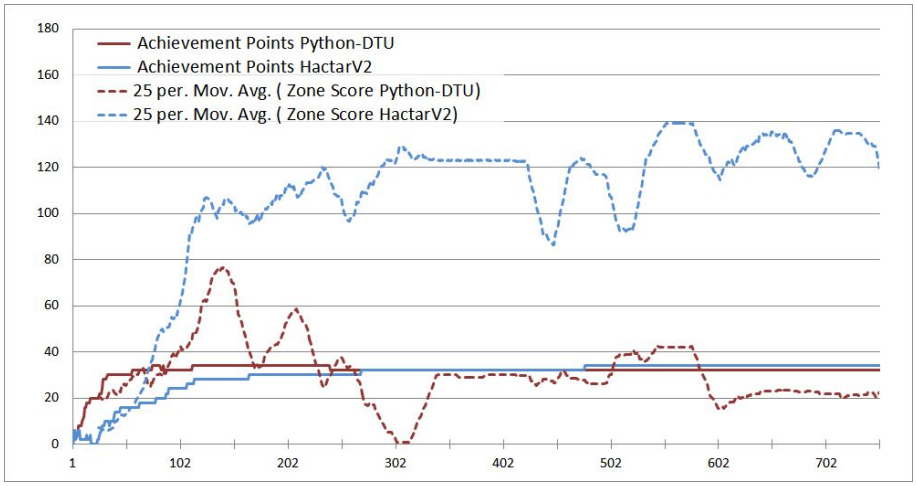


Figure 2: HactarV2 vs. PythonDTU.

so that they could disable opponent agents with a single hit. A cycle began in which HactarV2's saboteurs would alternate attacking (and instantly disabling) each of the opponent repairers. Since they were not recharging at the same time, they could ensure that on every step at least one of them would be attacking. Python's repairers contributed to this cycle with their behavior: on every step the enabled repairer would repair the disable one, but also receive an attack and become disabled for the next step. Then, on the next step the choice of action for the just disabled repaired would be to recharge while waiting to be repaired, enabling this vicious circle to continue almost indefinitely. Figure 3 shows a particular step during this cycle.

The node where all this took place was one of the high-valued ones, so it remained most of the time in HactarV2's domination, although some sporadic incursions from other PythonDTU's agents changed that for a few steps. The two saboteurs of PythonDTU where disabled just before the above-mentioned cycle started, and of course were not repaired, so they did not represent a threat anymore. The rest of HactarV2 team could focus on maintaining a rather big, stable zone, which explains the big difference in the final score.

5.5.2 HactarV2 vs. Simurgh

Figure 4 shows that the simulation of HactarV2 vs. Simurgh began with a clear domination of the zone-score from HactarV2 team. Several battles took place right from the beginning, and many of the Simurgh's agents became prematurely disabled as result. Both teams attempted to improve their sabo-

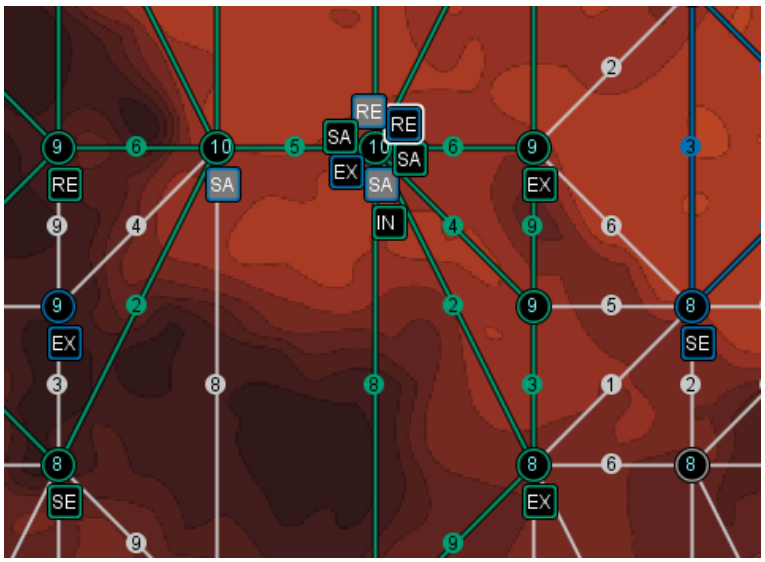


Figure 3: HactarV2 vs. PythonDTU screenshot.

teurs with the early achievement-points obtained, but Simurgh's saboteurs prioritized the *buy* action over attacking, even when sharing location with other saboteurs; they often got attacked and the *buy* attempt failed. After the first steps, the many disabled agents from Simurgh tended to regroup themselves with the repairers, but HactarV2's saboteurs remained close and kept those agents busy most of the simulation. The Simurgh agents could not handle the situation well, with the repairers repairing with no apparent role-based priority, and the saboteurs often skipping actions or attempting buys instead of attacking. The rest of the agents from HactarV2 managed to build and maintain a valuable zone in the center of the map, which ensured the team the big difference in the final score. The fewer free agents from Simurgh, on the other hand, could build some smaller zones towards the borders of the maps through the simulation, but these would only earn the team very few points.

5.5.3 HactarV2 vs. UCDBogtrotters

UCDBogtrotters suffered a lot of connection problems during the first day of competition (aproximately 40 percent of the actions were not sent in time to the server. See Subsection 5.4) and that is clearly reflected in Figure 5. HactarV2 was presented with almost no resistance from their opponents, and took advantage of it gathering several points. The peaks in the graph around

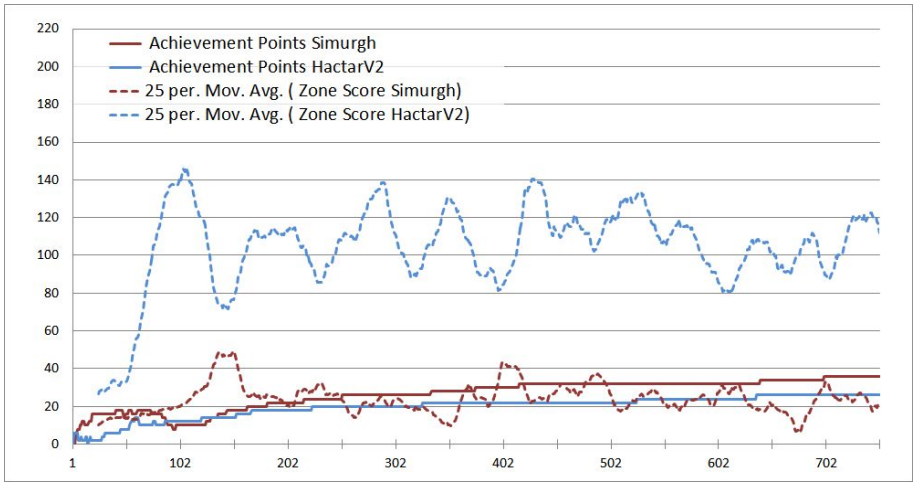


Figure 4: HactarV2 vs. Simurgh.

steps 100, 300, and 700 correspond to moments during the simulation in which all agents from UCDBogtrotters were disabled, thus giving HactarV2 domination of the entire map.

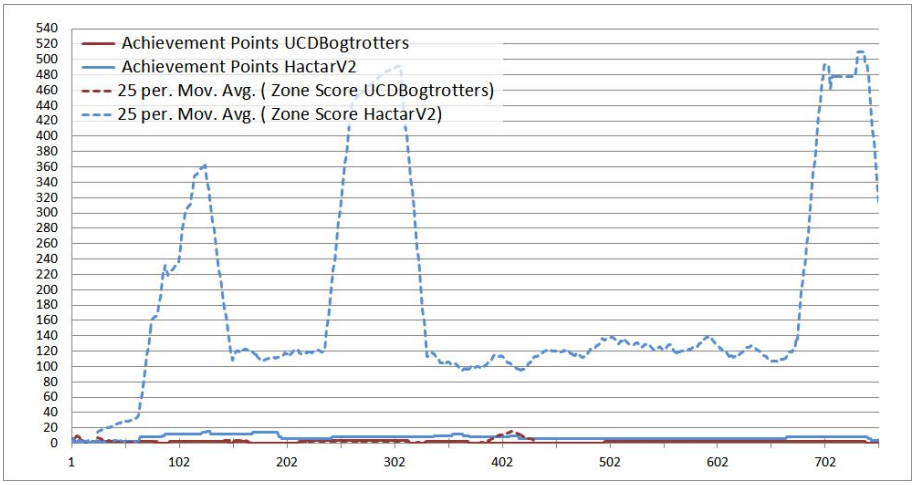


Figure 5: HactarV2 vs. UCDBogtrotters.

5.5.4 HempelsSofa vs. UCDBogtrotters

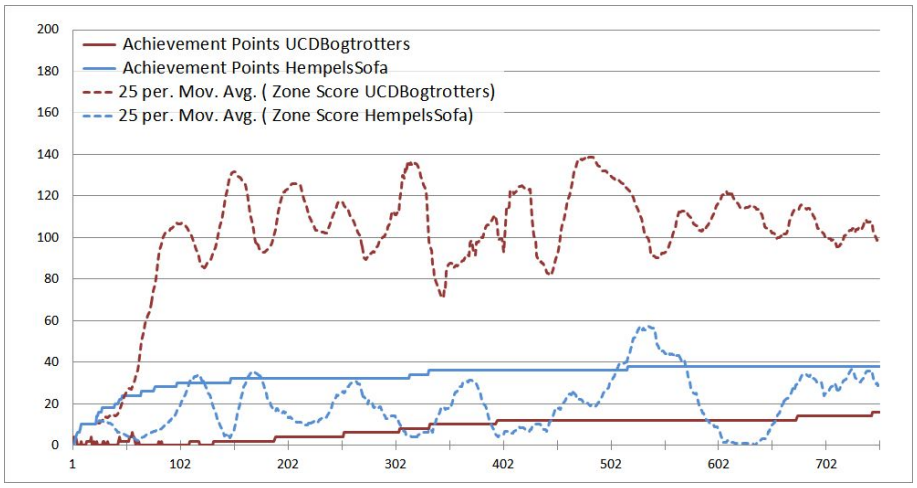


Figure 6: HempelsSofa vs. UCDBogtrotters.

Figure 6 shows an interesting simulation between HempelsSofa and UCDBogtrotters, with clear domination of the latter. During the initial phase it appears as if both teams were able to locate the most valuable nodes at the center of the map, and towards step 66 they both tried to build zones there. Figure 7 shows however a big difference between those zones: UCDBogtrotters' expands in more nodes, while HempelsSofa team tend to gather more agents in fewer nodes, resulting in a very small zone, that is even less worthy because of unprobed nodes.

Later during the simulation, around step 100, UCDBogtrotters managed to conquer the center of the map. HempelsSofa agents started alternating between moving in zone-formation around the center (with rather low zone-scores), and engaging in battle in the center, not very successfully. A couple of times during the simulation, the only few agents from HempelsSofa still enabled move around the center and become surrounded by agents from UCDBogtrotters, as shown in Figure 8, where UCDBogtrotters gain domination of almost the complete map, except for this few nodes in the center.

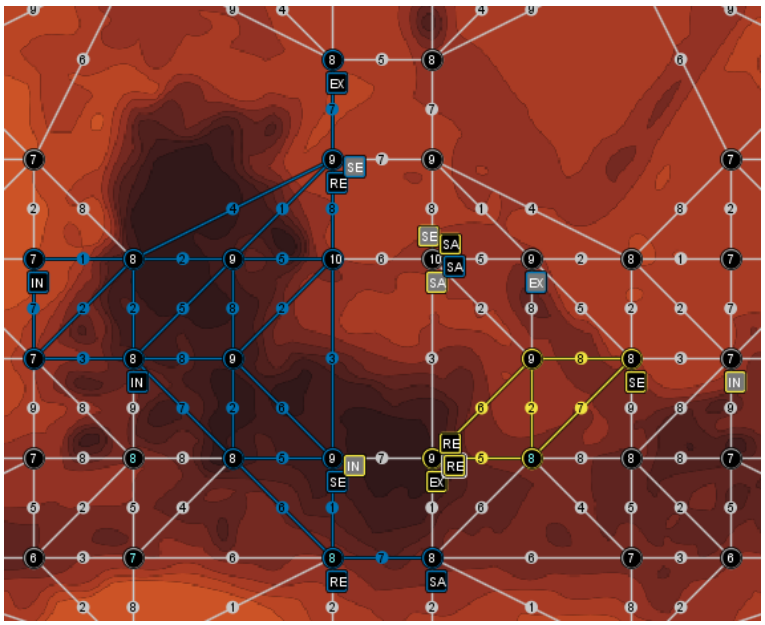


Figure 7: HempelsSofa vs. UCDBogtrotters screenshot.

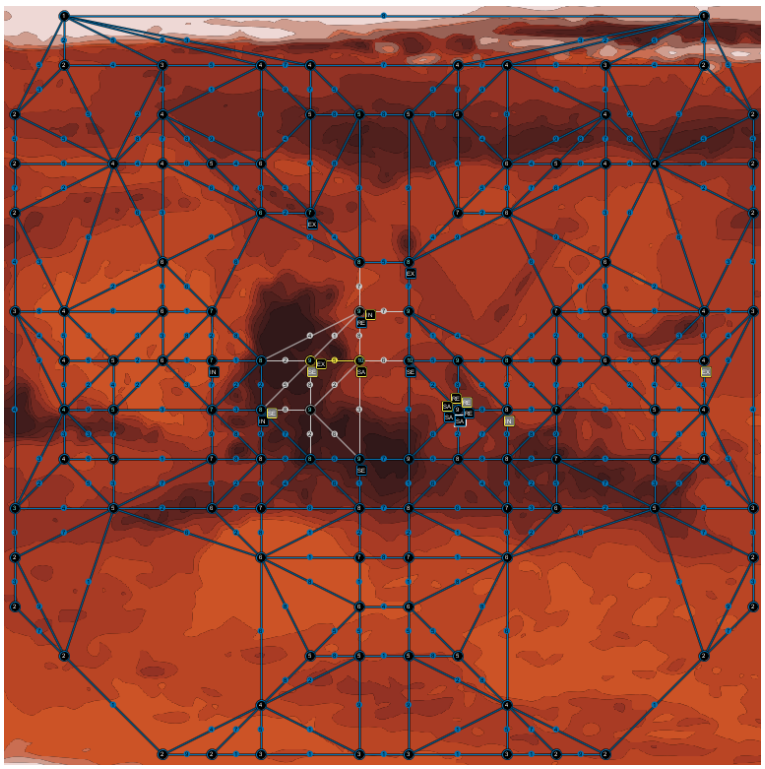


Figure 8: HempelsSofa vs. UCDBogtrotters screenshot.

5.5.5 PythonDTU vs. UCDBogtrotters

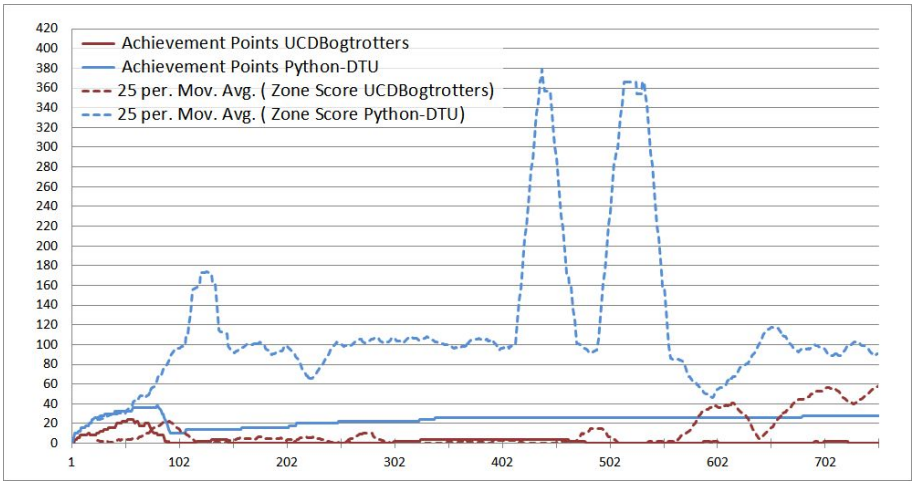


Figure 9: PythonDTU vs. UCDBogtrotters.

At the beginning of the match, that is in the first 50 steps, UCD seems to group most of the agents together on few nodes. This approach makes it easier for to be attacked, and less efficient when it comes to exploring the map, because they probed less nodes. Python, however, explores more, moves in relatively open groups, while maintaining ample zones. At step 57 Python manages to isolate a corner section of the map, establishing a rather big zone (see Figure 10). The value of the zone is 108, while only a quarter of the nodes in that very zone have been probed. At step 106 PythonDTU isolates a single UCD agent in the center of the map, conquering the entire map except for the isolated nodes.

At step 107 (and following) too many UCD agents, that is up to eight ones, stand on a single node, while the other two agents are disabled. Thus no zones are conquered. From PythonDTU, only saboteurs and repairers stand on the node. The explorer joins from time to time, apparently to win domination of that node for zone-making purposes. When PythonDTU gains domination of this node repeatedly, the team conquers the whole map. The team then also gains the probed-40-nodes achievement. The UCD saboteurs engage in battle and the repairers are constantly repairing other agents. But the remaining enabled agents do not leave the node. Instead they keep on parrying and recharging. Eventually the saboteurs leave the node, when they are disabled and sometimes being repaired at the same step).

At step 127 a PythonDTU saboteur follows a just repaired UCD saboteur

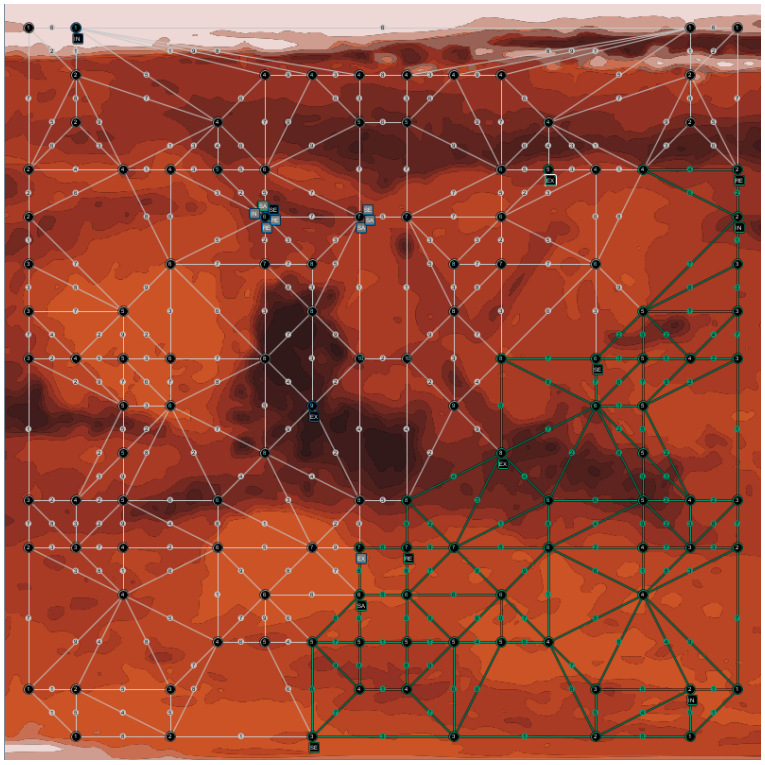


Figure 10: PythonDTU vs. UCDBogtrotters screenshot.

that leaves the previous node, thus ending the rather balanced situation. The rest of the agents move as well. PythonDTU maintains the previously conquered zone, while the free agents follow UCD's ones. We can observe similar situations in the two other peaks. At step 605 UCD establishes a bigger zone, but un-probed nodes result in PythonDTU getting bigger score. After that the UCD agents create a formation that is off-center.

5.5.6 Simurgh vs. d3lp0r

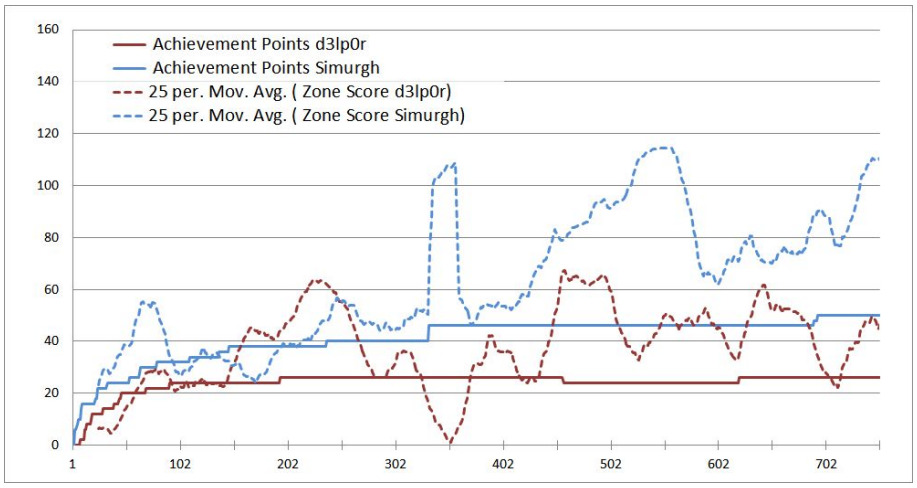


Figure 11: Simurgh vs. d3lp0r.

A first glimpse at Figure 11 reveals that the confrontation between Simurgh and d3lp0r was most of the time dominated by the former in terms of zone score, and completely in terms of achievement points. A fact that probably had a huge impact on this result was that a bug in d3lp0r's saboteurs made all of their attacks fail: internally, their processing used lowercase versions of identifiers for the enemy agents and they then sent this lowercase identifiers to the server which processed identifiers in a case-sensitive manner. Unfortunately, this bug remained undetected during the test phases previous to the competition (d3lp0r only took part in test matches against dummy-teams, which always used lowercase identifiers) and also during the first days of competition, since the message sent back to the agents for failed attack did not make the reason for the failure obvious.

Contributing to the big difference regarding the achievement points of the two teams was also an apparent limitation in the choice of action for d3lp0r's agents, most likely due to an strict focus in the zone-making component of the score. Surely the team was unable to obtain any attack-achievement, but it also did not earn any of the inspect-achievements nor any of the parry-achievements. An interesting fact of the game that is evidenced here is that almost not using any of the `parry` action gives the opposite team more than an instant advantage: It also means less parry-achievements for oneself and more attack-achievements for the opponents (Simurgh kept attacking constantly and earned up to the `attacked320` achievement) and this might

certainly reflect in the final score. The instant advantage of parrying can be argued, since it means consuming a turn and energy in an action that may be unrelated to that agent's current goal, and that is not guaranteed to be useful (i.e. the opponent saboteur may not attack that particular agent, or at all during that turn). Instead, d3lp0r team seemed to rely on the efficiency of their repairers, which was in fact quite good: most of the team's agents were enabled during most of the match, except for that peak in the graph around step 350, where Simurgh managed to disable all d3elp0r's agents and gain complete domination of the map. Nevertheless, the efficiency of d3lp0r's repairers also meant that Simurgh's saboteurs always had enabled opponent agents to attack and keep obtaining attack-achievements.

Simurgh's zone was more stable than d3lp0r's, as more of its agents tended to remain in a bigger single-zone formation, that started on an edge of the map and moved to the center towards the end of the simulation. The agents of d3lp0r, on the other hand, moved around the map in smaller groups. This, plus the fact that only Simurgh could count with all their agents enabled during the whole simulation, can explain the difference in their zone-scores.

5.5.7 Simurgh vs. PythonDTU

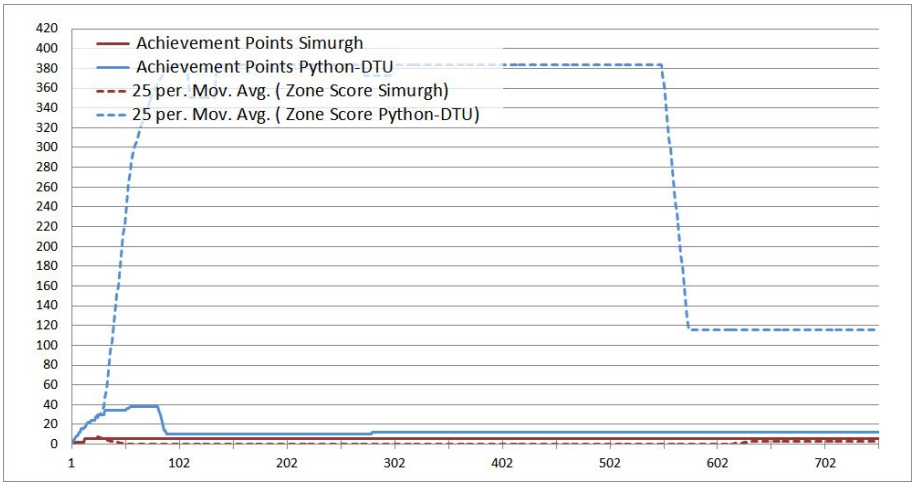


Figure 12: Simurgh vs. PythonDTU.

The match between Simurgh and PythonDTU (Figure 12) does not allow for much analysis. Simurgh had some severe problems in sending actions in time (see Subsection 5.4) which resulted in PythonDTU competing against a team of static agents. PythonDTU could easy disable all of Simurgh's agents quickly to conquer the whole map, and the situation kept going on until the game was well advanced. Then one of Simurgh's vehicles could be repaired, and PythonDTU lose domination of the complete map. Nevertheless PythonDTU's agent were already located forming a big zone in the center, that remained unchallenged in the rest of the simulation.

5.5.8 Simurgh vs. TUB

In the game Simurgh vs. TUB (Figure 13) stabilized after the first hundred steps with both teams competing for the center nodes, but with TUB doing it more efficiently, engaging less agents in the battle and leaving the rest around to compose bigger zones. As a result, although the domination of the central nodes fluctuated from one team to the other, TUB's zone were consistently more valuable than Simurgh's, as can be clearly seen in the graphic.

It can also be noted that both teams had a different approach regarding the achievement points: for Simurgh, they represented the major component of the score, so they didn't expend them, while TUB used part of them to improve their saboteurs.

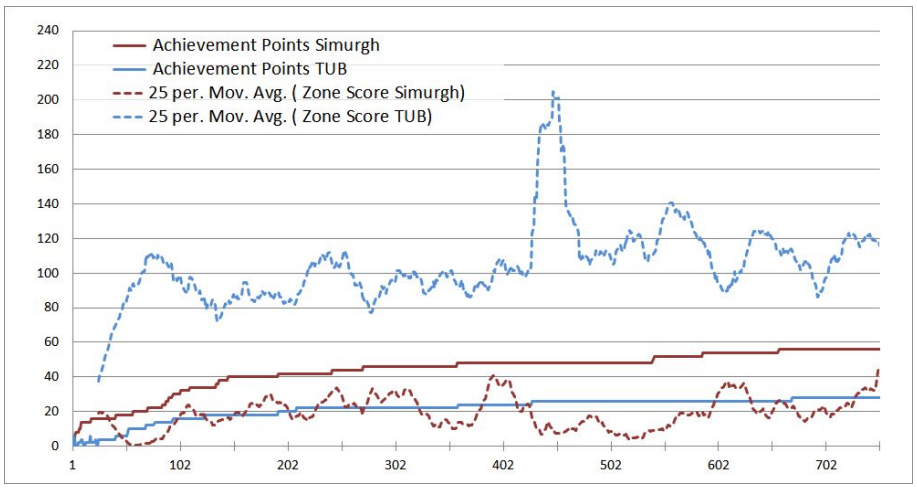


Figure 13: Simurgh vs. TUB.

5.5.9 Sorena vs. Nargel

The match between Sorena and Nargel (Figure 14) was a unique case during the competition, as the factor that defined the winner was the achievement points rather than the zone-score. For Sorena, the winner, the final score was composed by 15.496 points from the accumulated zone-score and 22.774 from accumulated achievement points, whereas for Nargel these values were 16.348 and 17.052 respectively.

Even more interesting is the fact that Nargel actually earned more achievement points than Sorena, but they spent some of them improving their agents, while Sorena spent none. The total accumulated points from the achievements they spent sum 7384 points, which is more than the difference towards Sorena in the final score. Nevertheless, this cannot lead to the conclusion that Nargel would have won the simulation had they kept all their achievement-points, since their performance regarding making zones and earning achievements would probably have been lower if they had not bought those improvements.

None of the teams were strong regarding zone-making; they were never able to build a zone worth more than 100 points, and their average zone values were close to 20. Both teams moved their agents forming small, erratic zones around the map.

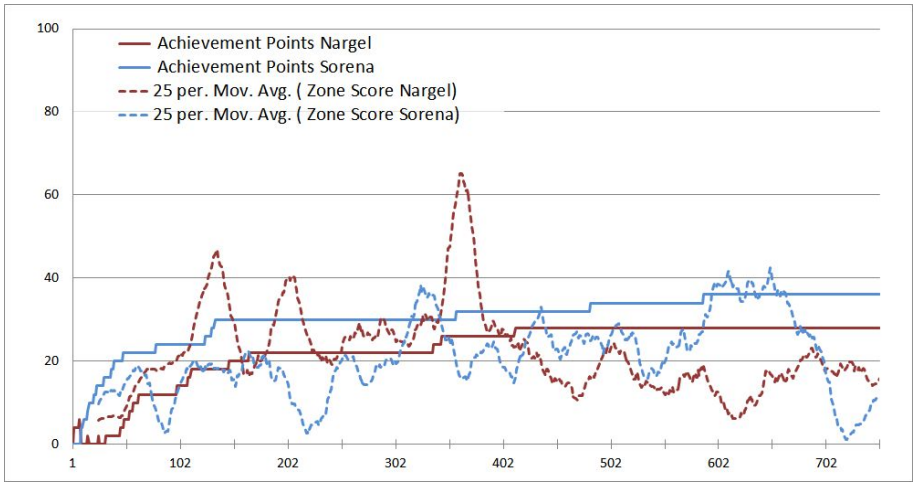


Figure 14: Sorena vs. Nargel.

5.5.10 Sorena vs. UCDBogtrotters

UCDBogtrotters clearly dominated the match against Sorena (Figure 15). From the beginning, UCDBogtrotters managed to place its agents on the center of the map and build a very valuable zone. Sorena's agents, on the other hand, wondered around the rest of the map, only dominating small zones intermittently.

Both teams had opposed approaches regarding achievement points: UCDBogtrotters spent most of the points for improving agents, clearly stating that zone-building was their main priority. In contrast, Sorena made their achievement-points its main score component, and spent none of them in improving agents. In this simulation though, the excellent zone-making performance from UCDBogtrotters was impossible for Sorena to compensate with their achievement points as they did in the closer match against Nargel.

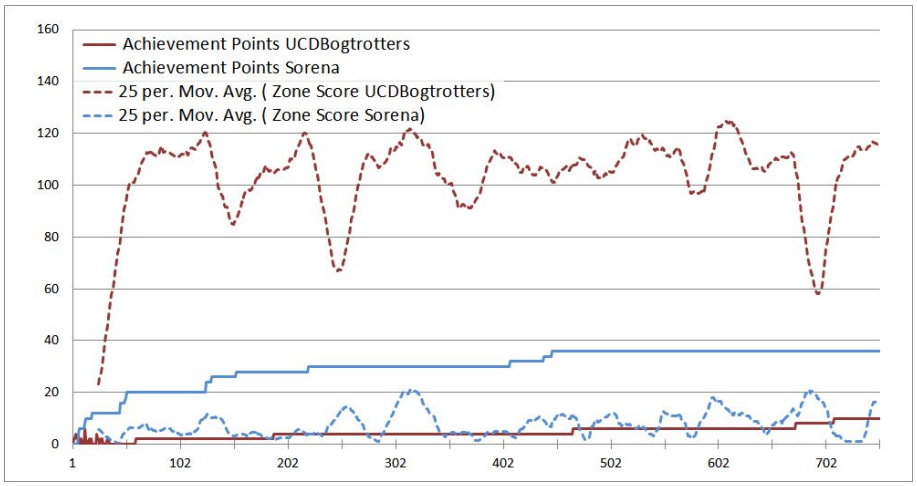


Figure 15: Sorena vs. UCDBogtrotters.

5.5.11 TUB vs. HactarV2

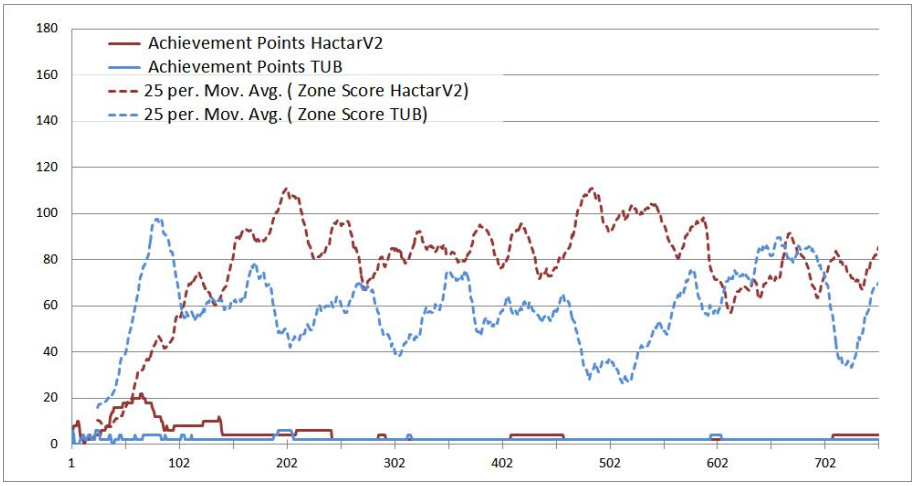


Figure 16: TUB vs. HactarV2.

TUB vs. HactarV2 (Figure 16) was a good match between two well-performing contenders. Both teams located the best nodes at the center of the map and engaged in a fierce battle for their domination, using aggressive strategies. HactarV2 managed to keep the central nodes for most of the time, and thus make a difference in the score to win the simulation.

A configuration with all saboteurs and repairers from both teams battling in a single node repeated several times during the simulation, in different nodes. Other agents joined them from time to time but remained mostly in a position useful for zone domination.

In terms of achievement points, the two teams also behaved very similarly to each other, spending most of these points in improvements for their saboteurs. The achievements also reflect the aggressive strategies used: both teams reached `achieved320`. But also some differences can be noticed here, as HactarV2 reached `parried160` while TUB did not parry at all.

5.5.12 TUB vs. PythonDTU

The match presented in Figure 17 between TUB and PythonDTU started with both teams going for the best nodes at the center of the map, first with a short period of domination from TUB, but then reverted by PythonDTU, that maintained the domination until the end of the match. The graph suggests that PythonDTU started slower, exploring more, and that around step

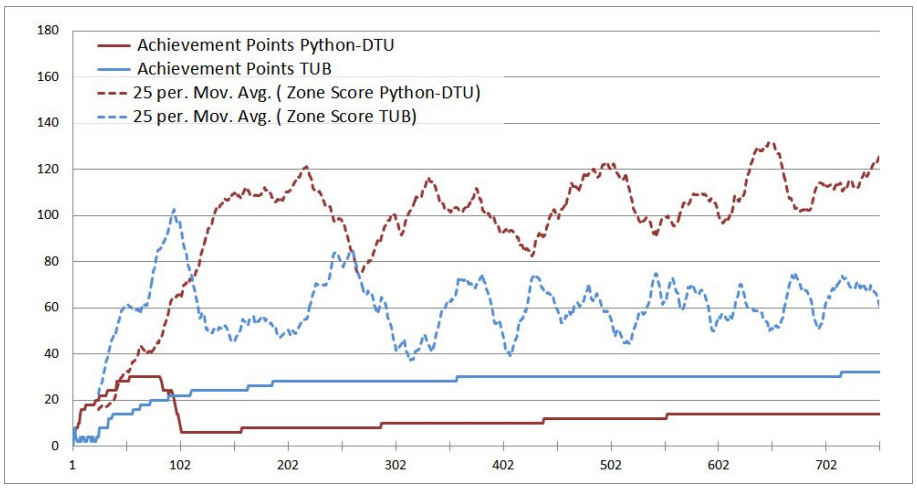


Figure 17: TUB vs. PythonDTU.

80 the switched focus to zone-domination, also executing at that time their buying strategy.

From then on, PythonDTU managed to place its agents forming a stable zone in the center, and most of them remained in position for most of the time, with the exception of the saboteurs and the repairers, that moved around while fighting against TUB's. The agents from TUB moved around the center also building zones but more unstable ones, as they where moving around.

Both team used achievement points to improve saboteurs only. TUB did this right away, whereas PythonDTU delayed the buys for some steps and ultimately spent more achievement points, getting its saboteurs more health than TUB's (and more than TUB's strength).

5.5.13 TUB vs. UCDBogtrotters

Figure 18 shows a rather unusual shape in the zone-score evolution of UCDBogtrotters. Analyzing the match, the conclusion is that the reason for this relies on the strange behavior of the explorer agents: on the first few steps, the explorers began the simulation as usually expected, that is, moving around and probing nodes. But after a very few steps, they ran across a TUB's saboteur. The explorers attempted to parry, which is not permitted for their role, and received the attacks becoming disabled. They got repaired around step 50, but they did not resume probing and instead joined the other agents for dominating a zone, even though only 8 nodes had been probed at that time

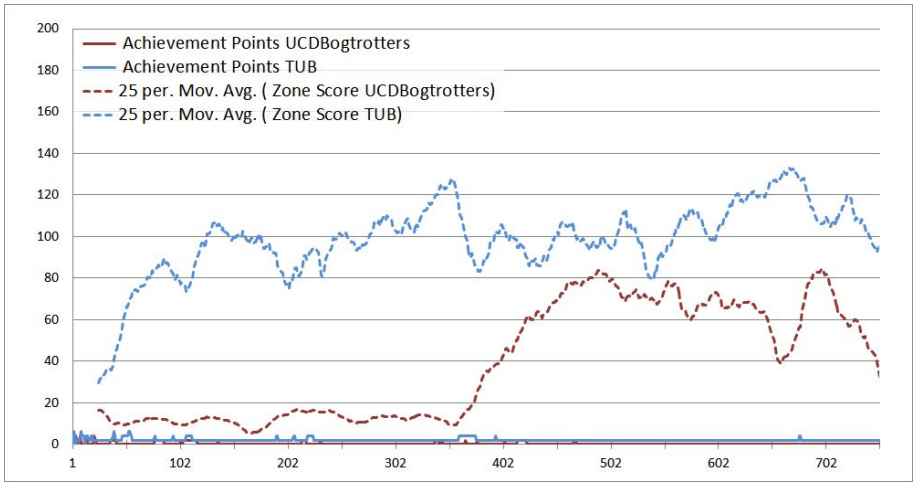


Figure 18: TUB vs. UCDBogtrotters.

and none of those nodes belonged to the zone.

Until around step 350, UCDBogtrotters maintained this zone in one side of the map, dominating up to 20 nodes. But since none of those nodes were probed, the team could only get up to 20 points for that zone. Then, explorer started moving and probing again until around step 400, and the team shifted it zone towards the center (where TUB had its zone). This ultimately suggests that UCDBogtrotters implemented a exploring phase at the beginning of the simulation that lasted for around 50 steps, and that is was restarted during the game.

TUB on the other hand found the best nodes in the center of the map at the beginning, and then moved in zone-formation around those nodes, earning higher zone values. For the fist half of the match they could do this unchallenged. Then, when UCDBogtrotters agents came closer, they had to engage in battle, but managed to keep keep the battle off center while retaining the best nodes.

5.6 Teams' Statistics

In this section, we focus on the second simulation and describe the overall behavior of each agent implementing one of the five roles. For a detailed description of all roles please have a look at the Appendix starting at page 35. Note, that the values describe the percentage of used actions on average, so they do not sum up to 100. However, the values reflect the real values quite accurately. Additionally, we will focus on the five teams *HactarV2*, *Python-DTU*, *TUB*, *UCDBogtrotters* and *Simurgh*.

5.6.1 *HactarV2*

The explorer used mainly the actions `recharge` and `goto`. 55% of all executed actions where the `recharge` actions (against *Python-DTU* it was even more than 80%) and approximately 30% the `goto`-action. Only used 1-2% was the `survey`-action. The `buy`-action was not used once and the `probe`-action was executed 5%. All actions that were tried to execute were also successfully executed. The repairer did not `buy` anything either and executed `goto` in 30% of the cases. This action was always successful. The action `parry` was executed 6% but only one out of four was successful. 53% of all actions were the `recharge`. `repair` was used in 8% and `survey` in 2%. The saboteur used 20% the `goto` action. `parry` was not used, `survey` less than 1% and `buy` just 1%, `attack` 41%. and `recharge` 35%. The sentinel executed the `goto`- action in 29% the cases. `parry` (5%), `survey` (2%) and `buy` (not used) were almost never used. The `recharge`-rate was around 64%. The inspector used the action `goto` in 36% the cases. `recharge` was done 56% and `inspect` (4%) and `survey` (1%) were almost never used. The action `buy` was not used at all.

In summary, only the saboteur was buying new equipment. Additionally, it was using its special action `attack` a lot. The other agents were walking around and using their special actions only from time to time.

5.6.2 *Python-DTU*

The explorer used `recharge` a lot (85%). And the `goto` in 10% the cases. The action `survey` was executed less than 1%. `buy` was never used and `probe` in 4 of 100 times. The repairer did not use `buy` and `parry`.`goto` was executed 11%, `recharge` 51%. `repair` 35% and `survey` almost never (1%). The saboteur did not use `parry` or `survey` and almost never `buy` (less than 1%). `goto` (15%) , `attack` (20%) and `recharge` (47%) were used more often. The sentinel was mostly only recharging (`recharge` 77%). `buy` was not used, `parry` in 2% the cases and `survey` just in 4%. The action `goto` was the second most used action (12%). The inspector was recharging a lot (84%). Otherwise it

was running around (goto 12%). inspect (less than 1%), (survey 2%) and buy (0%) were rarely used.

The saboteur was the only one buying new equipment but even this role did not use it a lot. The repairer repaired a lot of agents. All other agents were recharging quite often.

5.6.3 *TUB*

The explorer used recharge just in 34% the cases. The most often executed action was goto (52%). survey (5%) and probe (7%) were used to analyze the topology. However, the agent did not try to improve its values, i.e., the action buy was never used. The repairer did not execute buy nor parry. Also survey was used in less than 1%. Instead it was trying to repair (29%) other agents. Therefore it had to go the agents (goto 33%) and sometimes to recharge (33%). The saboteur did not try to defend itself (parry was never used), and did not analyze the topology (survey less than 1%). buy was only used in 1% the cases. Instead it tried to find opponents (goto 20%) and attack (32%) them. The recharge-rate was around 36%. The sentinel Did not execute parry or buy and almost never tried to survey (1%). But it was moving a lot goto 54% and recharging (recharge 44%) when necessary. The inspector, finally, was walking around (goto 55%) and recharging (recharge 43%). The inspect-acton was used – as well as survey – less than 1% and buy was not used at all.

The agents of team *TUB* were walking around quite often, the repairer was working very well. The saboteur tried to find and deactivate the opponents. However, the team did not try to improve their agents by buying new things.

5.6.4 *UCDBogtrotters*

This team had some problems with the connection, so their agents were sending invalid actions (or no actions in time) quite often. The explorer executed recharge in 43% the cases, goto in 23%. survey only 2% as well as probe. The buy-action was never used. parry not allowed but tried to use 7%. The repairer bought in less than 1% the cases something. It was going around sometimes (goto 7%) and trying to defend itself (parry 24%). recharge was used 33 of 100 times. The actual repair was executed in 14 out of 100 actions. survey, finally, was almost never used (less than 1%). The saboteur was walking around (goto 27%), trying to attack 14% others. parry was used just in one match but was never successful due to lack of energy. survey was executed in less than 1% the cases. buy only 1%, recharge quite often (24%). The sentinel was not really working (a lot of invalid messages) (goto 11%, parry 8%, survey less than 2%,buy less than 1%, recharge 51%). The inspector was going around (goto 17%) trying to inspect other agents (inspect 3%) but did not buy anything. Also the

`survey` was almost never used (less than 1%). Instead `recharge` was the main action (42%). However, this agent had some bug because it was sending a lot of invalid messages and even tried to `parry` although it was not allowed for this role.

5.6.5 *Simurgh*

Since the team *Simurgh* was not sending actions in time for the first two days, we only present the data from the one match against *HactarV2*. The explorer recharged in 17% the cases and was going around a lot (`goto` 38%). The `survey`-action was executed 4%. `probe` in 11 of 100 actions. `buy` was never used but `parry` (although not allowed) was tried to use in this match. The repairer did not `buy` anything either but tried to defend itself (`parry` 8%). If not attacked it was searching for agents to repair (`goto` 15%) and repaired them (`repair` 21%). When energy was missing it was recharging (`recharge` 32%). Finally, the `survey`-action was executed in less than 1% the cases. The saboteur did not try to defend itself (`parry` was not used), and did rarely `attack` (12%). Aside, the `goto` (12%) and `recharge` (16%) were the main actions. `survey` (less than 1%) and `buy` (less than 1%) were almost never used. These two agents did not that often send actions in time. The sentinel was just going around (`goto` 35%), defending itself when necessary (`parry` 4%), surveying the topology (`survey` 1%) and recharging sometimes (`recharge` 39%). `buy` was not used. Lastly, the inspector walked around (`goto` 35%) and recharged (`recharge` 41%) whenever lacking energy. The `inspect`-action was executed in less than 1% cases. This holds for `survey` also (1%). The action `buy` was not used. Additionally, `parry` was tried to use in 4% the cases although forbidden by the role.

6 Conclusion and Future Work

In this technical report, we extensively evaluated the Multi-Agent Programming Contest 2011 edition. The seventh edition of the competition introduced a completely new scenario that renewed the enthusiasm from the community and presented new challenges. The provided problem required the use of several multi-agent techniques and served as benchmark for comparing different approaches to the implementation of agents. As such, we presented here enough material to compare teams on a deeper level than just looking at the scores and we pointed out key points that affected those scores. From the analyses made, it became clear that in order to perform well in the competition, a reliable underlying platform for supporting the agent's execution was as vital as a the game-related strategies implemented.

The Multi-Agent Programming Contest will take place again in 2012 following the same track of the 2011 edition both in terms of the scenario and

the post-competition analysis. Slight modifications to the Agents-on-Mars scenario will be introduced and the number of agents competing simultaneously will be increased, augmenting the challenge. At the same time, more emphasis will be put in the recollection of information during the competition, to keep broadening the ways in which we assess the performances of the participating teams.

Acknowledgments. This work was funded by the NTH Focused Research School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.

References

[Behrens et al., 2010] Behrens, T., Dastani, M., Dix, J., Köster, M., and Novak, P., editors (2010). *Special Issue about Multi-Agent-Contest*, volume ?? of *Annals of Mathematics and Artificial Intelligence*. Springer, Netherlands.

[Behrens et al., 2012a] Behrens, T., Dix, J., Hübner, J., Köster, M., and Schlesinger, F. (2012a). Multi-agent programming contest 2011 edition documentation. Technical Report IfI-12-01, Clausthal University of Technology.

[Behrens et al., 2012b] Behrens, T., Köster, M., Schlesinger, F., Dix, J., and Hübner, J. F. (2012b). The multi-agent programming contest 2011: A résumé. In Boissier, O., Bordini, R. H., and Dennis, L., editors, *Programming Multi-Agent Systems, 9th International Workshop (ProMAS 2011)*, volume 7217 of *Lecture Notes in Computer Science*. Springer.

[Behrens et al., 2009] Behrens, T. M., Dastani, M., Dix, J., and Novák, P. (2009). Agent contest competition: 4th edition. In Hindriks, K. V., Pokahr, A., and na, S. S., editors, *Programming Multi-Agent Systems, 6th International Workshop (ProMAS 2008)*, volume 5442 of *Lecture Notes in Computer Science*, pages 211–222. Springer.

[Dastani et al., 2005] Dastani, M., Dix, J., and Novák, P. (2005). The first contest on multi-agent systems based on computational logic. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI*, volume 3900 of *Lecture Notes in Computer Science*, pages 373–384. Springer.

[Dastani et al., 2006a] Dastani, M., Dix, J., and Novák, P. (2006a). The first contest on multi-agent systems based on computational logic. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems (CLIMA*

VI), volume 3900 of *Lecture Notes in Artificial Intelligence*, pages 373–384. Springer. 6th International Workshop.

[Dastani et al., 2006b] Dastani, M., Dix, J., and Novák, P. (2006b). The second contest on multi-agent systems based on computational logic. In Inoue, K., Satoh, K., and Toni, F., editors, *Computational Logic in Multi-Agent Systems, 7th International Workshop, CLIMA VII*, volume 4371 of *Lecture Notes on Computer Science*, pages 266–283. Springer.

[Dastani et al., 2007] Dastani, M., Dix, J., and Novák, P. (2007). The second contest on multi-agent systems based on computational logic. In Inoue, K., Satoh, K., and Toni, F., editors, *Proceedings of CLIMA '06, Revised Selected and Invited Papers*, Lecture Notes in Artificial Intelligence, pages 266–283. Springer.

[Dastani et al., 2008a] Dastani, M., Dix, J., and Novák, P. (2008a). Agent contest competition - 3rd edition. In Dastani, M., Ricci, A., El Fallah Seghrouchni, A., and Winikoff, M., editors, *Proceedings of ProMAS '07, Revised Selected and Invited Papers*, Lecture Notes in Artificial Intelligence. Springer.

[Dastani et al., 2008b] Dastani, M., Dix, J., and Novák, P. (2008b). Agent contest competition - 4th edition. In *Proceedings of Sixth international Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *LNAI*. Springer Verlag.

A Detailed Team Statistics

In this Appendix we present the team's statistics for each role. However, we focus only on the second simulation and the seven interesting matches.

The tables consist of six columns. The "Status" can be either "not used" if the action was not used at all or "not allowed" if the action is not allowed to be performed by that particular role. "Action" refers to the name of the action and "Frequency" to the amount of successful and not successful executions. "Success" describes the percentage of successful executions. The "Overall" rate tells us how often this action was executed in comparison to all other actions and the last column shows how often this action was used successfully.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	92 of 92	100,00%	12,27%	12,27%
	probe	21 of 21	100,00%	2,80%	2,80%
	recharge	613 of 613	100,00%	81,73%	81,73%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	14 of 14	100,00%	1,87%	1,87%
not used	buy				
	goto	122 of 122	100,00%	16,27%	16,27%
	probe	24 of 24	100,00%	3,20%	3,20%
	recharge	584 of 584	100,00%	77,87%	77,87%
	skip	0 of 4	0,00%	0,53%	0,00%
	survey	16 of 16	100,00%	2,13%	2,13%

Table 8: HactarV2 vs. Python-DTU – HactarV2 Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	69 of 69	100,00%	9,20%	9,20%
	parry	0 of 2	0,00%	0,27%	0,00%
	recharge	646 of 646	100,00%	86,13%	86,13%
	repair	8 of 8	100,00%	1,07%	1,07%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	20 of 20	100,00%	2,67%	2,67%
not used	buy				
	goto	258 of 258	100,00%	34,40%	34,40%
	parry	0 of 17	0,00%	2,27%	0,00%
	recharge	429 of 429	100,00%	57,20%	57,20%
	repair	32 of 32	100,00%	4,27%	4,27%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	5 of 5	100,00%	0,67%	0,67%

Table 9: HactarV2 vs. Python-DTU – HactarV2 Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	532 of 532	100,00%	70,93%	70,93%
	buy	4 of 4	100,00%	0,53%	0,53%
	goto	25 of 25	100,00%	3,33%	3,33%
not used	parry				
	recharge	179 of 180	99,44%	24,00%	23,87%
	skip	0 of 9	0,00%	1,20%	0,00%
not used	survey				
	attack	527 of 527	100,00%	70,27%	70,27%
	buy	4 of 4	100,00%	0,53%	0,53%
	goto	27 of 27	100,00%	3,60%	3,60%
not used	parry				
	recharge	185 of 186	99,46%	24,80%	24,67%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 10: HactarV2 vs. Python-DTU – HactarV2 Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	69 of 69	100,00%	9,20%	9,20%
	parry	0 of 7	0,00%	0,93%	0,00%
	recharge	647 of 648	99,85%	86,40%	86,27%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	18 of 18	100,00%	2,40%	2,40%
not used	buy				
	goto	88 of 88	100,00%	11,73%	11,73%
	parry	12 of 17	70,59%	2,27%	1,60%
	recharge	631 of 632	99,84%	84,27%	84,13%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	6 of 6	100,00%	0,80%	0,80%

Table 11: HactarV2 vs. Python-DTU – HactarV2 Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	162 of 162	100,00%	21,60%	21,60%
	inspect	41 of 41	100,00%	5,47%	5,47%
	recharge	525 of 527	99,62%	70,27%	70,00%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	11 of 11	100,00%	1,47%	1,47%
not used	buy				
	goto	222 of 222	100,00%	29,60%	29,60%
	inspect	42 of 44	95,45%	5,87%	5,60%
	recharge	464 of 466	99,57%	62,13%	61,87%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	9 of 9	100,00%	1,20%	1,20%

Table 12: HactarV2 vs. Python-DTU – HactarV2 Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	83 of 83	100,00%	11,07%	11,07%
	probe	31 of 33	93,94%	4,40%	4,13%
	recharge	630 of 632	99,68%	84,27%	84,00%
	skip	0 of 2	0,00%	0,27%	0,00%
not used	survey				
not used	buy				
	goto	60 of 60	100,00%	8,00%	8,00%
	probe	29 of 29	100,00%	3,87%	3,87%
	recharge	648 of 648	100,00%	86,40%	86,40%
	skip	0 of 13	0,00%	1,73%	0,00%
not used	survey				

Table 13: HactarV2 vs. Python-DTU – Python-DTU Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	34 of 34	100,00%	4,53%	4,53%
not used	parry				
	recharge	342 of 342	100,00%	45,60%	45,60%
	repair	359 of 359	100,00%	47,87%	47,87%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	8 of 8	100,00%	1,07%	1,07%
not used	buy				
	goto	34 of 34	100,00%	4,53%	4,53%
not used	parry				
	recharge	339 of 340	99,71%	45,33%	45,20%
	repair	364 of 364	100,00%	48,53%	48,53%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	7 of 7	100,00%	0,93%	0,93%

Table 14: HactarV2 vs. Python-DTU – Python-DTU Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	26 of 26	100,00%	3,47%	3,47%
	buy	1 of 1	100,00%	0,13%	0,13%
	goto	24 of 24	100,00%	3,20%	3,20%
not used	parry				
	recharge	688 of 689	99,85%	91,87%	91,73%
	skip	0 of 10	0,00%	1,33%	0,00%
not used	survey				
	attack	49 of 61	80,33%	8,13%	6,53%
	buy	2 of 2	100,00%	0,27%	0,27%
	goto	25 of 25	100,00%	3,33%	3,33%
not used	parry				
	recharge	654 of 654	100,00%	87,20%	87,20%
	skip	0 of 8	0,00%	1,07%	0,00%
not used	survey				

Table 15: HactarV2 vs. Python-DTU – Python-DTU Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	115 of 115	100,00%	15,33%	15,33%
	parry	0 of 20	0,00%	2,67%	0,00%
	recharge	579 of 579	100,00%	77,20%	77,20%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	31 of 31	100,00%	4,13%	4,13%
not used	buy				
	goto	123 of 123	100,00%	16,40%	16,40%
	parry	0 of 158	0,00%	21,07%	0,00%
	recharge	429 of 429	100,00%	57,20%	57,20%
	skip	0 of 15	0,00%	2,00%	0,00%
	survey	24 of 25	96,00%	3,33%	3,20%

Table 16: HactarV2 vs. Python-DTU – Python-DTU Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	79 of 79	100,00%	10,53%	10,53%
	inspect	4 of 4	100,00%	0,53%	0,53%
	recharge	640 of 641	99,84%	85,47%	85,33%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	17 of 17	100,00%	2,27%	2,27%
not used	buy				
	goto	98 of 98	100,00%	13,07%	13,07%
	inspect	5 of 6	83,33%	0,80%	0,67%
	recharge	617 of 620	99,52%	82,67%	82,27%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	17 of 17	100,00%	2,27%	2,27%

Table 17: HactarV2 vs. Python-DTU – Python-DTU Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	300 of 300	100,00%	40,00%	40,00%
	probe	30 of 30	100,00%	4,00%	4,00%
	recharge	410 of 410	100,00%	54,67%	54,67%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	4 of 4	100,00%	0,53%	0,53%
not used	buy				
	goto	286 of 286	100,00%	38,13%	38,13%
	probe	25 of 25	100,00%	3,33%	3,33%
	recharge	425 of 425	100,00%	56,67%	56,67%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	8 of 8	100,00%	1,07%	1,07%

Table 18: HactarV2 vs. Simurgh – HactarV2 Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	289 of 289	100,00%	38,53%	38,53%
	parry	6 of 19	31,58%	2,53%	0,80%
	recharge	374 of 376	99,47%	50,13%	49,87%
	repair	44 of 45	97,78%	6,00%	5,87%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	15 of 15	100,00%	2,00%	2,00%
not used	buy				
	goto	301 of 301	100,00%	40,13%	40,13%
	parry	16 of 40	40,00%	5,33%	2,13%
	recharge	331 of 332	99,70%	44,27%	44,13%
	repair	59 of 59	100,00%	7,87%	7,87%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	12 of 12	100,00%	1,60%	1,60%

Table 19: HactarV2 vs. Simurgh – HactarV2 Repairer.

Status	Action	Frequency	Success	Overall	O. Success
not used	attack	198 of 316	62,66%	42,13%	26,40%
	buy	7 of 7	100,00%	0,93%	0,93%
	goto	196 of 196	100,00%	26,13%	26,13%
	parry				
	recharge	214 of 223	95,96%	29,73%	28,53%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%
not used	attack	167 of 282	59,22%	37,60%	22,27%
	buy	7 of 7	100,00%	0,93%	0,93%
	goto	223 of 223	100,00%	29,73%	29,73%
	parry				
	recharge	221 of 230	96,09%	30,67%	29,47%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 20: HactarV2 vs. Simurgh – HactarV2 Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	322 of 322	100,00%	42,93%	42,93%
	parry	14 of 28	50,00%	3,73%	1,87%
	recharge	370 of 372	99,46%	49,60%	49,33%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	19 of 19	100,00%	2,53%	2,53%
not used	buy				
	goto	319 of 319	100,00%	42,53%	42,53%
	parry	30 of 40	75,00%	5,33%	4,00%
	recharge	367 of 368	99,73%	49,07%	48,93%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	17 of 17	100,00%	2,27%	2,27%

Table 21: HactarV2 vs. Simurgh – HactarV2 Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	351 of 351	100,00%	46,80%	46,80%
	inspect	44 of 44	100,00%	5,87%	5,87%
	recharge	338 of 339	99,71%	45,20%	45,07%
	skip	0 of 4	0,00%	0,53%	0,00%
	survey	12 of 12	100,00%	1,60%	1,60%
not used	buy				
	goto	355 of 355	100,00%	47,33%	47,33%
	inspect	40 of 41	97,56%	5,47%	5,33%
	recharge	332 of 335	99,10%	44,67%	44,27%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	10 of 10	100,00%	1,33%	1,33%

Table 22: HactarV2 vs. Simurgh – HactarV2 Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	284 of 418	67,94%	55,73%	37,87%
not allowed	parry	0 of 20	0,00%	2,67%	0,00%
	probe	91 of 91	100,00%	12,13%	12,13%
	recharge	128 of 129	99,22%	17,20%	17,07%
	skip	52 of 65	80,00%	8,67%	6,93%
	survey	27 of 27	100,00%	3,60%	3,60%
not used	buy				
	goto	293 of 424	69,10%	56,53%	39,07%
not allowed	parry	0 of 14	0,00%	1,87%	0,00%
	probe	81 of 81	100,00%	10,80%	10,80%
	recharge	131 of 134	97,76%	17,87%	17,47%
	skip	64 of 67	95,52%	8,93%	8,53%
	survey	30 of 30	100,00%	4,00%	4,00%

Table 23: HactarV2 vs. Simurgh – Simurgh Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	120 of 134	89,55%	17,87%	16,00%
	parry	36 of 61	59,02%	8,13%	4,80%
	recharge	246 of 262	93,89%	34,93%	32,80%
	repair	162 of 255	63,53%	34,00%	21,60%
	skip	32 of 35	91,43%	4,67%	4,27%
	survey	3 of 3	100,00%	0,40%	0,40%
not used	buy				
	goto	112 of 132	84,85%	17,60%	14,93%
	parry	84 of 102	82,35%	13,60%	11,20%
	recharge	230 of 252	91,27%	33,60%	30,67%
	repair	160 of 226	70,80%	30,13%	21,33%
	skip	29 of 35	82,86%	4,67%	3,87%
	survey	3 of 3	100,00%	0,40%	0,40%

Table 24: HactarV2 vs. Simurgh – Simurgh Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	89 of 125	71,20%	16,67%	11,87%
	buy	5 of 13	38,46%	1,73%	0,67%
	goto	84 of 96	87,50%	12,80%	11,20%
not used	parry				
	recharge	111 of 121	91,74%	16,13%	14,80%
	skip	383 of 393	97,46%	52,40%	51,07%
	survey	2 of 2	100,00%	0,27%	0,27%
	attack	93 of 133	69,92%	17,73%	12,40%
	buy	5 of 9	55,56%	1,20%	0,67%
	goto	99 of 113	87,61%	15,07%	13,20%
not used	parry				
	recharge	128 of 134	95,52%	17,87%	17,07%
	skip	353 of 359	98,33%	47,87%	47,07%
	survey	2 of 2	100,00%	0,27%	0,27%

Table 25: HactarV2 vs. Simurgh – Simurgh Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	259 of 305	84,92%	40,67%	34,53%
	parry	37 of 65	56,92%	8,67%	4,93%
	recharge	258 of 264	97,73%	35,20%	34,40%
	skip	86 of 100	86,00%	13,33%	11,47%
	survey	8 of 16	50,00%	2,13%	1,07%
not used	buy				
	goto	260 of 324	80,25%	43,20%	34,67%
	parry	20 of 39	51,28%	5,20%	2,67%
	recharge	319 of 322	99,07%	42,93%	42,53%
	skip	38 of 48	79,17%	6,40%	5,07%
	survey	17 of 17	100,00%	2,27%	2,27%

Table 26: HactarV2 vs. Simurgh – Simurgh Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	263 of 316	83,23%	42,13%	35,07%
	inspect	4 of 4	100,00%	0,53%	0,53%
not allowed	parry	0 of 11	0,00%	1,47%	0,00%
	recharge	347 of 347	100,00%	46,27%	46,27%
	skip	56 of 64	87,50%	8,53%	7,47%
	survey	8 of 8	100,00%	1,07%	1,07%
not used	buy				
	goto	259 of 288	89,93%	38,40%	34,53%
	inspect	4 of 4	100,00%	0,53%	0,53%
not allowed	parry	0 of 34	0,00%	4,53%	0,00%
	recharge	270 of 273	98,90%	36,40%	36,00%
	skip	134 of 143	93,71%	19,07%	17,87%
	survey	8 of 8	100,00%	1,07%	1,07%

Table 27: HactarV2 vs. Simurgh – Simurgh Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	149 of 149	100,00%	19,87%	19,87%
	probe	72 of 72	100,00%	9,60%	9,60%
	recharge	513 of 513	100,00%	68,40%	68,40%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	8 of 8	100,00%	1,07%	1,07%
not used	buy				
	goto	154 of 154	100,00%	20,53%	20,53%
	probe	66 of 66	100,00%	8,80%	8,80%
	recharge	507 of 509	99,61%	67,87%	67,60%
	skip	3 of 15	20,00%	2,00%	0,40%
	survey	6 of 6	100,00%	0,80%	0,80%

Table 28: HactarV2 vs. UCDBogtrotters – HactarV2 Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	219 of 219	100,00%	29,20%	29,20%
not used	parry				
	recharge	499 of 499	100,00%	66,53%	66,53%
	repair	17 of 17	100,00%	2,27%	2,27%
	skip	0 of 2	0,00%	0,27%	0,00%
	survey	13 of 13	100,00%	1,73%	1,73%
not used	buy				
	goto	201 of 201	100,00%	26,80%	26,80%
	parry	6 of 13	46,15%	1,73%	0,80%
	recharge	457 of 461	99,13%	61,47%	60,93%
	repair	47 of 47	100,00%	6,27%	6,27%
	skip	3 of 7	42,86%	0,93%	0,40%
	survey	21 of 21	100,00%	2,80%	2,80%

Table 29: HactarV2 vs. UCDBogtrotters – HactarV2 Repairer.

Status	Action	Frequency	Success	Overall	O. Success
not used	attack	96 of 96	100,00%	12,80%	12,80%
	buy	12 of 12	100,00%	1,60%	1,60%
	goto	219 of 219	100,00%	29,20%	29,20%
	parry				
	recharge	408 of 414	98,55%	55,20%	54,40%
	skip	2 of 7	28,57%	0,93%	0,27%
	survey	2 of 2	100,00%	0,27%	0,27%
not used	attack	78 of 78	100,00%	10,40%	10,40%
	buy	12 of 12	100,00%	1,60%	1,60%
	goto	239 of 239	100,00%	31,87%	31,87%
	parry				
	recharge	404 of 409	98,78%	54,53%	53,87%
	skip	2 of 10	20,00%	1,33%	0,27%
	survey	2 of 2	100,00%	0,27%	0,27%

Table 30: HactarV2 vs. UCDBogtrotters – HactarV2 Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	179 of 179	100,00%	23,87%	23,87%
	parry	5 of 6	83,33%	0,80%	0,67%
	recharge	531 of 532	99,81%	70,93%	70,80%
	skip	1 of 8	12,50%	1,07%	0,13%
	survey	25 of 25	100,00%	3,33%	3,33%
not used	buy				
	goto	189 of 189	100,00%	25,20%	25,20%
	parry	1 of 2	50,00%	0,27%	0,13%
	recharge	528 of 528	100,00%	70,40%	70,40%
	skip	4 of 12	33,33%	1,60%	0,53%
	survey	19 of 19	100,00%	2,53%	2,53%

Table 31: HactarV2 vs. UCDBogtrotters – HactarV2 Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	235 of 235	100,00%	31,33%	31,33%
	inspect	26 of 28	92,86%	3,73%	3,47%
	recharge	467 of 467	100,00%	62,27%	62,27%
	skip	4 of 12	33,33%	1,60%	0,53%
	survey	8 of 8	100,00%	1,07%	1,07%
not used	buy				
	goto	208 of 208	100,00%	27,73%	27,73%
	inspect	31 of 31	100,00%	4,13%	4,13%
	recharge	482 of 485	99,38%	64,67%	64,27%
	skip	1 of 9	11,11%	1,20%	0,13%
	survey	17 of 17	100,00%	2,27%	2,27%

Table 32: HactarV2 vs. UCDBogtrotters – HactarV2 Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	84 of 84	100,00%	11,20%	11,20%
	probe	9 of 160	5,62%	21,33%	1,20%
	recharge	67 of 67	100,00%	8,93%	8,93%
	skip	372 of 378	98,41%	50,40%	49,60%
	survey	52 of 61	85,25%	8,13%	6,93%
not used	buy				
	goto	46 of 46	100,00%	6,13%	6,13%
	probe	12 of 133	9,02%	17,73%	1,60%
	recharge	46 of 46	100,00%	6,13%	6,13%
	skip	490 of 496	98,79%	66,13%	65,33%
	survey	20 of 29	68,97%	3,87%	2,67%

Table 33: HactarV2 vs. UCDBogtrotters – UCDBogtrotters Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy	1 of 18	5,56%	2,40%	0,13%
	goto	20 of 20	100,00%	2,67%	2,67%
	parry				
	recharge	117 of 119	98,32%	15,87%	15,60%
	repair	70 of 94	74,47%	12,53%	9,33%
	skip	483 of 492	98,17%	65,60%	64,40%
	survey	4 of 7	57,14%	0,93%	0,53%
not used	buy	0 of 4	0,00%	0,53%	0,00%
	goto	47 of 47	100,00%	6,27%	6,27%
	parry				
	recharge	269 of 271	99,26%	36,13%	35,87%
	repair	81 of 250	32,40%	33,33%	10,80%
	skip	155 of 167	92,81%	22,27%	20,67%
	survey	4 of 11	36,36%	1,47%	0,53%

Table 34: HactarV2 vs. UCDBogtrotters – UCDBogtrotters Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	62 of 245	25,31%	32,67%	8,27%
	buy	12 of 35	34,29%	4,67%	1,60%
	goto	4 of 4	100,00%	0,53%	0,53%
	parry	0 of 7	0,00%	0,93%	0,00%
	recharge	112 of 121	92,56%	16,13%	14,93%
	skip	326 of 335	97,31%	44,67%	43,47%
	survey	2 of 3	66,67%	0,40%	0,27%
	attack	57 of 269	21,19%	35,87%	7,60%
	buy	0 of 7	0,00%	0,93%	0,00%
	goto	26 of 29	89,66%	3,87%	3,47%
	parry	0 of 102	0,00%	13,60%	0,00%
	recharge	186 of 186	100,00%	24,80%	24,80%
	skip	128 of 137	93,43%	18,27%	17,07%
	survey	16 of 20	80,00%	2,67%	2,13%

Table 35: HactarV2 vs. UCDBogtrotters – UCDBogtrotters Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy	1 of 34	2,94%	4,53%	0,13%
	goto	41 of 41	100,00%	5,47%	5,47%
	parry				
	recharge	53 of 53	100,00%	7,07%	7,07%
	skip	595 of 600	99,17%	80,00%	79,33%
	survey	3 of 22	13,64%	2,93%	0,40%
not used	buy	0 of 9	0,00%	1,20%	0,00%
	goto	43 of 44	97,73%	5,87%	5,73%
	parry				
	recharge	38 of 38	100,00%	5,07%	5,07%
	skip	618 of 627	98,56%	83,60%	82,40%
	survey	13 of 32	40,62%	4,27%	1,73%

Table 36: HactarV2 vs. UCDBogtrotters – UCDBogtrotters Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	24 of 24	100,00%	3,20%	3,20%
not used	inspect				
	recharge	28 of 30	93,33%	4,00%	3,73%
	skip	673 of 680	98,97%	90,67%	89,73%
	survey	8 of 16	50,00%	2,13%	1,07%
not used	buy				
	goto	51 of 52	98,08%	6,93%	6,80%
not used	inspect				
	recharge	49 of 49	100,00%	6,53%	6,53%
	skip	617 of 620	99,52%	82,67%	82,27%
	survey	11 of 29	37,93%	3,87%	1,47%

Table 37: HactarV2 vs. UCDBogtrotters – UCDBogtrotters Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	56 of 56	100,00%	7,47%	7,47%
	probe	25 of 25	100,00%	3,33%	3,33%
	recharge	652 of 655	99,54%	87,33%	86,93%
	skip	1 of 14	7,14%	1,87%	0,13%
not used	survey				
not used	buy				
	goto	102 of 102	100,00%	13,60%	13,60%
	probe	32 of 32	100,00%	4,27%	4,27%
	recharge	599 of 605	99,01%	80,67%	79,87%
	skip	2 of 11	18,18%	1,47%	0,27%
not used	survey				

Table 38: Python-DTU vs. UCDBogtrotters – Python-DTU Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	93 of 93	100,00%	12,40%	12,40%
not used	parry				
	recharge	501 of 510	98,24%	68,00%	66,80%
	repair	131 of 131	100,00%	17,47%	17,47%
	skip	1 of 9	11,11%	1,20%	0,13%
	survey	7 of 7	100,00%	0,93%	0,93%
not used	buy				
	goto	86 of 86	100,00%	11,47%	11,47%
not used	parry				
	recharge	502 of 503	99,80%	67,07%	66,93%
	repair	149 of 149	100,00%	19,87%	19,87%
	skip	1 of 4	25,00%	0,53%	0,13%
	survey	8 of 8	100,00%	1,07%	1,07%

Table 39: Python-DTU vs. UCDBogtrotters – Python-DTU Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	181 of 393	46,06%	52,40%	24,13%
	buy	7 of 7	100,00%	0,93%	0,93%
	goto	104 of 104	100,00%	13,87%	13,87%
not used	parry				
	recharge	236 of 241	97,93%	32,13%	31,47%
	skip	1 of 5	20,00%	0,67%	0,13%
not used	survey				
	attack	166 of 383	43,34%	51,07%	22,13%
	buy	7 of 7	100,00%	0,93%	0,93%
	goto	120 of 120	100,00%	16,00%	16,00%
not used	parry				
	recharge	227 of 230	98,70%	30,67%	30,27%
	skip	1 of 10	10,00%	1,33%	0,13%
not used	survey				

Table 40: Python-DTU vs. UCDBogtrotters – Python-DTU Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	64 of 64	100,00%	8,53%	8,53%
	parry	45 of 65	69,23%	8,67%	6,00%
	recharge	591 of 592	99,83%	78,93%	78,80%
	skip	1 of 6	16,67%	0,80%	0,13%
	survey	22 of 23	95,65%	3,07%	2,93%
not used	buy				
	goto	47 of 47	100,00%	6,27%	6,27%
	parry	9 of 12	75,00%	1,60%	1,20%
	recharge	655 of 655	100,00%	87,33%	87,33%
	skip	1 of 7	14,29%	0,93%	0,13%
	survey	29 of 29	100,00%	3,87%	3,87%

Table 41: Python-DTU vs. UCDBogtrotters – Python-DTU Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	93 of 93	100,00%	12,40%	12,40%
	inspect	5 of 5	100,00%	0,67%	0,67%
	recharge	623 of 628	99,20%	83,73%	83,07%
	skip	1 of 9	11,11%	1,20%	0,13%
	survey	15 of 15	100,00%	2,00%	2,00%
not used	buy				
	goto	74 of 74	100,00%	9,87%	9,87%
	inspect	5 of 5	100,00%	0,67%	0,67%
	recharge	638 of 642	99,38%	85,60%	85,07%
	skip	1 of 6	16,67%	0,80%	0,13%
	survey	23 of 23	100,00%	3,07%	3,07%

Table 42: Python-DTU vs. UCDBogtrotters – Python-DTU Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	257 of 265	96,98%	35,33%	34,27%
not allowed	parry	0 of 95	0,00%	12,67%	0,00%
	probe	16 of 33	48,48%	4,40%	2,13%
	recharge	329 of 330	99,70%	44,00%	43,87%
	skip	15 of 21	71,43%	2,80%	2,00%
	survey	6 of 6	100,00%	0,80%	0,80%
not used	buy				
	goto	293 of 308	95,13%	41,07%	39,07%
not allowed	parry	0 of 57	0,00%	7,60%	0,00%
	probe	24 of 29	82,76%	3,87%	3,20%
	recharge	325 of 326	99,69%	43,47%	43,33%
	skip	17 of 23	73,91%	3,07%	2,27%
	survey	6 of 7	85,71%	0,93%	0,80%

Table 43: Python-DTU vs. UCDBogtrotters – UCDBogtrotters Explorer.

Status	Action	Frequency	Success	Overall	O. Success
	buy	1 of 3	33,33%	0,40%	0,13%
	goto	31 of 31	100,00%	4,13%	4,13%
	parry	114 of 186	61,29%	24,80%	15,20%
	recharge	242 of 289	83,74%	38,53%	32,27%
	repair	169 of 217	77,88%	28,93%	22,53%
	skip	14 of 21	66,67%	2,80%	1,87%
	survey	3 of 3	100,00%	0,40%	0,40%
not used	buy				
	goto	9 of 9	100,00%	1,20%	1,20%
	parry	103 of 284	36,27%	37,87%	13,73%
	recharge	217 of 243	89,30%	32,40%	28,93%
	repair	141 of 192	73,44%	25,60%	18,80%
	skip	15 of 21	71,43%	2,80%	2,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 44: Python-DTU vs. UCDBogtrotters – UCDBogtrotters Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	119 of 155	76,77%	20,67%	15,87%
	buy	12 of 17	70,59%	2,27%	1,60%
	goto	298 of 298	100,00%	39,73%	39,73%
not used	parry				
	recharge	253 of 258	98,06%	34,40%	33,73%
	skip	15 of 22	68,18%	2,93%	2,00%
not used	survey				
	attack	84 of 103	81,55%	13,73%	11,20%
	buy	6 of 17	35,29%	2,27%	0,80%
	goto	336 of 336	100,00%	44,80%	44,80%
not used	parry				
	recharge	267 of 271	98,52%	36,13%	35,60%
	skip	15 of 23	65,22%	3,07%	2,00%
not used	survey				

Table 45: Python-DTU vs. UCDBogtrotters – UCDBogtrotters Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
	buy	7 of 10	70,00%	1,33%	0,93%
	goto	139 of 139	100,00%	18,53%	18,53%
	parry	92 of 191	48,17%	25,47%	12,27%
	recharge	356 of 367	97,00%	48,93%	47,47%
	skip	16 of 24	66,67%	3,20%	2,13%
	survey	19 of 19	100,00%	2,53%	2,53%
not used	buy				
	goto	158 of 158	100,00%	21,07%	21,07%
	parry	87 of 297	29,29%	39,60%	11,60%
	recharge	254 of 266	95,49%	35,47%	33,87%
	skip	15 of 23	65,22%	3,07%	2,00%
	survey	5 of 6	83,33%	0,80%	0,67%

Table 46: Python-DTU vs. UCDBogtrotters – UCDBogtrotters Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	180 of 181	99,45%	24,13%	24,00%
	inspect	12 of 12	100,00%	1,60%	1,60%
not allowed	parry	0 of 194	0,00%	25,87%	0,00%
	recharge	323 of 324	99,69%	43,20%	43,07%
	skip	16 of 30	53,33%	4,00%	2,13%
	survey	9 of 9	100,00%	1,20%	1,20%
not used	buy				
	goto	304 of 304	100,00%	40,53%	40,53%
	inspect	4 of 17	23,53%	2,27%	0,53%
not allowed	parry	0 of 49	0,00%	6,53%	0,00%
	recharge	349 of 349	100,00%	46,53%	46,53%
	skip	19 of 29	65,52%	3,87%	2,53%
	survey	2 of 2	100,00%	0,27%	0,27%

Table 47: Python-DTU vs. UCDBogtrotters – UCDBogtrotters Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	257 of 257	100,00%	34,27%	34,27%
	probe	38 of 38	100,00%	5,07%	5,07%
	recharge	439 of 439	100,00%	58,53%	58,53%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	9 of 9	100,00%	1,20%	1,20%
not used	buy				
	goto	291 of 291	100,00%	38,80%	38,80%
	probe	30 of 30	100,00%	4,00%	4,00%
	recharge	402 of 403	99,75%	53,73%	53,60%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	19 of 19	100,00%	2,53%	2,53%

Table 48: HactarV2 vs. TUB – HactarV2 Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	196 of 196	100,00%	26,13%	26,13%
	parry	39 of 133	29,32%	17,73%	5,20%
	recharge	229 of 253	90,51%	33,73%	30,53%
	repair	147 of 151	97,35%	20,13%	19,60%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	11 of 11	100,00%	1,47%	1,47%
not used	buy				
	goto	210 of 210	100,00%	28,00%	28,00%
	parry	19 of 132	14,39%	17,60%	2,53%
	recharge	228 of 237	96,20%	31,60%	30,40%
	repair	153 of 158	96,84%	21,07%	20,40%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	8 of 8	100,00%	1,07%	1,07%

Table 49: HactarV2 vs. TUB – HactarV2 Repairer.

Status	Action	Frequency	Success	Overall	O. Success
not used	attack	329 of 329	100,00%	43,87%	43,87%
	buy	13 of 13	100,00%	1,73%	1,73%
	goto	178 of 178	100,00%	23,73%	23,73%
	parry				
	recharge	208 of 221	94,12%	29,47%	27,73%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%
not used	attack	307 of 307	100,00%	40,93%	40,93%
	buy	13 of 13	100,00%	1,73%	1,73%
	goto	171 of 171	100,00%	22,80%	22,80%
	parry				
	recharge	240 of 249	96,39%	33,20%	32,00%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	3 of 3	100,00%	0,40%	0,40%

Table 50: HactarV2 vs. TUB – HactarV2 Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	264 of 264	100,00%	35,20%	35,20%
	parry	54 of 99	54,55%	13,20%	7,20%
	recharge	354 of 361	98,06%	48,13%	47,20%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	18 of 18	100,00%	2,40%	2,40%
not used	buy				
	goto	272 of 272	100,00%	36,27%	36,27%
	parry	50 of 86	58,14%	11,47%	6,67%
	recharge	370 of 378	97,88%	50,40%	49,33%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	7 of 7	100,00%	0,93%	0,93%

Table 51: HactarV2 vs. TUB – HactarV2 Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	299 of 299	100,00%	39,87%	39,87%
	inspect	5 of 5	100,00%	0,67%	0,67%
	recharge	425 of 427	99,53%	56,93%	56,67%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	13 of 13	100,00%	1,73%	1,73%
not used	buy				
	goto	330 of 330	100,00%	44,00%	44,00%
	inspect	31 of 32	96,88%	4,27%	4,13%
	recharge	368 of 371	99,19%	49,47%	49,07%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	8 of 8	100,00%	1,07%	1,07%

Table 52: HactarV2 vs. TUB – HactarV2 Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	369 of 370	99,73%	49,33%	49,20%
	probe	55 of 57	96,49%	7,60%	7,33%
	recharge	277 of 279	99,28%	37,20%	36,93%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	36 of 36	100,00%	4,80%	4,80%
not used	buy				
	goto	374 of 374	100,00%	49,87%	49,87%
	probe	61 of 61	100,00%	8,13%	8,13%
	recharge	261 of 263	99,24%	35,07%	34,80%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	45 of 45	100,00%	6,00%	6,00%

Table 53: HactarV2 vs. TUB – TUB Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	209 of 209	100,00%	27,87%	27,87%
not used	parry				
	recharge	268 of 279	96,06%	37,20%	35,73%
	repair	250 of 250	100,00%	33,33%	33,33%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%
not used	buy				
	goto	155 of 155	100,00%	20,67%	20,67%
not used	parry				
	recharge	269 of 278	96,76%	37,07%	35,87%
	repair	306 of 308	99,35%	41,07%	40,80%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 54: HactarV2 vs. TUB – TUB Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	165 of 249	66,27%	33,20%	22,00%
	buy	12 of 12	100,00%	1,60%	1,60%
	goto	190 of 190	100,00%	25,33%	25,33%
not used	parry				
	recharge	273 of 284	96,13%	37,87%	36,40%
	skip	0 of 14	0,00%	1,87%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%
	attack	252 of 330	76,36%	44,00%	33,60%
	buy	10 of 10	100,00%	1,33%	1,33%
	goto	124 of 124	100,00%	16,53%	16,53%
not used	parry				
	recharge	272 of 275	98,91%	36,67%	36,27%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 55: HactarV2 vs. TUB – TUB Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	401 of 401	100,00%	53,47%	53,47%
not used	parry				
	recharge	335 of 337	99,41%	44,93%	44,67%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	5 of 5	100,00%	0,67%	0,67%
not used	buy				
	goto	393 of 393	100,00%	52,40%	52,40%
not used	parry				
	recharge	337 of 340	99,12%	45,33%	44,93%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	7 of 7	100,00%	0,93%	0,93%

Table 56: HactarV2 vs. TUB – TUB Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	395 of 395	100,00%	52,67%	52,67%
	inspect	2 of 2	100,00%	0,27%	0,27%
	recharge	340 of 342	99,42%	45,60%	45,33%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	5 of 5	100,00%	0,67%	0,67%
not used	buy				
	goto	418 of 418	100,00%	55,73%	55,73%
	inspect	6 of 6	100,00%	0,80%	0,80%
	recharge	311 of 315	98,73%	42,00%	41,47%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%

Table 57: HactarV2 vs. TUB – TUB Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	105 of 105	100,00%	14,00%	14,00%
	probe	32 of 32	100,00%	4,27%	4,27%
	recharge	599 of 601	99,67%	80,13%	79,87%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%
not used	buy				
	goto	43 of 43	100,00%	5,73%	5,73%
	probe	25 of 25	100,00%	3,33%	3,33%
	recharge	672 of 674	99,70%	89,87%	89,60%
	skip	0 of 8	0,00%	1,07%	0,00%
not used	survey				

Table 58: Python-DTU vs. TUB – Python-DTU Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	106 of 106	100,00%	14,13%	14,13%
not used	parry				
	recharge	304 of 311	97,75%	41,47%	40,53%
	repair	311 of 320	97,19%	42,67%	41,47%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	5 of 5	100,00%	0,67%	0,67%
not used	buy				
	goto	119 of 119	100,00%	15,87%	15,87%
not used	parry				
	recharge	308 of 330	93,33%	44,00%	41,07%
	repair	279 of 286	97,55%	38,13%	37,20%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	5 of 5	100,00%	0,67%	0,67%

Table 59: Python-DTU vs. TUB – Python-DTU Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	214 of 215	99,53%	28,67%	28,53%
	buy	7 of 7	100,00%	0,93%	0,93%
	goto	228 of 228	100,00%	30,40%	30,40%
not used	parry				
	recharge	267 of 294	90,82%	39,20%	35,60%
	skip	0 of 6	0,00%	0,80%	0,00%
not used	survey				
	attack	243 of 243	100,00%	32,40%	32,40%
	buy	7 of 7	100,00%	0,93%	0,93%
	goto	159 of 159	100,00%	21,20%	21,20%
not used	parry				
	recharge	238 of 333	71,47%	44,40%	31,73%
	skip	0 of 8	0,00%	1,07%	0,00%
not used	survey				

Table 60: Python-DTU vs. TUB – Python-DTU Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	98 of 98	100,00%	13,07%	13,07%
	parry	1 of 5	20,00%	0,67%	0,13%
	recharge	607 of 607	100,00%	80,93%	80,93%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	32 of 32	100,00%	4,27%	4,27%
not used	buy				
	goto	101 of 101	100,00%	13,47%	13,47%
	parry	8 of 18	44,44%	2,40%	1,07%
	recharge	595 of 596	99,83%	79,47%	79,33%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	28 of 28	100,00%	3,73%	3,73%

Table 61: Python-DTU vs. TUB – Python-DTU Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	84 of 84	100,00%	11,20%	11,20%
not used	inspect				
	recharge	649 of 652	99,54%	86,93%	86,53%
	skip	0 of 10	0,00%	1,33%	0,00%
	survey	4 of 4	100,00%	0,53%	0,53%
not used	buy				
	goto	102 of 102	100,00%	13,60%	13,60%
not used	inspect				
	recharge	632 of 636	99,37%	84,80%	84,27%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	6 of 6	100,00%	0,80%	0,80%

Table 62: Python-DTU vs. TUB – Python-DTU Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	407 of 407	100,00%	54,27%	54,27%
	probe	48 of 48	100,00%	6,40%	6,40%
	recharge	251 of 254	98,82%	33,87%	33,47%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	34 of 34	100,00%	4,53%	4,53%
not used	buy				
	goto	391 of 391	100,00%	52,13%	52,13%
	probe	51 of 51	100,00%	6,80%	6,80%
	recharge	255 of 260	98,08%	34,67%	34,00%
	skip	0 of 12	0,00%	1,60%	0,00%
	survey	36 of 36	100,00%	4,80%	4,80%

Table 63: Python-DTU vs. TUB – TUB Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	229 of 229	100,00%	30,53%	30,53%
not used	parry				
	recharge	207 of 214	96,73%	28,53%	27,60%
	repair	295 of 297	99,33%	39,60%	39,33%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%
not used	buy				
	goto	207 of 207	100,00%	27,60%	27,60%
not used	parry				
	recharge	197 of 199	98,99%	26,53%	26,27%
	repair	325 of 332	97,89%	44,27%	43,33%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	4 of 4	100,00%	0,53%	0,53%

Table 64: Python-DTU vs. TUB – TUB Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	308 of 313	98,40%	41,73%	41,07%
	buy	4 of 4	100,00%	0,53%	0,53%
	goto	159 of 159	100,00%	21,20%	21,20%
not used	parry				
	recharge	261 of 262	99,62%	34,93%	34,80%
	skip	0 of 11	0,00%	1,47%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%
	attack	370 of 374	98,93%	49,87%	49,33%
	buy	4 of 4	100,00%	0,53%	0,53%
	goto	93 of 93	100,00%	12,40%	12,40%
not used	parry				
	recharge	270 of 271	99,63%	36,13%	36,00%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 65: Python-DTU vs. TUB – TUB Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	404 of 404	100,00%	53,87%	53,87%
not used	parry				
	recharge	330 of 331	99,70%	44,13%	44,00%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	7 of 7	100,00%	0,93%	0,93%
not used	buy				
	goto	385 of 385	100,00%	51,33%	51,33%
not used	parry				
	recharge	349 of 354	98,59%	47,20%	46,53%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	4 of 4	100,00%	0,53%	0,53%

Table 66: Python-DTU vs. TUB – TUB Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	406 of 406	100,00%	54,13%	54,13%
	inspect	2 of 2	100,00%	0,27%	0,27%
	recharge	328 of 333	98,50%	44,40%	43,73%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%
not used	buy				
	goto	380 of 380	100,00%	50,67%	50,67%
	inspect	10 of 10	100,00%	1,33%	1,33%
	recharge	348 of 351	99,15%	46,80%	46,40%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	4 of 4	100,00%	0,53%	0,53%

Table 67: Python-DTU vs. TUB – TUB Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	412 of 413	99,76%	55,07%	54,93%
	probe	42 of 44	95,45%	5,87%	5,60%
	recharge	259 of 260	99,62%	34,67%	34,53%
	skip	0 of 3	0,00%	0,40%	0,00%
	survey	30 of 30	100,00%	4,00%	4,00%
not used	buy				
	goto	392 of 392	100,00%	52,27%	52,27%
	probe	60 of 61	98,36%	8,13%	8,00%
	recharge	250 of 250	100,00%	33,33%	33,33%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	39 of 40	97,50%	5,33%	5,20%

Table 68: TUB vs. UCDBogtrotters – TUB Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	359 of 359	100,00%	47,87%	47,87%
not used	parry				
	recharge	271 of 279	97,13%	37,20%	36,13%
	repair	99 of 99	100,00%	13,20%	13,20%
	skip	0 of 8	0,00%	1,07%	0,00%
	survey	5 of 5	100,00%	0,67%	0,67%
not used	buy				
	goto	333 of 333	100,00%	44,40%	44,40%
not used	parry				
	recharge	265 of 270	98,15%	36,00%	35,33%
	repair	137 of 137	100,00%	18,27%	18,27%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 69: TUB vs. UCDBogtrotters – TUB Repairer.

Status	Action	Frequency	Success	Overall	O. Success
not used	attack	137 of 265	51,70%	35,33%	18,27%
	buy	11 of 11	100,00%	1,47%	1,47%
	goto	200 of 200	100,00%	26,67%	26,67%
	parry				
	recharge	260 of 266	97,74%	35,47%	34,67%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%
not used	attack	221 of 306	72,22%	40,80%	29,47%
	buy	12 of 12	100,00%	1,60%	1,60%
	goto	139 of 139	100,00%	18,53%	18,53%
	parry				
	recharge	272 of 283	96,11%	37,73%	36,27%
	skip	0 of 9	0,00%	1,20%	0,00%
	survey	1 of 1	100,00%	0,13%	0,13%

Table 70: TUB vs. UCDBogtrotters – TUB Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	422 of 422	100,00%	56,27%	56,27%
not used	parry				
	recharge	314 of 316	99,37%	42,13%	41,87%
	skip	0 of 6	0,00%	0,80%	0,00%
	survey	6 of 6	100,00%	0,80%	0,80%
not used	buy				
	goto	438 of 438	100,00%	58,40%	58,40%
not used	parry				
	recharge	298 of 300	99,33%	40,00%	39,73%
	skip	0 of 5	0,00%	0,67%	0,00%
	survey	7 of 7	100,00%	0,93%	0,93%

Table 71: TUB vs. UCDBogtrotters – TUB Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	433 of 433	100,00%	57,73%	57,73%
	inspect	3 of 3	100,00%	0,40%	0,40%
	recharge	299 of 300	99,67%	40,00%	39,87%
	skip	0 of 12	0,00%	1,60%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%
not used	buy				
	goto	427 of 427	100,00%	56,93%	56,93%
	inspect	7 of 8	87,50%	1,07%	0,93%
	recharge	306 of 306	100,00%	40,80%	40,80%
	skip	0 of 7	0,00%	0,93%	0,00%
	survey	2 of 2	100,00%	0,27%	0,27%

Table 72: TUB vs. UCDBogtrotters – TUB Inspector.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	229 of 229	100,00%	30,53%	30,53%
not allowed	parry	0 of 40	0,00%	5,33%	0,00%
	probe	23 of 23	100,00%	3,07%	3,07%
	recharge	431 of 433	99,54%	57,73%	57,47%
	skip	4 of 10	40,00%	1,33%	0,53%
	survey	15 of 15	100,00%	2,00%	2,00%
not used	buy				
	goto	153 of 153	100,00%	20,40%	20,40%
not allowed	parry	0 of 121	0,00%	16,13%	0,00%
	probe	12 of 13	92,31%	1,73%	1,60%
	recharge	441 of 441	100,00%	58,80%	58,80%
	skip	4 of 12	33,33%	1,60%	0,53%
	survey	10 of 10	100,00%	1,33%	1,33%

Table 73: TUB vs. UCDBogtrotters – UCDBogtrotters Explorer.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	153 of 153	100,00%	20,40%	20,40%
	parry	34 of 73	46,58%	9,73%	4,53%
	recharge	279 of 283	98,59%	37,73%	37,20%
	repair	174 of 225	77,33%	30,00%	23,20%
	skip	4 of 8	50,00%	1,07%	0,53%
	survey	6 of 8	75,00%	1,07%	0,80%
not used	buy				
	goto	83 of 83	100,00%	11,07%	11,07%
	parry	14 of 69	20,29%	9,20%	1,87%
	recharge	377 of 382	98,69%	50,93%	50,27%
	repair	147 of 192	76,56%	25,60%	19,60%
	skip	4 of 12	33,33%	1,60%	0,53%
	survey	11 of 12	91,67%	1,60%	1,47%

Table 74: TUB vs. UCDBogtrotters – UCDBogtrotters Repairer.

Status	Action	Frequency	Success	Overall	O. Success
	attack	200 of 200	100,00%	26,67%	26,67%
	buy	12 of 12	100,00%	1,60%	1,60%
	goto	228 of 256	89,06%	34,13%	30,40%
not used	parry				
	recharge	258 of 272	94,85%	36,27%	34,40%
	skip	4 of 9	44,44%	1,20%	0,53%
	survey	1 of 1	100,00%	0,13%	0,13%
	attack	113 of 113	100,00%	15,07%	15,07%
	buy	9 of 10	90,00%	1,33%	1,20%
	goto	312 of 313	99,68%	41,73%	41,60%
not used	parry				
	recharge	291 of 303	96,04%	40,40%	38,80%
	skip	5 of 11	45,45%	1,47%	0,67%
not used	survey				

Table 75: TUB vs. UCDBogtrotters – UCDBogtrotters Saboteur.

Status	Action	Frequency	Success	Overall	O. Success
	buy	5 of 6	83,33%	0,80%	0,67%
	goto	57 of 57	100,00%	7,60%	7,60%
	parry	100 of 148	67,57%	19,73%	13,33%
	recharge	505 of 515	98,06%	68,67%	67,33%
	skip	4 of 11	36,36%	1,47%	0,53%
	survey	13 of 13	100,00%	1,73%	1,73%
not used	buy				
	goto	56 of 56	100,00%	7,47%	7,47%
	parry	65 of 95	68,42%	12,67%	8,67%
	recharge	565 of 569	99,30%	75,87%	75,33%
	skip	4 of 11	36,36%	1,47%	0,53%
	survey	18 of 19	94,74%	2,53%	2,40%

Table 76: TUB vs. UCDBogtrotters – UCDBogtrotters Sentinel.

Status	Action	Frequency	Success	Overall	O. Success
not used	buy				
	goto	119 of 119	100,00%	15,87%	15,87%
	inspect	24 of 24	100,00%	3,20%	3,20%
not allowed	parry	0 of 19	0,00%	2,53%	0,00%
	recharge	564 of 568	99,30%	75,73%	75,20%
	skip	4 of 15	26,67%	2,00%	0,53%
	survey	5 of 5	100,00%	0,67%	0,67%
not used	buy				
	goto	93 of 93	100,00%	12,40%	12,40%
	inspect	42 of 42	100,00%	5,60%	5,60%
not allowed	parry	0 of 19	0,00%	2,53%	0,00%
	recharge	578 of 578	100,00%	77,07%	77,07%
	skip	4 of 12	33,33%	1,60%	0,53%
	survey	6 of 6	100,00%	0,80%	0,80%

Table 77: TUB vs. UCDBogtrotters – UCDBogtrotters Inspector.

Part II
In-Depth Team Descriptions

Multi-Agent Programming Contest 2011 Techinical Report Template

[Authors]

[Affiliation]

Abstract. Please follow the given template structure for your submission. For each section, we are proposing some questions that the reader of the paper should be able to answer after reading it. (Please note that the technical report format is not like the "form" used in the registration template. A *paper* style should be used.)
If you encounter any problems with some of the questions please do not hesitate to request clarification on the mailing list.
Max number of pages: 10
Deadline: **20/10/2011**
Format: LNCS
`http://www.springer.com/computer/lncs?SGWID=0-164-6-793341-0`
Please submit your contribution via `http://https://www.easychair.org/conferences/?conf=mapc11` (menu TechReport).

1 Introduction

1. What was the motivation to participate in the contest?
2. What is the (brief) history of the team? (MAS course project, thesis evaluation, ...)
3. What is the name of your team?
4. How many developers and designers did you have? At what level of education are your team members?
5. From which field of research do you come from? Which work is related?

2 System Analysis and Design

1. Did you use multi-agent programming languages? Please justify your answer.
2. If some multi-agent system methodology such as Prometheus, O-MaSE, or Tropos was used, how did you use it? If you did not, please justify.
3. Is the solution based on the centralisation of coordination/information on a specific agent? Conversely if you plan a decentralised solution, which strategy do you plan to use?
4. What is the communication strategy and how complex is it?
5. How are the following agent features considered/implemented: *autonomy*, *proactiveness*, *reactiveness*?

6. Is the team a truly **multi**-agent system or rather a centralised system in disguise?
7. How much time (man hours) have you invested (approximately) for implementing your team?
8. Did you discuss the design and strategies of you agent team with other developers? To which extend did your test your agents playing with other teams.

3 Software Architecture

1. Which programming language did you use to implement the multi-agent system?
2. How have you mapped the designed architecture (both multi-agent and individual agent architectures) to programming codes, i.e., how did you implement specific agent-oriented concepts and designed artifacts using the programming language?
3. Which development platforms and tools are used? How much time did you invest in learning those?
4. Which runtime platforms and tools (e.g. Jade, AgentScape, simply Java, ...) are used? How much time did you invest in learning those?
5. What features were missing in your language choice that would have facilitated your development task?
6. Which algorithms are used/implemented?
7. How did you distribute the agents on several machines? And if you did not please justify why.
8. To which extend is the reasoning of your agents synchronized with the receive-percepts/send-action cycle?
9. What part of the development was most difficult/complex? What kind of problems have you found and how are they solved?
10. How many lines of code did you write for your software?

4 Strategies, Details and Statistics

1. What is the main strategy of your team?
2. How does the overall team work together? (coordination, information sharing, ...)
3. How do your agents analyze the topology of the map? And how do they exploit their findings?
4. How do your agents communicate with the server?
5. How do you implement the roles of the agents? Which strategies do the different roles implement?
6. How do you find good zones? How do you estimate the value of zones?
7. How do you conquer zones? How do you defend zones if attacked? Do you attack zones?

8. Can your agents change their behavior during runtime? If so, what triggers the changes?
9. What algorithm(s) do you use for agent path planning?
10. How do you make use of the buying-mechanism?
11. How important are achievements for your overall strategy?
12. Do your agents have an explicit mental state?
13. How do your agents communicate? And what do they communicate?
14. How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?
15. Is most of you agents' behavior emergent on and individual and team level?
16. If you agents perform some planning, how many steps do they plan ahead.
17. If you have a perceive-think-act cycle, how is it synchronized with the server?

5 Conclusion

1. What have you learned from the participation in the contest?
2. Which are the strong and weak points of the team?
3. How suitable was the chosen programming language, methodology, tools, and algorithms?
4. What can be improved in the context for next year?
5. Why did your team perform as it did? Why did the other teams perform better/worse than you did.
6. Which other research fields might be interested in the Multi-Agent Programming Contest?
7. How can the current scenario be optimized? How would those optimization pay off?

Short Answers

Please provide short answers to all the questions in a separate section. This does not count for the 10 pages limit.

Bogtrotters in Space

Dominic Carr, Sean Russell, Balazs Pete, G.M.P. O'Hare, Rem W. Collier

University College Dublin

Abstract. This is the fourth year in which a team from University College Dublin has participated in the Multi-Agent Programming Contest[1]. This paper describes the system that was created to participate in the contest, along with observations of the team's experiences in the contest. The system itself was built using the AF-TeleoReactive and AF-AgentSpeak agent programming languages running on the Agent Factory platform. Unlike in previous years where a hybrid control architecture was used, this year the system was implemented using only agent code and associated actions, sensors, modules and platform services.

1 Introduction

This years entry to the 2011 Multi-Agent Programming Contest was designed and built using the Agent Factory framework, which provides support for the development and deployment of agent-based applications using a variety of Agent-Oriented Programming (AOP) languages [Collier, 2002]. As with the previous years entry, our approach was centered around the use of two specific AOP languages: *AF-AgentSpeak*, an AgentSpeak variant that is based on Jason [Bordini et al., 2008] and *AF-TeleoReactive*, an implementation of Nilssons Teleo-Reactive programming language [Nilsson, 1994]. Both of these langages were implemented using the Common Language Framework [Russell et al., 2011], a set of reusable components that aims to simplify the prototyping of AOP languages.

This is the fourth year in which a team from University College Dublin has participated in the contest. Last year [rus,] the team performed well in the herding scenario coming in third in the contest, that entry building on the work of the two preceding years [Jordan et al.,][Dragone et al., 2009]. This years team included four members from the previous year: Rem Collier (lecturer), Sean Russell (Ph.D. Student), Dominic Carr (Ph.D. Student) and Gregory O'Hare (professor). Rem and Gregory are active researchers in the area of Multi-Agent Systems and AOP languages, Sean and Dominic are working in the area of agent-enabled Wireless Sensor Networks. The final member of the team was Balazs Pete, a 2nd year undergraduate student who worked on the contest as part of a summer internship.

Our primary motivation to compete in the contest was to test and debug AgentFactory, and further refine and direct its development trajectory. As in in previous years we were strongly motivated to provide new researchers with practical exposure to AOP using Agent Factory. The contest fits this need, well providing an interesting problem to solve. We also wanted to drive language development within AgentFactory.

To this end, one of our goals was to remove our dependency on a behaviour-based architecture that had previously been used to implement core behaviours of the system. Instead, we aimed to increase our utilization of AOP languages, and to replace the behavioural layer with Teleo-Reactive functions. Additionally, we sought to use the Environment Interface Standard (EIS)[Behrens et al., 2011] integration provided with the contest server instead of building a custom solution. Further details of our approach can be found in section 3.

[1] http://www.multiagentcontest.org/2011

2 System Analysis and Design

The development model used was based on a team programming approach to system development that was adopted for the previous contest [rus,]. At any point in time, one team member was actively engaged with coding and the other team members provided strategy analysis, debugging assistance etc. Whenever the team identified a possible strategy, a ”champion” was assigned to seperate from the main group and to flesh out the idea. Once finished, the ”champion” then presented the idea to the rest of the group. If accepted by the group, the idea was prioritised and added to the to do list.

The overall approach adopted was decided upon at the start of our involvement in the project. In essence, our objective was to maintain the centralised task allocation model used in the previous architecture, but to replace the low level behaviours with teleo-reactive functions and EIS integration. All of our analysis and design work was targeted at solutions which were compatible with this general architecture. We did not consider any of the software engineering methodologies outlined in the literature. Where necessary, Agent UML Protocol Diagrams were used to illustrate interaction protocols and pseudo code, adapted to our planning language, was used to outline plans. For the AF-TeleoReactive programs, a function hierarchy diagram was used to outline designs.

In our architecture coordination of information is carried out through the use of a number of platform services, which represent shared platform wide resources for the agents. The primary service is the map service, in simplest terms this service could be viewed as a whiteboard where agents could post relevant information to be made available to other agents, secondly it also allowed some analysis of the data to be performed e.g. identifying the highest value zones or routing between two vertices in the graph. Team coordination was achieved through the use of another platform service, the group service, which was used by the leader agent to assign tasks to the individual agents.

The attributes of *autonomy*, *pro-activeness* and *reactiveness* were implemented by making use of two AOP languages and more specifically through the structure of the *AF-TeleoReactive* programs which can allow the agents to bypass their assigned task in favour one of their own goals.

Our system is a true multi-agent system with centralised coordination. The choice of centralised coordination was made in an effort to allow the rapid prototyping of different task allocation strategies during development while abstracting from communication issues. In order to facilitate this centralised coordination, a leader agent was specified to complement the in-situ ATPV agents. This strategist agent performs some rudimentary analysis of the graph to determine the cluster of vertices with the highest value, then based on the structure of the graph a number of vertices are selected on which to position agents, which are then associated with individual agents.

During the course of the contest or prior to this we did not discuss our strategies with any other teams, this was primarily in an effort to remain competitive as our experience from participation in last years contest highlighted the importance of effective strategies. To this end we only participated in a single test match which we used to ensure we could successfully connect to the server and perform actions in a reasonable time.

The implementation of the system took approximately 600 hours, the majority of this time can be attributed to an internship undertaken by Balazs Pete over a 12 week period from June until the end of August. The remainder of the time can be accounted for when the rest of the team picked up the development two weeks before the contest.

3 Software Architecture

As in previous years, our system utilised Agent Factory (AF) [Collier, 2002] as our underlying agent technology. AF is an open-source Java-based framework that provides support for the development and deployment of agent-oriented applications. Specifically, it provides a generic Run-Time Environment (RTE) for deploying agent-based systems that is based on the FIPA standards [Poslad et al., 2000] together with a set of *development kits* that facilitate the implementation of diverse agent types, ranging from custom agent architectures to agent programming languages.

The RTE consists of a set of configurable agent platforms that contain the machinery necessary to deploy these agent types together with support for the deployment of platform-level resources, known as platform services, that are shared amongst the agents residing on the platform. Other support includes a range of monitoring and inspection tools that aid the developer in debugging their implementations.

The development kits provide the core agent interpreter/architecture together with appropriate customisations for the AF tool support. This will typically include a set of plugins for the AF Debugging Tool and an Eclipse plugin for the AF IDE (which is a set of plugins for Eclipse). For the purposes of this competition, we made use of two of the AOP language development kits packaged with AF, which are introduced next.

3.1 AF-TeleoReactive

AF-TeleoReactive is based on Nils Nilsson's Teleo-Reactive agent paradigm [Nilsson, 1994] which was designed to react to a changing environment (hence reactive) whilst still performing actions which take it to it's goal (hence teleo, meaning goal oriented). The functional components of AF-TR agents are represented by an ordered list of production rules.

An example of a production rule would be $K \rightarrow A$, where the element K represent conditions on the input from the sensors or the model of the environment, and the element A represent an action on the environment. When a sequence is being interpreted it is scanned from the top until it comes across a rule whose condition is satisfied. The corresponding action is then performed and the interpreter is then restarted from the top of the list.

Information about the current state of the environment is gathered via a set of *Sensors*: Java classes that convert raw sensor data into beliefs that are added to the agents belief set. To handle the potentially dynamic nature of the environment that the agent is sensing, beliefs stored in the AF-TR belief base do not persist by default. Instead they are wiped at the start of each iteration of the agent interpreter. To cater for beliefs that should persist, consideration must be given to this when creating the sensor, which allows the programmer to define which types of beliefs should persist. Whether a belief should persist or not depends on the nature of the item being observed. For instance, in the context of the agent contest, it would safe to adopt a temporal belief regarding the position of a edge within the map (which by its very nature cannot move) whereas a belief about the location of a enemy agent will change over time.

AF-TeleoReactive was developed based on the notion of blind commitment, is so far as the agent will continue performing an action until its actions have modified the environment sufficiently to cause another condition to fire. As such it is assumed that the continuous execution of an action will cause such a change in the environment.

3.2 AF-AgentSpeak

AF-AgentSpeak is based on Jason [Bordini et al., 2008], a purpose-built agent-oriented programming language that implements an extended and improved version of Rao's AgentSpeak(L) language [Rao, 1996]. The language consists of a set of plan rules, examples of which are shown in Fig. 1. Each plan rule consists of a triggering event, a context and a plan containing a number of actions that should be adopted if the plan rule is selected.

The deliberation cycle of AF-AgentSpeak is an adaptation of the algorithm used in Jason that is compliant with the AF common language framework.

1. An event is selected from the set of internal and external events.
2. All plan rules triggered by this event are then selected.
3. The list of rules is reduced to those whose context evaluates to true.
4. From this list a single plan rule is selected and added to a new or existing intention stack depending on whether it is a sub plan or new plan respectively.
5. The next step from each of the agents current intentions is executed in parallel by the agent.

As with Jason, AF-AgentSpeak offers an extended suite of functionality that is not available in the original version AgentSpeak(L). This includes support for inheritance, partial plans, abstract plans, and plan overriding as described in [jor, 2011]; and an extended set of plan operators, including: *foreach* (plan expansion), *while* (loops), *if* (selection), *wait* and *when* (delayed execution), and = (assignment). This extended planning language is provided as part of the Common Language Framework [Russell et al., 2011].

```
#agent Leader

module groups -> com.agentfactory.mapc.GroupModule;
module map -> com.agentfactory.mapc.MapModule;

@initialization
+initialized : true  <-
    groups.setup(groups, [
        team(home, [Repairer, Inspector, Sentinel, Sentinel, Explorer, Explorer, Inspector]),
        team(support, [Repairer]),
        team(away, [Saboteur, Saboteur])]),
    @setup;

+sim(end) : true <-
    groups.reset,
    map.resetService,
    .println("resetting"),
    @setup;

#partial @setup <-
    .abolishAll(strategy(?t, ?x)),
    groups.setShared([cash(0), stepNo(0)]);
...
```

Fig. 1. Example AF-AgentSpeak code from the Leader agent

Figure 1 contains part of the code for the Leader agent. As can be seen in this figure, the Leader agent accesses the Group and Map services through two purpose-built modules (these same modules are used by the AF-TeleoReactive agents). On creation, the leader sets

up 3 groups: a *home* group, which is responsible for building and holding the zone; an *away* group, containing the Saboteurs, and which is charged with the task of attacking the enemy; and a *support* group which contains a single Repairer and is responsible for supporting the away group. Once the groups are setup, the *@setup* partial plan is invoked, which does additional configuration steps. A partial plan is used here because the code is common to both the initialization of the agent and the resetting of the agent when the current simulation ends.

3.3 Core Architecture

The core architecture of the system is shown in Figure 3, the agents communicate through the use of the two platform services *MapService* and *GroupService*. No agent communicates directly with another agent wether it represents an ATPV or the leader which has no embodiement in the simmulation. Receiving shared data is done without any requirment to request or subscribe to particular forms of data. All agents automatically pull all relevant information during the perception phase of their execution, allowing the agent to use the shared beliefs as if they were it's own percepts. An example of this is shown in the first rule in Figure 2, this is taken from the repair function, which is activated when a dissabled agent is on the same vertex as a Repairer. The first percept is one shared through the MapService and the second and third are shared through the GroupService. The second line of the example is taken from the *doAction* function where the *?action* variable is the first parameter of the function (in this case repair) and the *?param* variable is the second (the name of the agent to be repaired).

```
agentInfo(?name,?team,?ver,Saboteur,disabled) & memberOf(home) & groupOrder(away, ?name, primary)
          -> doAction(repair,?name, .nil)

step(?step) & (?action == repair) & ~doneActionForStep(?step, ?act) -> eis.perform(repair(?param))
```

Fig. 2. Example of role based action selection

Agent communication with the server is handled through the Agent Factory EIS layer and the eismassim jar, this is shown by the second rule in Figure 2 whereby the action and parameters are passed to the service in the form of a predicate.

The agents were not distributed across multiple machines, implementation of this would have required a modification to the platform service. this was not done due to time constraints.

4 Strategies, Details and Statistics

4.1 Strategy

The overall team strategy is the combination of a number of role dependent strategies and the over all zone creation strategy. As discussed in Section 3.3 agent coordination and information sharing is done through a number of whiteboard type services. Based on information shared through these services, the agents will perform different actions based on their role or assigned tasks.

4.2 Agent Mental State

In general the prevailing goals which drive the agents are to hold the positions assigned by the leader, but not all agents were assigned positions. Secondary to this goal some agents may override this goal with their own periodically. An example of this would be that *Inspectors* periodically decide to refresh the teams knowledge of the attributes of the enemy and as such move to inspect each one. Agents can change their behaviour based on a number of factors such as the conditions shown in Figure 4, or the exceptions is given in Table 1.

Saboteurs are not assigned tasks, they have a priority based attack system where the priority targets are Saboteurs then Repairers and then everyone else, when en-route to a target a Saboteur will attack any enemy agent the happens to be on the same vertex. Saboteurs also use a targeting separate targeting system when two friendly Saboteurs are on the same vertex ensuring that they both selected different targets.

Repairers are also not assigned tasks directly, rather they are associated with a Saboteur which they are charged to support. To mitigate the risk of repairers being disabled when traveling to repair a teammate, we elected to have the disabled agent travel to the Repairer. This decision also allowed us to keep the two repairers in relatively close proximity to their assigned Saboteur to minimise the risk of one of them being disabled.

The agents contain an explicit mental state composed of the percepts received from the server as well as all internally generated beliefs and those received from platform services.

There are both implicit and explicit hierarchies within the structure of the agents. Informally the Leader agent is responsible for assigning tasks to a sub group of the ATPV agents. Additionally the Repairers were tasked in a support role following the Saboteurs but did not answer to them explicitly.

Agent path planning was done using a form of breadth first search which can ignore edges above a certain weight, this algorithm was chosen as it seemed that the number of steps taken to travel the a distance was more important than the amount of energy expended.

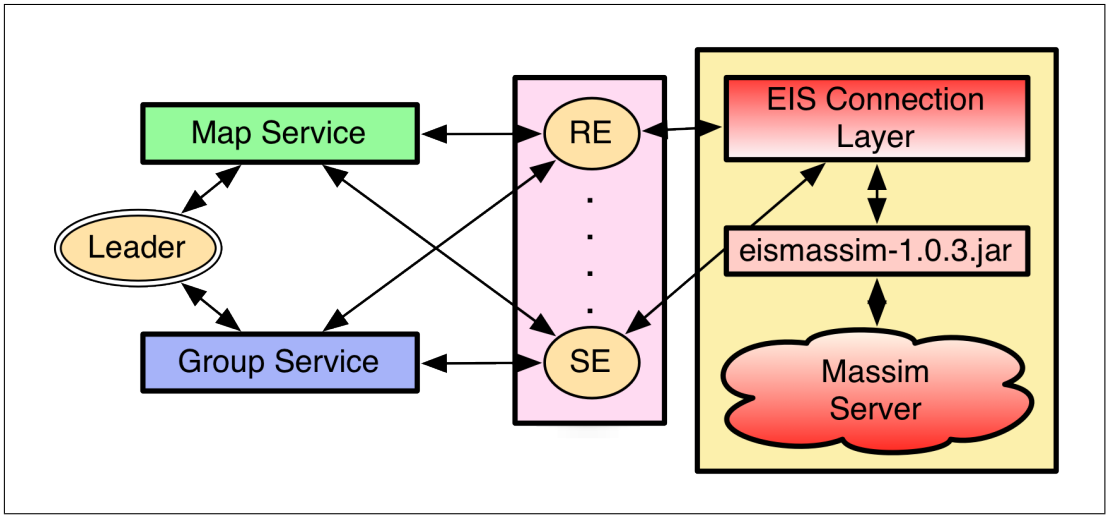


Fig. 3. Core Architecture

Table 1. Agent Autonomic Actions

Role	Condition	Action
Inspector	Enemy with data older than 100 steps	Inspect the enemy
All agents	Become disabled	Move to closest Repairer
Explorers	Map becomes dominated (all vertices owned)	Adopt goal to probe every vertex
All Agents (Except Saboteurers)	Enemy Saboteur on vertex	Parry presumed attack
Repairer	Disabled teammate on vertex	Repair teammate

4.3 Implementation Details

The topology of the map was not analysed in any great detail, basic clustering was performed to determine the best position to use as the centre of the captured zone, this is modelled on breadth first search where the cumulative value of the vertices is summed. When the value is not known it is assumed to be 1.

The agents use the eismassim package in order to communicate with the server, this is achieved through an existing integration of EIS with AgentFactory.

The role of an agent is captured during the initial percept and stored for reference within the code, in this way during the execution of the agent. An example of this is given in figure 4, the first line represents the goal that we should know the role of each of the enemy agents and the second rule states that we should refresh the information we hold on that agent if we haven't inspected them in the last 100 steps.

```
role(Inspector) & agentInfo(?name,?team,?vertex,?role,?s) & (?role == none) -> inspectAll
role(Inspector) & lastInspected(?n,?s)  & step(?step) & (?step > ?s+100) -> inspectEntity(?n)
```

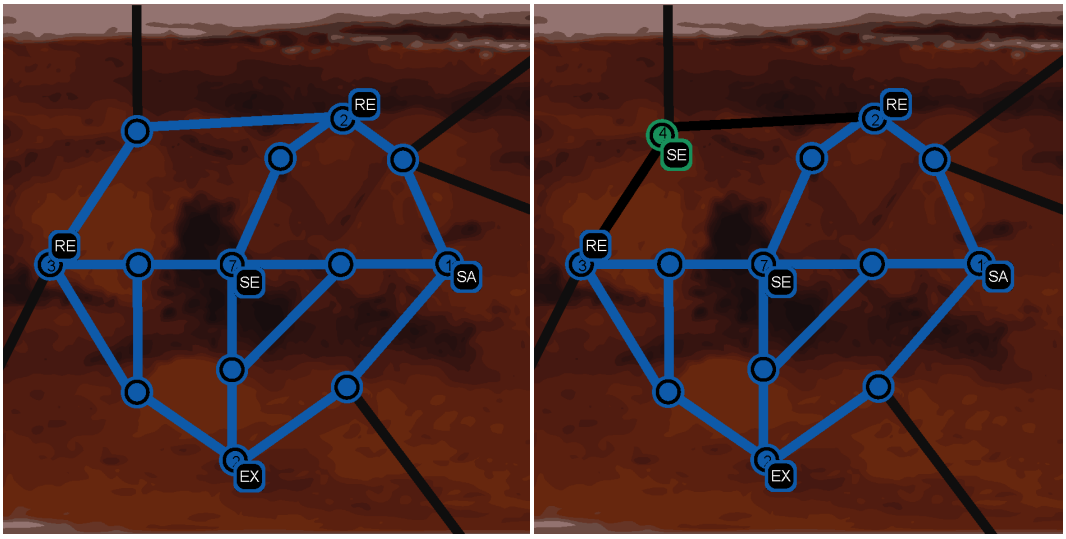
Fig. 4. Example of role based action selection

4.4 Zones

Zone selection was very simplistic and was not concerned with the actual value we would attain if we we holding certain positions, instead we opted to use a simple clustering algorithm based on breadth first search to identify a good position to use as the centre of our zone.

Zone defense was designed around a well known concept in ship design whereby it is divided into a number compartments, the basic principle is that when agents are positioned two steps away from other agents, and sometimes the centre of the zone, this naturally creates a pattern where subzones exist within the zones. An example of this is shown in Figure 5(a), when an enemy attempts to break the frontier of the zone the compartment collapses but the rest of the zone remains, this is shown in Figure 5(b).

We did not specifically implement a team behavior for attacking enemy zones but this occasionally emerges out of aggressive Saboteur behavior, or frontiers may be broken by an Inspector approaching to inspect enemy agents.



(a) Zone configuration (b) Zone configuration under attack

Fig. 5. Subgraphs showing zone configuration

4.5 Buying Algorithm

We used a complex recursive buying function designed to maximise the effectiveness of the our Saboteur agents, it was designed to react to the attributes of the enemy agents always ensuring that based on our current knowledge we could disable any of the enemies agents with one hit and survive one hit from the enemy saboteurs. The function was designed recursively such that when the cash was available a number of agents could purchase simultaneously based on a hard coded priority.

The achievement points were very important in our strategy as a number of our victories were based on the fact that our Saboteurs we able to dominate the enemy Saboteurs. Secondarily we relied on excess achievement points being spent on a very high visibility for one of the teams Sentinels in order to keep as much information on the enemy movements as current as possible.

As an aside during the final game when it was clear that we could not increase our standing in the league table we attempted a strategy whereby the team did not spend any of the achievement points. By the end of the simulation we had amassed 54 points resulting in a much more competitive and exciting simulation.

5 Conclusion

From our participation in the contest were were enable to evaluate and test the functionality of modifications made to the languages we were using. The contest proves every year to be an impetus for development of new features and thorough testing within the languages created using the Common Language Framework. It has further affirmed the stability of the chosen languages for their roles within the architecture of the system, In that Af-Agentspeak is suited for the organisational role of the leader and AF-TeleoReactive is suited for the more reactive and time dependent control of the ATPV units in the simulation.

The team overall performed well, the individual roles of the agents sometimes combining to work very well together, at an individual level the strategy of the agents worked quite well but on some occasions the team as a whole became uncoordinated. This resulted from a error in our approach to development, focusing too much on the low level individual strategies than the higher level coordination strategies.

As in our conclusion last year we believe that the results of the competition reflect the effectiveness of the team strategies. In both of the games with the top two teams we were outclassed in terms of strategy and as a result we were comprehensively beaten. Our closest match was with third placed TUB, despite losing all the simulations, it was much closer and much more exciting than any of the games with the lower ranked teams.

For next years contest some changes we would consider beneficial would be to introduce more static assignment of roles to agents, we believe that this could allow a greater diversity in capabilities and characteristics of the ATPVs.

References

[rus,] From bogtrotting to herding: a ucd perspective. *Annals of Mathematics and Artificial Intelligence*, pages 1–20.

[jor, 2011] (2011). Reuse by inheritance in agent programming languages. *Intelligent Distributed Computing V*, pages 279–289.

[Behrens et al., 2011] Behrens, T., Hindriks, K., and Dix, J. (2011). Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61:3–38.

[Bordini et al., 2008] Bordini, R., Hübner, J., and Wooldridge, M. (2008). *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience.

[Collier, 2002] Collier, R. W. (2002). *Agent Factory: A Framework for the Engineering of Agent-Oriented Applications*. PhD thesis, School of Computer Science and Informatics.

[Dragone et al., 2009] Dragone, M., Lillis, D., Muldoon, C., Tynan, R., Collier, R., and O'Hare, G. (2009). Dublin bogtrotters: Agent herders. *Programming Multi-Agent Systems*, pages 243–247.

[Jordan et al.,] Jordan, H., Treanor, J., Lillis, D., Dragone, M., Collier, R., and O'Hare, G. AF-ABLE in the multi agent contest 2009. *Annals of Mathematics and Artificial Intelligence*, pages 1–21.

[Nilsson, 1994] Nilsson, N. (1994). Teleo-reactive programs for agent control. *Arxiv preprint cs/9401101*.

[Poslad et al., 2000] Poslad, S., Buckle, P., and Hadingham, R. (2000). The FIPA-OS agent platform: Open source for open standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, pages 355–368.

[Rao, 1996] Rao, A. (1996). AgentSpeak (L): BDI agents speak out in a logical computable language. *Agents Breaking Away*, pages 42–55.

[Russell et al., 2011] Russell, S., Jordan, H., O'Hare, G. M. P., and Collier, R. W. (2011). Agent factory : A framework for prototyping logic-based aop languages. In *In Proceedings of the Ninth German Conference on MultiAgent System Technologies MATES 2011*.

Short Answers

Introduction

1. To involve new researchers with agent-oriented development, and to drive development of Agent Factory.
2. This is the Fourth year in which a UCD team has entered the contest. The team has had different configurations over the years, with Dr. Collier as our constant.
3. UCD Bogtrotters
4. The team had 4 developers, all of who were involved in the system design. We have 2 PhD candidates, one lecturer, and one undergraduate student.
5. Sean Russell and Dominic Carr are involved in WSN research, Balazs Pete is still undertaking undergraduate education, Dr. Collier is an active researcher in the Agents field.

System Analysis and Design

1. Due to time constraints, our system was not specified or designed using any particular multi-agent system methodology.
2. No, all information is disseminated to all agents.
3. Communication is performed through platform services providing simple whiteboard type functionality.
4. The attributes of *autonomy*, *pro-activeness* and *reactiveness* were realised in the use of Agent-Oriented Programming languages and through the structure of the agent programs which allow bypassing of the assigned task in favour one of their own goals.
5. Our system is a true multi-agent system with centralised coordination.
6. 600 man hours.
7. We did not discuss our strategies with other developers. We participated in one test match to test the connection and to assess if actions were completed in a timely manner, more testing was not carried out due to time constraints.

Software Architecture

1. The Java programming lanquage was used as it forms the basis for AgentFactory.
2. The system was implemented using the AF-TeleoReactive and AF- AgentSpeak multi-agent programming languages running on the Agent Factory platform.
3.
4. Our development platform was the Eclipse IDE with the inclusion of the AgentFactory plugin. All members were familiar with Eclipse and, new members spent 1-2 hours learning how to best use the plugin.
5. AgentFactory.
6. During initial development a number of features were identified as missing from AF-TeleoReactive such as parallel action execution and executing actions for all available bindings. These were implemented prior to the start of the competition.
7. Parallel action execution and executing actions for all available bindings simplified the development task greatly.
8. A modified version of breadth first search designed to be aware of the weight of edges and maximum energy of the agent.
9. The agents were not distributed across multiple machines, implementation of this would have required a modification to the platform service. this was not done due to time constraints.

10. The reasoning cycle of the agent was asynchronous with respect to the receive-percepts/send-action cycle, as such we had to prevent the agent from performing more than one action in a single simulation step.
11. The most difficult part of the development process was the synchronisation of individual agent behaviours to have the intended combined effect.
12. Java Code: 3863, Agent Code: 407, Total: 4270

Strategies, Details and Statistics

1. The overall team strategy is the combination of a number of role dependent strategies and the over all zone creation strategy.
2. The leader agent assigns task to the other agents, information is shared though platform services.
3. The results of map analysis are exploited to find the best position to use as the centre of the captured zone.
4. The agents use the eismassim package to communicate with the server.
5. All agents have the same source code, an agents role was stored as a predicate with the agents beliefs and used to determine the actions the agent would consider/perform.
6. We employed a simple clustering algorithm to identify a position to use as the centre of our zone.
7. Zone defence was based on subdivision of zones. When agents are positioned two steps away from other agents, this creates sub-zones within a zone, which can persist if the frontier is breached. We did not implement an aggressive behaviour, but this occasionally emerges out of the Saboteurs or Inspectors behaviour.
8. Agents can change their behaviour based on a number of factors such as the presence of enemy agents or being disabled.
9. Breadth first search.
10. We exploited the buying mechanism to maximise the effectiveness of the our Saboteur agents, the algorithm was reactive to the current knowledge of enemy capabilities.
11. The buying mechanism was used to attempt to gain supremacy in individual battles between saboteurs.
12. Yes, they have a mental state made up of internal beliefs and shared information from the platform service.
13. Agents communicate through a shared a platform service. The leader assigns tasks through this service.
14. There are both implicit and explicit hierarchies within the structure of the agents. Informally the Leader agent is responsible for assigning tasks to a sub group of the ATPV agents. Additionally the Repairers were tasked in a support role following the Saboteurs but did not answer to them explicitly.
15. Individually the agents performed as expected, but at a team level emergent behaviour was observed.
16. The leader agents perform advance planning as regards placement of ATPV agents in the future steps.

Conclusion

1. The suitability of different AOP languages to different positions within a multi agent system.
2. Strong: Individual agent strategy, Weak: overall team strategy.

3. The chosen programming languages were very suited to the roles they were applied to.
4. More Diversity between agent roles.
5. Strategy was the primary factor in the final tournament rankings.
6. The contest may be of interest to researchers in the broad AI field who are not directly working with agents.
7. Increasing the size of the maps and the number of agents would create a more challenging contest.

An Argumentative Approach for a BDI Agent

Iñaki Garay, Diego Marcovecchio, Leonardo Molas, Emiliano Montenegro,
Fernando Sisul, Manuel Torres, Sebastián Gottifredi, Alejandro García,
Diego Martínez, and Guillermo Simari

Universidad Nacional del Sur
{igarai,diegomarcov,emm.montenegro,fsisul,jmtorresluc}@gmail.com,
{lam,sg,ajg,dcm,grs}@cs.uns.edu.ar

Abstract. This report presents the design and results of the d3lp0r multi-agent system developed by the LIDIA team for the Multi-Agent Programming Contest 2011 (MAPC). The d3lp0r agents use a BDI architecture extended with planning and argumentation (via Defeasible Logic Programming) to model a cooperating team operating in a dynamic and competitive environment.
In particular, the main goal of this report is to describe the chosen architecture, the communication scheme and the way argumentation was put to use in the agent's reasoning process and the technical details thereof.

1 Introduction

The d3lp0r system was developed in the context of the Multi-Agent Programming Contest 2011 (MAPC) [Behrens et al., 2010] hosted by the Clausthal University of Technology [1]. The LIDIA (Laboratorio de Investigación y Desarrollo en Inteligencia Artificial, Artificial Intelligence Research and Development Laboratory) research group was established in 1992 at the Universidad Nacional del Sur. The d3lp0r team was formed incorporating six graduate students, two Ph.D. students and three professors. The undergraduate students fully developed the system, while the Ph.D. students and professors provided guidance. The group's main motivation was to apply argumentation [Prakken and Sartor, 1997,Rahwan and Simari, 2009] [Bench-Capon and Dunne, 2007] via defeasible logic programming (DeLP [Garcia and Simari, 2004]) in a BDI based agent [Amgoud et al., 2008], in the context of a multi-agent gaming situation, and to test the integration of the different technologies used.

2 System Analysis and Design

Despite many man-hours dedicated to design in the early stages of the competition, the development team's lack of experience in multi-agent systems

[1] More information in www.tu-clausthal.de

made several changes and additions necessary and precluded the use of design methodologies specific to multi-agent systems. Nevertheless, our approach was more than satisfying, resulting to be modular, correct and in close correspondence with the literature.

The solution follows a decentralised architecture in which agents run completely decoupled in different processes with no shared state.

In addition to the agent processes, the system design includes an independent "percept server", through which percepts are communicated among agent team members via a broadcast mechanism running on standard network sockets. Each agent handles his own connection to the MASSim server, and upon receiving its percept, retransmits it to the percept server. The percept server joins all percepts into a "global percept", and sends each agent the set difference between its own and the global percept. The agent then enters its reasoning phase and decides which action it will send back to the MASSim server. Other than the percept server mechanism, there is no communication among team agents. This design was chosen for its minimal complexity.

Agents can also be configured to run in a standalone mode, in which they will not use the percept server and thus have no communication with the rest of the team. Team performance drops noticeably in this case, as the actions are less informed.

Agents are completely autonomous meaning that decision-making takes place individually at the agent level, with no intervention from human operators or a central intelligence agency within the system, and that decisions made by an agent are influenced solely by the current simulation state and the results of previous steps. Despite the sharing of all percepts among the team agents in the initial phase of the turn, no control variables or instructions are included. The agent architecture developed is based on the BDI model [Rao and Georgeff, 1991], and is explained in detail in further sections.

The agents' behavior can be considered proactive, given they pursue their selected intentions over time, that is, they have persistent goals. Plans for achieving intentions are recalculated and followed for the number of steps requiered, unless the goal in question becomes impossible or no longer relevant.

Approximately 1500 man-hours were invested in the team development. Experience from a previous instance of the MAPC was shared with our teams by members of the ARGONAUTS team from TU Dortmund[Hölzgen et al., 2011]. Although the initial plan was to run tests against other agent teams prior to the competition, time constraints made this impossible.

3 Software Architecture

This section details the implementation and module organization of the proposed multi-agent system.

3.1 Programming languages, platforms and tools

The agent system was implemented using Python 2.7 and SWI Prolog 5.10.5. Language integration was achieved using the *pyswip* library[2], which facilitates the execution of Prolog queries from Python. The implementation of Defeasible Logic Programming (DeLP) by the LIDIA [Garcia and Simari, 2004] was used for the deliberative process, in which desires and intentions are set. The standard Python and SWI-Prolog debugging tools were used. DeLP includes a graphical viewer for dialectical trees, allowing visualization of which arguments attack others and facilitating debugging of the defeasible rules employed. These languages and platforms were well-known at the start of the project, and were chosen for precisely those reasons.

No multi-agent programming languages / platforms / frameworks were used due to a lack of familiarity on behalf of the development team. Also, integrating or extending an existing framework with queries to defeasible rules was initially considered more difficult than the straighforward approach taken.

Python's amenity to rapid application development and "batteries-included philosophy" facilitated implementing the communication layer to the MASSim server, parsing of perceptions, rapid addition of planned features and bug correction. DeLP's capability to deal with conflicting pieces of information was also very helpful in order to implement the decision-making module.

3.2 Implementation

The system was implemented as a collection of independent operating system processes, the percept server (PS from here onwards) and each agent running in its own address space. The agents are started individually and synchronize via the PS. Each one handles its own connection to the MASSim and percep servers, as the *eismassim* package provided by the contest organizers was not used to avoid the difficulty of integrating yet another language and runtime (Java) with the ones being used.

Fig. 2 shows the structure and flow of control and information within the decision making module implemented in Prolog and DeLP.

Agent main program. The agent main program is implemented in Python, and handles all communication with the servers, XML parsing, processing the information in the percept into a form suitable for assertion in the agent's knowledge base, and generation of the XML representing the action taken which is returned to the MASSim server.

Initialization consists of opening the connections to the MASSim server, authenticating, opening the connection to the PS, and starting the Prolog engine. The main program loop is then entered, in which messages from the MASSim server are received and parsed.

[2] `http://code.google.com/p/pyswip/`

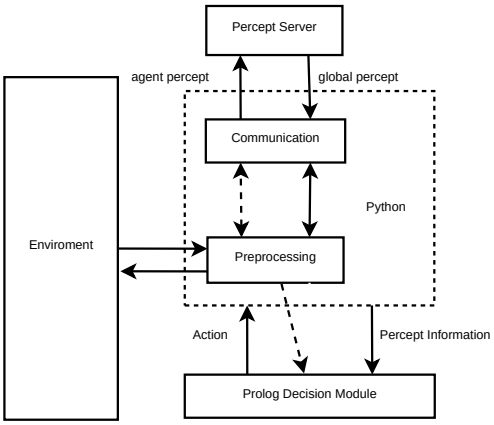


Fig. 1. Agent architecture in a flow chart-like diagram. Dashed arrows represent process flow, solid lines represent data flow.

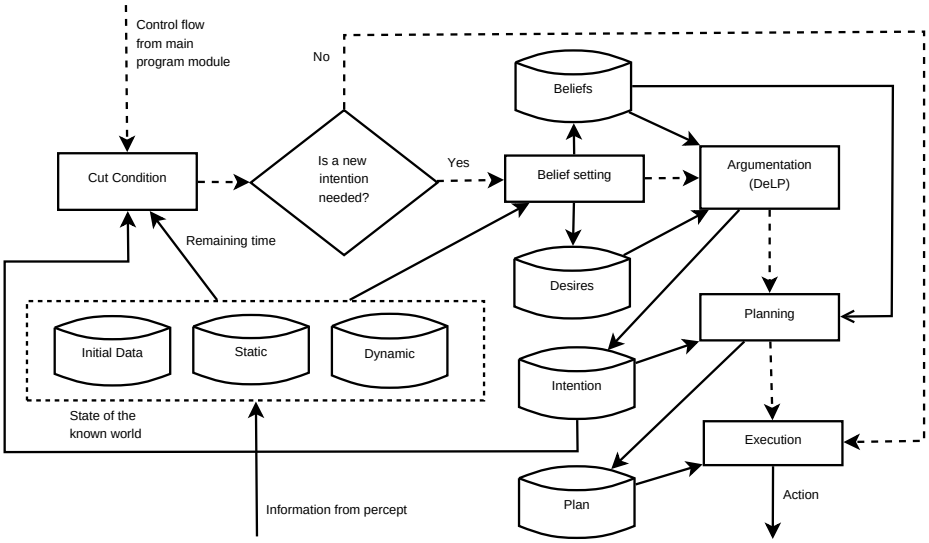


Fig. 2. Architecture of the Prolog decision module

When a message of type `sim-start` is received, initial information present in the message such as the agent's role and the simulation parameters are asserted into the agent's knowledge base and the perceive-act loop is started.

Each iteration of the perceive-act loop expects a `request-action` message from the MASSim server and parses the XML into a Python dictionary. Elements in the percept are divided into a "public" section, which is sent to the PS to be shared with other team agents and a "private" section.

The agent will then send the percept to the PS and await the global percept containing the remaining information perceived by the team. The global percept is merged with its own, and asserted into the agent's knowledge base, establishing the agent's beliefs. Note that no information is included in the percept other than what is received in the percept.

The decision making module implemented in Prolog is then queried for the next action to be performed by the agent. Once control flow returns to the Python program with the determined action, the corresponding XML message is generated and sent to the MASSim server.

Percept Server. The PS maintains a connection for each agent. The connection handling methods encode the associate array into a form suitable for conversion into a set datastructure, which is then sent over the network. On each iteration, the PS waits for each agent's data, performs a union of all data sets, and returns to each agent the set difference between the data the agent sent and the total union. Figures 3 and 4 show example percepts after parsing, before being sent to the percept server.

```
{ 'surveyed_edges' : [ ],
  'vis_verts'      : [ { 'name': 'vertex65',
                         'team': 'none' },
                       ...  ],
  'vis_ents'       : [ { 'node':   'vertex97',
                         'status': 'normal',
                         'name':   'a6',
                         'team':   'A' },
                         ...  ],
  'inspected_ents' : [ ],
  'vis_edges'      : [ { 'node1': 'vertex141',
                         'node2': 'vertex65' },
                       ...  ],
  'position'       : [ { 'node':       'vertex141',
                         'vis_range':  '1',
                         'health':     '6',
                         'name':       'self',
                         'max_health': '6' } ],
  'probed_verts'   : [ ] }
```

Fig. 3. A sample public section of a percept, after parsing.

Beliefs-Desires-Intentions Once the remaining information is received from the percept server, the data is then asserted into the agent's knowledge

```
{ 'total_time':      2000L, 'zone_score': '0',
  'last_step_score': '20',  'timestamp':  '1323732915832',
  'strength':        '0',   'energy':     '11',
  'money':           '12',  'max_energy_disabled': '16',
  'last_action':     'recharge', 'max_health':     '6',
  ... }
```

Fig. 4. A sample private section of a percept.

base. A collection of predicates queried from the main Python module are in charge of verifying that information is not overwritten, and that redundant information is not inserted. Most predicates which represent the state of the environment include a parameter bound to the turn the information was perceived, so that information can be considered "stale" if the difference between the current turn and the turn the information was acquired is too large.

Beliefs in the knowledge base are represented as terms, arguments to the predicate `b/1`. Desires are also represented as Prolog terms; possible desires are `expansion` (increase the value of a zone), `explore` (probe nodes and increase the team's knowledge of the graph), `regroup` (move closer to team members), `seekforrepair` (move closer to an agent with the *repairer* role), `selfdefense`, `parry`, `stay` (do not move), `buy`, `probe`, `repair`, `attack`. The desires an agent may entertain depend on the agent's role.

The intention is selected from the set of possible desires the agent may entertain. If the agent already has an intention stored, the *cut condition* checks whether it makes sense to keep trying to fulfill it. It is a series of simple conditions that review the state of the world.

Then, if there is not any committed intention, or the cut condition decides it is not interesting to keep it, the *beliefs setting process* is started. It generates the possible desires for this step, according to what is stored in the knowledge base, and, for each one of them, the beliefs needed. The decision-making module is implemented in DeLP [Rotstein et al., 2007] [Ferretti et al., 2008], a defeasible logic programming language that uses argumentation [Besnard et al., 2008] to reason with conflicting information. Given the set of possible desires and beliefs set by the previous module, it selects the best desire, returning it as the intention that the agent commits to achieve.

All the plans for all the desires were previously calculated and stored as beliefs, since the amount of steps that they take is used by the argumentation module. The *planning* module selects the one corresponding to the selected intention, and stores it. Then, the execution module only gets the plan, and returns to Python the first action in it. For both search algorithms, the Depth First Search and the Uniform Cost Search, we added conditions that could cut several branches, when they were expanding to unwanted nodes. This

conditions were set by the caller, since they depend on the context of the problem.

For the UCS, we first used a simple stack implemented with a list, to keep track of the frontier, because of Prolog's inability to work with arrays. This would have allowed us to develop a heap data structure, to be used in a priority queue. Lately, we found a Prolog library that implemented this data structure, and the migration was pleasantly straightforward.

However, if the process flow comes from the other branch of Fig. 1 (that is, after the cut condition, the agent has an intention), the execution is not that simple. Since skipping the decision- taking makes this branch insignificant in terms of time, we decided to recalculate the plan. This might help us when a better path is discovered, even though this is unlikely.

Deliberation and DeLP. In DeLP[Garcia and Simari, 2004], knowledge is represented using facts, strict rules and defeasible rules. Facts and strict rules are ground literals representing firm information that cannot be challenged. *Defeasible Rules* (d-rules) are denoted $L_0 \prec L_1, \ldots, L_n$ (where L_i are literals) and represent tentative information. These rules may be used if nothing could be posed against it. A d-rule *"*$Head \prec Body$*"* expresses that *"reasons to believe in Body give reasons to believe in Head"*. A DeLP program is a set of facts, strict rules and defeasible rules.

Strong negation is allowed in the head of program rules, and hence, may be used to represent contradictory knowledge. From such a program contradictory literals could be derived, however, the set of facts and strict rules must possess certain internal coherence (it has to be non-contradictory).

To deal with contradictory information, in DeLP, *arguments* for conflicting pieces of information are built and then compared to decide which one prevails. The prevailing argument is a *warrant* for the information that it supports. In DeLP, a query L is *warranted* from a program if a *non-defeated* argument that supports L exists.

```
selfDefense(1000) -<
    myStatus(normal), canParry,
    myPosition(Node), enemySaboteurPosition(Node).

canParry -< myRole(repairer).  canParry -< myRole(saboteur).
canParry -< myRole(sentinel).
~canParry <- myEnergy(Energy), less(Energy, 2).
```

Fig. 5. Desire `SelfDefense`

Figure 5 is an example of our representation of the possible intentions written in DeLP. Self defense has a weight of 1000. It is a high priority intention given that the average weight of the intentions is around 150. `myStatus`,

`myPosition`, `enemySaboteurPosition`, `myRole` and `myEnergy` are facts of the knowledge base. `canParry` arguments will support or defeat arguments for selfdefense intention.

3.3 Difficulties encountered

The most difficult problems were related to optimization. Much of our time was spent in reducing the complexity of our algorithms, and the times they were called.

Our initial plan was to distribute agents on several machines. Each agent runs as a separate process, and communicates with others via TCP sockets. After some experience and benchmarking, agents were run on one machine, due to For both search algorithms, the Depth First Search and the Uniform Cost Search, we added conditions that could cut several branches, when they were expanding to unwanted nodes. This conditions were set by the caller, since they depend on the context of the problem.

For the UCS, we first used a simple stack implemented with a list, to keep track of the frontier, because of Prolog's inability to work with arrays. This would have allowed us to develop a heap data structure, to be used in a priority queue. Lately, we found a Prolog library that implemented this data structure, and the migration was pleasantly straightforward.

Our initial plan was to distribute agents on several machines. After some benchmarking agents were run on one machine due to performance issues. Having the option to easily try both approaches was a benefit of the proposed design.

In total, the system consists of 1336 lines of Python, 5059 lines of Prolog pertaining strictly to the agent, belief setting and auxiliary predicates, and 355 lines of DeLP rules, both defeasible and strict. The DeLP interpreter consists of 4494 lines of Prolog. These figures includes commentaries and blank lines.

4 Strategies, Details, and Statistics

In this section, we will explain the main characteristics of the team's overall strategy, as well as several implementation details, such as algorithms used and agents' organization.

4.1 Strategy

The main strategy of the team consists of detecting profitable zones from the explored nodes, and positioning the agents correctly to maintain, defend and expand the zones formed.

Every agent is concerned with the formation and expansion of zones, beyond its role. The decision-taking process is responsible for calculating and selecting the most beneficial intention. This selection process is based on the

gain in terms of score, the need of the team for the execution of a role-specific action, or the benefit that the agent is currently contributing to the team.

Agents coordination is implicit, only involving the percept information and in particular excluding processed beliefs and control variables. The agents do not communicate intentions or plans.

Agents do not change their behavior during runtime, maintaining the same strategy throughout all simulation stages.

Zone conquering. The exploration of the map is done gradually. Actions related to the exploration (probe, survey) are weighed along with all other actions and selected when considered important. This occurs to a greater extent during the initial steps of the simulations, when the team lacks knowledge of the map and other actions are unnecessary. Agents make no assumptions regarding the map topology.

Behavior is not primarily focused on finding new zones, but agents do attempt to expand and maximize the points of the existing ones. They calculate whether they are part of a zone or not. This is achieved by checking the color of the current node and whether a neighbor of it is also colored by the agent team (if this is not the case, the node does not increase the zone points). If an agent is not part of any zone it tries to regroup with its teammates.

When a zone is formed and an agent is part of it, for each potentially beneficial neighbor node, the agent calculates how much points the team would gain if it moves, and tries to expand the zone. If the expansion intention is selected and acted upon, then a new better zone is implicitly conquered.

Coloring algorithm. This estimations are done with our reimplementation of the coloring algorithm used by the MASSim server. The information is used by the decision taking module. Our approach makes several assumptions that facilitate the application of the algorithm in a map that has not been completely explored.

Attacking and defending. Both attacking and defense of zones are implicitly implemented. Saboteurs prefer to attack enemies that are near, so if an agent of another team enters our team's zone, it will be attacked by the saboteurs in the zone. This is the most likely scenario, unless the saboteur's position is so important that it decides to stay in the formation in order to keep the zone.

The same happens with enemies in their own zones. Zones are not intentionally destroyed, but any agent that is part of a zone may be attacked, affecting possibly the structure of the enemy zone.

Agents of other roles can also implicitly defend a zone. For example, an agent can go to a node that has one agent of each team, with the purpose of coloring the contested node and defending the zone.

Buying. Agents follow a list of predefined buying actions, when the necessary amount of money is reached. This behavior follows the idea of getting some specific skill upgrades that the team considered important to achieve early in the simulations.

Achievements. Achievements are not explicitly taken under consideration. That is, the agents' reasoning process is not affected by the possibility of completing achievements. However, the team can manage to achieve a significant number of them, which results naturally from the agents' behavior. This fact let the development team avoid the need of adding special features dedicated to the seek of achievements.

4.2 Implementation

Here are some implementation details of the different parts of the agents.

Mental state. Agents have a complete and explicit mental state. It consists of a set of components, such as beliefs, desires, intentions, and plans. The belief base includes the information obtained from the perceptions, as well as different kinds of beliefs required by the decision-taking module. The desires are set every step that the agent decides to select a new intention. The intentions and plans kept in the knowledge base are those that the agent is currently carrying out.

Path planning. Path planning is implemented with an Uniform Cost Search [Russell and Norvig, 2003]. We tried to minimize the amount of steps required to achieve the goal, rather than the energy spent. The returned result is a list of actions to be done, rather than a list of nodes.

Since this algorithm can be called several times in one step, and given that the actual amount of steps spent by an intention is taken under consideration by the decision-taking module, it was crucial to perform several optimizations in it. In the end, this allowed us to run all the agents in a single machine during the competition.

The plans are as long as the selected intention requires. This may sound excessive, but the possible goals were previously selected for their potential, taking into consideration their distances (in nodes, not in actions). However, plans are recalculated in every step, as explained earlier.

4.3 Agents' organization

The decision-taking module makes use of other agents' status, but there is neither negotiation nor intentions exchange, so the team performance is emergent on individual behavior.

Referring to our actual programming, all the agents have a strong core of common code, which is all the Python part, that servers as a receive-percepts/send-action client of the server; the Percept Server; and an important part of the Prolog code. This includes all the utilities used, the implementation of the BDI architecture, the structure of the decision-taking module, and a considerable part of the arguments used, that are common to all the roles.

Apart from all this, each role has a couple of separate files, that have specific code, including the arguments used in the decision-taking module, and the setting of the beliefs needed for those arguments. Here is where the individual behavior is set, since the specific actions that can be done by each role are taken into consideration here.

5 Conclusion

In this section, we make some final comments about the contest and our experience.

5.1 Our team, and its development

Being our first experience building a system this size, we learned several lessons about working in large projects, such as setting standards and synchronizing versions of the technologies used.

In LIDIA, our teammates have done an important amount of research in argumentation and multi-agent systems, providing valuable experience and guidance.

We believe the right decisions were taken regarding the programming languages. DeLP resulted suitable for the implementation of the decision-making module since it was flexible enough to develop our argumentation approach.

Several hotfixes that were written and deployed during the competion due to the lack of testing. Avoiding this undesireable situation is one of our main priorities for next year's competition.

Our lack of experience in this kind of contests, unexpected network and latency problems, as well as some bugs that caused critical performance issues, caused our team to lose several matches that could have been won otherwise.

5.2 Our thoughts about possible optimizations to the contest

More information associated to nodes, e.g. absolute coordinates, would help in the implementation of directed search decreasing execution time.

Strategically, the early dominance of the center area played an important part of a good candidate to win a match. It would be useful to try other variations, such as making the borders more important, or others shapes of the map, such as stretched, in form of V, X, O, etc. This would benefit teams that

explicitly and thoughtfully look for and conquer good zones, rather than benefiting teams that assume that only one good zone exists in the middle of the map.

More feedback from the server would be appreciated, especially regarding errors. This is important for detection of bugs involving communication, i.e. problems with the connection, files sent.

Test matches in earlier stages of development would reduce technical errors during the competition. Teams reimplementing the eismassim communication functionality are vulnerable to errors difficult to foresee.

Finally, we think both Robotics and Game AI are interesting fields that could benefit from participating in the contest.

References

[Amgoud et al., 2008] Amgoud, L., Devred, C., and Lagasquie, M. (2008). A constrained argumentation system for practical reasoning. In *Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 429–436.

[Behrens et al., 2010] Behrens, T., Dastani, M., Dix, J., Köster, M., and Nov'ak, P. (2010). The multi-agent programming contest from 2005-2010: From collecting gold to herding cows. *Annals of Mathematics and Artificial Intelligence*, 59:277–311.

[Bench-Capon and Dunne, 2007] Bench-Capon, T. J. M. and Dunne, P. E. (2007). Argumentation in artificial intelligence. *Artif. Intell.*, 171(10-15):619–641.

[Besnard et al., 2008] Besnard, P., Doutre, S., and Hunter, A., editors (2008). *Computational Models of Argument: Proceedings of COMMA 2008, Toulouse, France, May 28-30, 2008*, volume 172 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Ferretti et al., 2008] Ferretti, E., Errecalde, M., García, A. J., and Simari, G. R. (2008). Decision rules and arguments in defeasible decision making. In [Besnard et al., 2008], pages 171–182.

[Garcia and Simari, 2004] Garcia, A. and Simari, G. (2004). Defeasible logic programming: An argumentative approach. *Theory and Practice of Logic Programming (TPLP)*, 4:95–138.

[Hölzgen et al., 2011] Hölzgen, D., Vengels, T., Krümpelmann, P., Thimm, M., and Kern-Isberner, G. (2011). Argonauts: a working system for motivated cooperative agents. *Ann. Math. Artif. Intell.*, 61(4):309–332.

[Prakken and Sartor, 1997] Prakken, H. and Sartor, G. (1997). Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, 7(1):25–75.

[Rahwan and Simari, 2009] Rahwan, I. and Simari, G. R. (2009). *Argumentation in Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition.

[Rao and Georgeff, 1991] Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a bdi-architecture. In *Second Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484.

[Rotstein et al., 2007] Rotstein, N. D., Garcia, A. J., and Simari, G. R. (2007). Reasoning from desires to intentions: A dialectical framework. In *AAAI Conference on Artificial Intelligence*, pages 136–141.

[Russell and Norvig, 2003] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition.

Short Answers

Introduction

Question 1. What was the motivation to participate in the contest?

The group's main motivation was to apply argumentation via defeasible logic programming (DeLP) in a multi-agent gaming situation and to test the integration of the different technologies used.

Question 2. What is the history of the team?

The LIDIA research group was established in 1992 in the Universidad Nacional del Sur, and it is the first time our team participates in the contest.

Question 3. What is the name of your team?

The team's name is d3lp0r.

Question 4. How many developers and designers did you have? At what level of education are your team members?

The d3lp0r team was formed incorporating six graduate students, two Ph.D. students and three professors.

Question 5. From which field of research do you come form? Which work is related?

The LIDIA research group has been working in Artificial Intelligence and Argumentation via Defeasible Logic Programming for almost 20 years now, and the DeLP server technology developed has been used in the contest.

System Analysis and Design

Question 1. If some multi-agent system methodology such as Prometheus, O-MaSE, or Tropos was used, how did you use it? If you did not what were the reasons?

No design methodology specific to multi-agent systems was used.

Question 2. Is the solution based on the centralization of coordination / information on a specific agent? Conversely if you plan a decentralized solution, which strategy do you plan to use?

The solution follows a decentralized architecture in which agents run completely decoupled in different processes. Agents share no memory and decision-making takes place individually, even though every agent communicates part of his perception to the others.

Question 3. What is the communication strategy and how complex is it?

Percepts are communicated among agent members of the team via a broadcast mechanism developed as part of the multi-agent system. This design was chosen for its minimal complexity.

Question 4. How are the following agent features considered/implemented: autonomy, proactiveness, reactiveness?

Agents are completely autonomous; decision-making takes place individually at the agent level, with no intervention from human operators or a central intelligence agency within the system. Agents assign priorities to different possible goals depending on their desires, and plan in order to achieve the most valuable goal. This results in more autonomous way in which an agent acts. The agents' behavior can be considered proactive, given they pursue their selected intentions over time, that is, they have persistent goals.

Question 5. Is the team a truly multi-agent system or rather a centralized system in disguise?

The team is a truly multi-agent system, and has no centralized characteristics beyond the sharing of all percepts among the team agents.

Question 6. How much time (man hours) have you invested (approximately) for implementing your team?

About 1500 hs.

Question 7. Did you discuss the design and strategies of your agent team with other developers? To which extent did your test your agents playing with other teams?

Experience from a previous instance of the MAPC was shared with our teams by members of the ARGONAUTS team from TU Dortmund[Hölzgen et al., 2011]. Although the initial plan was to run tests against other agent teams prior to the competition, time constraints made this impossible.

Software Architecture

Question 1. Which programming language did you use to implement the multi-agent system?

The agent system was implemented using Python 2.7 and SWI Prolog 5.10.5. DeLP, a defeasible logic language, was used as a service within Prolog.

Question 2. Did you use multi-agent programming languages? Why or why not to use a multi-agent programming language?

No multi-agent programming languages/platforms/frameworks were used. Being the first time we participated in the contest, we decided not to use technologies that we had absolutely no experience on. Besides, one of our goals was to develop our own platform in order to keep developing in the future.

Question 3. How have you mapped the designed architecture (both multi-agent and individual agent architectures) to programming codes i.e., how did you implement specific agent-oriented concepts and designed artifacts using the programming language?

The perception is processed by the Python program, that parses the XML. Then, it sends it to the Percept Server that every step merges all perceptions, and delivers them back to the agents. The Python code asserts all the perception into Prolog, then querying it for the next action to be executed. Prolog handles all the decision making, argumentation and planning, and returns the action binded to a variable to Python, that then generates with it an XML to be sent to the server.

Question 4. Which development platforms and tools are used? How much time did you invest in learning those?

All our code was written using either vim (on Linux) or Notepad++ (on Windows). We used no IDEs, but occasionally we did use the SWI-Prolog integrated debugger.

Question 5. Which runtime platforms and tools (e.g. Jade, AgentScape, simply Java, ...) are used?

How much time did you invest in learning those? Python and Prolog were the chosen languages for the development of the system. Most of us had already worked with both of them, so we did not spend much time learning those.

Question 6. What features were missing in your language choice that would have facilitated your development task?

Question 7. What features of your programming language has simplified your development task?

Python's amenity to rapid application development and 'batteries-included philosophy' facilitated implementing the communication layer to the MASSim server, parsing of perceptions, rapid addition of planned features and bug correction. We made use of Prolog's declarative nature to model states of the world, and it also made it more straightforward to implement search algorithms.

Question 8. Which algorithms are used/implemented?

Search algorithms, as Uniform Cost Search and Depth First Search, as well as the zone-coloring algorithm were implemented in Prolog. The implementation of Defeasible Logic Programming (DeLP) by the LIDIA was used for the deliberative process.

Question 9. How did you distribute the agents on several machines? And if you did not please justify why.

Initial plans were to distribute agents on several machines. Each agents runs as a separate process, and communicates with others via TCP sockets. After some experience and benchmarking, agents were run on one machine due to performance issues. Having the choice was a benefit of the proposed design.

Question 10. To which extend is the reasoning of your agents synchronized with the receive-percepts/send-action cycle?

All the reasoning is done after receiving the percepts, and before sending the action.

Question 11. What part of the development was most difficult/complex? What kind of problems have you found and how are they solved?

The most difficult problems were related to optimization. Much of our time has been spent in reducing the complexity of our algorithms, and the times they are called.

Question 12. How many lines of code did you write for your software?

Total LOC is 5842.

Strategies, Details, and Statistics

Question 1. What is the main strategy of your team?

The main strategy of the team consists of detecting profitable zones from the explored vertices, and positioning the agents correctly to maintain, defend and expand the zones. Every agent, beyond its role, is concerned with the formation and expansion of zones. The decision-taking process is responsible for calculating and selecting the most beneficial intention, which may be focused in the zone conquering (if possible), or not.

Question 2. How does the overall team work together? (coordination, information sharing, ...)

The agents coordinate in an implicit way. This is, the information shared consists only of the perception received, without having neither preprocessed beliefs, nor control variables. The agents do not communicate their intention, or plans, so any coordination that they may have has been achieved implicitly.

Question 3. How do your agents analyze the topology of the map? And how do they exploit their findings?

Agents make no assumption about the map topology. The exploration of the map is done gradually, as a result of the reasoning process.

Question 4. How do your agents communicate with the server?

Some functionality provided by the eismassim library was reimplemented in a connection library in Python.

Question 5. How do you implement the roles of the agents? Which strategies do the different roles implement?

Agents recover their assigned role from the simulation start message. Each role has a couple of separate files, that have specific code, including the arguments used in the decision-taking module, and the setting of the beliefs needed for those arguments. All agents follow the same concept. Every agent is concerned with the formation and expansion of zones, beyond its role.

Question 6. How do you find good zones? How do you estimate the value of zones?

Agents are not primarily focused on finding new zones, but they attempt to expand and maximize the points of the existing ones. They calculate whether they are part of a zone or not. If an agent is not being part of any zone, it tries to regroup with its teammates. When a zone is formed, and the agent is part of it, for each potentially beneficial neighbor node, the agent calculates how much points would the team gain if it moves, and tries to expand the zone. This estimations are done with our reimplementation of the coloring algorithm used by the MASSim server.

Question 7. How do you conquer zones? How do you defend zones if attacked? Do you attack zones?

Both attacking and defense of zones are implicitly implemented. saboteurs prefer to attack enemies that are near, so if an agent of another team enters our team's zone, it will be attacked by the saboteurs in the zone. The same happens with enemies in their own zones. Zones are not intentionally destroyed, but any agent that is part of a zone may be attacked, affecting possibly the structure of the enemy zone.

Question 8. Can your agents change their behavior during runtime? If so, what triggers the changes?

Our agents do not change their behavior during runtime. This feature was analyzed, but the team did not have enough time to finish its implementation.

The approach proposed consists in adding a phase indicator that holds different phase values like 'exploration', or 'expansion'. The phase is updated at runtime considering elements like the number of steps or the team's overall performance in the simulation in progress.

The phase component is weight as an extra factor that modifies the potential benefit of each action, so that its inclusion directly affects the action-selection process.

Question 9. What algorithm(s) do you use for agent path planning?

Path planning is implemented with an Uniform Cost Search. What we tried to minimize was the amount of steps required to achieve the goal, rather than the spent energy. The returned result is a list of actions to be done.

Question 10. How do you make use of the buying-mechanism?

Agents follow a list of predefined buying actions, when the necessary amount of money is reached. This behavior follows the idea of getting some specific skill upgrades that the team considered important to achieve early in the simulations.

Question 11. How important are achievements for your overall strategy?

Achievements are not explicitly taken under consideration. The agents' reasoning process is not affected by the possibility of completing achievements. However, the team can manage to achieve a significant number of them, which results naturally from the agents' behavior.

Question 12. Do your agents have an explicit mental state?

Agents have a complete and explicit mental state. It consists of a set of components, such as beliefs, desires, intentions, and plans.

Question 13. How do your agents communicate? And what do they communicate?

Agents only communicate their perceptions via a perception server implemented in Python. On each perceive/act cycle, agents receive the percept from the MASSim server, separate the information which will remain private and which will be shared. The public part of the percept is sent to the percept server, which performs a union of all percept and send the difference back to each agent. After receiving the joint percept, the agents enter a belief setting phase, and later an argumentation phase.

Question 14. How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?

There is no agent hierarchy, and given the decision-making process takes place individually for each agent, there is no organization between them. The only organization that they have is the proper given by the environment, which is the roles.

Question 15. Is most of your agents behavior emergent on an individual and team level?

The decision-taking module makes use of other agents' status, but there is neither negotiation nor intentions exchange, so the team performance is emergent on an individual behavior.

Question 16. If your agents perform some planning, how many steps do they plan ahead?

The agents make plans as long as the selected intention requires. This may sound excessive, but the possible goals were previously selected for their potential, taking in consideration their distance (in nodes, not in actions). However, plans are recalculated in every step.

Conclusion

Question 1. What have you learned from the participation in the contest?

Being our first experience building a system this size, we learned several lessons about working in large projects, such as setting standards and synchronizing versions of the technologies used.

Question 2. Which are the strong and weak points of the team?

In LIDIA, our teammates have done an important amount of research in argumentation and multi-agent systems. This allowed the team to take advantage of previous experiences. As a weak point can be considered the lack of experience in large projects.

Question 3. How suitable was the chosen programming language, methodology, tools, and algorithms?

In retrospective, we may have taken the right decisions regarding the programming languages. DeLP resulted to be suitable for the implementation of the decision-making module since it was flexible enough to develop our argumentation approach.

Question 4. What can be improved in the context for next year?

There were several hotfixes that were written and deployed at the same time we were facing our competitors due to the lack of testing in the actual context of the competition. This situation should obviously not happen, and adding much more real testing is one of our main priorities for next year's competition.

Question 5. Why did your team perform as it did? Why did the other teams perform better/worse than you did?

We had several problems that did not let us perform as good as we expected. Our lack of experience in this kind of contests, unexpected network and latency problems, as well as some bugs that caused critical performance issues, caused our team to lose several matches that could have been won otherwise.

Question 6. Which other research fields might be interested in the Multi-Agent Programming Contest?

Both Robotics and Gaming AI are interesting fields that could benefit from participating in the contest.

Question 7. How can the current scenario be optimized? How would those optimization pay off?

More information for the nodes, including something useful for a directed search (i.e., absolute coordinates), would help in the implementation of a A* search (which would decrease execution time). Defining the most valuable zones randomly would benefit teams that thoughtfully look for and conquer good zones, rather than teams that assume that the center of the map is the most valuable zone and do not explore the rest.

More feedback from the server would be appreciated, specially involving errors.

Finally, we think it would be really helpful that we have test matches in a more early stage, in order to have more time to correct errors in the client.

HactarV2: An Agent Team Strategy Based on Implicit Coordination

Marc Dekker, Pieter Hameete, Michiel Hegemans, Sebastiaan Leysen, Joris van den Oever, Jeff Smits, and Koen V. Hindriks

Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
agent-contest@mmi.tudelft.nl

Abstract. In this paper we report on the design and implementation of our multi-agent system, called HactarV2, for the Agent Contest 2011. HactarV2 has been implemented in the agent programming language GOAL. One of the main challenges of the Agent Contest is to design a *decentralized* multi-agent system that is able to strategically compete with other agent teams. To address this challenge, the strategy of HactarV2 is based on implicit coordination between agents and there is no central manager that keeps track of all information. The aim, moreover, has been to minimize the communication between agents. Because initially agents are randomly placed on the map, in the first phase of the game the agents individually explore the map and update each other.In the second phase of the game, which starts when the agents have located high value nodes on the map, the agents group together and act as a swarm to maintain and possibly expand the zone on the map that is occupied by the agents. All the work put in by our team payed off and HactarV2 won the Agent Contest without losing a single game. And although we did not get the highest score nor the highest score difference we did get the lowest score against and the highest single game score.

1 Introduction

The scenario of the Multi-Agent Contest 2011, annually organized by Clausthal University of Technology[Behrens et al., 2010a], involves agents that aim to explore and exploit the best zones on the planet Mars. The scenario assumes that water wells have been discovered on Mars and it is now possible to populate the planet. In order to do so, however, it is important to explore Mars, to locate the best water wells, and to occupy those places. Of course, our team of agents is not the only team that is exploring Mars but it competes with another team that has the same goal.

The map of Mars is represented as a graph where nodes denote places and have a value which indicates the amount of water that is present. A node is connected to all its neighboring nodes except for those nodes where the map is mirrored to obtain a symmetric map. Each team consists of 10 agents for

exploring Mars territory. Five different roles are available and each role is assigned to one pair of agents. *Explorers* can determine the value of nodes; *Sentinels* have a larger visibility range and can provide more information about what happens on the planet; *Inspectors* can determine the roles and status of enemy agents; *Saboteurs* have the ability to attack and disable opponent agents; and, finally, *Repairers* are able to restore disabled agents back to a working state.

An elaborate scoring scheme is present which determines the overall scores each round and points are accumulated over a 1000 rounds. Points can be scored by effectively using the capabilities of a role; for example, so-called achievement points are assigned when performing multiple successful attacks. Achievement points can be utilized for purchasing upgrades for agents or to contribute more directly to the team score. Another way to score points is to locate high valued nodes in the graph, positioning agents on that node, and defend them from the opposing team. By occupying larger zones of nodes, moreover, more points are scored.

In this paper we report on the design and development of our multi-agent system (MAS) called HactarV2. The HactarV2 MAS has been developed at Delft University of Technology by the authors, a team of six students and Koen Hindriks, who supervised the team.

The team name is based on the super computer Hactar from the book "Life, the Universe and Everything"[Adams, D., 1982]. Hactar is a computer whose components reflect the pattern of the whole. After failing its intended purpose it gets pulverized and scattered through the universe. Still operational, Hactar proceeds to slowly recombine and become a cloud of particles. It then tries to destroy the universe only to be thwarted by the main protagonist's terrible cricket skills. We think that HactarV2 is a fitting name for our MAS because initially all agents are scattered all over the map as their positions are randomly allocated and then the agents later during the game recombine into a swarm.

2 System Analysis and Design

During the design and development of our MAS we have used the agile software development approach Scrum [Schwaber, 1995]. Scrum is an iterative, incremental framework for project management. It emphasizes short development cycles in which clear targets are set to build and extend the functionality of a system. We have used the open-source platform iceScrum[iceScrum, 2009] for managing and maintaining our development process and for collecting all ideas concerning strategy choices, implementation issues and optimization problems. iceScrum supports working with virtual sticky notes for representing tasks that can be assigned to people and for keeping track of progress. We decided not to use an agent-based development methodology such as Prometheus[Padgham and Winikoff, 2003] because the team did not have sufficient experience with any of these methods.

In the design of our MAS we opted for a completely decentralized approach. One of the main reasons for this choice has been the communication overhead that is needed when a central manager would have been introduced. This would have required the MAS to perform multiple reasoning cycles to exchange messages between all agents and a manager agent. Our agents, however, use all available time within a simulation round (2 seconds) to decide which action to take. As a consequence, messages are received only in the next simulation round and there is no time within a single round to process messages. Information exchanged between agents thus would possibly be outdated which makes message exchange ineffective. This applies in particular to information about positions and the zone that is occupied by agents but, of course, less so to information about the map itself because the map itself remains the same throughout the game.

Instead of exchanging many messages with a central manager that controls what agents do, we designed our agents to base their decisions mainly on the information that is perceived by the agent itself. This ensures that the information is up-to-date and reduces the need for communication.

Message exchange between agents is used in particular (i) for informing other agents about the structure of the map, (ii) between disabled agents and repairers for requesting a repair, and (iii) between non-saboteur agent to saboteurs for reporting locations of opponent agents. Moreover, an agent will only send these messages if it is sure that the receiving agent does not perceive this information itself.

The main reason for exchanging map information between agents is that they need to have all available map information to ensure adequate performance and to avoid doing probe and survey actions that already have been performed by others. Therefore, information acquired by performing a probe or survey action is broadcast to all other agents. This may require each agent to process up to 90 messages that are received from the other 9 agents per round. Because processing all received map information each round requires extensive updating of the belief base, we have had to pay quite some time to optimizing these updates. The benefit of sharing all map information, however, is that the exploration process can be optimized in terms of speed and efficiency (i.e. avoid duplicating actions that have already been performed) and agents are able to do path finding completely autonomously. A downside of this design, of course, is that all agents have to process the map information which would not have been the case in a centralized design.

Our decentralized design implies that agents make decisions autonomously. One potential issue that may arise in such a setup is that different agents will attempt to achieve the same goals. In order to prevent this, agents have been equipped with the capability to predict what other agents will do. This allows agents to decide who will actually adopt a particular goal. We discuss these capabilities in more detail in Section 4.

2.1 Testing

While developing our MAS, we have put a lot of effort in testing and analyzing test results. We considered testing to be very important as it greatly facilitates the evaluation of various ideas and strategies that we tried during our short development cycles. A great benefit of extensive testing has been that it quickly made us familiar with all the details of the simulation scenario.

We have used various test strategies. Initial testing focused on whether the MAS behavior was coherent. At this stage we used dummy agents as opponents that did nothing during a simulation (performed skips each round). In subsequent stages, when we had a reasonably performing MAS, we tested against the Java teams that were supplied by the organizers. These tests were particularly interesting as they informed us about the way our agents reacted to very different agents. We also performed tests in which our most recent MAS played against older versions in order to verify whether we actually improved our MAS over time. During these tests we observed suboptimal behavior that we believe could only have been found because the strategy of the opposing MAS of our earlier versions was still quite similar. We have also participated in all test matches that were organized by the contest organizers.

One testing tactic that we used while debugging our MAS involved the use of an edited XML configuration file for a simulation which granted our team 2 million seconds to send their actions. This made it easier to pause the agent system and study the state of the MAS at a time when something went wrong. As a result, bugs were found more easily.

In total we have spent roughly 500 man hours on implementing our MAS. Around 200 hours were spent on increasing performance and solving other problems and the remaining time was spent on implementing and debugging the multi-agent strategy.

3 Software Architecture

The HactarV2 MAS has been programmed completely in the agent programming language GOAL [Hindriks, 2009,Hindriks et al., 2001,goa, 2011]. All team members were familiar with GOAL because it is being taught as a first year course in the Computer Science curriculum at Delft. In this section, we briefly discuss GOAL and some aspects related to how we structured our code.

3.1 GOAL

GOAL is a logic-based agent programming language. One of its main strengths is that it facilitates the development of high-level strategies for agents. The current version of GOAL uses Prolog to represent the knowledge, beliefs, and goals of an agent. Prolog is a declarative programming language. Prolog programs consist of rules and facts which describe *what* is the case and computation is a form of theorem proving instead of the usual procedural style where

a programmer needs to write programs that dictate *how* something is to be computed. Using Prolog, GOAL agents can derive new facts from their beliefs about the environment and the goals they want to achieve. GOAL agents derive their choice of action from their beliefs and goals by means of condition-action rules.

We have found that it is quite important to pay attention to the predicates used to represent the environment in Prolog. This is important to ensure readability of the program code and for performance reasons. Our initial representation of the map, for example, by two different predicates that were used for separately storing information about nodes and edges turned out to decrease performance.[1] We replaced this representation for another representation in which the id of a node, its value, and a list of outgoing edges is stored by means of a single predicate. This representation increased in particular the performance of queries that retrieved the neighbors of a node. But more importantly, by using this representation a significant reduction in the size of the belief base of an agent has been realized.

GOAL is distributed with a complete IDE for programming, testing and debugging a multi-agent system. The platform fully supports the Environment Interface Standard [Behrens et al., 2010b]. This allowed us to focus completely on the strategic aspects of the contest and we did not have to spent time on lower-level details, such as sending actions to and analyzing the XML files that are received from the simulation server.

The complete system (i.e. MAS and environment interface) was run on a single high end machine. Development on a single machine has been easier. We have considered whether we should distribute the MAS on multiple machines, which is supported by GOAL, mainly for performance reasons. Because our MAS turned out to be efficient enough to run it on a single machine, we have not investigated this possibility any further.

3.2 Agent Structure

GOAL agents execute a classic Observe-Decide-Act (ODA) cycle. At the start of each reasoning cycle of a GOAL agent, all percepts are collected and processed. Our agents start a new cycle after receiving information from the simulation server that a new round has started. After processing the percepts, agents process the messages that they have received. GOAL provides various built-in actions for updating an agent's state. The basic idea of processing percepts first and thereafter messages is that what the agent has observed itself then can be used to verify the information received through messages. GOAL provides a special module for processing events such as percepts and messages which is called the *event module*. After processing all events in order to ensure the agent's mental state is up-to-date again, the agent proceeds

[1] We could trace performance of Prolog queries by means of the logging functionality provided by GOAL.

with deciding what to do next. This choice is made by evaluating condition-action rules which are part of the agent program. These action rules are triggered in another module called the *main module* which is executed after processing the rules in the event module has finished. When a choice is made, the choice of action is subsequently sent to the environment.

GOAL agents are sets of *modules*. There are three types of modules that have a special role, including the event and main module mentioned above and a third module called *init module* for initializing the agent. A programmer, however, can add as many modules as he needs. For example, code related to communication, percept handling, navigation, as well as role specific tasks have been placed in separate modules. The main benefits of the use of modules are that it significantly reduces the chance of code duplication and that it facilitates multiple programmers to each focus on a specific part of the code while still maintaining a clear overview of the overall structure of the MAS.

Every agent in our MAS has a similar structure. In order to prevent version problems and code copying, we decided to make one master agent which can handle all different roles. By using an organization of code that is very similar to the Strategy design pattern, each agent uses role specific modules while sharing standard modules related to e.g. navigation. One shared main module deals with all the role-independent information and strategy choices that are common to all agents. This includes, for example, the processing of percepts that are received from the environment, basic communication, and navigation tasks. The function of our main module is to connect all the smaller sub-modules and jump back and forth between them. Once all the common tasks have been executed the main module checks the role of the agent in question and selects role-specific modules that handle all tasks and decisions that concern that specific role. We can see the module selection happening in code snippet 1 displaying the main program section for the agents. The first condition checked is that no action has been selected yet this round, if an action has been execute the program section will be done. Next the agent checks if it is disabled right now if so it will start the disabled module. Next the agent checks if we control the whole map because if we do we will not have to swarm but can do other things. If this is not the case the role specific module of the agent is run and if this still does not yield an action it will go explore. The total program code consists of 1758 lines of code spread over 18 files.

4 Strategies, Details and Statistics

The strategy of our MAS distinguishes two phases. In the first phase, discussed mainly in Section 4.1, agents do not yet act as a team. This corresponds to the initial state of the game in which agents are randomly placed on the map. In the second phase, agents act as a team in order to occupy high valued zones on the map. This phase is discussed in Section 4.2. As path

```
program {
%Only try to find a new action when one was not chosen in this step yet
if bel(not(doneAction)) then {

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

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

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

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

%Otherwise skip (should never happen)
if bel(true) then skip.
}
}
```

Code 1: The main program section of HactarV2 agents. This section is run each turn and decides what modules will be used this turn. Options are checked linear. (Lines beginning with % are comments.)

planning is quite important in the game, we discuss our approach to finding routes on the map separately in Section 4.3.

4.1 Individual Agent Strategy

At the beginning of a game all agents move and act on their own. Per role a different strategy has been designed. We also designed *defensive* strategies for each role and a buying strategy which we discuss at the end of the section.

The goal of an *explorer* agent at the start of the game is to find the highest valued node on the map. We call this node the *optimum*. This strategy works because the map generator produces maps that have one cluster of higher valued nodes at the center of the map. Once the optimum is found, the name of this node is sent to the other agents and a swarm can be formed to occupy the zone around this node. The first phase ends when the optimum has been found.

The strategy of an explorer for finding the optimum consists of the following rules of which the code can be seen in code snippet 2. If the node at which the explorer is located has not yet been probed, an explorer agent will first probe it. Similarly, if the edges going out of the node have not been surveyed, an explorer agent will survey these edges. Moving to another node depends on the value of this node and that of neighboring nodes. The agent will go to a node that has not yet been probed only if the current node has a lower value than the last visited node and that node is connected to the current node and the last visited node. Otherwise the agent will go back to the last visited node. There is one exception to this rule. If there is a neighboring node which has a higher value than this one (which was found by the other explorer), the agent will go there. The agent will also try to go to a neighboring node that is not close to a (potentially) dangerous opponent. If there are no safe nodes, the agent will take a chance and go to a potentially dangerous node anyway. The agent will conclude that the optimum has been found if no move can be made according to the previously outlined rules. This conclusion may not always be right (the algorithm outlined is not perfect) but works pretty well in practice.

Once the optimum has been found, an explorer agent will join the other agents and start swarming as a team around the optimum. It will continue to go to and probe nodes when nearby nodes are found that not yet have been probed. Doing so allows it to find even higher valued nodes than the node that is currently believed to be the optimum. If it finds a new optimum, the agent will inform the other agents. The action selection rules for these decisions can be seen in code snippet 3. The first option is that the agent is not in the zone anymore so it will have to head their. The next option is that it is in the zone and a node in the zone has not been probed yet so it will got their to probe it. If neither of these options are applicable it will go into normal swarm behavior.

As a defensive strategy, an explorer basically will try to run away as quickly as possible from nodes it considers to be unsafe. A node is considered to be

```
%go into defensive mode if it is not save here
if bel(not(noFlee), currentPos(Here), not(safePos(Here)))
then defense.

%Probe your node if it is unprobed
if bel(currentPos(Here), needProbe(Here), me(Name), team(Team),
  findall(Agent,
    (visibleEntity(Agent,Here,Team,_), role(Agent,'Explorer')),
    Agents),
  agentRank(Agents,Name,Rank) )
then selectProbe(Rank).

%Survey edges with unknown weight around the current node if any
if bel(currentPos(Here), needSurvey(Here), agentRankHere(Rank) )
then selectSurvey(Rank).

%If in optimumZone and walking to optimum, then stop moving
if bel(inOptimumZone, optimum(Opt), gotoPath(L), last(L, Opt),...)
then cancelMove + swarm.

%Move to a location on the map.
if a-goal(currentPos(X)) then move.

%If optimum has not been found yet go look for it.
if a-goal(optimum) then searchOptimal.

%Otherwise do swarm move
if true then swarmProbe.
```

Code 2: The main decision making code for the explorer role. The program goes through the options in a linear fashion.(Lines beginning with % are comments.)

```
%Get back to buddies if not in the zone, dont stray away from the group
if bel(not(inOptimumZone), optimum(X), currentPos(Pos),
  path(Pos,X,[Here,Next|Path],_))
then adopt(currentPos(X)) + insert(gotoPath([Next|Path]), noFlee) + move.

%Find the closest unprobed node in the optimum zone, and go there
if bel(currentPos(Start), pathClosestNonProbed(Start,
  NonProbedVertex, [Here,Next|Path], Dist),
  verticesNearOptimumZone(Vertices),
      member(NonProbedVertex, Vertices))
then advancedGoto(Next) + insert(gotoPath([Next|Path]), noFlee)
  + adopt(currentPos(NonProbedVertex)).

%Nothing to probe, participate in swarm
if true then swarm.
```

Code 3: The action selection code for an Explorer agent while it is in swarm mode.(Lines beginning with % are comments.)

unsafe if on that node an opponent agent is located that is a saboteur or the role of that agent is unknown. All other nodes are considered to be safe. Explorer agents will only move to safe nodes.

A *sentinel* moves on the map using the same exploration strategy as discussed above. It tries to keep its distance from (potentially) dangerous opponents. After an explorer finds a zone worth defending around the optimum, the sentinel will go there as well, join the other agents, and start swarming around the optimum. We experimented with sentinels that bought a lot of sensors in order to be able to see a large part of the map. But we discarded this strategy for the sentinel because it is very expensive and did not seem to benefit the performance of our MAS.

As a defensive strategy, a sentinel will parry when an opponent saboteur is present on the same node. The idea is that in this way, if the saboteur attacks, our team can gather parry achievements. If the role of an opponent agent is unknown, a repairer will also initially parry. However, when the first parry turned out to be unnecessary, with a 50% chance, a sentinel agent will ignore opponents with unknown roles on the same node. Moreover, in case multiple agents are defending against one opponent saboteur, agents will leave the node with a 50% chance whenever their parry was useless. This 50/50 choice prevents that one opponent saboteur will keep multiple agents busy on a node.

The *inspector* uses the same basic exploration strategy and swarming behavior that most agents use. The main difference is that the priority of an inspector agent is to inspect opponents that have not been inspected yet. After doing so, the information obtained is shared with all other agents. In addition an inspector will repeatedly inspect an opponent saboteur every 50

rounds. This enables the saboteurs from our MAS to keep track of the state of opposing saboteurs.

As a defensive strategy, an inspector will run away if the opponent agent is known to be a saboteur. However, if the opponent agent is unknown, it will move towards it to inspect it.

Agents that are disabled will ask for help from a *repairer* agent. Disabled agents will move towards the repairer that is closest after informing it that the agent wants to be repaired by the repairer. If a repairer is not already committed to another agent, it will also start moving towards the disabled agent that requested for help. Whenever a repairer gets a request to repair another repairer, however, it will drop its current commitment and start to move towards that repairer. A repairer will also drop its current commitment when it gets disabled itself and start moving towards the other repairer. Disabled agents send a path to the repairer they are moving to, to prevent the repairer from having to calculate a path towards the disabled agent and to save time.

Repairers use exactly the same defensive strategy as sentinels.

The *saboteurs* start the match in search and destroy mode. They receive information from all agents that is useful for locating opponent agents. They will move towards a last known location of an opponent agent that is closest to their own position and attack that agent. While testing we found that this often resulted in the opponent having fewer agents available at the start of the game which reduced the effectiveness of our opponent.

When the saboteur has decide to attack someone it will use the module saboteurAttack to accomplish the attack. The cod of this module is given in code snippet 4. If the target of the attack is on the same node as the saboteur it will be attacked except if the target successfully parried the attack of the saboteur last turn and another enemy agent is present it will attack the other agent. If it can not attack the enemy on the same node it will recharge. If the target is on another node the saboteur will head for that node.

Buying can be important during the game but we consider achievement points to be more important. Therefore our buying strategy is designed to keep our achievement points as high as possible and to spend less money than the opponent does. As the amount of money available has quite an impact on the points that are scored each round, we decided to only upgrade our saboteurs.

The upgrading of saboteurs is aimed at ensuring two things: (i) our saboteurs have 1 health point more then the highest strength of any of the opponent saboteurs and (ii) the strength of our saboteurs is equal to the highest number of health points of any of the opponent saboteurs. If both of these goals are realized, then our saboteurs will survive a blow of an opponent saboteur while destroying them in a single hit.

We start buying upgrades for saboteurs immediately so that they are better then our opponents from the start. After that we wait for input from the inspectors about the opponent saboteurs before upgrading further. Very important is that we will always attack opponent agents at our own location

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

Code 4: The module show what the saboteur does when it has decided to attack something. (Lines beginning with % are comments.)

before upgrading. This prevents opponent saboteurs from interrupting our agents and also interrupts whatever the opponent agent is doing. As a result our score is lower than that of our opponents in the first 100 or so steps but this investment pays off in the remainder of the match; this effect can be clearly seen in Figure 1.

An assumption we have made is that the opponent team gives higher priority to upgrading saboteurs than to upgrading other roles. But even if this assumption would not be true, the idea is that upgrading of other roles is not that important because our saboteurs cannot be attacked by these roles.

Saboteurs do not have a "defensive" strategy but will always attack because they are designed to be superior to the opponents agents.

4.2 Swarming Strategy

When explorer agents have found the optimum (i.e. the node with highest value) and all other agents have been informed about this, these agents will start moving towards the optimum. Because all map information is shared between agents, the probability that each agent will be able to find a path to the optimum is quite high. If an agent is not able to find such a path, it will continue surveying edges until it finds a path to the optimum. Finding an optimum moves the game to the second phase in our strategy. We call this phase the *swarming* phase.

In figure 2 we show what the result of swarming looks like during a game. The HactarV2 agents have formed a zone around the high value nodes in the center. In figure 2 this is the group of almost black nodes. To the upper right of the HactarV2 zone is the zone created by the agents of the opposite team

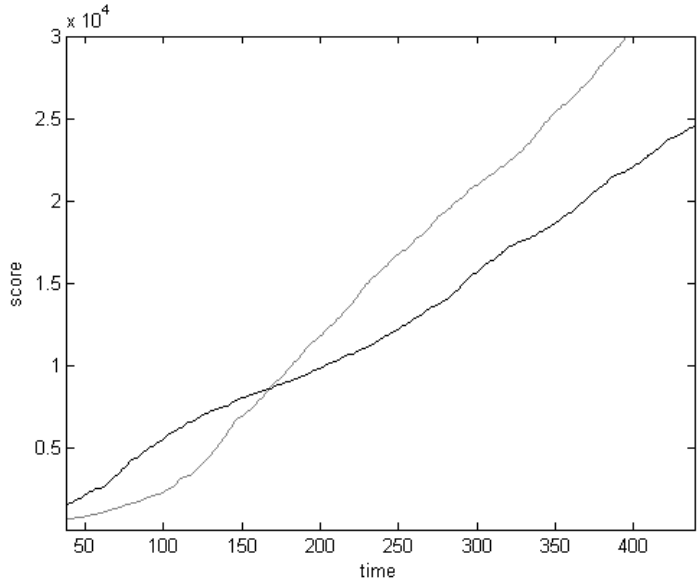


Fig. 1. Scores HactarV2(Gray) vs TUB(Black)

which are colored gray in figure 2. They are forced to built their zone away from the high value nodes and because of this are at a clear disadvantage.

In order to explain the behavior of the agents when they enter the swarm phase, we call this behavior *swarming*, we introduce several concepts. An agent that is part of our team is said to be *dependable* if that agent can effectively participate as a team member in the swarming behavior. Other agents can derive (predict) whether an agent can be depended on because the roles of all friendly agents are known and all of our agents use similar code. These facts allow other agents to predict what kind of action another agent will perform in the next round and to derive from that whether that agent will be dependable.

Agents that are located on the same node are ranked and assigned a unique number called the *agent's rank*. This rank is used when multiple agents are predicted to perform the same action. In that case, based on their rank it is decided which agent will perform that action; the other agents will then perform another action. This rank-based mechanism allows agents to divide tasks among themselves without the need for communication while at the same time ensuring that each agent performs a unique action whenever possible.

Another concept we use is that of one agent being *connected* to two other agents. A link between two agents is said to exist if there are at most two edges that connect the nodes on which these agents are located and these nodes are owned by our team. An agent is said to be *connected* if that agent has links

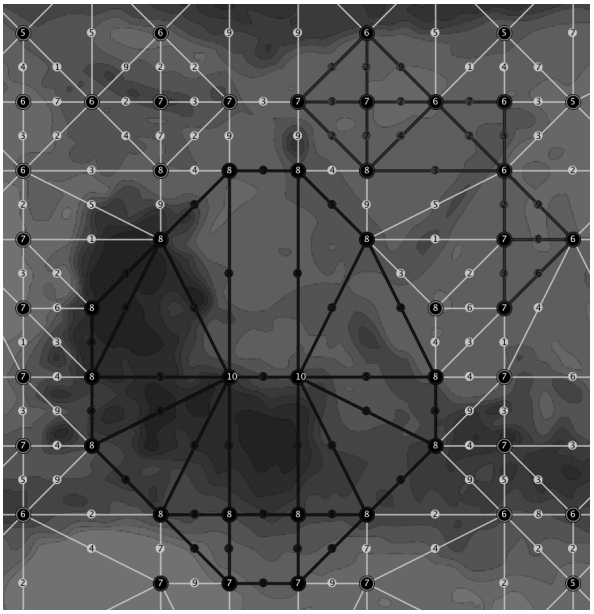


Fig. 2. The swarm created by HactarV2 in the middle with opponents push out of the center.

with at least two other dependable agents. The nodes that connected agents are located on are called *swarm positions*.

The zone of nodes which is owned by our team and contains the optimum is called the *optimum zone*. All agents maintain a list of nodes that are part of this zone as well as a list of those nodes that are just outside this zone. An agent is said to be inside the optimum zone if the node that the agent is on only has edges to nodes that are part of the optimum zone. An agent is said to be on the edge of the optimum zone if the agent is on a node that is part of the optimum zone that is connected to a node that is not part of that zone.

While moving towards the optimum zone the agents will constantly check if they are already in a swarming position that is part of that zone. If that is not the case, they will continue moving towards the optimum zone. All agents that try to become part of the swarm intend to be within the optimum zone. This means that our MAS will create at most one swarm.

The main action selection code for when an agent is in the optimum zone can be seen in code snippet 5. It shows that once an agent has arrived in the optimum zone, the agent will determine the highest valued node directly outside the optimum zone and move towards that node. By using this tactic the swarm will always expand in the direction of the highest valued nodes not yet owned by our MAS. When the agent reaches a position on the edge of the swarm it will consider all nodes it can move to. In addition it will pre-

dict to which nodes its connected agents may want to move. Based on this information, an agent can determine without communication if it can make the best move compared to any of the agents it is connected to. If that is the case, an agent will move to expand the zone, and otherwise it will stay on its current node in order to ensure that the agents will still be connected. (To ensure connectedness it is not required that only one agent moves to expand the zone and we can use the agent rank discussed above to determine which agents can move. We do not discuss the rather complicated details here, however.)

If for some reason an agent is no longer connected with agents in the optimum zone, it will start moving back towards the optimum until it reconnects to agents that are part of the swarm. Using this set of rules we obtain a very robust swarm. They allow agents to hook up with the swarm but also allow agents with more urgent tasks to leave without disturbing the swarm area too much.

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

  %Make a list of swarm positions and get my current rank
  if bel(expandDest(List), List = [[Value, Vertex]|_], me(Id),
      agentRankHere(Rank))
  then{
    %Taking other agents into acount choice optimal location to go to
    if bel(not((connectedAgent(Id, Agent), currentPos(Agent, Pos),
        bestExpandDest(_, Value2, Pos), Value2 >= Value)))
    then gotoSplit(Rank, List).

    %Lower rank if not leader and choice destination
    if bel(not(kingOfTheHill), Rank2 is Rank-1)
    then gotoSplit(Rank2, List).

    %In optimum zone but not on a swarm position so not being usefull
    %Move back to optimal position and try again
    if bel(currentPos(Pos), not(swarmPos(Pos)), optimum(Opt),
        path(Pos, Opt, [Here,Next|Path], _)) then advancedGoto(Next).
  }

  %otherwise take the recharge action.
  if true then recharge.
}
```

Code 5: The program section of the swarm module. When the agent is actively part of the swarm it will follow these instructions. (Lines beginning with % are comments.)

If our zone is being threatened by opposing agents our saboteurs will defend it. Opponent agents are considered threatening if they are not disabled and located at a node that is just outside the current optimum zone. While a saboteur defends the zone it is no longer considered to be dependable. That way the swarm will not be disrupted by unpredictable movements from the saboteur. If a threatening opponent agent moves away from our current zone before it is attacked, the saboteur will become part of the swarm again. Saboteurs that take part in the swarm will not actively look for or attack opponent agents. This is consistent with our main strategy: to ensure that the MAS occupies the optimum zone and the opponent agents cannot occupy a zone that is worth more points.

Especially when the opposing agents had a very similar strategy that involves capturing nodes with the highest values it is difficult for our agents to occupy and maintain a large optimum zone. In these games the points that are obtained by achievements can determine the difference between winning and losing. Attack and parry achievements yield the most points while the inspect achievement yields the fewest points. Probed and zonescore achievements typically also yield few points except when agents occupy almost the entire map. Therefore, we decided to focus on attack and parry achievements.

4.3 Path Planning

In principle our agents do not plan actions in advance. Agents decide every reasoning cycle which action to perform based on the rules that are part of the agent program. Path planning is an exception to some extent because an agent will compute a path to move from a node A to another node B. So, an agent plans at most n rounds ahead where n is the length of a path that is followed. Following a path, however, can quite easily be disrupted if an alternative action is considered a better option.

The path finding algorithms we have used are based on an implementation of Dijkstra's shortest path algorithm in Prolog that is available on the web [Barker, 1999]. Our agents use seven, slightly different versions of this algorithm. These versions differ from each other in what they aim for. The most basic version simply searches for a path between two nodes A en B. But we also have versions for finding a path to the closest non-probed node or non-surveyed edge. Yet other versions search for a path to the closest repairer or closest opponent agent.

5 Conclusion

We have enjoyed the Agent Contest experience very much. It has also taught us various valuable lessons. Maybe the most important thing that we have learned from participating in the Agent Contest is that the continuous testing of a MAS can greatly help to improve the effectiveness of that MAS. We became particularly aware of this after the initial testing against the dummy

agent team (agents that performed skip actions) resulted in basic strategies that also worked against teams developed by other groups.

A second lesson that we have learned is that the design of the structure of an individual agent and the MAS can make a big difference during the development of a MAS. We benefited in particular from the module concept that is supported by GOAL[Hindriks, 2009,goa, 2011]. The strategy associated with each role has been coded in a separate module and there are dedicated modules for functions such as navigation and communication that are shared by agents. This greatly facilitated dividing coding tasks within the team. Maintaining a clear and concise documentation of the system has also been important in this regard. We have documented in particular the *ontology* used and shared by all agents. The ontology documents all predicates used to represent the beliefs, goals, percepts, actions, and knowledge rules of agents and briefly explains the meaning of each predicate and how its parameters should be instantiated. The ontology facilitates keeping track of what everyone is doing and has saved us a lot of time.

We believe that the key strength of our MAS resides in the performance of our saboteur and repairer but also in the manner our agents team up and form quite robust swarms. The buying strategy of our saboteur has also been quite effective. The final contest, however, also demonstrated some points that still can be improved. For example, we observed that sometimes both saboteurs attacked one and the same target and that our swarm sometimes is not as robust as it should be.

Our MAS performed very well and we won the Agent Contest. As mentioned above, the strategy of our saboteurs and repairers proved the key to success. Our saboteurs are aggressive at the start of a game but put on a tight leash when a swarm is formed to ensure that opponent agents cannot get their agents near the high valued nodes. Our repairers make sure that as many agents as possible are active and are able to participate in the swarm.

Mainly because of our buying strategy, we usually start with a lower score than our opponents. This is made up for, however, when the highest valued node is found by our agents and we start to gather more and more achievement points. This is clearly illustrated in Figure 3. In this figure the gray line represents the score of HactarV2 and the black line represents the score of the Python-DTU team. For the first 275 steps HactarV2's score stays below that of Python-DTU's but thereafter HactarV2 takes and keeps the lead.

To write our code we used the agent programming language GOAL and the IDE that is distributed with this language. This was the first time that GOAL has been used in the Multi-Agent Programming Contest and we are very satisfied with its performance. We have deviated from some of the guidelines for programming GOAL agents for reasons of efficiency but otherwise GOAL proved to be very suitable for the programming of, for example, the fast and efficient path planning algorithm that we used.

We think that the Mars scenario is a good and interesting scenario. It poses some interesting challenges with respect to coordinating agents. We

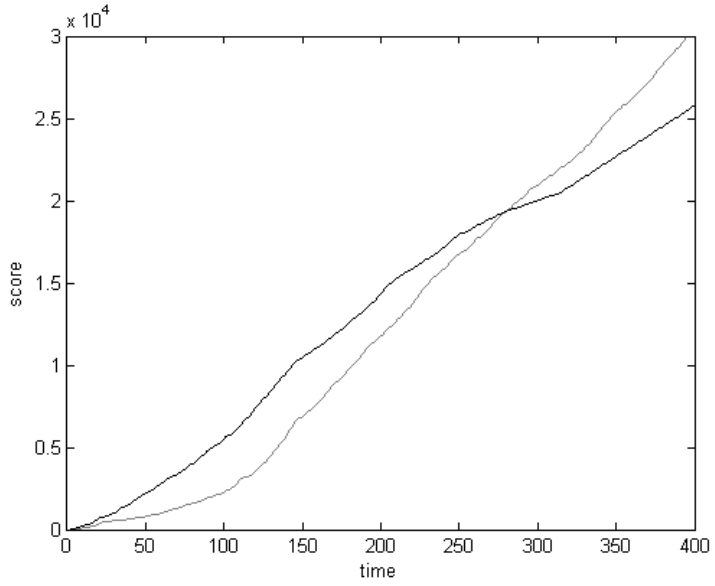


Fig. 3. Scores HactarV2(Gray) vs Python-DTU(Black)

believe that scaling up the contest by having more agents would create an even bigger challenge. Maybe one quite interesting idea is to be able to earn and dynamically add new agents during the game, for example, based on certain achievements. More pragmatically, as we discussed above, we have found testing to be very important for the success of our MAS and we believe that it is a good idea to start contest wide testing earlier and in phases. In a first phase - which we think should start about a month before the contest - a general test to check for and identify possible problems related to establishing a connection and to communication is advised. Thereafter in the second phase test matches should be organized that allow teams to test their strategies.

We also think that adding local optimums in the map would make the contest more interesting. This would require more complicated strategies for swarming and would provide a bigger challenge for the exploration of the map. Finally, we think it would be a good idea to add an extra achievement that promotes the cooperation of agents. This achievement should not replace the achievement for the area that is controlled but would count the number of agents that share a connection each turn and sum this amount over all turns. This would reward teams whose agents cooperate above teams whose agents move around individually.

References

[goa, 2011] (2011). `http://mmi.tudelft.nl/trac/goal`.

[Adams, D., 1982] Adams, D. (1982). *Life, the Universe and Everything*. Harmony Books.

[Barker, 1999] Barker, C. (1999). LPA Win-Prolog goodies. A WWW collection of Prolog source snippets. *http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm*.

[Behrens et al., 2010a] Behrens, T., Dastani, M., Dix, J., Köster, M., and Novak, P. (2010a). The multi-agent programming contest from 2005-2010: From collecting gold to herding cows. *Annals of Mathematics and Artificial Intelligence*, pages 277–311.

[Behrens et al., 2010b] Behrens, T., Hindriks, K., and Dix, J. (2010b). Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, pages 1–35.

[Hindriks, 2009] Hindriks, K. (2009). Programming rational agents in goal. In *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US.

[Hindriks et al., 2001] Hindriks, K., Boer, F. d., Hoek, W. v. d., and Meyer, J. (2001). Agent programming with declarative goals. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 248–257. Springer Berlin / Heidelberg.

[iceScrum, 2009] iceScrum (2009). icescrum, free and opensource platform for your agile developments. *http://www.icescrum.org/en/*.

[Padgham and Winikoff, 2003] Padgham, L. and Winikoff, M. (2003). Prometheus: a methodology for developing intelligent agents. In *Proceedings of the 3rd international conference on Agent-oriented software engineering III*, AOSE'02, pages 174–185. Springer-Verlag.

[Schwaber, 1995] Schwaber, K. (1995). Scrum development process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134.

Short Answers

Introduction

1. Different team members had different motivations but generally speaking everyone was interested in coding agents for games.

2. All students that participated in the team either assisted with the multi-agent systems project or participated in that project. In this project, students have to program agents that control bots in the first-person shooter game Unreal Tournament. The MAS project is part of the first-year bachelor curriculum at Delft University.

3. HactarV2

4. All six student members of our team contributed to developing and designing our MAS. All students were first-year or second-year Bachelor students.

5. Multi-Agent Systems and Artificial Intelligence.

System Analysis and Design

1. We did not use any agent-based development methodology. The team did not have sufficient experience with any of these methodologies.

2. Our MAS uses a decentralized strategy. Our strategy is based on an implicit coordination mechanism where agents predict actions that other agents will perform and decide which action to perform themselves autonomously, taking these predications into account. Agents exchange information about edges, nodes, and inspected opponents. Additionally, there are some messages exchanged between all agents and repairers and saboteurs.

3. All information about the map such as edge weights and node values as well as information obtained by inspection about opponent agents is broadcast to all agents. Other information such as repair requests or information on locations of opponent agents is communicated only with agents that make use of this information.

4. *Autonomy*: By communicating the all map information between all agents, at some point in time each agent knows the complete map. Using the map information and information received through percepts, the agent then decides on which action to perform autonomously. It tries to take into account what other agents will do while making these decisions.
   Proactiveness: GOAL explicitly differentiates between a goal and belief base. Each agent has beliefs and goals, accompanied by a set of rules for

reasoning. Using the beliefs in combination with the rules it will conclude what actions to perform in order to reach its goals, making the agent proactive.
Reactiveness: Even when agents are committed to particular goals they will continuously monitor their environment and interrupt its goal-driven behavior temporarily. After performing more urgent actions in response to information received from the environment it will continue with its goal.

5. Our team consists of multiple agents that each execute their own program. Moreover, each agent maintains its own information and there is no centralized information store nor a central manager that coordinates the actions of agents. Each agent communicates directly with all other agents and makes decisions autonomously.
6. Around 500 man hours have been spent on our implementation.
7. Other than some basic strategy discussions on IRC we tried to keep our design as secret as possible. We have utilized every possible testing opportunity that was given to our advantage.

Software Architecture

1. GOAL
2. We used the agent programming language GOAL [Hindriks, 2009,goa, 2011]. GOAL uses Prolog as a knowledge representation language.
3. We used modules to code roles and specific functions that were part of our design.
4. We used the GOAL agent platform, which our entire team was already familiar with.
5. We have used the GOAL platform.
6. None
7. Apart from the modules mentioned above, certain features of the IDE of GOAL which made it possible to insert goals and beliefs manually and to perform queries on the belief and goal bases of agents were helpful. Other features included measuring the speed of certain queries and functionality to pause the agents and check their belief and goal bases.
8. Many customized Dijkstra algorithms and a Breadth First Search.
9. We did not distribute the agents on several machines even though GOAL facilitates this. We felt that our agents should be coded sufficiently efficient to run on a single fast desktop. This would simplify testing and prevent us from spending time on getting a distributed system to run.

10. Agents receive percepts and messages sent in the previous cycle, process them, send messages and do an action.
11. The most complex and difficult part of development was keeping the code as efficient as possible. We solved this by constantly improving our skills in the GOAL and Prolog programming.
12. 1758

Strategies, Details and Statistics

1. The main strategy of our teams it to occupy the zone with the highest valued nodes. Our team of agents builds a swarm around the highest valued node on the map. The swarming agents are used to defend and expand this zone.
2. The only information passed around is information about surveyed and probed nodes, as well as inspected opponents and the location of a repairer. Once an agent is disabled it will transmit its position (or rather the shortest path) to the nearest repairer. Agents also report visible opponents to saboteurs when they think the saboteurs can't see them.
   Agents implicitly coordinate their activities. For example, in order to prevent two agents from performing the same action we implemented a priority system based on agent ranks. While deciding what to do next an agent's rank is taken into account to avoid duplicating actions.
3. All agents perform survey actions. The explorers will try to find the node with the highest value right from the start and will send information about this node to all other agents. Agents will then move to this node and form a swarm. Once a swarm has been established the explorers are used to probe all nodes inside our occupied zone to increase our zonescore.
4. We use the provided EISMASSIM interface to communicate with the server.
5. Each role implements a different strategy though all will join the swarm if they do not have a task with a higher priority. See the paper for extensive discussion of these roles.
6. Our explorers search for the highest valued node on the map, assuming that this node is more or less in the center of the map. See the paper for details.
7. When our agents have found the start of our zone we conquer it, because all our agents will go there and form a swarm. If there are already opponents in this area, the saboteurs will attack these. When we have established a zone the saboteurs will defend it by attacking any opponents

that come near this zone. When defending our agents they first check if the opponent perceived is a known opponent; if the opponent agent is not a saboteur it is ignored. See for more details the paper.

8. Our agents are able to change their behavior at runtime because they decide what to do next every round. In general at the start of the match the agents will explore the map and when an optimum has been found they will change their behavior to start swarming around the optimum. The trigger here is the broadcast message of discovering said optimum. Repairers and saboteurs will switch from swarming to respectively repairing an agent and attacking an opponent whenever the need arises.

9. Our path finding algorithms are all based on an implementation of Dijkstra's shortest path algorithm in Prolog which we found on the website of Colin Barker[Barker, 1999]. We have seven different versions. All versions use the same basic core of the Dijkstra algorithm, which in essence uses the Dijkstra greedy rule to determine which node to visit next, but they differ in that they all try to achieve something else. We have a version that tries to find a path between nodes A and B but we also have versions for finding a path to the closest non-probed node or non-surveyed node. Yet other versions find a path to the nearest repairer or nearest opponent.

10. Since money has quite an impact on how many points you get per round we decided to only upgrade our saboteurs in such a way that they will always have 1 more health point than the highest strength of the opponent saboteurs. The strength of our saboteurs will always be equal to the highest health point of the opponent saboteurs. This way they will be able to survive one blow of an opponent saboteur while disable them in one hit.

11. Since the accumulation of achievements can account for a large part of the score, we focused on gathering all achievements. This, for example, is the reason that our agents do not flee, but prefer to parry.

12. Yes

13. They share information about those nodes that have been fully explored, about probed node values and their own location in certain situations.

14. The organization is implicit. Each agent has it's own (slightly) different behavior and all organization derives from their own behavioral patterns and inferences about the other agents.

15. The behavior is emergent on an individual level. Agents share a few goals as a team but mainly derive what to do next from their own beliefs about the environment and their knowledge about what other agents may do.

16. Generally each step a re-evaluation of the agents actions takes place. Only when trying to reach specific locations some actual planning takes place. In that case, a path is computed to get to some location on the map.

Conclusion

1. We found that testing is very important. Additionally, we found that a modular organization of code and maintaining clear and concise documentation is very beneficial for working as a team.

2. Our team has multiple strong points but the most important are: Our buying strategy, swarm location selection, the usage of our saboteur and the interaction between the repairer and disabled agents. Important weak point are that our swarm can sometimes to easily be disrupted and that our saboteurs have a tendency to go after and attack the same target.

3. We found GOAL a very suitable language to use for the multi-agent contest. It provided us with all the functions we needed. The used path planning algorithm is fast and efficient enough that it could be used without having to worry about time constraints.

4. We found that testing our MAS was very useful. It might be considered to start testing earlier and maybe in phases. In an initial test only the connection and communication with the server can be tested. This gives teams time to fix any issues with this before the actual contest. After this test, actual matches can be played to let teams test their strategies.

5. Our team preformed as well as it did because we made sure all separate parts were robust and dynamic. We paid extra attention to the saboteur and repairer roles as we were convinced that a key to success is to maximize the number of active agents on our team and minimize the number of active agents on the opposing side. These two ideas combined with our optimized swarming system gave us the ability to outperform our opponents.

6. Compiler construction and Algorithms.

7. To make the contest more interesting, one idea is to add local optimums. This would add an extra challenge to the exploration of the map because the agents would have to make sure they have found the actual optimum and not just a local one. Another idea would be to be able to “earn” new agents dynamically during the game based on e.g. certain achievements. Finally, an achievement for swarming behavior could stimulate coordinated behavior of agent teams.

Nargel: A Multi-Agent System Implemented through MaSE and JADE

Vahid Rafe, Majid Fandrousi, Rosa Yousefian, Sina Hamedheidari

Department of Computer Engineering, Faculty of Engineering
Arak University – Arak 38156-8-8349, Iran
v-rafe@araku.ac.ir, m-fandrousi@arshad.araku.ac.ir, r-yousefian@arshad.araku.ac.ir,
s-hamedheidari@arshad.araku.ac.ir

Abstract. For this year's multi-agent contest, finding water storages on Mars, we developed a decentralized Multi-Agent System (MAS) named "Nargel". In this paper, we present the agents' architecture as well as the methodology, team strategies, coordination and cooperation mechanisms which are used to design and implement the Nargel. To design the Nargel, MaSE methodology is used while for implementing our MAS, the JADE platform along with the Java programming language is used. In Nargel, there is no supervisor. In fact, agents share their percepts with each other. Additionally, coordination and cooperation are done with the help of agent speaking mechanism which is provided by the JADE platform.

Keywords: Multi-agent systems . MaSE . JADE . Agent speaking

1 Introduction

The color of planet Mars resembles bloody war scenes. So that, ancient Babylonians called that "Nargel", the God of war [nar,]. Because this year's contest took place on Mars and it was a battle of life (finding water), we had chosen "Nargel" as our team name.

The motivation to participate in this contest was about doing a project for advanced software engineering course. As in developing multi-agent systems there are many challenges, hence we decide to contribute to this contest to develop a multi-agent system.

All the team members except the first author are MSc Software Engineering students. The first author is an assistant professor. Actually, he is the team supervisor. He has also contributed in the last year tournaments [Rafe et al., 2011] and we were made familiar with this contest by him. At first, there were six members in the team, but during the design phase two of them left and the rest (four members) continued the work. Our team consists of two designers, one developer and one supervisor.

2 System Analysis and Design

Obviously the most significant concept in multi-agent systems is decentralization of information and cooperation between agents. It means that if an agent affords all tasks about keeping information and leading the agent cooperation, the other agents cannot act autonomously. For this reason, our aim is to develop a decentralized system in which agents behave autonomously. To do so, each agent has a local world model which stores its percepts and tries to cooperate with other agents to reach a common goal.

Our system agents are autonomous and behave individually, but in cooperation with each other. All behaviors are directed to the main goal of the system.

Also, our agents are proactive. In fact, the agent must decide how and when to do a special job. Escaping from a perilous situation is a simple example of proactiveness in our system. In Nargel when an agent sees an opponent agent on a neighbor node knows this situation as a dangerous situation and escapes from that.

The agents are also reactive, such that they can react upon the opponents actions. For example when there is an attack, they can react and defend themselves against the opponents.

The MaSE (Multi-agent Systems Engineering) methodology was used to analyze and design our system. MaSE is a full-lifecycle methodology for analyzing, designing, and developing MASs. MaSE uses a number of graphically based models derived from standard UML models to describe the types of agents in a system and their interfaces to other agents [Bergenti et al., 2004].

In a nutshell, the general operation of MaSE methodology is divided in two phases: analysis and design. Each phase has some diagrams but the most important ones are Goal hierarchy, Role model for the analysis phase and System architecture for the design phase. Our system goal hierarchy diagram is shown in Fig. 1.

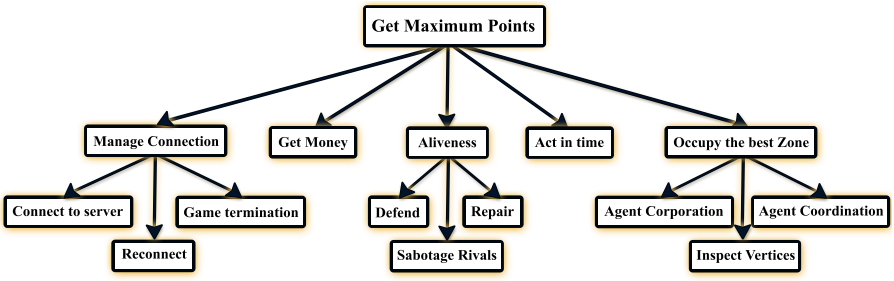


Fig. 1. goal hierarchy diagram.

To create this diagram, we studied this year's scenario perfectly. Then, we extracted the requirements from it and finally, we defined and created our goal hierarchy based on the extracted requirements. In this diagram, goals are abstractions of the detailed requirements. In the next step of the analysis phase we defined roles and created role model based on the goals in the goal hierarchy.The role model is the most important diagram in the analysis phase. Indeed, the role model indicates the policy of design and implementation of the system. The role model and system architecture diagrams of our system will be explained in the rest of the paper.

MaSE is the basis for the agentTool development system. The agentTool is a graphically based, fully interactive software engineering tool, which fully supports each step of MaSE analysis and design. The agentTool also supports automatic verification of inter-agent communications, semi-automated design, and code generation for multiple MAS frameworks. MaSE and agentTool are both independent of any particular agent architecture, programming language, or communication framework [Bergenti et al., 2004].

Unfortunately, we couldn't complete the MaSE phases for this project. So we lost the opportunity of using agentTool to produce codes from models. In this project we worked on most important models such as goal model and role model, and programmed the concepts introduced by these models. No tools were used to program them.

Communication between agents is being done only in early steps of the match to group and to gather agents in a specific position. In the rest of the match, each agent interacts with the environment via its related data structures. Agent communications are done via the agent speaking feature which is provided by the JADE platform. The JADE handles sending and receiving messages between agents and we do not deal with its details. We only composed two separate files for each individual agent, which contain the codes corresponding to sending and receiving messages. Also, we call sending and receiving methods from the agent code. Since every agent can then speak by using these two files to its teammates via the JADE platform. There are not many varieties of messages and just a few messages with simple structure are used to exchange information. More details about agent communication via the JADE will be explained in the rest of the paper.

To design and to implement our MAS we did not consult anybody out of our team. Also, our team members spent totally about 500 hours to design and develop Nargel. Before the tournament we tested Nargel during some matches with HempelsSofa, Sorena and two dummy systems provided by the organizers. additionally, we participated in all the test matches set up by the organizers.

3 Software Architecture

Since we were not familiar with multi-agent programming languages and we did not have enough time to learn them, so we used Java programming language to develop Nargel.

The main architecture concepts of our system are keeping environmental information and establishing communication between the agents to reach a common goal. For keeping information, each individual agent keeps necessary information (like information about the environment, the opponents etc) in some proper data structures. The used data structures are two dimensional matrices for storing environmental information and some variables for storing temporary data, messages et cetera.

A couple of agents define the goals with the help of each other. Defining goals and informing other agents about them is done by agent speaking mechanism. To handle agent speaking we used the JADE run time platform.

About 50 hours were spent to get familiar with that platform and to learn how to take advantage of JADE's capabilities and features in our system. Since the JADE framework is thoroughly implemented in Java, this helped us to use the JADE facilities in Java without any problem. The JADE platform has RMA (Remote Monitoring Agent) which allows controlling the life cycle of the agent platform and all the registered agents. The distributed architecture of JADE allows also remote controlling, where the GUI is used to control the execution of agents and their life cycle from the remote host [Bellifemine et al., 2007]. Our system architecture is shown in Fig. 2.

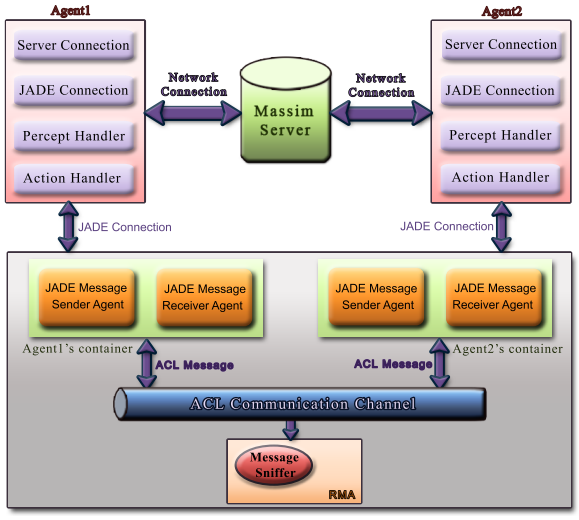


Fig. 2. Our proposed architecture.

Each agent uses a network connection to connect to the MASSim server. Through this connection it can receive percepts from the server and sends action to the server. Communication with the server is done by the use of eismassim package.

Every agent itself makes and runs a sender agent and receiver agent on the JADE platform in order to speak with other agents. The architecture of JADE platform is such that there is a specific container for every agent and the sender and receiver agents which were ran by the team agents are added to it. In the JADE platform the agents exchange their messages via ACL (Agent Communication Language) messaging protocol. Through the Sniffer component of RMA we can monitor all the transmitted messages between agents [Bellifemine et al., 2007].

Despite the fact that the agents had capability to run on several machines, we ran all agents on a single machine because the system workload was low. Before the main tournament (i.e. during the test matches), our agents had many problems in sending actions because of the low bandwidth, filtering restrictions et cetera. But we solved these problems for the main tournaments. In each step, an agent decides and defines its action according to its previous knowledge and current percepts. So, the agent reasoning is completely synchronized with receiving percepts/sending actions cycle.

When implementing the system we encountered some hard and time consuming issues. One was to handle sending actions and receiving percepts. Due to the lack of time and getting rid of the complexities of this problem we used the related codes from dummy agents provided by the organizers. The other concern was about implementing a proper zone making strategy. We proposed a simple approach to solve this problem. This simplicity caused that our agents could not make big enough zones during the contest. Additionally, sometimes enlarging zones destroyed the current zone completely.

Since the contest environment was a graph, the most crucial algorithm to be used was finding the best path between nodes. To this point we used Dijkstra algorithm [Corman et al., 2001] which became more applicable as the contest goes on (environmental information was incomplete in the early steps of the contest and gradually became complete).

We programmed Nargel system in about 3000 lines which are for both implementing strategies and agent communication.

4 Strategies, Details and Statistics

Information about the contest graph was being stored in a matrix named Environmental matrix. This information contained the distance between nodes (i.e. weight of edge between each two nodes). At the beginning, it was assumed that there were no edges between nodes. After a while, when adjacent nodes was recognized and distance between nodes was found by the survey action –which is done by the agents– the environmental matrix was updated. During the match, this matrix was being used for path finding op-

erations. However, our agents did not analyze the topology of the map. In addition, each individual agent kept some data structures:

- Visible Vertices: In this matrix every agent keeps visible vertices.
- Visible Edges: In this matrix every agent keeps visible edges. Each entry contains the source and the destination vertices of an edge.
- Neighbors: Every agent keeps the id of its neighbor nodes in this vector.
- Visible Opponents: In this matrix, position and status (being enabled or disabled) of the visible opponents is being kept.
- Visible Teammates: In this matrix, position and status (being enabled or disabled) of the visible teammates is being kept.

In the beginning of the match, two agent groups are created, one for making and keeping zones and the other for attacking and disturbing the opponent zones. In order to divide agents to these groups, two saboteur agents inform each other about the number of teammates they see. Since each group has a saboteur, the saboteur which sees more teammates joins to the zone maker group and the other saboteur joins to the disturber group. Then, saboteur1 will announce to the other agents that it belongs to the zone maker group while the other belongs to the disturber group. two saboteurs decide on specifying their groups with negotiation via agent speaking and with the use of environmental information. It is also the case for repairers, that is one repairer which sees more teammates joins to the zone maker group and the other repairer joins to the disturber group.

Now we will describe the functionality of these two groups in details:
Zone maker group: In addition to the saboteur, this group consists of seven other agents (i.e. two explorers, two sentinels, two inspectors and one repairer which sees more teammates). The saboteur which sees more teammates says its current position to other members of this group. In fact, in this time a planning is done for the entire match, that is, the area of the map where the team wants to make zones is being specified. Each member keeps the mentioned position in order to go to that position if it becomes disabled in the future.

As soon as the message arrives agents try to get to that position. Each agent for finding the shortest path to the target position uses Dijkstra algorithm and also crosses edges which has enough energy to go from them. During their movement to the target node to gain achievements and to know the environment better (i.e. graph status), sentinel agents and repairers survey, two explorers probe and inspector agents inspect the match environment. This algorithm is shown in Fig. 3.

When an agent reaches that position, it will change its goal to zone making. Since then, the agents will behave to reach the new goal. To make a zone each agent moves to a neighbor node with the following priorities:

1. the node not belonging to our team without any teammate and opponent

```
if (any opponent agent threats) escape to a safer node
if (current energy > 80% of the maximum energy) {
    if (the target is a direct neighbor)
        goto the target node;
    else
        path = Dijkstra(currentPosition, target);
    //path[0] is the first node. path[0] == -1 means no path found.
    if (path[0] == -1) goto a random neighbor;
    if (path[0] != -1 and currentEnergy is enough) goto path[0];
}
```

Fig. 3. algorithm of gathering agents on a specific position

2. the node without any teammate and opponent on it
3. the node without any opponent on it

The used algorithm is illustrated in Fig. 4. Actually, we did not consider the topology of the graph to find the most valuable nodes for making our zones on them. One of the reasons of this decision was that the repairer does not move and other agents must not spread widely on the graph.

In every situation except the threatening one the explorer, inspector and sentinel agents do their special actions after each move (i.e. explorer probes, inspector inspects and sentinel surveys). In our repair strategy, the repairer stops on the specified announced node after reaching to it. Since the two groups (the zone maker and the disturber) may locate far from each other we took this strategy. In such a situation if a repairer became disabled, the process of repairing it would have been too hard. For this reason, the saboteur and repairer agents stop on the saboteur's initial position and did not move until the end of the match. If an opponent agent wanted to attack the repairer, the saboteur agent would have defended it. When an agent becomes disabled it calls a path finding function with its position and the repairer position as its input parameters. This function with having the source node, the destination node and the environmental matrix produces a possible path with the minimum cost by the use of Dijkstra algorithm. Then it tries to reach the repairer and stay there until its health becomes maximum then it leaves the node and continues its working based on its goal. The repairer agent will repair each teammate on its node if its health is not maximum. The most significant advantage of this strategy is that the opponent team cannot disable our entire agents and captures the whole graph because if the repairer agent becomes disabled, it still can repair the saboteur agent on its node. In this way, perhaps in very few steps all the graph can be captured. Another advantage of being in a fixed position for the repairer is that the disabled agent needs to call the path finding function just one time to find the best path. Only in some special situations such as threatening, the disabled agent needs to call the function again because its initial position may be changed because of the escaping. Otherwise, if the repairer moves, the

```
node1 = node2 = node3 = -1;
 for (i=0 to the number of neighbors){
    dangerous = friendsOnIt = inOurZone = false;
    // checking the third priority
    for (j = 0 to the number of visible opponents)
        if (neighbors[i] == visibleOpponents[j].position){
            dangerous = true;
            break;
        }
    if (!dangerous) node3 = i;
    // checking the second priority
    for (j = 0 to the number of visible friends)
        if (neighbors[i] == visibleFriends[j].position){
            friendsOnIt = true;
            break;
        }
    if (!friendsOnIt and !dangerous) node2 = i;
    // checking the first priority
    for (j = 0 to the number of visible vertices)
        if (neighbors[i] == visibleVertices[j] and
            visibleVertices[j].team == ourTeam){
            inOurZone = true;
            break;
        }
    if (!inOurZone and !friendsOnIt and !dangerous) node1 = i;
}
if (node1 != -1) goto node1;
if (node2 != -1) goto node2;
if(node3 != -1) goto node3;
```

Fig. 4. Zone making algorithm

disabled agent have to call path finding function every step to find a path to the repairer position.

Disturber Group: This group consists of one Saboteur agent and one Repairer agent. The reason of choosing these agents is that the saboteur is capable to destroy opponent's and when it becomes disabled, the Repairer can help it. Another reason to choose two agents for this job is that at least two nodes are needed to make a zone [Behrens et al., 2011]. These two agents move in such a way that they can make the zone during the path. The saboteur agent waits until its repairer reach to its position. As soon as the repairer arrives, the disturbing starts. The saboteur agent chooses its next destination node based on a priority. The priorities are as follows:

1. the node belonging to the opponent in which there is an opponent‘s agent
2. the node belonging to the opponent in which there is no opponent‘s agent
3. a random node regarding to its current energy and the weight of the edge

The priority of these movements was adjusted due to their hurting rates.

The algorithm is presented in Fig. 5. If the saboteur agent and an opponent are on the same node, our saboteur will not move until the opponent agent becomes disabled. The saboteur agent after each movement waits for the repairer. The explained strategy is divided into some roles, which are shown by the role model in Fig. 6. The model in Fig. 6 is produced based on MaSE notation. Role definitions are captured in a MaSE [Wood, 2000] role model which includes information on interactions between role tasks and is more complex than traditional role models. Roles are denoted by rectangles. Each role is associated with a number of tasks. Tasks are denoted by ovals attached to the corresponding role. Lines between tasks denote communications protocols with the arrow pointing from the initiator to the respondent. Solid lines indicate external communications [Bergenti et al., 2004].

The tasks are generally derived from the goals for which a task is responsible. For instance, the Zone Aggressor role is responsible for attaining disturbing goal. Therefore, to accomplish this goal, the role must be able to detect the opponent zone and to select the best path. Hence, we have created one task for this role: Disturb.

When an agent, during its movement to a specific node, detects an opponent on the neighbor node escapes to another node that does not contain any opponent. Also, if our agent placed with an opponent on the same node, our agent would have defended itself with parry action in case of having enough energy. These behaviors are emergent on the individual level but others like zone making or disturbing are important in the team cooperation.

If during a simulation the team reaches to a specified level of zone score (e.g. 70), the agents will not move and stick to their positions. In other words, agents try to keep the built zone to reach the zone keeping goal. By this strategy the team can gain this score for some steps. But unfortunately because of

```
node1 =  node2 = -1;
 for (i = 0 to the number of neighbors){
    dangerous = inOpponentZone = false;
    // checking the second priority
    for (j = 0 to the number of visible opponents)
        if (neighbors[i] == visibleOpponents[j].position){
            dangerous = true;
            break;
        }
    if (dangerous) node2 = i;
    // checking the first priority
    for (j = 0 to the number of visible vertices)
        if (neighbors[i] == visibleVertices[j] and
            visibleVertices[j].team == opponentTeam){
            inOpponentZone = true;
            break;
        }
    if (inOpponentZone and dangerous) node1 = i;
}
if (node1 != -1) goto node1;
if (node2 != -1) goto node2;
goto a random node;
```

Fig. 5. Disturbing algorithm

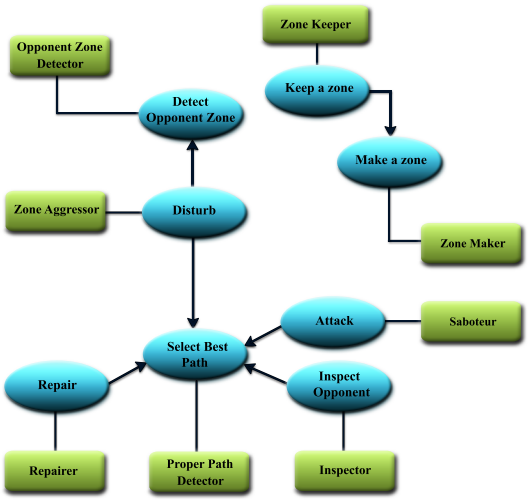


Fig. 6. The role model.

not having a perfect and powerful defensive strategy we had two problems. First one, we could not defend our zones and a few steps after building a zone we lost it immediately because of opponents disturbing and the second, we could not make big zones with a lot of nodes. Since during the zone expansion, again the opponents disturbed the zone and broke it down completely.

In the case of our buy strategy, the disturber group agents buy energy. The saboteur of this group also increases its disturbing power. But we limited these purchases to get the scores of achievements in the remaining steps of the simulation.

For all our routing issues such as gathering in an initial position or moving toward the repairer we used Dijkstra algorithm.

Because of the decentralized development of the team, we did not have a hierarchical structure or a leader agent. We only had a temporary implicit hierarchical mood in the early steps of the simulation. In the early steps of the match, the two saboteur agents behave like a leader to gather the other agents to make the zone maker and the disturber groups. Since then, there is no hierarchical structure and leader, instead each agent will do its role autonomously and will corporate with other agents to reach their common goals.

5 Conclusion

By attending in this year competitions we became familiar with the structure of multi-agent systems and programs. We got to know the people who are active in this field. Also, we learned how to design and implement autonomous agents. These competitions also provided an opportunity for us to learn agent based methodologies and work with special tools to develop multi-agent systems. Among these the MaSE methodology and the JADE platform had the most impact.

In the case of the methodology, the tools and the used algorithms to implement our system the only problem was the low runtime speed of the JADE platform.

Such these competitions provide an appropriate field for multi-agent programming [Behrens et al., 2010]. To improve this field and make it more interesting it is better to use software and hardware beside each other as an agent instead of just software agent because software agents cannot illustrate real world constraints and problems. Of course this idea forces a geographical constraint for setting up the contest. Since real environments are not free from danger for multi-agent systems it is better consider them in the contest to make it more real and very close to the actual world. For example, in Mars scenario, it will be more interesting if some nodes are being considered as holes or pits so that if an agent moves to that node it will be omitted and disappeared from the match. Also, placing some aliens in the environment to bother both teams can make it more attractive. Team agents should have the power to kill and learn them. For example an instructor role can considered

to learn an alien and make it as a teammate. Also, it will be more interested if the buying and increasing the physical properties of an agent be reasonable (i.e. a level for these items is being considered). Finally, it will be more challengeable if the environment be more complex (e.g. by increasing the number of agents), so each team has to distribute on different machines and should consider high performance algorithms.

The strong points of our team were preventing the opponent from occupying the whole graph and attacking the opponent agents that could be seen obviously during the matches. However, inability to make the big zones and defend them due to the lack of a powerful defensive mechanism, was our team weakness point. Since this year was our first experience to attend in such competitions and we were not enough familiar with multi-agent programming and agent based systems we could not build a perfect system. On the other hand not having enough time and human resources to build a perfect program caused that we did not get good results in this year competitions.

An important point about our team strategy in gathering agents was that, during their movement to the specified node in the case of facing an opponent in the neighbor nodes they changed their path by escaping, so they might become far from the target instead of closing to it. This matter especially is too crucial for the repairer agent of the zone maker group. During the matches this case bothered our team too much. Thus, we found that if the repairers did the grouping task instead of the saboteurs, it would have been very better because in this case the zone maker repairer would have stayed in its initial position and the saboteur like other agents would have reach to the repairer without any serious problem.

But for the next year tournament we are going to gather enough specialists, use from powerful tools and languages and spend more time to implement better strategies.

References

[nar,] Website. `http://www.haftaseman.ir/webdb/article.asp?id=851`.

[Behrens et al., 2010] Behrens, T., Dastani, M., Dix, J., Köster, M., and Nov'ak, P. (2010). The multi-agent programming contest from 2005-2010: From collecting gold to herding cows. *Annals of Mathematics and Artificial Intelligence*, 59:277–311.

[Behrens et al., 2011] Behrens, T., Koster, M., Schlesinger, F., Dix, J., and Hubner, J. (2011). Multi-agent programming contest scenario description. 2-4.

[Bellifemine et al., 2007] Bellifemine, F., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. John Wiley and Sons Ltd.

[Bergenti et al., 2004] Bergenti, F., Gleizes, M.-P., and Zambonelli, F. (2004). *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Kluwer Academic. 107-125.

[Corman et al., 2001] Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. McGraw Hill Book Company, second edition. 495-501.

[Rafe et al., 2011] Rafe, V., Nikanjam, A., and Rezaei, M. (2011). Galoan: A multi-agent approach to herd cows. *Annals of Mathematics and Artificial Intelligence, (AMAI), Springer*, 61(4):333–348.

[Wood, 2000] Wood, M. F. (2000). *Multiagent systems engineering: A methodology for analysis and design of multiagent systems*. Master's Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Ohio, USA.

Short Answers

1.1 The motivation to participate in this contest was about doing a project for advanced software engineering course.
1.2 The team was working on a software engineering project.
1.3 Nargel.
1.4 Our team consists of two designers, one developer and one supervisor. All team members except the supervisor are MSc students.
1.5 Our field of research is Software Engineering.
2.1 The MaSE (Multi-agent Systems Engineering) methodology was being used to analyze and design our system.
2.2 We developed a decentralized system in which agents behaved autonomously, kept the environmental information and tried in cooperation with each other to reach a common goal.
2.3 Agent communications was done via agent speaking feature which is provided by the JADE platform. JADE handles sending and receiving messages between agents and we didn't deal with its details.
2.4 Our agents are autonomous because they can act without any interfere from a leader and are reactive such that they can react upon the opponent's actions and also they have proactiveness because they can predicts some situations and acts in time.
2.5 Yes, It is a truly multi-agent system.
2.6 Our team spent about 500 hours to design and develop Nargel system.
2.7 No we did not consult with anyone out of the team. We had tested Nargel system during some matches with HempelsSofa, Sorena and two dummy systems provided by the organizers and participated in all the test matches set up by the organizers.
3.1 Java.
3.2 No, because we were not familiar with the multi-agent programming languages and we did not have enough time to learn them.
3.3 We implemented agent oriented concepts by the JADE facilities in Java.
3.4 We did not use any.
3.5 We used the JADE runtime platform and about 50 hours were spent to learn how to take advantage of the JADE capabilities and features in our system.
3.6 There was nothing missed.
3.7 The JADE framework is thoroughly implemented in Java, this helped us to use the JADE facilities in Java without any problem.
3.8 Dijkstra.
3.9 We ran all agents on a single machine because the system workload was low.
3.10 In each step an agent decides and defines its action according to its previous knowledge and current percepts. So the agent reasoning is completely synchronized with receiving percepts/sending actions cycle.

3.11 One was to handle sending actions and receiving percepts. To solve this, we used the related codes from dummy agents. The other was implementing the zone making strategy which we used a simple approach for that.
3.12 We programmed Nargel system in about 3000 lines which are for both implementing strategies and agent communication.
4.1 Our team strategy is to consider both making zones and disturbing opponent zones. Actually, we worked on both contexts.
4.2 For keeping information, each individual agent keeps the necessary information in some data structures. Defining goals and informing all agents about them is being done by agent speaking mechanism.
4.3 Our agents did not analyze the topology of the map.
4.4 Communication with server is done by the use of eismassim package.
4.5 In every situation except the threatening one the explorer, inspector and sentinel agents do their special actions after each movement.
4.6 Our agents did not estimate the value of the zones.
4.7 We did not conquer any zones and also we did not defend our zones against opponent attacks, instead, we attacked the opponent zones.
4.8 No they cannot.
4.9 We used Dijkstra algorithm for all our routing issues.
4.10 The disturber group agents buy energy. The saboteur of this group also increases its disturbing power.
4.11 We limited the purchases to get the scores of achievements in the remaining steps of the simulation.
4.12 No.
4.13 A few messages with simple structure were being used to exchange information and cooperate. This messaging was done by JADE.
4.14 There is no hierarchical structure and leader, instead each agent will do its role autonomously and will corporate with other agents to reach their common goals.
4.15 Behaviors like parrying and escaping are emergent on the individual level but others like zone making or disturbing are for team benefits.
4.16 At the first of the match a position is being defined from which zone making begins. Thus a planning is done for the entire match that is the area of the map where the team wants to make zones is being specified.
5.1 We knew the structure of multi-agent systems and learned how to make a computer program autonomous and also got familiar with agent based methodologies and special tools to develop multi-agent systems.
5.2 The strength points of our team were preventing the opponent from occupying the whole graph and attacking the opponent agents. However inability to make big zones and defend them was our team weakness point.
5.3 In the case of methodology, tools and used algorithms to implement our system the only problem was the low runtime speed of the JADE platform.
5.4 We didn't saw any defect in this year contest.

5.5 Because this year was our first experience and on the other hand not having enough time and human resource to build a perfect program caused that we didn't get good result in this year competitions.

5.6 To improve this field and make it more interesting it is better to use software and hardware beside each other as an agent instead of just software agent.

5.7 Some nodes can be considered as holes so that if an agent moves to that node it will be disappeared. Some aliens can be placed in the environment to bother both teams. Teams should have the power to kill and learn them.

Implementing a Multi-Agent System in Python

Mikko Berggren Ettienne, Steen Vester, and Jørgen Villadsen*

Department of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, Building 321, DK-2800 Kongens Lyngby, Denmark

Abstract. We describe the solution used by the Python-DTU team in the Multi-Agent Programming Contest 2011, where the scenario was called Agents on Mars. We present our auction-based agreement, area controlling and pathfinding algorithms and discuss our chosen strategy and our choice of technology used for implementing the system. Finally, we present an analysis of the results of the competition as well as propose areas of improvement.

1 Introduction

This paper documents our solution to the Multi-Agent Programming Contest 2011 (MAPC) [Behrens et al.,2011,Behrens et al.,2010] as the Python-DTU team.

The aim of MAPC is to stimulate research in the area of Multi-Agent Systems (MAS). It is a returning competition which has been held every year since 2005. The challenge is to solve a cooperative task in a dynamic environment using a multi-agent system. This year's MAPC presented a new scenario called Agents on Mars in which two opposing teams control 10 agents and compete to control "zones" of a graph in a discrete time world.

This year's contest was the 7th edition of MAPC. Every year participants have stated their implementation language/framework and submitted their source code along with a short report describing their solution. In 2005 MAPC was built on a "Food-Gatherers" scenario, 2006-2007 presented a "Goldminers" scenario and 2008-2010 presented a "Cows and Cowboys" scenario. This year again presented a new scenario "Agents on Mars" making it unfeasible to build a solution from earlier year's implementations. Throughout the years many participants have used existing MAS frameworks, in particular Jason [Bordini et al.,2007] and JIAC [Hirsch et al.,2009] which are both open source and implemented in Java, while other participants have implemented their own MAS frameworks. Naturally MAS frameworks can be reused in different scenarios and framework experiences from earlier years are worth considering. We participated in the contest in 2009 and 2010 as the Jason-DTU

* Corresponding author: jv@imm.dtu.dk

team since we used the Jason platform and its agent-oriented programming language AgentSpeak [Boss et al.,2010,Vester et al.,2011]. We performed well but for the contest this year, with the new and more complex scenario, we decided to focus on an auction-based agreement approach and to implement the multi-agent system in the programming language Python.

Our observation from this and previous years is that because of the complex nature of the scenarios, choosing which strategies to apply poses the greatest challenge. Compared to that, the actual requirements for a supporting framework are not overwhelming which led us to implement our own framework. While Jason had some immediate benefits, e.g. with regards to agent communication, we regularly encountered problems where we would have preferred to have complete control over every aspect of the implementation. Thus our decision to implement our own framework for this year's competition was evident. We chose Python as we think it is in many ways superior with respect to development speed and succinctness compared to Java, C#, C++ and other languages that we have experience with. Furthermore Python supports multiple programming paradigms, including the functional, which proved quite effective for this setting. This is also confirmed by the final implementation which takes up very few lines of code compared to earlier years and yet proved to be very effective.

We used approximately 400 man hours in total for implementing the system and for participating in the official test matches. We discussed agent designs and strategies with other teams during the competitions only.

2 Supplementary Competition Description

The environment can conveniently be represented as a graph with each water well being a vertex and an edge $e = (u, v)$ between any two connected wells u and v. Furthermore we can describe the entities of a game as follows:

- A - The set of agents in the game where $|A| = 20$.
- T - The set of team names where $|T| = 2$.
- $G = (V, E)$ - The graph of the game.

Note that each team always consists of 10 agents. A game is then described by the static mappings:

- $team : A \to T$, mapping each agent to a team.
- $weight : E \to \{1, 2, \ldots, 10\}$, assigning weights to all edges.
- $value : V \to \{1, 2, \ldots, 10\}$, assigning values to all vertices.

From here it follows that every external state of a game is given by the additional mappings (the internal states of the agents are not considered in an external state):

- $pos : A \to V$, mapping all agents to a vertex in the game graph.
- $score : T \to \mathbb{N}$, mapping each team to their current score.

- $money : T \to \mathbb{N}$, mapping each team to their current amount of money.
- $zones : T \to \mathbb{N}$, mapping each team to their current zone score.

An agent's role determines which actions it is allowed to perform, its maximum health and energy, its strength and its visibility range. The five possible roles are defined in table 1.

Explorer	Actions:	skip, goto, probe, survey, buy, recharge
	Maximum energy:	24
	Maximum health:	4
	Strength:	0
	Visibility range:	2
Repairer	Actions:	skip, goto, parry, survey, buy, repair, recharge
	Maximum energy:	16
	Maximum health:	6
	Strength:	0
	Visibility range:	1
Saboteur	Actions:	skip, goto, parry, survey, buy, attack, recharge
	Maximum energy:	14
	Maximum health:	3
	Strength:	3
	Visibility range:	1
Sentinel	Actions:	skip, goto, parry, survey, buy, recharge
	Maximum energy:	20
	Maximum health:	1
	Strength:	0
	Visibility range:	3
Inspector	Actions:	skip, goto, inspect, survey, buy, recharge
	Maximum energy:	16
	Maximum health:	6
	Strength:	0
	Visibility range:	1

Table 1. Roles

An agent can increase its maximum energy, maximum health, strength and visibility range with the buy action. Thus we can use the following mappings to describe the internal state of the game, i.e. the state of the agents:

- $mh : A \to \mathbb{N}$, mapping every agent to its current maximum health.
- $h : A \to \mathbb{N}$, mapping every agent to its current health.

- $me : A \to \mathbb{N}$, mapping every agent to its current maximum energy.
- $e : A \to \mathbb{N}$, mapping every agent to its current energy.
- $s : A \to \mathbb{N}$, mapping every agent to its current strength.
- $vr : A \to \mathbb{N}$, mapping every agent to its current visibility range.

And the constant mappings:

- $role : A \to \{Repairer, Saboteur, Explorer, Sentinel, Inspector\}$, mapping each agent to a role.
- $actions : \{Repairer, Saboteur, \ldots\} \to 2^{\{skip, goto, \ldots\}}$, mapping each role to a set of actions.

Furthermore we have $\forall a(a \in A \to (mh(a) \geq h(a) \geq 0 \wedge me(a) \geq e(a) \geq 0))$. In each simulation step an agent can perform one action. In general there is a probability of 1 percent that an action will fail. The definitions of all actions are given in table 2. A failed action has no effect, but will still decrease the energy of an agent as specified in table 2.

2.1 Scoring

The winner of a game is the team with the highest score when the game ends. The score for a team t is computed as:

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

Where steps is the number of simulation steps.

Money When a team reaches an achievement, its amount of money is increased by 2. The achievements are distributed in different categories and money will only be given the first time an achievement is reached. For a team t the achievements are defined as follows:

- `Zone`: $zones(t) \geq 10, zones(t) \geq 20, zones(t) \geq 40, \ldots$
- `Probing`: Probing i unprobed vertices for $i \geq 5, i \geq 10, i \geq 20, \ldots$
- `Surveying`: Surveying i unsurveyed edges for $i = 10, i \geq 20, i \geq 40, \ldots$
- `Inspecting`: Inspecting i uninspected* opponents for $i \geq 5, i = 10$
- `Attacking`: Attacking i opponents for $i \geq 5, i \geq 10, i \geq 20, \ldots$
- `Parrying`: Parrying i opponent attacks for $i \geq 5, i \geq 10, i \geq 20, \ldots$

Note that attacking or parrying the same opponent multiple times count towards new achievements whereas this is not the case for probes, surveys and inspects.

* However inspecting an already inspected agent a will count towards an inspect achievement if either of $s(a), mh(a), me(a), vr(a)$ has changed since the previous inspection of a.

For every agent $a \in A$

skip This action will never fail and has no effect.

recharge This action increases $e(a)$ by 50% if $h(a) > 0$ and by 30% if $h(a) = 0$.
The action will fail in step s if a is attacked in step s.

attack a must specify the name of an agent b to attack.
The action requires $pos(a) = pos(b) \wedge h(a) > 0 \wedge role(a) = Saboteur \wedge e(a) \geq 2$
and will decrease $e(a)$ by 2. If b performs a parry action in the same step
the attack action fails. Otherwise if $h(b) \geq s(a)$, $h(b)$ is decreased by $s(a)$
else $h(b)$, is decreased to 0.

goto a must specify the vertex v it wants to go to.
The action requires $(pos(a), v) \in E \wedge e(a) \geq weight((pos(a), v))$
and will decrease $e(a)$ by $weight((pos(a), v))$.

probe The values $value(v)$, $v \in V$ are initially unknown to a.
This action will reveal $value(pos(a))$ to a.
The action requires $role(a) = Explorer \wedge e(a) \geq 1$ and will decrease $e(a)$ by 1.
The action will fail in step s if a is attacked in step s.

survey The weights $weight(edge)$, $edge \in E$ are initially unknown to a.
This action will reveal $weight((s,t))$ where $(s,t) \in E$ for all edges (s,t) where both s and t
are reachable from $pos(a)$ in $vr(a)$ or less steps.
I.e. let $k = vr(a)$, then there exists vertices $pos(a) = v_0, v_1, \ldots, v_i = s$
such that $(v_{j-1}, v_j) \in E$ for $1 \leq j \leq i$ and $i \leq k$.
and there exists vertices $pos(a) = u_0, u_1, \ldots, u_i = t$
such that $(u_{j-1}, u_j) \in E$ for $1 \leq j \leq i$ and $i \leq k$.
The action requires $h(a) > 0 \wedge e(a) \geq 1$ and will decrease $e(a)$ by 1.
Again the action will fail in step s if a is attacked in step s.

inspect This action will inspect the internals of all opponents b if $pos(b)$ is reachable
from $pos(a)$ in one or less steps, i.e. $(pos(a), pos(b)) \in E$ or $pos(a) = pos(b)$.
The internals are the values given by the mappings describing the state of b.
The action requires $role(a) = Inspector \wedge team(a) \neq team(b) \wedge h(a) > 0 \wedge e(a) \geq 2$
and will decrease $e(a)$ by 2. Again the action will fail in step s if a is attacked in step s.

repair a must specify the name of an agent b to repair.
The action requires $role(a) = Repairer \wedge a \neq b \wedge e(a) \geq 2$
and will decrease $e(a)$ by 2 and increase $h(b)$ to $mh(b)$

buy The agent must specify either max. health, max. energy, strength or visibility range
and the chosen parameter will be increased for agent a by 1.
The action requires $h(a) > 0 \wedge e(a) \geq 2 \wedge money(team(a)) > 2$
and will decrease $e(a)$ and $money(team(a))$ by 2.
Also, to buy strength it is required that $role(a) = Saboteur$.
Again the action will fail in step s if a is attacked in step s.

Table 2. Actions

Zones A $zone$ is a connected subgraph with at least two nodes whose vertices are colored according to section 2.2. The value of a zone $Z \subseteq V$ occupied by a team $t \in T$ is given as $\sum_{v \in z} z\_val(v)$ where $z\_val(v) = 1$ if v has not been probed by an agent of team t and $z\_val(v) = value(v)$ if v has been probed by an agent of team t. $zones_s$ for a team t is the sum of all zone values at step s for team t. Figure 1 shows an example of zone scoring.

2.2 Graph Coloring Algorithm

In this section the rules for coloring nodes of the graph will be presented. An algorithm is run every time step to determine which team owns which parts of the graph. The algorithm runs in three steps as follows:

1. For every vertex v, if team t_1 has more agents standing on the vertex than team t_2 then t_1 owns v.
2. For every vertex v not owned by a team after the previous step, if team t_1 owns more neighbor vertices of v than team t_2 after step 1 then t_1 owns v. For this rule to apply however, t_1 must own at least two neighbor vertices of v after step 1.
3. For every vertex v not owned by a team after the previous steps, if team t_1 owns a node on all paths π from v to an agent of team t_2 after step 1 and 2, then t_1 owns v.

In figure 2 we give an example of an application of each rule. Rule 3 is the most complicated rule to understand, but it can be interpreted by considering the vertices owned after step 1 and 2 as a frontier of vertices. If there are any vertices encapsulated within that frontier and no opponent agents are, then the vertices encapsulated will also be owned by the team having the frontier. The rules for coloring are very important to understand completely since the team that is consistently best at obtaining zones of nodes with the highest value will often be the winner of a match. In our strategy we focus only on rule 1 and 2, since rule 3 is quite hard to use in practice, especially when opponents are moving all the time. One of the most important applications of rule 3 is that if all agents of one team are disabled, then the other team will own all vertices in the graph since disabled agents cannot be used for controlling areas. Thus, it is very expensive if one team is completely disabled even if it is only for a few steps.

3 System Analysis and Design

We did not use any multi-agent system methodology because we preferred to have complete knowledge and control of every part of the implementation. We chose a decentralized solution where agents shared percepts through shared data structures and coordinated actions using distributed algorithms. Our agreement based auction algorithm heavily relies on communication and is

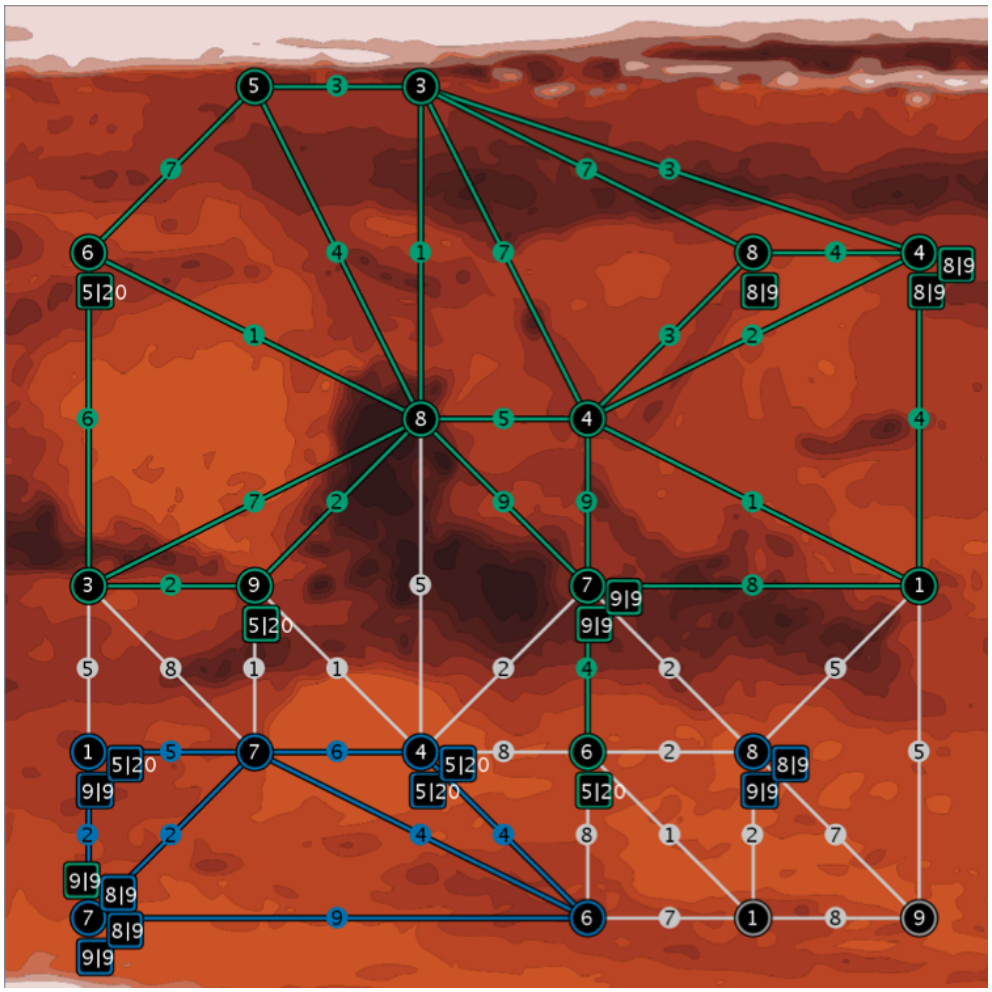


Fig. 1. An scenario showing zone scoring, reprinted from [Behrens et al.,2011]. Vertex values and edge weights are depicted directly on vertices and edges respectively. The squares located around vertices are agents of team $green$ or team $blue$. Accordingly the green team controls the green zone and the blue team controls the blue zone. Assuming that the blue team has probed all vertices in their zone, the value of their zone is 25.

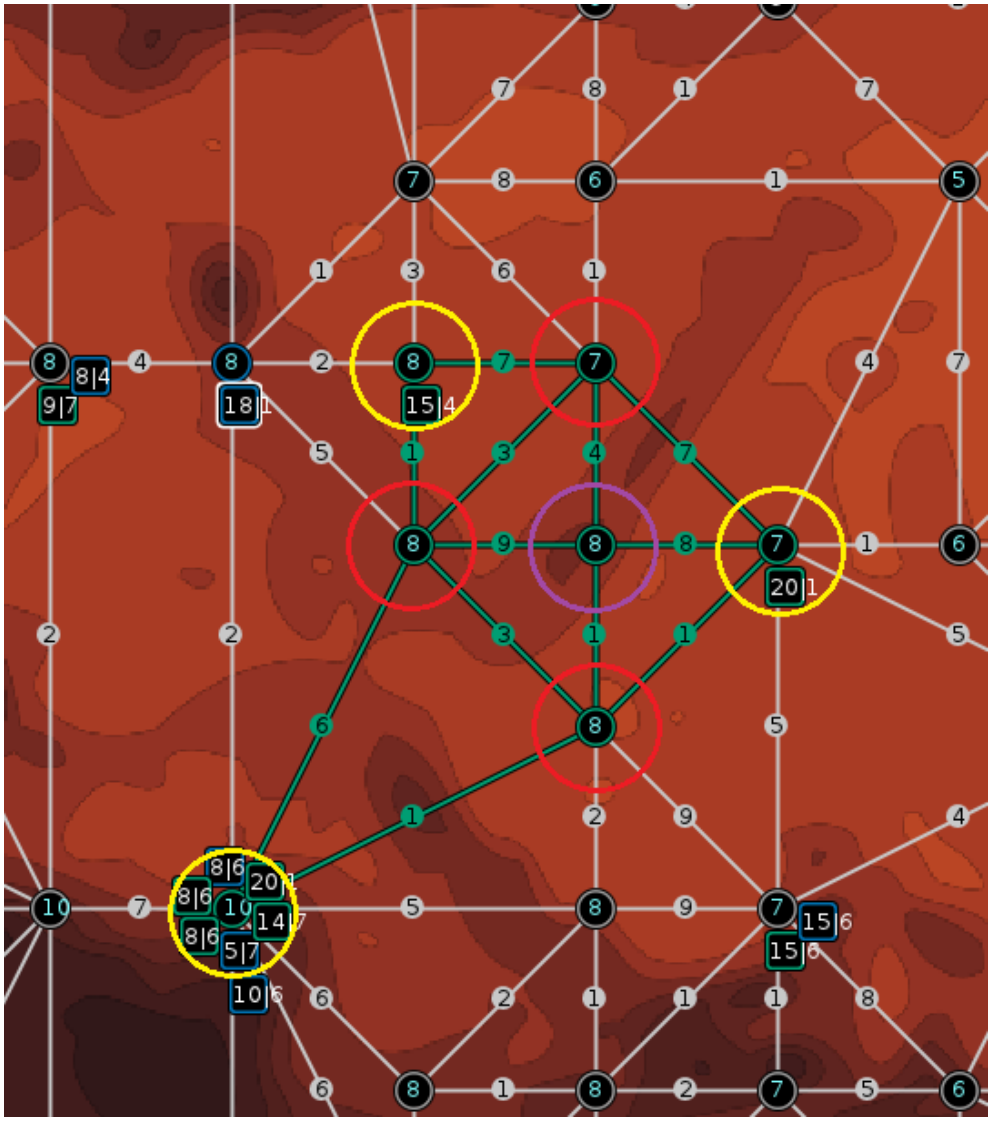


Fig. 2. A scenario showing applications of the three rules of coloring. The vertices marked with a yellow circle are colored according to rule 1, the vertices marked with a red circle are colored according to rule 2 and the vertex marked with a purple circle is colored according to rule 3

part of how agents decide on goals. Each agent acts on its own behalf based on its local view of the world.

In the following we describe our decentralized solution to agent cooperation using a distributed auction algorithm.

3.1 The Agreement Problem

Many situations arise where a subset of our agents must cooperate to solve a task. For example, we use most of our agents to survey the graph in the beginning of every match. To do this our agents need to agree on who surveys which parts of the graph. In the same way our saboteurs have to agree on which opponents to attack and our repairers must agree on which of our agents to repair. Some agents might also be more suitable for a goal than others (because of special abilities, shorter distance to goal, etc.). We would like to assign agents in such a way that as many goals as possible are accomplished in as little time as possible, since accomplishing goals quickly gives a higher score. Before assigning goals to agents we start by assigning *benefits* to each goal for each agent, such that the benefit of a goal is high if the goal is important to solve and such that the benefit will be higher the faster the agent can accomplish it (e.g. the shorter the distance from the agent to the goal is, taking the energy of the agent into account). We would then hope to achieve the following properties when designing an algorithm to assign goals to agents:

1. The total benefit of the assigned goals should be as high as possible. Preferably optimal or close to it.
2. The running time of the algorithm should be fast, since we need to assign goals to agents at every time step in the competition and still have time left for other things such as environment perception, information sharing, etc.
3. The algorithm should be distributed between the agents resembling a true multi-agent system.
4. It should not be necessary for the agents to have the same beliefs about the state of the world in order to agree on an assignment.
5. The algorithm should be robust. If it is possible, our agents should be able to agree on an assignment even if some agents break down or some communication channels are broken.

The algorithm described in the following is inspired by [Zavlanos et al.,2008,Bertsekas et al.,1991] and achieves this with some compromises while still satisfying every point to some extent.

3.2 Auction Algorithm

To solve the problem we use an auction algorithm in which agents will make bids against each other on the goals that they would like to pursue. The rules

of the auction will be designed so we can be sure that the algorithm will terminate in a finite number of rounds and that all agents are assigned to different goals at termination. Also, the assignment will be near optimal in a sense that is defined later in this section.

We assume that there are n agents and at least n different goals, such that there always exists a feasible assignment of a distinct goal to each agent. It is also assumed that the agents are using a network of communication channels where all pairs of agents are not necessarily connected at all times. Though we do assume that the graph of the communication network is connected at all times. When designing an auction each goal i will at a given time t have a *price* denoted $p_i(t)$. Initially, we let $p_i(0) = 0$ for all goals i. The price of a goal will be the highest bid made by an agent on that goal (except when the price is 0). As in a real world auction, agents will now place bids on the goals which give them the largest *net value*. Here we define the net value of agent i for goal j by

$$\textit{net value}_{ij} = \textit{benefit}_{ij} - p_j$$

which is the benefit the agent will get from the goal minus the price of the goal.

In each bidding round, agents place bids according to their local information about the current prices. If an agent has not currently placed the highest bid on any goal then the agent will place a bid on the goal which maximizes his current net value according to his current knowledge of the prices of the goals. The highest bid as well as local information about the current prices and current highest bidders of goals are in each round sent to all neighbour agents, i.e. agents to which a communication channel exists. In addition local beliefs are updated according to the prices received from neighbour agents. If several agents have made the same bid for a goal, the agent with the highest index will win the goal. The updated values are then used by the agents to calculate the *net values* for the next bidding round. Thus, in one bidding round at time step t the algorithm works by letting each agent i do the following:

1. Receive the newest prices and owners of all goals. Update the local belief base if there are higher bids on any of the goals that the agent did not already know about. This includes updating the *net values* of the goals. Also, the agent may have lost a goal it owned in the previous round.
2. If the agent is not currently the owner of a goal, it will place a bid on the goal j with the highest *net value* according to its belief base. It does so by setting itself as owner of j and increasing p_j by $\gamma = v_i - w_i + \varepsilon$ where v_i is the net value of j and w_i is the net value of the goal with the second highest net value. ε is a positive number which is a parameter of the algorithm that influences the running time and the quality of the final assignment. Generally speaking, a low value of ε gives high quality assignments but longer running time.

An example run is shown in figure 3 where $\varepsilon = 1$ and goal benefits are integers. This was also the case when we used the algorithm in practice which gave us a very short running time. In practice we simulated a complete communication network topology by using a shared database of bids between the agents.

In general the algorithm terminates when n rounds without new bids occurs, in which case all agents have an assigned goal. In our case with a complete communication network, it can terminate as soon as one round without new bids occurs assuming that no communication channels are broken. For a proof that the algorithm does in fact terminate no matter what choice of $\varepsilon > 0$ and no matter the choice of structure of the connected communication network we refer to [Zavlanos et al.,2008]. Here it is also proven that the algorithm terminates in $O(\Delta n^2 \lceil \frac{max_{i,j}\{benefit_{ij}\} - min_{i,j}\{benefit_{ij}\}}{\varepsilon} \rceil)$ and that the final assignment obtained by the algorithm is within $n\varepsilon$ of being optimal. Here n is the number of participants and $\Delta \leq n - 1$ is the maximum network diameter, which is the longest distance between two vertices in the communication network, which is practically reduced to 1 when using a shared data structure as we did in practice.

This choice of algorithm gives quite good solutions with respect to maximizing total benefit, the running time was no problem during competition and the computation is completely distributed between the agents. The agents do not need to share beliefs about the world state, but only need to use their local belief base to approximate their own benefits for different goals. Finally, the algorithm will work even if some communication channels break down which should make the solution more robust than a centralized approach in some environments.

4 Software Architecture

The competition is built on the Java MASSim-platform and the Java EISMASSim framework is distributed with the competition files. This framework is based on EIS [Behrens et al.,2009] and abstracts the communication between the server and the agents to simple Java method-calls and call-backs. To utilize this framework we started out with the Java implementation of Python called Jython which in contrast to Python can import Java libraries and classes.

A true multi-agent system allowing distribution of the agents was not enforced by the competition rules. However we took up this challenge as it posed some interesting distribution problems as seen in section 3.1. To support agent communication in our multi-agent system we started out using the Apache ActiveMQ as a messaging server which offers clients for all popular programming languages.

Using the EISMASSim Java framework together with ActiveMQ clients written in Python and glueing it all together with Jython gave some performance issues when exchanging percepts between the agents. We found that each component performed well when tested in a controlled context

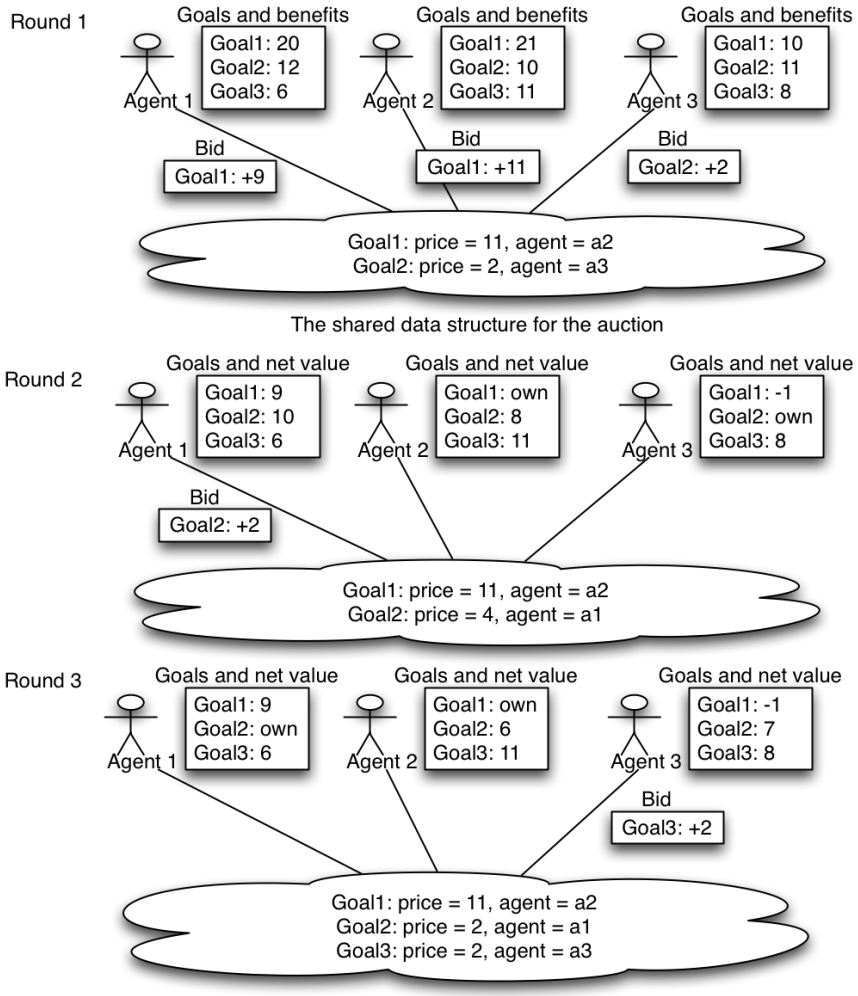


Fig. 3. An example of an auction between three agents. In round 1 the auction data structure is empty before the bidding and thus goal benefits are equal to goal net values. Each agent places a bid on its preferred goal. The bids are calculated as the difference between the two best goals plus ϵ. The data structure stores the highest bid and the corresponding agent for each goal. Both agent 1 and agent 2 bid on goal 1 and agent 1's bid is discarded as it is lower than agent 2's bid. In round 2 the net values have changed as the new bids in the shared data structure are considered. Agent 2 and 3 have not been overbidden, so they won't bid in this round and does now consider themselves owners of the goals they bid on in round 1. However agent 1 overbids agent 3 on goal 2 as the shared data structure shows. Now in round 3 agent 1 has become the owner of goal 2 and agent 3 bids on goal 3 as this is the best goal for it considering the latest bids in the data structure. Now all agents are assigned a goal and the auction ends. We see in this case that we do not get the optimal solution (agent 1 = goal 1, agent 2 = goal 3, agent 4 = goal 2, total benefit = 42) but instead a solution very close to the optimal (total benefit = 41).

and thus the issues were with the interaction between the components. We decided to skip Jython, ActiveMQ and EISMASSim and to instead follow a much cleaner Python-only implementation. Even though some work was needed to implement the protocol specific parts which EISMASSim handled, this left us with a more flexible implementation of which we had complete knowledge and control of every part and relieved us from most of the performance issues.

We did not have time to implement our own messaging server with a simple text-based protocol but instead choose to use a set of shared data structures for agent communication. This ensured great performance and was possible because distributing the agents on different computers was not necessary.

4.1 Modeling the Environment

Each agent keeps an internal model of the environment. The environment is trivially modeled as a graph using simple data structures and classes. Every agent is responsible for parsing the messages it receives from the competition server and updating its model accordingly. The agents are also responsible for sending new percepts to the other agents of the team.

Agent positions are represented by a two way mapping allowing retrieval of all agents at a given position, or the position of a given agent. Due to the non-static nature of the agents and their limited visibility range, agents must be removed from this mapping when moving out of the visible area for an agent to avoid inconsistency between the "true" world as represented by the server and the internal world of the agent. The agents also share their percepts of other agents and their own position. However it does happen that an enemy agent moves out of vision for all team agents. In this case we keep the agent in the position mapping allowing inconsistency until it becomes visible again and the mapping is updated. An agent can then use this inconsistent information if it needs to locate the "disappeared" agent. Thus instead of randomly searching in non-visible areas, it can start searching in the area where the agent disappeared.

4.2 Goal Searching

A goal is contained in the abstract type we call Action which is not only used for the agreement auction itself, but also for the agents to carry out the necessary steps to achieve a goal when they have won it in the auction. In our implementation, satisfying a goal implies reaching a specific vertex and then performing some action (in some cases the skip action). The Action type is defined as follows:

```
1 class Action():
2     def __init__(self, goal, type, vertex=None, cost=0, path=[], arg='', length = 0):
3         self.goal = goal
4         self.type = type
5         self.arg = arg
```

```
6           self.vertex = vertex
7           self.cost = cost
8           self.path = path
9           self.length = length
```

Here *goal* is the name used to distinguish goals during the auction, *type* is the action the agent must perform when reaching the vertex given in *vertex* and *arg* is a possible argument for the action to be executed. *cost* is subtracted from a constant and the result is used as the goal benefit needed for the auction. The *path* contains a list of vertices the agent must follow to reach the vertex given in *vertex* and *length* is simply the length of this path.

The code below is part of the *get_goals* method which returns a set of Actions. It shows how a saboteur performs a goal search before participating in an agreement auction. The RUNTIME parameter is a constant determining when the team strategy changes from *achievement* to *area controlling* as further described in section 5.

If the current agent is a saboteur and attacking is enabled we check if the team is currently on the *area controlling* strategy in line 5. If this is the case, the agent's knowledge of the currently controlled zone is updated in line 6. Line 7 is a custom best-first search returning a set of Actions. The first parameter is a string to which an opponent name will be appended. This will be given as the goal name to the Action constructor while the second parameter is given as the type. The rest of the Action constructor parameters are given directly from within the search. The third parameter indicates where to start the search from, namely the position of the agent. The fourth parameter is a pointer to a function that determines which vertices are valid goals and which are not. In this case a valid goal is a vertex in the currently controlled zone with a non-disabled opponent agent placed on it. The last parameter indicates that the search can stop when 2 valid goals are found, because only the two saboteurs will bid on these goals. The for-loop on lines 8-9 lowers the found Actions' cost value equally by a constant so great that they will always be chosen over other possible Actions. If less than 2 Actions are found, other Actions are needed, as the agent must be sure to win the auction for at least one goal. To find additional goals another similar search is run, however this time the fourth parameter is a new function pointer. This function will validate every vertex that is not in the currently controlled zone and has a non-disabled agent placed on it.

If the current phase is instead the *achievement* phase, the code jumps from line 5 to line 12 and in this search the currently controlled zone is not computed and thus no vertices will be ignored by the validating function.

```
1  if
2  ...
3  elif self.type == SAB:
4          if DO_ATTACK:
5              if self.runtime > RUNTIME:
6                  self.get_expand()
7                  goals = self.bfs('attack_owned_', ATTACK, start, self.get_opponent_in_owned, 2)
8                  for g in goals:
9                      g.cost = g.cost − 100
10                 if len(goals) < 2:
```

```
11                      goals.extend(self.bfs('attack_', ATTACK, start, self.get_opponent, 2))
12              else:
13                  goals = self.bfs('attack_', ATTACK, start, self.get_opponent, 2)
14          else:
15              goals = self.bfs('survey_', SURVEY, start, self.is_unsurveyed, 10)
```

It might still be the case that less than two Actions have been found. This is handled at a later point in the *get_goals* function. Actions helping area controlling are added if the strategy is *area controlling* and survey actions are added if the strategy is *achievement*. Note that the agent must then have at least 10 different goals as it now possibly auctions against all the other 9 agents. If the agent still has not found enough Actions, ignore actions with low benefit will be added to the Action set until sufficient actions are available.

5 Strategies, Details and Statistics

Considering the complexity of the environment in combination with the nondeterminism introduced by the opposing team's agents, it is clear that classical planning approaches will not suffice for this scenario. We instead let the agents implement a greedy top-level strategy by calculating prioritized sets of goals at each simulation-step. The goals depend on the agent's role, the state of the agent's internal world and how far the simulation has progressed as described in section 4.2. The strategy is greedy as agents does not consider subsequent goals when deciding on a set of goals.

5.1 General Strategy

In the Agents on Mars scenario there are two main ways to earn points. The first is achievement points which are given to a team if they achieve some goals cf. [Behrens et al.,2011]. 2 points were rewarded for reaching an achievement. This means that the team will get 2 points every step for the remainder of the match, unless the points are used to buy special abilities for the agents. It follows that if we are interested in making achievements it makes most sense to do so as early as possible in a match, since this will give us points in every time step for the rest of the match. Another interesting thing is that the number actions required to get achievements in each area increases exponentially. For example, one gets 2 achievement points when the team has done 5 successful attacks, then another 2 after 10 successful attacks, then 20 successful attacks, then 40 successful attacks, then 80 successful attacks, etc. This means that if we are to maximize our earning from achievements, it is probably a good idea to be versatile and good at all the different kinds of achievements. For example, after 160 attacks it will be much easier to survey 5 edges than attack opponents another 160 times (though attacking opponents gives other desirable benefits as well).

The other way to earn points is by controlling areas which gives as many points as the controlled areas are worth every time step cf. [Behrens et al.,2011].

But before we have probed the vertices of the graph each vertex we control will only give 1 point per time step instead of its real value (which is in the interval $\{1, ..., 10\}$ in the competition). It seems like an obvious choice to let our two explorer agents probe a number of vertices in the beginning of a match before trying to have our agents control an area with high value. In the meantime the other agents can try to do as many achievements as possible. In the competition we used the first 80 time steps to get as many achievements as possible and thereafter we would try to control an area with as high value as possible for the rest of the match. We will refer to the two strategies as the *achievement* strategy and the *area controlling* strategy respectively. The choice of the 80 time steps is based on our experience of how long it typically takes our agents to reach a reasonable number of achievement points while also discovering some "valuable" parts of the graph. This can clearly be refined and is not necessarily the best choice on bigger maps than what was typically used in the competition.

5.2 Achievement Strategy

The goal of this phase is to explore the graph to get information about the map structure while getting as many achievement points as quickly as possible. This naturally also involves probing as many nodes as possible which will prepare us for the *area controlling* phase.

To be versatile and obtain different kinds of achievements, most of the agents will perform the task that is unique to them given their role during this phase of the game. Explorers will probe vertices and typically reach 60-80 probes before the phase is over. Sentinels will survey and inspectors will inspect other agents and start surveying if all opponents are inspected before the phase ends. Saboteurs will attack non-disabled opponent agents, prioritizing repairers and saboteurs over other agent types. Repairers will survey if no team agents are disabled, otherwise they will repair team agents, prioritizing a disabled repairer over other agents. When choosing between multiple possibilities, i.e. different unprobed vertices, multiple disabled agents, etc. the agents will always choose the closest target where the distance is calculated using the pathfinding algorithm given in section 6.2 taking path length, the number of vertices on the path and the agents' recharge rates into account.

5.3 Area Controlling Strategy

As the vertices with high values are typically placed close to each other in the maps of the competition, both teams will usually not have any choice but to try to get control of as much of this area as possible. We will call this area the *good* area. Because if we try to control areas in other parts of the graph, the opponent team will get control of the good area, leading to us losing the match. In our area controlling strategy the saboteurs will still try to attack

opponents and the repairers will also use the same strategy as in the achievement phase. If our opponents try to get control of the good area as well, our saboteurs and repairers will also be in the good area and indirectly help us to get control of as many vertices as possible in this area. The other six agents will place themselves on strategically important vertices given by the area controlling algorithm in section 6.1 to give us control of as valuable an area as possible. If there is a part of the good area that has not been probed, our explorers will probe that part before helping to control the area so we will not miss out on any area points due to unprobed vertices. In addition, the agents capable of parrying attacks will do so, while they try to control vertices giving us achievement points and making the time spent by the opponent saboteurs for each successful attack longer.

5.4 Putting It All Together

In the above we have omitted the discussion of how the agents agree on who does what when conflicts can occur. The assignment problem is simply solved using the strategy described in section 3.1. In this way each agent will specify his benefit for the different goals according to his beliefs about the world and then the agents will in a distributed manner negotiate an assignment that gives as large benefit for the whole team as possible. Also, the assignment algorithm guarantees that the disabled agents are divided among the repairers in a way such that they will not try to repair the same agent. The same concept applies to agents with survey goals and any other type of goal which more than one agent is capable of accomplishing. We did not cover what our agents will do when the whole graph is surveyed before the 80 steps are over. In this case the agents will start using the area controlling strategy one by one. This makes the transition between the two strategies natural and our coordination algorithm will make sure this is done automatically.

Our solution came in as number two in the final ranking. Out of 24 matches we lost all three matches against the team taking the best ranking and only one other match. The total score of all teams was also compared and in this category our team came in as number one scoring almost 20% more than the second best which was the overall winner. Even though the total score didn't count in the final ranking, we still think that it is very important and that it suggests that our solution had very much potential. It is very hard to point out the exact reasons that we lost some matches. This is because matches cannot be directly compared due to the random map generation and because one action may have great side-effects in one match but not in another. However it seemed like the winning team only did better than us on some key points, especially in upgrading their saboteurs and that we were equally fit in most other areas. For some areas we even did better than the winning team, which is backed by the fact that our team got the highest overall score.

6 Additional Algorithms

In this section we describe the area controlling and pathfinding algorithms used in the solution. We give the algorithms in pseudo-code and discuss important aspects such as running-time, correctness and termination.

6.1 Area Controlling

Given the nondeterminism of the environment and the graph coloring described in section 2.2 it is clear that it is unfeasible to search for an optimal solution to zone controlling. Therefore the algorithm is very much designed by trial and error while taking area robustness, agent count and zone value into account. Two parameters directly influence the value of a zone $Z \subseteq V$, namely the size of the zone $|Z|$ and the values of its vertices $value(v), v \in Z$. Two very similar algorithms, AREA1 and AREA2, are used to decide on an area. The only difference is that the algorithm AREA2 tries to find an area which is free of opponent agents. If this is not possible then AREA1, which ignores opposing agents, is used instead. The pseudo-code is provided as algorithm 1 and algorithm 2.

Informally AREA2 first finds the most valuable vertex v in the graph. Now let C be the set of vertices with distance 2 to v. The vertices in C form a frontier around v such that all paths from v to vertices with distance greater than 2 to v goes through a vertex from C. Starting with a random vertex $u \in C$ the following vertices will be selected from C such that they have distance 2 to a previously selected vertex from C. The final selection will include v and every second node on the path created by the frontier.

The idea is then to place an agent on each of the selected vertices. When not considering opposing agents this construction will create a zone containing all selected vertices and additionally at least the vertices between v and the frontier and vertices outside of the frontier connected to two of the selected vertices in the frontier cf. section 2.2.

First AREA2 is run with $i = 6$ (at most 6 agents will participate in area controlling as repairers and saboteurs will keep repairing and sabotaging). Thus the middle vertex v and 5 vertices from the frontier are selected. If the number of vertices in the frontier is greater than 10 only part of the frontier will be covered by the final selection. If AREA2 fails to return an opponent-free area of the desired size, AREA1 is run with $i = 6$. If the number of vertices in the frontier is less 10, vertices between v and the frontier are randomly selected until 6 vertices are selected in total, thus increasing the area's robustness against opposing agents.

All nodes with distance 3 to v are probed if their value is unknown and the area is reconstructed at every simulation step making it dynamically adapt to the placement of the opposing team's agents. Testing has shown that the size of the circle around v (distance 2 to v) is a good tradeoff between easy protection from opposing agents and zone size. See figure 4 for an example of an area and the corresponding agent placement.

Algorithm 1 AREA1$(G = (V, E), i)$

```
R ← ∅
u_1 ← NIL
for v ∈ V do
    if u_1 = NIL ∨ value(u_1) < value(v) then u_1 ← v
    end if
end for
R ← R ∪ {u_1}
for u ∈ V do
    candidate ← NIL
    for v ∈ R do
        if DISTANCE(u, v) < 2 ∨ u ∈ R then
            candidate ← NIL
            break (inner for-loop)
        end if
        if DISTANCE(u, v) = 2 ∧ DISTANCE(u_1, v) = 2 then
            candidate ← u
        end if
    end for
    if candidate is not NIL then
        R ← R ∪ {candidate}
    end if
    if i = |R| then return R
    end if
end for

for v ∈ V do
    if DISTANCE(u_1, v) = 1 ∧ v ∉ R then
        R ← R ∪ {v}
    end if
    if i = |R| then return R
    end if
end for
return R
```

where DISTANCE is defined for vertices s and $t \in V$ as:

$$\text{DISTANCE}(s,t) = i \Leftrightarrow \exists_{v_0,\ldots,v_i \in V}(s = v_0 \wedge v_i = t \wedge (v_{j-1}, v_j) \in E \text{ for } j \in \{1,\ldots,i\}) \\ \wedge \neg\exists_{v_0,\ldots,v_k \in V}(s = v_0 \wedge v_k = t \wedge (v_{j-1}, v_j) \in E \text{ for } j \in \{1,\ldots,k\} \wedge k < i)$$

AREA1 returns the set of selected vertices R. The first for-loop runs exactly $|V|$ iterations. The following outer for-loop iterates at most $|V|$ times while the inner for-loop iterates at most $i - 1$ times. Checking for distance 2 which is done inside the inner for-loop can trivially be done with a breadth-first search in time $b^2 \leq |V|^2$ where b is the number of neighbors and we thus get the total running time $O(|V|^3)$ as i is a constant and AREA1 is thus polynomial in $|V|$. This running-time is however very pessimistic as the branching factor is less than 10 for all maps we have seen so far. Assuming a constant branching factor we could instead obtain a running time linear in $|V|$. In all loops every element in a set of vertices is considered exactly once, and as the number of different vertices is finite, the algorithm will always terminate even if vertices are added to the sets inside the loops.

Algorithm 2 AREA2($G = (V, E), i$)

```
R ← ∅
u_1 ← NIL
for v ∈ V do
    if u_1 = NIL ∨ value(u_1) < value(v) then u_1 ← v
    end if
end for
R ← R ∪ {u_1}
for u ∈ V do
    if OPPONENTFREE(u) then
        candidate ← NIL
        for v ∈ R do
            if DISTANCE(u, v) < 2 ∨ u ∈ R then
                candidate ← NIL
                break (inner for-loop)
            end if
            if DISTANCE(u, v) = 2 ∧ DISTANCE(u_1, v) = 2 then
                candidate ← u
            end if
        end for
        if candidate is not NIL then
            R ← R ∪ {candidate}
        end if
        if i = |R| then return R
        end if
    end if
end for
return R
```

where DISTANCE is defined as in AREA1

and OPPONENTFREE is defined so that a vertex u is opponent free for a team t if

$$\neg\exists v, a(team(a) \neq t \wedge pos(a) = v \wedge (v = u \vee (u, v) \in E))$$

The running time and termination of AREA2 is identical to AREA1 except from the OPPONENTFREE check made inside the second outer for-loop. However this check can trivially be done with a breadth-first search in time $O(b) \leq |V|$ and we again get the total running time $O(|V|^3)$ for AREA2 which similarly is polynomial in $|V|$.

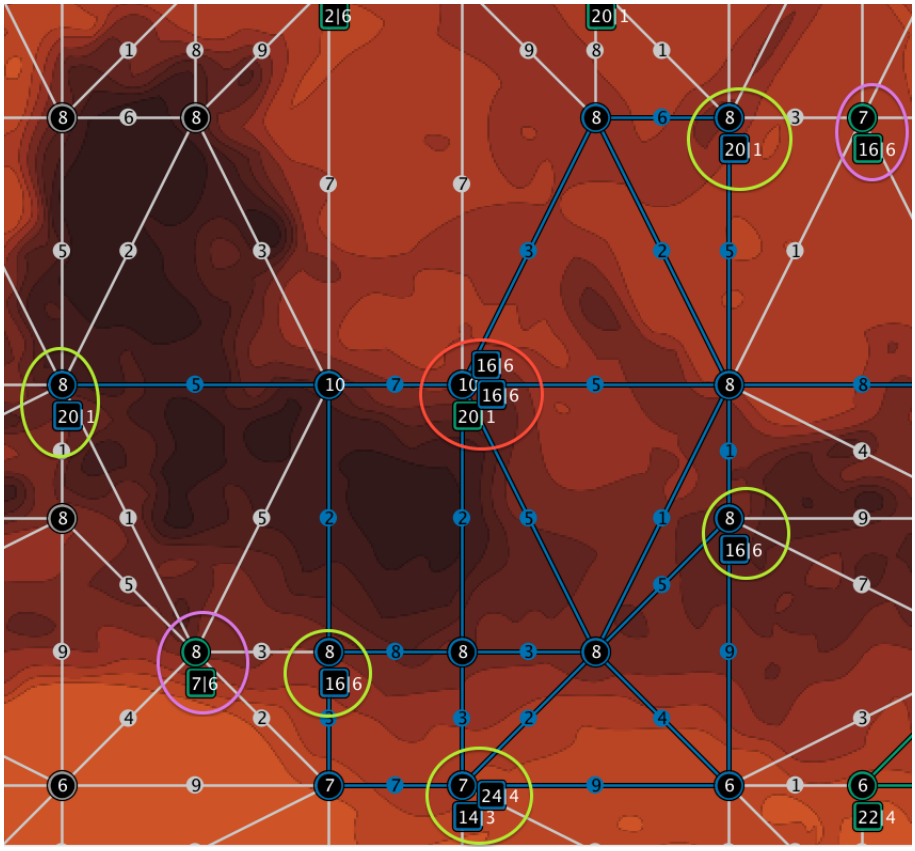


Fig. 4. A scenario showing area controlling by the blue team. The vertex v marked by a red circle has value 10 (the highest possible value). The vertices marked by a yellow circle covers part of the frontier around v. All of these vertices have distance 2 to v and distance 2 to their selected neighbors in the frontier. The purple circles show that the zone is not opponent-free and has therefore been constructed by AREA1, as no opponent-free zones could be constructed.

6.2 Pathfinding

Pathfinding in scenarios similar to that of the competition is usually done using a best-first algorithm such as A*. This however requires an evaluation function and that some heuristic can be applied to the domain, in many cases euclidian distance can be used. However this was not possible in this particular domain as no physical placement or positions of the vertices are given.

To avoid exponential running times of breadth-first and similar search algorithms we instead implemented and tweaked an all-pair shortest path algorithm to allow dynamic vertex addition. However we did not use this algorithm in the competition mainly for the two following reasons.

First, in the early development phase we suspected our dynamic all-pair shortest path algorithm as a candidate for some performance issues. However we discovered that the issues were I/O related, and found that we had plenty of processing time and power to perform multiple stock graph search algorithms at each simulation step. Secondly, the importance of not only considering the edge weights but also the number of vertices on a path as described shortly, led to the decision to use the algorithm described in the following section for path-finding. We could instead have tweaked the dynamic all-pair shortest path algorithm to also consider the number of vertices on a path, but this seemed to be nontrivial. However the all-pair shortest path algorithm could still be highly relevant in case of less processing time or bigger graphs.

Pathfinding in Discrete Time Given a weighted undirected graph $G = (V, E)$, it is trivial to find the shortest path between two vertices u and $v \in V$ using stock graph search algorithms such as best-first search. However the given scenario yields for a pathfinding algorithm not only taking the total path length but also the number of vertices on the path into account. This becomes evident when considering the discrete nature of the competition scenario.

In general it is desirable for an agent to reach a given goal which might involve moving from a vertex u to another vertex v in as few time-steps as possible. Thus enabling the agent to reach more goals throughout a simulation. Also the possible points given for reaching a goal will have a greater effect on the final score, the earlier the goal is reached cf. section 2.1. But reaching goals clearly also requires energy. Only the skip and recharge actions does not consume energy and clearly these will not directly help an agent reach any goals.

Moving from a vertex u to another vertex v through a path p will consume the sum of the edge weights on p from the agent's energy. Choosing the shortest path will therefore minimize the required energy. However it also requires $n - 1$ time steps to move from u to v through p where n is the number of vertices on p. As seen in figure 5 the shortest path in terms of edge weights might not be the shortest path in terms of required time-steps. To

this extend there is however, an important relation between time-steps, energy and path length. Assume that an agent a has $e(a) = 20$, $pos(a) = v_1$ and $h(a) > 0$ at time-step s in figure 5. Choosing the shortest path p_1 in regards to edge weights (v_1, v_2, v_3, v_4), a will reach v_4 at time-step $s + 3$ with $e(a) = 14$. Choosing the shortest path p_2 in regards to required time-steps (v_1, v_5, v_4), a will reach v_4 at time-step $s + 2$ with $e(a) = 13$. a can then perform a recharge action and will thus be in a better position ($e(a)$ would be higher), at time-step $s + 4$ if a chooses p_2. And thus clearly p_2 is the optimal choice in this situation.

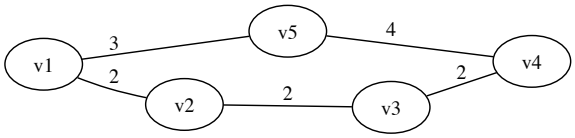


Fig. 5. Getting from vertex v_1 to vertex v_4 an agent can choose between path v_1, v_5, v_4 with total cost 7 which requires 2 time-steps or path v_1, v_2, v_3, v_4 with total cost 6 which requires 3 time steps.

We give the following definition of a best-path:

Definition A path p between two vertices u and v is an ordered set of vertices $w_0, \ldots w_i$ where $i \geq 1$ such that $u = w_0, \ldots, w_i = v$ and $(w_{j-1}, w_j) \in E$ for $1 \leq j \leq i$. The length of p is $length(p) = i$ and the cost of p for an agent a is given as:

$$\begin{aligned} cost(p) &= \sum_{1}^{i} (weight(e_i) + rr(a)) \\ &= i * rr(a) + \sum_{1}^{i} weight(e_i) \\ &= i * rr(a) + edgecost(p) \end{aligned}$$

Where $e_i = (w_{i-1}, w_i)$ and $rr : A \to \mathbb{N}$ is the recharge rate for a agent defined as:

$rr(a) = 50\% * me(a)$ rounded to the nearest integer, if $h(a) > 0$
$rr(a) = 30\% * me(a)$ rounded to the nearest integer, if $h(a) = 0$

Finally p is the best-path for a if there is no path p' such that:

$$cost(p) > cost(p')$$

Energy Overflow Unfortunately choosing a path using the definition above will not always lead to the optimal path choice. Consider the two paths from v_1 to v_4 in figure 6: Let a be an agent with $me(a) = 14$, $h(a) > 0$ and $e(a) = 14$ at time-step s. Then $p_1 = v_1, v_2, v_3, v_4$, $length(p_1) = 3, cost(p_1) = 30$ and $p_2 = v_1, v_5, v_4$, $length(p_2) = 2, cost(p_2) = 29$ and thus $cost(p_2) < cost(p_1)$ and p_2 should be the optimal path. However choosing p_2, a will have to recharge after traversing (v_1, v_5) before it can traverse the edge (v_5, v_4) and thus a will arrive at v_4 at time-step $s + 3$ with $e(a) = 4$. If a instead choose p_1 it could walk the complete path without recharging and thus reach v_4 at time step $s + 3$ with $e(a) = 5$. But then p_1 is clearly better than p_2 for a and it follows that p_2 is not optimal.

The reason that p_1 comes out as the better choice in this case is that a does not get the full potential of the necessary recharge on p_2. The recharge could potentially increase a's energy by 7, but because $me(a) = 14$ and $e(a) = 9$ when the recharge action is carried out, a only increases its energy by 5. If these two energy points wasn't lost, which we denote as energy overflow, p_2 would indeed be the optimal choice for a. Note that the order of the edges is also important. If we swap the weight of edge (v_1, v_5) and (v_5, v_4), p_2 would also be the optimal choice.

We stick to our definition of best-path as it really is optimal when there is no overflow and still gives a very decent path even with energy overflow, because there is a bound of the energy overflow, which in addition only rarely happens in this domain as we shall see in the following sections.

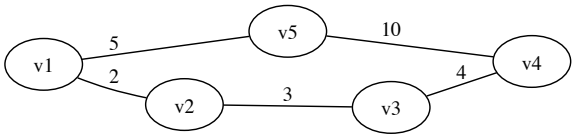


Fig. 6. Getting from vertex v_1 to vertex v_4 an agent can choose between path v_1, v_2, v_3, v_4 with total edge-weight 9 with length 3 or path v_1, v_5, v_4 with total edge-weight 15 and length 2.

Bounded Energy Overflow Energy overflow can only happen if an agent a needs to traverse an edge $(u, v) \in E$ and $e(a) < weight((u, v)) \wedge e(a) >$

$(me(a) - rr(a))$ and the energy overflow is then given as $overflow = e(a) + rr(a) - me(a)$. But as there is an upper bound on all edge weights of 10 cf. section 2 it is always the case that $e(a) < 10$ when overflow happens. But from here it follows that overflow can never happen if $h(a) = 0$ as $e(a) + rr(a) - me(a) < 0$ when $e(a) < 10$ due to the recharge rates and maximum energy of the different agent roles. It also follows that the greatest energy overflow possible is 2 for the saboteur, 1 for the repairer and the inspector and that overflow will never happen for explorers and sentinels. Lastly energy overflow will only happen for a repairer or inspector a if $e(a) = 9$ and a needs to traverse an edge with weight 10, and will only happen for a saboteur a if $e(a) = 9$ and the following edge has weight 10 or if $e(a) = 8$ and the following edge has weight 10 or 9. After an occurrence of energy overflow for agent a, it must be the case that $e(a) = me(a)$ thus it requires traversal of an edge to bring $e(a) < 10$ so another energy overflow can happen. Thus for a path of length i the maximum overflow for a is bounded by $i/2 \cdot 2 = i$ assuming that a starts with $e(a) = me(a)$. In most cases there is no energy overflow or the actual energy overflow is much lower because it will only happen in rare situations due to the random map generation and the required edge constellation.

Optimal with No Energy Overflow Under the assumption that energy overflow will never happen, it is clear that every recharge action for an agent a will increase $e(a)$ by $rr(a)$. Thus recharges will be carried out by a whenever needed. But under this assumption we can ignore the maximum energy restriction for a when considering paths. This is because it then makes no difference if a recharges when required while walking a path p, or if a recharges enough to walk the complete path before starting the walk. But then we can say that a path p from u to $v \in V$ is better than a path p' from u to $v \in V$ if at least one of the following points holds:

Let a start at time-step s and from there recharge enough to walk the complete path and let $e(a)_{p''}$ denote the amount of energy a has after walking a path p'':

1. a arrives at v at time-step $s + i, i > 0$ for both paths but $e(a)_p > e(a)_{p'}$ (p is obviously better than p')
2. a arrives at v at time-step $s + i$ after recharging and walking p and at time-step $s + j$ after walking p', where $0 < i < j$ and $e(a)_p + j - i * rr(a) > e(a)_{p'}$ (p is shorter than p' and a can use the saved time-steps to recharge and is then in a better position at time $s + j$ by choosing p over p')
3. a arrives at v at time-step $s + i$ after recharging and walking p and at time-step $s + j$ after walking p', where $0 < j < i$ and $e(a)_{p'} + j - i * rr(a) < e(a)_p$ (p' is shorter than p but using the saved time-steps to recharge will not leave a in a better position at time $s + i$)

And by the definition of best-path at least one of these points must holds if $cost(p) \leq cost(p')$. This is easily seen as cost is composed by the total edge

weight + (the path length × recharge rate of the agent) where the multiplication implicitly expresses that the possible "saved time-steps" are worth as much as an agent could increase its energy by using them to recharge. The pseudo-code for finding the best-path is provided as algorithm 3.

7 Conclusion

We have implemented an auction based agreement algorithm which turned out to be a very good solution for cooperation between the agents. We have found a close to optimal solution to the non-trivial problem of pathfinding in discrete time. Tweaking our solution with prioritized attacks and repairs have also proven very effective for the given scenario.

Even though the nature of the competition and the time limitation encourages very domain specific solutions, we have considered genuine multi-agent challenges such as agreement, cooperation and communication. Python has proved to be a suitable programming language for implementing a multi-agent system. We did not encounter any programming language specific problems or limitations and many features of Python helped us develop an effective yet very compact solution. We were mostly satisfied with the behavior of our agents, however there is still room for improvement. Our general approach and strategies turned out to be very effective, but because of the time limitation we could not implement all of our ideas. Especially our vertex expansion algorithm could have been further optimized and our buying strategy could have been dynamic so that it took the opposing team's strategy into account.

Acknowledgements

Thanks to Andreas Schmidt Jensen, John Bruntse Larsen and Niklas Christoffer Petersen for comments.

References

Behrens et al.,2011. Tristan Behrens, Jürgen Dix, Jomi Hübner, Michael Köster, and Federico Schlesinger. *Multi-Agent Programming Contest — Scenario Description — 2011 Edition*, available online `http://www.multiagentcontest.org/`, 2011.

Zavlanos et al.,2008. M.M. Zavlanos, L. Spesivtsev, and G.J. Pappas. *A Distributed Auction Algorithm for the Assignment Problem*, Proceedings of the 47th IEEE Conference on Decision and Control, Cancun Mexico, 2008.

Bertsekas et al.,1991. D.P. Bertsekas, and D.A. Castanon. *Parallel synchronous and asynchronous implementations of the auction algorithm*, Parallel Computing 17, 707–732, 1991.

Boss et al.,2010. Niklas Skamriis Boss, Andreas Schmidt Jensen, and Jørgen Villadsen. *Building Multi-Agent Systems Using Jason*, Annals of Mathematics and Artificial Intelligence 59, 373-388, 2010.

Algorithm 3 BESTPATH($G = (V, E), u, v$)

```
Q ← ∅
R ← ∅
S[u] ← 0
T[u] ← NIL
ENQUEUE(Q, u, 0)
while Q ≠ ∅ do
    c ← DEQUEUE(Q)
    while c ∈ R and Q ≠ ∅ do
        c ← DEQUEUE(Q)
    end while
    if c = v then return (S, T)
    end if
    R ← R ∪ {c}
    for s ∈ V and s ∉ R do
        if (c, s) ∈ E then
            S[s] ← weight(c, s) + S[c] + rr(a)
            T[s] ← c
            ENQUEUE(Q, s, S[u])
        end if
    end for
end while
return NIL
```

DEQUEUE returns the element enqueued with the lowest priority

When v is found S and T is returned from which respectively the path cost and the path can be extracted. The algorithm returns NIL if there is no path between u and v. In the outer loop only elements not in R are selected from V and are subsequently added to R thus the algorithm will always terminate. In the worst case the algorithm will consider all paths to all possible nodes thus the time complexity is $O(b^d)$ where b is the branching factor and d is the maximum path length. When v is dequeued from Q it has the lowest priority in Q. Clearly if u is dequeued again at a later time it will thus have the same or a higher priority. Also v could not have been dequeued at an earlier step with a lower priority as the algorithm would then have terminated. It is easily seen that the priority is equal to $cost(p)$ for the path p that can be extracted from T. As the algorithm considers all paths to all possible nodes it must therefore be the case that the path that can be extracted from T is the best-path path from u to v.

Vester et al.,2011. Steen Vester, Niklas Skamriis Boss, Andreas Schmidt Jensen, and Jørgen Villadsen. *Improving Multi-Agent Systems Using Jason,* Annals of Mathematics and Artificial Intelligence 61, 297-307, 2011.

Behrens et al.,2010. Tristan Behrens, Mehdi Dastani, Jürgen Dix, Michael Köster, and Peter Novák. *The Multi-Agent Programming Contest From 2005-2010: From Collecting Gold to Herding Cows*, Annals of Mathematics and Artificial Intelligence 59, 277-311, 2010.

Bordini et al.,2007. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*, Wiley 2007.

Hirsch et al.,2009. Benjamin Hirsch, Thomas Konnerth, and Axel Hessler. *Merging Agents and Services — The JIAC Agent Platform*, In Multi-Agent Programming (editors: Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini), Springer 2009.

Behrens et al.,2009. Tristan Behrens, Jürgen Dix, and Koen Hindriks. *The Environment Interface Standard for Agent-Oriented Programming — Platform Integration Guide and Interface Implementation Guide*, Department of Informatics, Clausthal University of Technology, Technical Report IfI-09-10 2009.

Short Answers

Introduction

1. We are affiliated with DTU Informatics (short for Department of Informatics and Mathematical Modelling, Technical University of Denmark, and located in the greater Copenhagen area). We participated in the contest in 2009 and 2010 as the Jason-DTU team since we used the Jason platform and its agent-oriented programming language AgentSpeak. We performed well but for the contest this year, with the new and more complex scenario, we decided to focus on an auction-based agreement approach and to implement the multi-agent system in the programming language Python.
2. The project is done as a special course in June-September 2011 with more or less the same team as in 2010.
3. The name of our team is Python-DTU.
4. The members of the team are Mikko Berggren Ettienne and Steen Vester, MSc students, and associate professor Jørgen Villadsen, PhD.
5. We are part of Algolog, the Algorithms and Logic section, which is responsible for the Efficient and Inteligence Software study line of the MSc in Computer Science and Engineering program.

System Analysis and Design

1. We did not use any multi-agent system methodology. We preferred a flexible implementation of which we had complete knowledge and control of every part of the implementation.
2. We did not implement a solution with a centralization of coordination/information on a specific agent. Rather we implemented a decentralized solution where agents shared percepts through shared data structures and coordinated actions using distributed algorithms.
3. Our communication strategy is to share all new percepts to keep the agents internal world models identical. Furthermore our agreement based auction algorithm heavily relies on communication and is part of how agents decide on goals.
4. Each agent acts on its own behalf based on its local view of the world which is updated through percepts and is thus autonomous and reactive. This is implemented as an agent-control-loop in which the agents decide which actions to execute based on their current view of the world. When a repairer and a disabled agent moves towards each other the repairer decides and announces who should take the last step so they won't miss each other. This proactiveness is implemented by considering the current energy and the paths of the agents.

5. Our solution is a true multi-agent system. However it is currently limited with regards to distributivity by the way the agent communication is implemented.
6. We invested approximately 400 man hours for implementing the system.
7. We discussed agent designs and strategies with other teams during the competitions. We also tested our solution multiple times against itself and the other teams participating in the official test matches.

Software Architecture

1. We used the programming language Python to implement the multi-agent system.
2. We did not use any multi-agent programming language. We have good experiences with Python and would rather get started than spend time learning a new programming language.
3. The agent is implemented as a class and any number of agents can be started with a loop where each is given a reference to the shared data structure. Most of our designed architecture is implemented using Python lists and dictionaries and for some artifacts we have implemented regular classes.
4. We used Ubuntu Linux and Mac OS X as development platforms and GEdit, Eclipse and TextMate as code editors/IDEs. A couple of hours were used to investigate advanced features of the editors.
5. We only used the Python runtime again on Ubuntu Linux and Mac OS X. We have used approximately 20 hours to investigate advanced features of Python.
6. We missed better editor support, code completion, syntax support and debugging capabilities for Python.
7. The following features of Python has simplified out development task: dynamically typed, concise and compact, no compilation, multiple built-in list functions and support for multiple programming paradigms (especially the functional).
8. Our implementation has mainly relied on custom best-first searches and a distributed auction-based agreement algorithm and a custom pathfinding algorithm tweaked for this domain.
9. We did not distribute the agents on several machines for two reasons. First, we had no need to, as we had plenty of computation power on a single machine to reason and send the action messages before the deadlines. Secondly the shared data structure in our implementation would have to be replaced by a message server and a simple protocol. We considered this but due to limited time we had to prioritize differently.

10. The agents are single threaded, all reasoning is done in-between the receive-percept and the send-action messages.
11. The most difficult parts of the implementation was the different custom algorithms. Especially the ones requiring some kind of synchronization between the agents. This was solved using the shared data structure in combination with timeouts.
12. We have written 1345 lines of code in total (including comments).

Strategies, Details and Statistics

1. Our main strategy is to use the first 10% of a simulation for exploring the map and inspecting the opponent. Hereafter we focus on controlling a valuable area while letting saboteurs attack and repairers repair.
2. Every agent share any new information it perceives with all other agents. An agent does also calculate a series of prioritized goals and is assigned to one of these after participating in the agreement auction. Information about other agents, etc. is stored in a shared data structure and can be updated by any agent.
3. The agents start out by probing and surveying most of the map. Edge weights are used for pathfinding and vertex values are used when deciding on an area to control.
4. A simple class inherited by each agent handles xml parsing, creation of xml-messages and sending and receiving them over a TCP/IP connection.
5. Every agent has a strategy related to its role. However all agents will follow the explorers strategy at a certain time-step if its own strategy doesn't yield any goals at that step.
6. Our strategy is to dynamically select the most valuable known vertex v on the map and probe every vertex with distance 3 to v. We then form a circle around v by selecting vertices with distance 2 to each other and distance 2 to v. Because valuable vertices are always clustered we don't estimate the zone value.
7. Our saboteurs prioritize attacking opponents inside the zone we control. This helps both conquering and defending zones. However, if the zone is clear of enemies, they will attack all over the map, including enemy zones.
8. Our agents changes behavior to follow the main strategy. This change from exploring to area controlling as triggered by a specific time step in the simulation. Also if their current strategy does not yield any goals they will temporarily adopt another strategy.

9. A customized best-first algorithm taking both agent energy, agent recharge rate, path length and the number of vertices on a path into account was used for pathfinding.
10. When we reach a certain point in the simulation, we upgrade our saboteurs health and strength with one point.
11. Achievements points are a big part of the overall score for our team, especially the ones that are obtained early in the simulation. Except from that they are not very important for our strategy.
12. Our agents does not have an explicit mental state.
13. Our agents mainly communicate and coordinate by placing bids on different goals in a shared data structure when participating in the auction-based agreement algorithm.
14. We have not taken an organizational approach, thus the organization of our agents is implicit.
15. All our agent behaves individually and autonomously to achieve the best for the team, taking their current position, health, etc. into account when deciding on a goal.
16. Our agents does not perform any planning other than pathfinding. A path can reach many steps ahead, but is re-calculated at each time-step.

Conclusion

1. Participating in this competition has greatly increased our experience with many of the practical, implementation and theoretical aspects and challenges of constructing a multi-agent system.
2. We had a very strong solution doing good against all teams and only loosing to one. In general we were good at disabling enemy agents, repairing team agents and controlling areas. A weak point was our static buying strategy which gave teams with a dynamic buying strategy an advantage against us.
3. We found Python very suitable for implementing an efficient solution with a manageable amount of code. Our custom algorithms have all proved very useful for this specific domain.
4. We think that changes in the balancing of the competition were announced too late. We suggest that important parameters should be settled on at least a month before the competition date. We were happy with the test runs, but we suggest that they are performed at an earlier point to avoid the problems many teams had with server communication.

5. Part of why we did good is because we could quickly implement the necessary framework in Python and focus on more important things like strategies and algorithms. Many other teams had great trouble with communication with the game server throughout the competition.
6. We think the contest has relevant elements for the following research areas: Algorithms, Game development, Game theory, Logic, AI.
7. We think that the scenario should be updated so that it is no longer feasible to share all percepts between all agents. Thus making it advantageous to implement a true multi-agent system, which is not the necessarily the case in the current scenario.

A Gaia-driven Approach for Competitive Multi-Agent Systems

Sahar Mirzayi, Vahid Nateghi And Fatemeh Eskandari

Department of Computer Engineering, Faculty of Engineering
Arak University – Arak 38156-8-8349, Iran
sahar.mirzayi@gmail.com
vahid.nateghi@gmail.com
fatemeh.eskandari.69@gmail.com

Abstract. This is a report on Simurgh team's participation in the 2011 multi-agent contest. The design and development process, architecture details, and team strategies for the multi-agent system have been discussed, along with experiences of the developers. Gaia methodology was used for the design and analysis of the Simurgh multi-agent system. The main strategy was obtaining a higher score through the support of agents with a better perceived strategic placement. Decision correction strategy was used to change the agent behavior, by taking the other conflicting team members' decisions, into account. Simurgh was implemented using Java language. Agents have the same degree of autonomy and the team is implemented in a completely decentralized fashion.
Keywords: Multi-Agent System, Gaia, Decentralized Coordination, Dynamic Role Assignment

1 Introduction

The multi-agent programming contest (MAPC) is held every year in the pursuit of expanding researches and testing frameworks and development environments for the Multi-Agent Systems (MAS) [Behrens et al.,2010]. In 2011, for the first time, Simurgh [1] team took part in the contest, in order to improve its knowledge of MAS. Taking part in MAPC was a project of "specialized studies in software engineering" course. Most of the material presented in the course was about analysis, design and development of MAS. The MAPC organizers introduced the "Mars" scenario for this year's competition, which was a complete redesign and different to the scenarios used in previous contests, especially in that the map was based on a graph this year. The Simurgh team decided to use Gaia methodology [Zambonelli et al.,2003] for the analysis and design of the agents and use Java as the implementation language.

[1] Simurgh is an old Persian myth, quite similar to the Phoenix, however, a Phoenix is believed to be quite small and to be reborn from its ashes, whereas a Simurgh is believed to be big, very wise, and can be called upon by burning one of its feathers.

The analysis phase started from Jan 2011 and because of multiple changes in the scenario, the development continued on until September.
Simurgh is compromised of three members, Sahar Mirzayi, Vahid Nateghi and Fatemeh Eskandari. Mirzayi is a Graduate student at Arak University. Her interests are soft computing and distributed systems. Nateghi and Eskandari are under graduate students at Arak University, interested in the field of robotics.

2 System Analysis and Design

Our team used Gaia methodology for the analysis and design of Simurgh because of its high reactiveness property, accessibility of its notation and modeling and its flexibility. Even though Gaia has the same precision property in comparison with Tropos [Castro et al.,2002], clearness and understandability of Gaia were the main reasons for our choice. Furthermore Gaia is appropriate for developing systems with small number of agent types. It is a general methodology that supports analysis and design of both the individual agent structure and the agent society in the MAS development process. According to Gaia, MAS is viewed as a composition of a number of autonomous and interactive agents existing in a society, in which each agent plays one or more specific roles.
Gaia defines the structure of MAS in terms of role model [Juan et al.,2002], the model which identifies the roles that agents have to play within the MAS, and the interaction protocols between the different roles. The objective of the Gaia analysis process is the identification of the roles and modeling the interactions between them.
the role model phase of Simurgh was analyzed using Gaia, the roles were identified as follows: the *Server Connector* role is responsible for connecting to the server, receiving perceptions and sending actions back. Agents should be capable of translating perceptions and extracting information within perceptions, this is the *Parser* role. The *Team Communicator* is another role which has to facilitate communication with other teammates and share perceptions and decisions with them. Moreover, the agents should explore the map and try to complete the world model; this is the operation for which the *World Explorer* role is responsible. The *Zone Holder* is a role which should try to create a zone or expand available zones. Some agents might be disabled by opponent's agents, the *Helper* role has to find these agents' positions and then approach and repair them. The *Water Well Explorer* role is responsible for finding water wells and increasing the team's score. Because in this year's scenario, opponents can attack each other the *Attacker* role is defined to blemish the nearest opponent's agents if there are any. The *Chaser* role is also defined for gathering position data on enemies' agents and to attack them. Because agents must have a plan to preserve themselves, the *Defender* role is also defined for parry. *Opponent Analyzer* role tries to inspect enemies in the maps' vertices. And there is also the *Savior* role, which helps team-

mates under opponent attack.
The roles are dynamically assigned to agents. The rules governing the dynamic role assignments are covered in section 4.
The requirements of the MAS were prioritized as goals. According to the scenario and our strategy, *supporting the best positioned agents, by other teammate, to obtain the highest score*, was our first priority. A very valuable goal is creating zones in the map by finding the nearest friend using the Dijkstra algorithm, but if there were already some zones, then the highest priority is assigned to the agents who can expand the most valuable available zone. Also, if there are some disabled agents, the important goal for the Repairer is to help these disabled teammates. So the priorities of goals for each role are different. The goals are as following:

- Move to a node for creating a zone
- Move to a node for expanding the zone
- Stand in a certain place to keep the zone
- Repair a disabled agent
- Move to a node for repairing a disabled agent
- Probe the node
- Move to a node for probing
- Parry
- Attack
- Move to a node for attacking opponents
- Move to a node for saving teammates
- Survey the map
- Buy

The most important goal is creating a zone. For each of the goals, a proper algorithm is designed and implemented according to team strategies, which will be discussed in section 4. All agents have the same degree of autonomy, which makes our implementation a true MAS.
Simurgh used a decentralized coordination and cooperation mechanism for implementing agents, which will be described in the following. In the beginning of each step, agents share their perception to other teammates via a shared channel and then update their perception according to the received perception messages from the channel. In each step, agents define some goals and inter-operate with other agents to make a final decision according to other agents' goals, autonomously. When agents share their goal with teammates, if there is a conflict between goals of two agents in the current step, one of them chooses to change its goal to prevent a conflict. It was decided that an agent with the lowest potential score for the team, changes its decision but according to deadline approach, this could not be implemented, therefore, in the final version, the agent which had to change its goal was chosen by its goal's priority. The agent should then replace its goal with the one that doesn't have any conflict with other teammates' goals. In conflicting situations, the agent with better energy proceeds and the other one

changes its action. The conflicts found in agents' decisions are listed below:

- If two Explorers try to explore a same node one of them is doing a useless action. So there is a conflict between two explore actions.
- If two agents try to survey the same node, one of them can do another action because the resulting perceptions are shared at the beginning of each step. In this case, the agent who has a larger visibility range does the action and the other one should change its decision.
- Inspections done in the same node have the same result and therefore, redundant. One of the agents should change its decision.
- When an agent tries to create a zone with a teammate, this teammate should not move to other nodes for any reason except repairs.
- When an agent is trying to expand its zone, other agents in the zone, which have a maximum distance of 2 from it, shouldn't move to nodes which have a maximum distance of more than 2 from the agent's new position.

Doing 2 Go-to actions with the goal of repairing a same agent, must not generate a conflict, because one of the repairers may face an attack and not be able to get to the disabled agent. After taking these precautions, conflicts are still possible because a second level of chosen goal message sharing was not implemented.
According to the above, two types of communication were needed: shared perception messages and shared chosen goal messages. Shared perception messages are sent and received at the beginning of each step and the shared chosen goal messages are exchanged after all agents make a decision about their goal. Because each agent communicates with its teammates via a shared channel, communication complexity in each step will be O(n), in which n is the number of agents. Gaia can transform the analysis models into a sufficiently low level of abstraction. The design phase of Gaia is agent model design [Castro et al.,2002]. The purpose of the agent model is to document the various types of agents that would be used in the system. Creating the agent model means aggregation of roles into agent types [Wooldridge et al.,2000]. The agent model of Simurgh is shown in figure 1. Agent types in the system are defined in MAPC scenario as Explorer, Repairer, Saboteur, Sentinel, and Inspector. Each agent should implement the Server Connector, Parser, Team Communicator and World Explorer roles. As we described in section 4 explorers and saboteurs don't implement the Zone Holder role. One of the saboteurs has attacker role and the other has chaser role.

3 Software Architecture

The difficulties faced during the implementation, mainly stem from goal choice. For example finding a good algorithm to create zones and keep them,

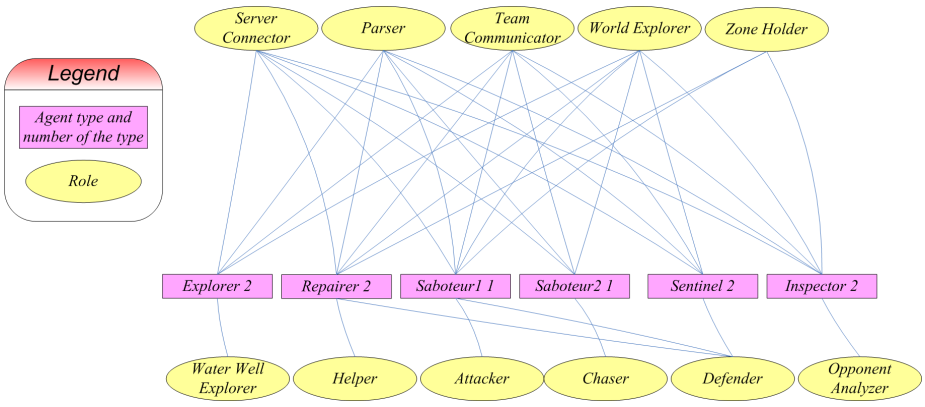


Fig. 1. Agent model diagram

required hundreds of test-runs. The communication complexity is not a major problem and doesn't force developers to use a specific runtime platform. Simurgh has been implemented using Java programming language and J2SE. The messaging between agents to share perceptions and decisions also uses apltk package features. As our team members were already familiar with Java, there was no learning curve for using the programming language.

The agent's architecture is shown in figure 2. Some of the ideas behind the design of agent architecture were adapted from [Galoan et al.]. The eis-massim package [Behrens et al., 2009] was used as the component for communication with the server. Inspired from black board design pattern [Wang et al.,2004], the channel is a shared data structure where all agents can push and pull messages. The perception updater module extracts information from perceptions and saves it in an understandable format for agents. The perception sender shares perception with other teammates. The role assignment chooses a role for the agent according to the perceived strategic placement. Then the proper goal is selected from the goal repository. The goals are sent to teammates for decision correction. If there is a conflict between two agents' goals, the conflict finder exposes them. The new role assignment process will be done, thereafter. In the end, proper actions are selected and sent to the server.

Java doesn't provide a convenient message broadcast mechanism. Moreover, synchronization points cause bottlenecks because the independent code lines after the point can't execute and agents should stop decision process until others reach the point. Using an object-oriented programming language accelerates the design and implementation process. Furthermore, using eis-massim, encapsulates connection handling details. Therefore, the whole implementation is just a bit more than 4000 lines of code.

The synchronization takes less than 1 percent of each step's length. All agents

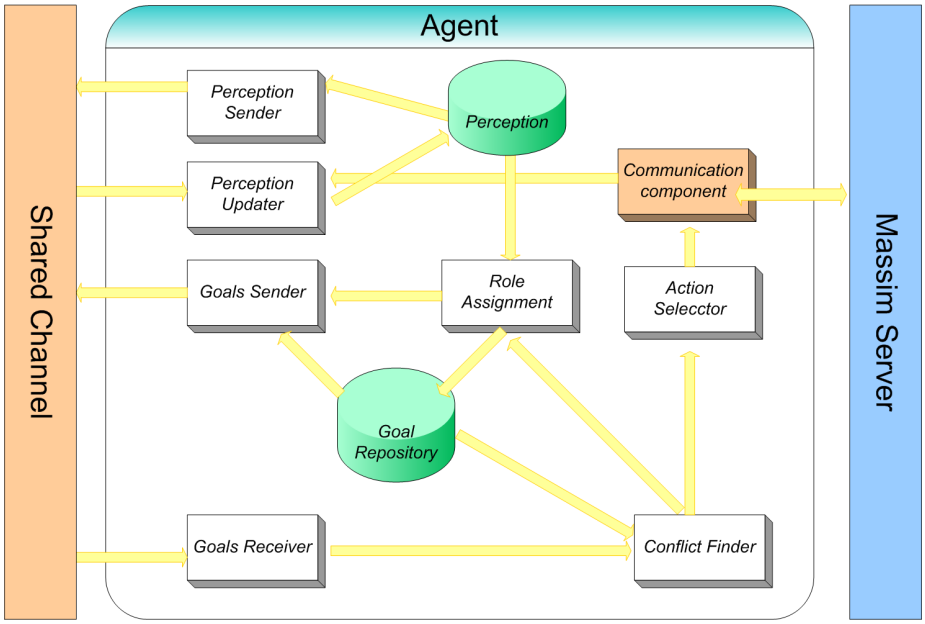


Fig. 2. The agent's architecture

execute on the same physical machine, as the computation cost of the agents is relatively low and as latency of connection to the server was one of our biggest barriers, distribution of agents did not appear to be a wise deployment decision.

4 Strategies, Details and Statistics

In order to get to know the environment and fill the agents' world model as soon as possible, in the first ten steps, all agents have the explorer role. They survey and go to new random vertices. After these steps, roles are assigned to agents iteratively during the rest of the tournament. Then each agent should follow a specific set of instructions according to its role.
Some criteria trigger agents to change their roles, such as: being disabled, being in a zone, being near a better zone, possibility of an attack, having a low level of energy and the existence of a disabled teammate.
If an agent is low on energy, it recharges itself. If an agent is disabled, it tries to approach a helper until it acquires energy. However a disabled repairer tries to repair other teammates first (helper role), and then tries to approach other repairers to acquire energy. The zone holder role (all agent types except explorers and saboteurs) first expands the zone by scattering away from teammate in the right manner. If the zone can not be expanded anymore

and there is a bigger nearby zone, an agent may leave the zone and join the other. If a zone holder senses a nearby probed node, it will change its position, so that the score of the node is achieved in the current step. If a zone holder doesn't find a zone it tries to find the nearest zone holder agent and create a zone with it. The water well explorer tries to survey new nodes in a way that new non-probed vertices are visited. the World Explorer role uses a taboo list to fill the world model quickly. If a savior receives a help message from a nearby teammate which it has a maximum distance of two, it will decide to attack the opponent in the next two steps. A chaser gathers opponents' position information and chases them with the goal of an attack. When an agent knows that there is an attacker in its node, it chooses the defender role and parries itself. An inspector chooses the opponent analyzer role when an attack occur.

As described earlier, each agent shares its world model updates with teammates and then sends their decisions and if there is a conflict, it will be changed in a way that the higher priority goal would be followed. At the end of each step, agents send their actions to the server using the eis-massim package.

The main strategy of Simurgh is Cooperation with teammates and supporting them to realize better goals. There is no central or hierarchal leadership and data doesn't route to any specific agent.

Each agent has a world model that contains every detail captured from perception messages. The most important element in the world model is the map, a weighted graph in which both edges and vertices are weighted. The other elements found in the world model are teammate positions, disabled teammates, repairer teammates, teammates who are in a zone, savior teammates, each zone's position and its score, explored water wells and enemies' positions. if agents don't have enough information about the world model (usually in the first steps of the tournament) they should move to nodes in a random fashion, in which case, a taboo list of visited nodes is kept to avoid visiting nodes twice for several steps to ensure that world exploring is done as soon as possible.

Each agent evaluates zones separately by the formula 1, as follow:

$$ZoneValue_{i,j} = \frac{\alpha * score_i}{\sum_k score_k} + \frac{\beta * distance_{ij}}{\sum_h distance_{hj}} \quad (1)$$

Where i is the zone's number, j is the agent's number, $score_k$ is the score of the k^{th} zone, $distance_{h,j}$ is the distance of j^{th} agent from nearest agent in the h^{th} zone. α and β are constant values which are chosen by experimental tests as α = 3 and β = 2 . If an agent has the zone holder role, it tries to join the zone with better Zone Value.

Finding enemies' zones and conquering them was not a part of our strategy but agents can defend zones using the savior role. For finding the shortest path to an agent, an improved Dijkstra algorithm [Horowitz et al.,2007] is used.

To improve the attacking power of the team, attacker and savior roles are allowed to use achievements' points and buy items like a shield, a sensor and three sabotage devices, when the team is in the risk of a huge attack. Although Simurgh uses achievement points to make purchases, no specific plan was developed to increase achievement points.
Agents don't have a mental state and plan for one step and if the priority doesn't force them to change their goal, the last goal is maintained. This goal has to be realized by exactly one agent. Although Our Agents' behaviors are planned individually, an agent must know how to provide valuable information to a specific teammate, for example, while chasing down enemy agents, some feedback from nearby teammate agents might be needed.

5 Conclusion

Simurgh members were new to the field of multi-agent programming. In that regard, MAPC was a very unique and valuable experience in design and implementation of MAS. The first lesson learned during the implementation is that multi-agent development methodologies are very diverse, and to make a proper choice of methodology, thorough investigations are needed. There are many multi-agent programming languages and tools, some of which directly implement the multi agent theory, while others extend existing languages to suit the MAS paradigm. The second lesson learned from MAPC was that different communication strategies are needed for decision making and realizing goals between agents living in a same society. Different approaches were examined and tested and choices were made based on the best results. Central leadership, hierarchal leadership and group-based goal realization with multiple leaders were surveyed; however, it was ultimately decided to have a monolithic society in which all agents can choose a goal according to their capabilities and coordination.
Because of the lack of reliable high speed connection and high delay times, we missed half of the tournaments and finally achieved the 6^{th} place. Some strength and weaknesses were found in the team design. For example, Simurgh could parry very well because if an agent inspects an attacker, it informs the team savior agents quickly. Choosing the helper role instead of zone holder has a higher priority for repairers, so sometimes our zone broke during the movement of our repairers. We did not put any emphasis on attacks in Simurgh, especially breaking enemies' zones, so other teams could gain score simply when competing against us. The zone expanding algorithm has some weaknesses that cause zone breaking. This is one of the problems we hope to resolve for the next year's contest. Furthermore, for the next MAPC we hope to improve our attacks and design an algorithm for finding opponent's zones'.
Gaia proved to be a good base methodology for Simurgh MAS development. It was flexible, clear and understandable. Moreover J2SE did not impose any worrying constraints on us.

In the scenario, coloring algorithms could be improved. For example, zones should be able to expand only in 3 or 4 levels of colored nodes, so that if most agents of a team are disabled, the opponent can't expand the zone to the whole map.

References

[Behrens et al.,2010] T. Behrens, et al., "The multi-agent programming contest from 2005-2010", Annals of Mathematics and Artificial Intelligence, vol. 59, pp. 277-311, 2010.

[Zambonelli et al.,2003] F. Zambonelli, et al., "Developing multiagent systems: The Gaia methodology", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 12, pp. 317-370, 2003.

[Castro et al.,2002] J. Castro, et al., "Towards requirements-driven information systems engineering: the Tropos project", Information systems, vol. 27, pp. 365-389, 2002.

[Juan et al.,2002] T. Juan, et al.,"ROADMAP: extending the gaia methodology for complex open systems", 2002, pp. 3-10.

[Wooldridge et al.,2000] M. Wooldridge, et al., "The Gaia methodology for agent-oriented analysis and design", Autonomous Agents and Multi-Agent Systems, vol. 3, pp. 285-312, 2000.

[Galoan et al.] V. Rafe, et al., "Galoan: a multi-agent approach to herd cows", Annals of Mathematics and Artificial Intelligence, pp. 1-16.

[Behrens et al., 2009] T. Behrens, et al., "The Environment Interface Standard for Agent-Oriented Programming Platform Integration Guide and Interface Implementation Guide", 2009.

[Wang et al.,2004] B. WANG, et al., "Active Blackboard Design Pattern for Distributed Agents Coordination", Computer Engineering, vol. 9, 2004.

[Horowitz et al.,2007] E. Horowitz, et al., "Fundamentals of data structures in C", Silicon Press, 2007.

Short Answers

- What was the motivation to participate in the contest?
  To improve team member's knowledge of MAS.
- What is the (brief) history of the team?
  Taking part in this contest was a project of "specialized studies in software engineering" course. Most of the material presented in the course was about analysis, design and development of MASes.
- What is the name of your team?
  Simurgh
- How many developers and designers did you have? At what level of education are your team members? From which field of research do you come from? Which work is related?
  Simurgh is compromised of three members, Sahar Mirzayi, Vahid Nateghi and Fatemeh Eskandari. Mirzayi is a Graduate student at Arak University. Her interests are soft computing and distributed systems. Nateghi and Eskandari are under graduate students at Arak University, interested in the field of robotics.
- If some multi-agent system methodology such as Prometheus, O-MaSE, or Tropos was used, how did you use it? If you did not what were the reasons?
  Our team used Gaia methodology for the analysis and design of Simurgh because of its high reactivness property, accessibility of its notation and modeling and flexibility. Even though Gaia has the same precision property in comparison with Tropos, clearness and understandability of Gaia were the main reasons for our choice.
- Is the solution based on the centralization of coordination/information on a specific agent? Conversely if you plan a decentralized solution, which strategy do you plan to use?
  Simurgh used a decentralized coordination and cooperation mechanism for implementing agents. In the beginning of each step agents share their perception to other teammates via a shared channel and then update their perception according to the received perception messages from the channel. In each step, agents define some goals and inter-operate with other agents to choose their final decision according to other agents' goals, autonomously. When agents share their goal to teammates, if there is a conflict between goals of two agents in current step, one of them chooses to change its goal to prevent a conflict.
- What is the communication strategy and how complex is it?
  The previous answer describes the communication strategy. Because each agent communicates with its teammates via a shared channel, communication complexity in each step will be O(n), in which n is the number of agents.
- How is the following agent features considered/implemented: autonomy, proactiveness, reactiveness?
  There is no central or hierarchical leadership. Each agent chooses a role

according to perceived information from the model and other teammate's goals. Then negotiate with other teammates and choose the final action.

- Is the team a truly multi -agent system or rather a centralized system in disguise?
  Yes it is a decentralized implementation.
- How much time (man hours) have you invested (approximately) for implementing your team?
  About 600 man hours.
- Did you discuss the design and strategies of your agent team with other developers? To which extent did you test your agents playing with other teams?
  We didn't discuss with other teams but we test our team in test matches and improve some defensive capabilities.
- Which programming language did you use to implement the multi-agent system?
  Java.
- Did you use multi-agent programming languages? Why or why not to use a multi-agent programming language?
  No, because of high learning curve and deadline approach.
- How have you mapped the designed architecture (both multi-agent and individual agent architectures) to programming codes, i.e., how did you implement specific agent-oriented concepts and designed artifacts using the programming language?
  Agents are distinct threads having a separate world model. Messages are sent via a shared channel using apltk package.
- Which development platforms and tools are used? How much time did you invest in learning those?
  Nothing.
- Which runtime platforms and tools (e.g. Jade, AgentScape, simply Java ...) are used? How much time did you invest in learning those?
  The communication complexity is not a major problem and doesn't force developers to use a runtime platform.
- What features were missing in your language choice that would have facilitated your development task?
  Java doesn't provide a convenient message broadcast mechanism. Moreover, synchronization points cause bottlenecks, because the independent code lines after the point can't execute and agents should stop decision process until other reach the point.
- What features of your programming language has simplified your development task?
  Using an object-oriented programming language accelerates the design and implementation process. Furthermore, using eis-massim, encapsulates connection handling details.
- Which algorithms are used/ implemented?
  Improved Dijkstra for path finding and some ideas in convex hull for zone creating is used.

- How did you distribute the agents on several machines? And if you did not, please justify why.
  All agents execute on the same physical machine, as the computation cost of the agents is relatively low and as latency of connection to the server was one of our biggest barriers, distribution of agents did not appear to be a wise design decision.
- To which extend is the reasoning of your agents synchronized with the receive-percepts/send-action cycle?
  The time of synchronization is less than 1 percent of each step length.
- What part of the development was most difficult/ complex? What kind of problems have you found and how are they solved?
  The difficulties faced during implementation, mainly stem from goal choice. For example finding a good algorithm to create zones and keep them, required hundreds of test-runs.
- How many lines of code did you write for your software?
  A bit more than 4000 lines of code.
- What is the main strategy of your team?
  Cooperation with teammates and supporting them to realize better goals.
- How does the overall team work together?
  In the beginning of each step agents share their perception to other teammates via a shared channel and then update their perception according to the received perception messages from the channel. In each step, agents define some goals and inter-operate with other agents to choose their final decision according to other agents' goals, autonomously. When agents share their goal to teammates, if there is a conflict between goals of two agents in current step, one of them chooses to change its goal to prevent a conflict.
- How do your agents analyze the topology of the map? And how do they exploit their findings?
  Each agent has a world model that contains every detail captured from perception messages. The most important element in the world model is the map, a weighted graph in which both edges and vertices weighed. The other elements found in the world model are teammate positions, disabled teammates, repairer teammates, teammates who are in a zone, savior teammates, each zone's position and its score, explored water wells and enemies' positions.
- How do your agents communicate with the server?
  Using eis-massim.
- How do you implement the roles of the agents? Which strategies do the different roles implement?
  If an agent has low energy, it recharges itself. If an agent is disabled, it tries to approach helper until it acquires energy. However a disabled repairer tries to repair other teammate first (helper role), and then try to approach other repairers to acquire energy. The zone holder role (all agent types except explorers and saboteurs) first expands the zone by scattering from teammate in the right manner. If it can't expand the zone and

there is a bigger nearby zone, it may leave this zone and join the other. If a zone holder senses a nearby probed node, it will change its position, so that the score of the node is achieved in the current step. If a zone holder doesn't find a zone it tries to find the nearest zone holder agent and create a zone with it. The water well explorer tries to survey new nodes in a way that new non-probed vertices are visited. World Explorer role uses a taboo list to fill the world model quickly. If a savior receives a help message from a nearby teammate who has at most two node distances from it, it will decide to attack the opponent in the next two steps. A chaser gathers opponents' position information and chases them with the goal of attack. When an agent knows that there is an attacker in its node, it chooses defender role and parries itself. Inspector chooses opponent analyzer role when an attack occur.

- How do you find good zones? How do you estimate the value of zones?
  Using following formula:

$$ZoneValue_{i,j} = \frac{\alpha * score_i}{\sum_k score_k} + \frac{\beta * distance_{ij}}{\sum_h distance_{hj}} \quad (2)$$

- How do you conquer zones? How do you defend zones if attacked? Do you attack zones?
  Finding enemies' zones and conquering them was not a part of strategy but agents can defend zones using the savior role.
- Can your agents change their behavior during runtime? If so, what triggers the changes?
  Some criteria trigger agents to change their role, such as: being disabled, being in a zone, being near a better zone, possibility of an attack, being at a low level of energy and existence of a disabled teammate.
- What algorithm(s) do you use for agent path planning?
  Improved Dijkstra.
- How do you make use of the buying-mechanism?
  To improve the attacking power of the team, attacker and savior roles are allowed to use achievements' points and buy items like a shield, a sensor and three sabotage devices, when the team is in the risk of a huge attack.
- How important are achievements for your overall strategy?
  Although Simurgh uses achievement points to make purchases, no specific plan was developed to increase achievement points.
- Do your agents have an explicit mental state?
  Yes, the world model described later.
- How do your agents communicate? And what do they communicate?
  It described later.
- How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?
  There is no central or hierarchal leadership and data doesn't route to any specific agent.

- Is most of your agents' behavior emergent on and individual and team level?
  Mostly individually.
- If your agents perform some planning, how many steps do they plan ahead?
  Agents plan for one step and if the priority doesn't force them to change their goal, the last goal is maintained.
- What have you learned from the participation in the contest?
  This contest was a very unique and valuable experience in design and implementation of MAS for the team. The first lesson learned during the implementation is that multi-agent development methodologies are very diverse, and to make a proper choice of methodology, through investigations are needed. There are many multi-agent programming languages and tools, some of which directly implement the multi agent theory, while others extend existing languages to suit the MAS paradigm. The second lesson learned from the contest was that different communication strategies are needed for decision making and realizing goals between agents living in a same society. Different approaches were examined and tested and choices were made based on the best results.
- Which are the strong and weak points of the team?
  Simurgh could parry very well because if an agent inspects an attacker, it informs the team savior agents quickly. Choosing the helper role instead of zone holder has a higher priority for repairers, so some times our zone broke during the movement of our repairers. We did not put any emphasis on attacks in Simurgh, especially breaking enemies' zones, so other teams could gain score simply when competing against us. The zone expanding algorithm has some weaknesses that cause zone breaking. This is one of the problems we hope to resolve for the next year's contest. Furthermore, for the next contest we hope to improve our attack and design an algorithm for finding opponent's zones'.
- How suitable was the chosen programming language, methodology, tools, and algorithms?
  Gaia proved to be a good base methodology for Simurgh MAS development. It was flexible, clear and understandable. Moreover J2SE did not impose any worrying constraints on us.
- What can be improved in the contest for next year?
  Some changes in the scenario may help.
- Why did your team perform as it did?
  Why did the other teams perform better/worse than you did?
  It described later.
- Which other research fields might be interested in the Multi-Agent Programming Contest?
  Software development methodologies, scheduling using MAS.
- How can the current scenario be optimized? How would that optimization pay off?
  In the scenario, coloring algorithms could be improved. For example,

zones should be able to expand only in 3 or 4 levels of colored nodes, so that if most agents of a team are disabled, the opponent can't make zone expanding to the whole map.

Sorena: Multi-agent system developed with JACK

Alireza Hasanpour, Naser Fallahi, Katayon Khatibifar

Department of Computer Engineering, Faculty of Engineering
Arak University – Arak 38156-8-8349, Iran
h_alireza0111@yahoo.com, n.fallahi87@yahoo.com, k.khatibifar@gmail.com

Abstract. This paper describes the process of developing a multi-agent game system, our team is called Sorena. The system is implemented in a graph like environment. At the beginning of the game, agents are distributed randomly in the game environment. The general goal of the agents is to win the game, which can be achieved by performing actions such as attack, probe, goto and communicating together. The analysis is done by **Prometheus** methodology. The **JACK** platform was used, in order to enable the agents to communicate efficiently. It also has *Design View* facility and *JackBuild* tool for compiling the code. Conclusions are drawn on how the chosen methodology, tools and programming language were suitable.

Keywords: multi-agent system, Prometheus, JACK programming language, action

1 Introduction

Because of the agent oriented nature of this contest and its specific algorithms, we decided to participate. We saw it as a great opportunity to apply some learnt theory in practice and gain useful multi-agent programming experience. The quality of the contest and its participants were also important reasons that motivated us. As a part of our course project, we were encouraged to participate in the contest and challenge our theoretical knowledge in a full capacity. We did build team of three to analyze and implement the multi-agent system which is called Sorena[1] (named after our team's title). The team consists of the following members:

- Alireza Hasanpour, M.Sc student, Department of Computer Engineering, Faculty of Engineering, Arak University, Iran, system developer
- Naser Fallahi, B.Sc student, Department of Computer Engineering, Faculty of Engineering, Arak University, Iran, system analyzer
- Katayon Khatibifar, M.Sc student, Department of Computer Engineering, Faculty of Engineering, Arak University, Iran, system analyzer

[1] Sorena was one of the ancient Iranian legends.

This paper includes an introduction to the Sorena, system analyze and design software architecture, the main strategy of our team. Finally, it presents conclusions on the lessons learnt and the areas that can be improved.

1.1 Related work

N. Yadav et al [Yaday et al.,2010] used Prometheus methodology and JACK programming language to design and implement their multi-agent system for the Multi-Agent Programming Contest(MAPC) of the year 2009. Their most focus was on designing their system with Prometheus methodology whereas our most Concentration in this year's competitions was on implementing our multi-agent system with the use of JACK language.

1.2 Overview of scenario

This section briefly explains the game's scenario which has been adapted from the contest's website. For more details see [MAPC,2011,Behrens et al.,2010]. This year's scenario is about finding water well in Mars. The game environment is a graph-like environment. Each team has ten agents which consist of tow *Repairer*, *Sentinel*, *Explorer*, *Inspector* and *Saboteur*. Each agent performs a specific action. The *Repairer*, *Explorer*, *Inspector* and *Saboteur* agents perform *repair*, *probe*, *inspect* and *attack* actions, respectively. Also all the agents can perform *goto*, *skip*, *recharge*, *buy* and *survey* actions and but only the *Repairer*, *Saboteur* and *Sentinel* agents can perform *parry* action.

The score is computed by summing up the values of the *zones* and the current achievements for each *step*. The *step* is number of game simulation *step* and the *zone* is a subgraph from the game's graph that agents occupy and color it according to the graph's coloring algorithm introduced in the scenario. The achievement is earning money by performing some specific actions for example doing the *probe*, *survey*, *inspect*, *attack*, and *parry* for 5 or 10 or 20 or times, correctly.

2 System Analysis and Design

The Prometheus methodology was used for design of this multi-agent system.

- This methodology can be used for development of intelligent agents which use goals, beliefs, plans, and events [Padgham et al.,2002,Padgham et al.,2004].
- The methodology facilitates tool support. The Prometheus Design Tool (PDT) [Thangarajah et al.,2005], which is freely available and could be plugged in Eclipse.
- Prometheus is mainly used by industrial software developers and undergraduate students [Padgham et al.,2002].

This methodology consists of three phases. Phase one is system specification. This phase focuses on identifying the goals, functionalities of the system, inputs (percepts) and outputs (actions) [Padgham et al.,2004]. Phase two is architectural design. This phase uses the outputs from the previous phase to determine type of the system agent and how they will interact [Padgham et al.,2004]. Phase three is detailed design. This phase concentrates on the internals of each agent and how it will complete its tasks within the whole system [Padgham et al.,2004].

In the first phase, the general goals of the system were specified as they are showed in Fig. 1, which also could be the main strategy of our team. To win the game, agents should make a *zone*, extend and keep it and collecting defined achievements. Attacking opponent team agents and destructing their zones prevents opponent team from scoring which also leads to win the game (see Fig. 1).

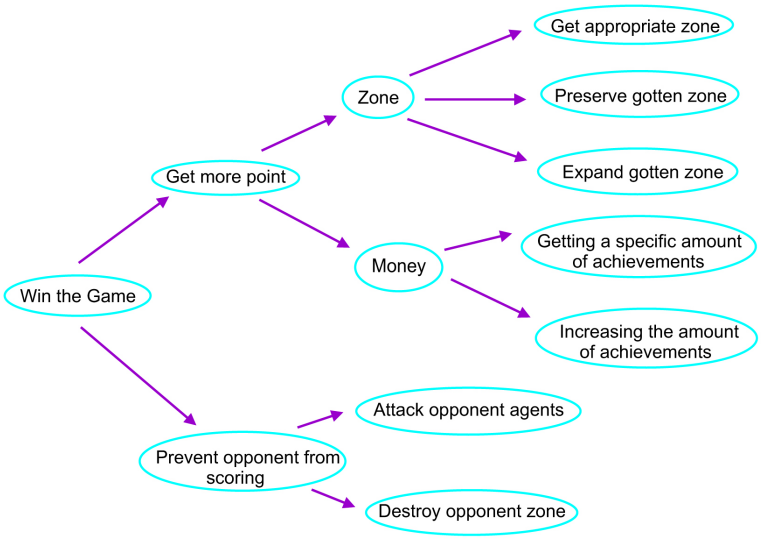


Fig. 1. Our proposed goal model

In order to make a *zone*, the designers decided that the agents being gathered by a *Sentinel* agent in every 15 *steps*. The reason of choosing this kind of agent is that it has wider overview of the environment in comparison with the others. This number of *steps* obtains from running of our code in the test match with other teams and with trial and error way. After the agents came together in neighborhood of the *Sentinel* agent, they walked to different po-

sitions randomly. These processes (gathering and walking) were being done repeatedly. This strategy chases the following goals:

1) Making and extending the zones
2) Unpredictability of the agents
3) Destruction of the opponent zones by random walks
4) Preventing the opponent's *Saboteur* agents in disabling our agents
5) Speeding up the process of repairing disabled agents
6) Agents' movement in all the map

In this multi-agent system communications is simple and obvious the types of communications are described as follow:

- The disabled agent calls *Repairer* agent by sending its position and the word "disabled" to the *Repairer*.
- A Sentinel agent calls other agents by send them the position of its neighbors and word "goto".
- Sending word "parry" to agents by *Inspector* agent

The agents work independently and share the results of their actions. For example after an agent does *survey* action, writes edges' weight in a shared matrix, so all agents could use it. However each agent plays an unique role, which satisfies its independency and autonomy. In the other hand, the agents have to move according the plan, therefore, they are proactive, too. Calling agents by *Sentinel* agent at the beginning of the game and answering to the call are samples of the agents' proactivity. The *Repairer* action is another example of the agent's capability to react in different situations. When a *Repairer* intends to make a *zone* but at the same time receives a message from a disabled agent, its priority reaction will be fixing that disabled agent.

Each agent has been assigned different roles. A number of these roles are exclusive and the base of this game is the agents' coordination. Our plans are decentralized and all the agents in the map communicate with each other, so they aren't controlled centrally.

We have spent 140 man hours for analyzing and designing this multi-agent system approximately, and it took us the same amount of time to implement it. Through this project we have discussed our plans, analysis, design and implementation with other participants. We did some preliminary test matches with *Nargel* team before the main contest.

3 Software Architecture

The JACK programming language was used to implement the agents. JACK is a totally multi-agent based programming language that provides many facilities for agent oriented concepts, like agent communication [Busetta et al.,1999] [Winikoff,2005]. Also JACK's graphical development environment simplifies developing of agent based applications [JACK-Manual, 2005]. Learning

how to work in this environment is simple for programmers who are familiar with object oriented environments. In addition of agent communication, this language provides the following facilities for programmers:

- A simulator to show agents communication
- *Design View* facility, which can model and present the designed system by graphical elements
- Capability of been compiled by *JackBuild*

Due to the technical difficulties, we could not connect to the remote server via JACK, so we added our communication agent codes to the organizers testing agent codes that existed in their web site [MAPC,2011]. In order to do this, we inherited testing agent from Agent class of JACK. So the agents were able to connect to the MASSim server [Behrens et al.,2009] as well as speak with each other through JACK.

In order to allow the agents to communicate with each other, first we implemented the capability of sending and receiving messages for all the agents, and then compiled it by JACK compiler. The Fig. 2 presents the *Saboteur* agent's code written in JACK agent programming language. In this code, the *PostEventandWait* method has been used to send a message to the other agents and waits for their respond. The *agentname* and the *Msg String* are parameters of this method.

```
1  package massim.javaagents.agents2011;
2
3  public agent SimpleSaboteurAgent extends Agent {
4      #handles event Utterance;
5      #sends event Utterance ev;
6      #posts event StartSaboteur sf;
7      #handles event StartSaboteur;
8      #uses plan RcvSaboteurMsg;
9      #uses plan SendSaboteurMsg;
10
11     public SimpleSaboteurAgent(String name)
12     {
13         super(name);
14     }
15
16     public void converse(String agentname,String Msg)
17     {
18         postEventAndWait(sf.startsaboteur(agentname,Msg));
19     }
20 }
```

Fig. 2. Saboteur agent's code

One of the JACK compiler's features is that after compiling codes it produces files with *java* type. Then we run these codes to get files with *class* type. And finally, we added these files to *Javaagent* package to build communication between the agents.

One of the disadvantages of this language is the lack of technical supporting. We had problems in connecting to the server, so we asked help from different people including JACK developers, but we did not get any responses to our questions.

Dijkstra was the main algorithm that we used for agent's path finding. This algorithm uses an adjacent matrix in order to find the correct path through the graph [Neapolitan et al.,2004]. As we have been completing this matrix through the game, the agents were able to find a better path at the later *steps* of the game.

We have used simple variables, linked lists and arrays and controlled our plans by simple variables. For example at the beginning of the gathering plan of agents, we initialized a variable, and after reaching an acceptable goal its value was changed.

Our agents are synchronized with the receive-percepts/send-action cycle. Server sends perception to the agents. They receive the perception and interpret them to update its internal world model. Each agents has three general goals: pre defined goals for every agent, collaborative goals with other agents (e.g.make a zone) and proper reaction against environmental condition. Based on the produced beliefs a proper goal will be selected and then by the communicating with other agent a plan will be selected. Accordingly, a proper action will be selected too(see Fig. 3).

Concerning the small number of agents (10) for each team and constant roles for each agent and using computers with high performance, we did not have to distribute agents on the several machines. Computer that we used for the contest was able to run the testing server for two teams at the same time and shows their map and agent walking without any interruption.

Learning JACK took us one month. The most difficult part was connecting to the server. We wrote 2000 lines for our software approximately.

4 Strategies, Details and Statistics

Each agent acts individually and according to a predefined plan. However, in some situations like lacking of a complete adjacent graph, they might walk randomly to a neighbor node to reach their goal. According to our strategy, agents should know the environment in order to walk towards the *Sentinel* agent. In order aiming this goal, we defined two shared lists which could be accessed by all the agents. For getting to know the environment all the agents use the *survey* action and save position of the nodes which have been surveyed in one of the lists and consequently weights of the edges are saved in a matrix. First, all the matrix's cells are filled by a very large number and after each *survey* action these cells which present the edges' weight change. It

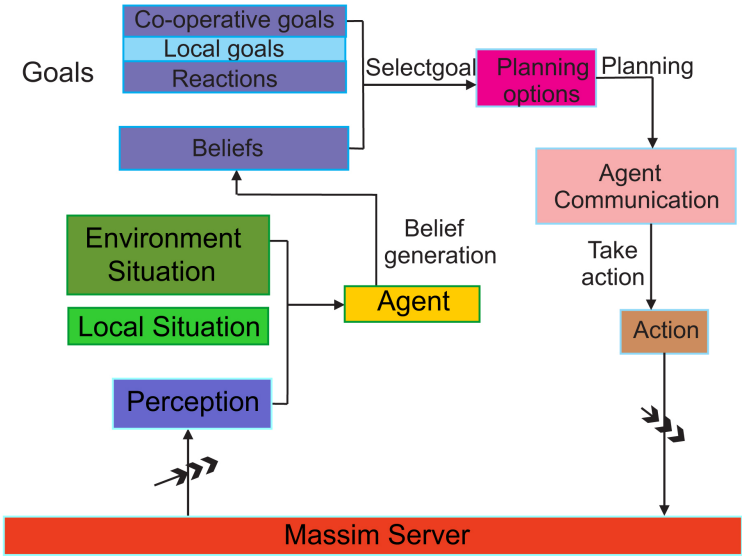
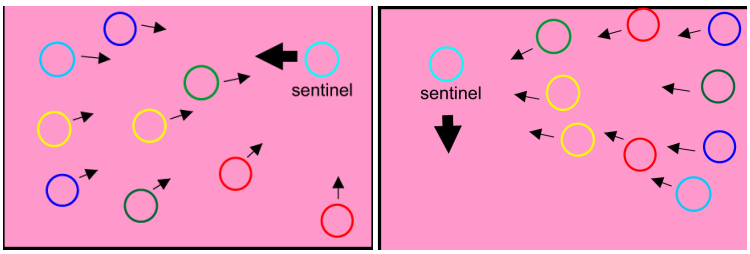


Fig. 3. Software Architecture

should be considered that for a better path finding and coming over the bad result of Dijkstra algorithm in some situations, we fill cells related to visited edges by a predefined constant number, too.

While *Sentinel* agent calls other agents to gather on its neighboring nodes, each agent in its movement path does its role. For example, if *Saboteur* agent faced an opponent agent on its way to the destination node will do the *attack* action. The *Explorer* agent also *probes* every node on its path. Meanwhile if an agent become disabled by the opponent *Saboteur* agent, it stops moving and sends a message to the *Repairer* agent which contains its node id and a string value to say it is disabled. The *Repairer* agent as soon as receive this message and seeing the word ”disable” in it, will run Dijkstra algorithm with the position of disabled agent and itself as destination and source respectively, and this algorithm sends a list of nodes -which are the path from *Repairer* agent to the disabled agents- to the repairer. By repairing the disable agent, these two agents will continue their move to the specified positions. If the agents move and communicate properly, they could extend their *zone* in all 15 *steps*. We are working on improving our strategy, in order to score higher in the next year contest. However, as the *Sentinel* agent moves around the map in every 15 *steps*, it calls the agents to the map’s nodes(see Fig. 4). With this strategy we could attack the opponent’s *zones* and destruct them and our gained *zone* had high points (more than 50 from our point of view) in some *steps* and it was less than 50 in some other *steps*.



(a) (b)

Fig. 4. Agents movement

The achievements' points are important, so we try to gain as much as money that we could and save it in order to get more scores in each *step*.

All of our agents are in the same level and none of them get orders from the others. As explained in section 3, we appoint the *Sentinel* agent to call the others and gather them together. The following piece of code shows how the *Sentinel* agent recalls the other agents to its neighboring nodes(see Fig. 4).

As it can be seen in the code(Fig. 5), the agents are distributed in the *Sentinel* agent's neighboring node. For example, if the *Sentinel* agent has 3 neighboring nodes, the other agents as a group of 3 accommodate themselves on each of the neighboring nodes. When the variable *firstgoal* is filled with the string value "goto", this code segment will run. The variable *ratio* keeps the number of neighboring teammate agents of *Sentinel* agent which is calculated by the division of total number of agents (i.e. 10) to the number of neighboring nodes of *Sentinel* agent. Then the *Sentinel* agent sends the position of its neighboring nodes to the other agents by calling "Converse" method. Of course the method "Converse" will call *PostEventandWait*, too(see Fig. 2). When the agents receive the message, they move toward the *Sentinel* and also perform their specific roles as they move.

5 Conclusion

Participating in the MAPC was a good opportunity for us to improve our agent oriented programming knowledge and compare our skills in this field with different teams from other countries. Furthermore we gained very useful insights which could be used to guide other students who want to learn this kind of programming theoretically or practically.

Hard working of members and acceptable functionality of determined strategy are strong points of our team. Although our weak strategy in keeping zones, some members who left the team during design phase and also spending too much time on learning JACK are weak points of our team. We

```
if(firstgoal=="goto" && !ggetName().equals("SorSen2"))
 {
        int ratio=10/neighbors.size();
        for(int i=0;i<neighbors.size();i++)
        {
            for(int j=ratio*i ;j<ratio*(i+1);j++)
            {
                if((agents.get(j).toString()).equals("SorSen1"))
                    continue;
                converse(agents.get(j),"goto"+neighbors.get(i));
                println(agents.get(j)+" goto "+neighbors.get(i));
            }
        }
        for(int i=0;i<10%neighbors.size();i++)
        {
            if((agents.get(9-i).toString()).equals("SorSen1"))
                continue;
            converse(agents.get(9-i),"goto"+neighbors.get(i));
            println(agents.get(9-i)+" goto "+neighbors.get(i));
        }
        firstgoal="";
 }
```

Fig. 5. Calling agents by Sentinel agent

chose the Prometheus methodology and JACK as programming language. JACK is an efficient language and in addition, Prometheus produces JACK pseudocode and there is a consistency between Prometheus and this language [Bergenti et al.,2004,Yaday et al.,2010]. One area that we are working on it, is improving the path finding algorithm. Our algorithm chooses the shortest path that may include more *steps*, but we are developing another algorithm which base on the number of *steps* in each path and we also working on using the agent-oriented concepts of JACK [JACK-Manual, 2005] for example *Database* and *Capability* instead of using form simple variables.

The scenario has been changed a lot at this year and there is room for improvement. We propose to add the capability of changing the agents' role by spending money. We also suggest to hold the contest in different countries and then each country's finalist compete against other countries opponent.

Due to the short preparation time and losing tow team members during the project we did not get a good result. We take this opportunity to congratulate the winning team.

Acknowledgments. We would like to acknowledge the help and support of the students who have participate in the MAPC in the previous years for their comment and feedbacks.

References

[Yaday et al.,2010] N. Yadav, C. Zhou, S. Sardina, and R. Rönnquist, *A bdi agent system for the cow herding domain*, Annals of Mathematics and Artificial Intelligence **59** (2010), 313–333.

[MAPC,2011] *the Multi-Agent Programming Contest 2011 organizer*, 2011, Available from: http://multiagentcontest.org

[Behrens et al.,2010] Tristan Behrens, Mehdi Dastani, Jürgen Dix, Michael Köster, and Peter Nov'ak, *The multi-agent programming contest from 2005-2010: From collecting gold to herding cows*, Annals of Mathematics and Artificial Intelligence **59** (2010), 277–311.

[Padgham et al.,2002] Lin, P. and W. Michael, Prometheus: a methodology for developing intelligent agents, in Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 12002, ACM: Bologna, Italy.

[Padgham et al.,2004] Lin. Padgham and Michael. Winikoff, *Developing intelligent agent systems : a practical guide*, Wiley, Chichester :, 2004 (English).

[Thangarajah et al.,2005] John Thangarajah, Lin Padgham, and Michael Winikoff, *Prometheus design tool*, Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (New York, NY, USA), AAMAS '05, ACM, 2005, pp. 127–128.

[Busetta et al.,1999] P Busetta, Ralph Rnnquist, A Hodgson, and A Lucas, *Jack intelligent agents - components for intelligent agents in java*, AgentLink News Letter **2** (1999), no. January, 2–5.

[Winikoff,2005] Winikoff, M., Jack Intelligent Agents: An Industrial Strength Platform Multi-Agent Programming, R. Bordini, et al., Editors. 2005, Springer US. p. 175-193.

[JACK-Manual, 2005] Agent Oriented Software Pty. Ltd., *Jack intelligent manuals*, 2005, Available from:http://www.agent-software.com

[Behrens et al.,2009] Tristan M. Behrens, Jrgen Dix, Koen V. Hindriks, *The Environment Interface Standard for Agent-Oriented Programming Platform Integration Guide and Interface Implementation Guide*, IfI-09-10, December 2009, 23 pp.

[Neapolitan et al.,2004] R.E. Neapolitan and K. Naimipour, *Foundations of algorithms using c++ pseudocode*, Algorithms Series, Jones and Bartlett, 2004.

[Bergenti et al.,2004] F. Bergenti, M. Gleizes, and F. Zambonelli, *Methodologies and software engineering for agent systems : the agent-oriented software engineering handbook*, Kluwer Academic, Boston [u.a.], 2004.

Short Answers

1.1 The quality of the contest and its participants were reasons that motivated us.
1.2 This contest was our course project.
1.3 Our team name is Sorena.
1.4 Our team consisted of three students -one system developers and two designers- that two of them were postgraduate students, one was undergraduate student.
1.5 All members of the team are educated in software engineering.
2.1 We used Prometheus methodology as: in the first phase, called system specification, we identified main goals and requirements of the system. In the next phase named architectural design, we specified the way in which agents communicate with each other or with server. And in the last phase of this methodology means detailed design, we analyzed behavior of the agents in all parts of the system.
2.2 No, We had a decentralized main strategy that all of the agents were gathered and then distributed in the map repeatedly.
2.3 Our agents communicated with each other for gathering, repairing and attacking by the simple strategy.
2.4 Because each of agent kinds had an unique role, which satisfies its the autonomy. Calling agents at the beginning of the program is a sample of proactiveness. Reacting of the Repairer to the message of a disabled agent shows reactiveness.
2.5 Yes, it's a truly multi-agent system.
2.6 We spend 140 man hours for implementing this system approximately.
2.7 Yes, we discussed with Nargel team and played some testing matches with them.
3.1 We used JACK programming language.
3.2 Yes, this language has many capabilities like facilitating agent communication.
3.3 We couldn't use all agent oriented concepts of JACK, so we used simple variables, arrays and linked lists to simulate these concepts.
3.4 We used JACK agent platform.
3.5 We used JACK runtime Environment and invested about one month to learn it.
3.6 Lack of a strong supporting for this language was our biggest problem.
3.7 Communication between the agents simply and quickly.
3.8 Our main and most important algorithm was Dijkstra which was used to path finding.
3.9 Because of small number of agents and using a computer with high performance, we did not distribute agents on several machines.
3.10 Our agents are synchronized with the receive-percepts/send-action cycle. After receiving percept from server, each agent analyzes it, prepared it plan then choose the best action to reach the goal and sent the action to the server.

3.11 We had problem in connecting to remote server by JACK, so we added our JACK agent communication codes into the organizers testing agent codes.
3.12 We wrote 2000 lines for our software approximately.
4.1 We gathered all agents by one specified agent and then dispersed them repeatedly.
4.2 Communication of agents has described in part 3-2. Also some variables like adjacent matrix of graph and list of surveyed nodes were shared between all the agents.
4.3 We converted graph of the map into a matrix, so we could use Dijkstra algorithm for path finding.
4.4 As we used codes of testing agents, we did not handle manner of communication of agents with the server.
4.5 During to do making *zone* plan, all the agents perform a specific role. For example, the Saboteur agent attacks the opponent agents.
4.6 We built a *zone* by gathering agents and extended it by moving agents to neighbor nodes and gathered them after some *steps* again.
4.7 We described in 4.6 how we made a *zone*, and because of gathering agents repeatedly, we could keep it, however it's a weak strategy. Also by moved agents to neighbor nodes, we could attack others' zones.
4.8 Our agents had constant behavior during the game.
4.9 We used Dijkstra algorithm.
4.10 We used score of achievement and did not buy anything.
4.11 We try to gain as much as money that we could and save it in order to get more scores in each *step*.
4.12 No, they do not have such state.
4.13 See 4.2.
4.14 All of our agents were in the same level and none of them got orders from another.
4.15 Each agent acts individually and according to a predefined plan.
4.16 By our strategy the agents were called every 15 *steps*, so they could build a *zone* in 15 *steps*.
5.1 Attending in this contest was a good opportunity for us to improve our agent oriented programming knowledge.
5.2 Hard working of the members and acceptable functionality of our strategy were strong points of our team. But weak strategy in keeping our zones was the negative point of our team.
5.3 We are content of using JACK programming language, Dijkstra algorithm for path finding, Prometheus methodology and PDT tool of it, although we could use them in a better way or optimize them.
5.4 Adding capability of changing agents' role by spending money.
5.5 Due to the short preparation time and losing tow team members during the project we did not get a good result, but the best teams by good strategy and spending sufficient time could get acceptable results.
5.6 This contest seems to be complete.
5.7 We think this scenario is the most interesting and exciting one between all the previous scenarios and we congratulate organizers of this contest.

TUB Team Description

Axel Heßler, Thomas Konnerth, Pawel Napierala and Benjamin Wiemann

Technische Universität Berlin, Germany

Abstract. We describe our contribution to the Multi-Agent Programming Contest 2011, which has been prepared by students and researchers of the DAI-Labor at TU Berlin, Germany, using the JIAC V agent framework and the agile JIAC methodology.

1 Introduction

Our team for the Multi-Agent Programming Contest 2011 [Behrens et al., 2010] is called "TUB" and has been developed in the course "Multi Agent Contest"[1] by the following students: Pawel Napierala and Benjamin Wiemann, supervised by the following agent researchers: Thomas Konnerth and Axel Heßler. We have invested 640 hours approximately to create the contest version of our system finished in third place.

2 System Analysis and Design

During the development of our team we roughly followed the JIAC methodology, which can be described as a bottom-up and agile methodology. We start with domain analysis, which is to build a first *ontology*: find the concepts of the domain, their structure and relationships to each other: agents, own team, opponents, nodes, edges, visited, probed, surveyed, weight. Parallel to that, the main requirements are collected: just one requirement in the case of the contest, to maximise the score.

In Figure 1, the first version of the domain model is shown. A *World* consists of *Vertices* and *Edges*. The world is occupied by *Bots* that are organized by *Teams*. Under certain circumstances bots form *Zones*, which are collections of vertices that are populated by only bots from one team. During many iterations the ontology is extended by more concepts, properties and associations. The ontology of the TUB team is directly transformed into Java classes. The transformation to OWL would have been possible but offers no added value in this scenario.

As a second step the methodology says: make a *role model* and a *user interface (UI) prototype*. A role is specified by the a number of capabilities or behaviours and the relationships to other roles. Identifying the roles was an easy task because they are easily collected from the scenario document. We

[1] Project 0435 L 774 at TU Berlin, Germany

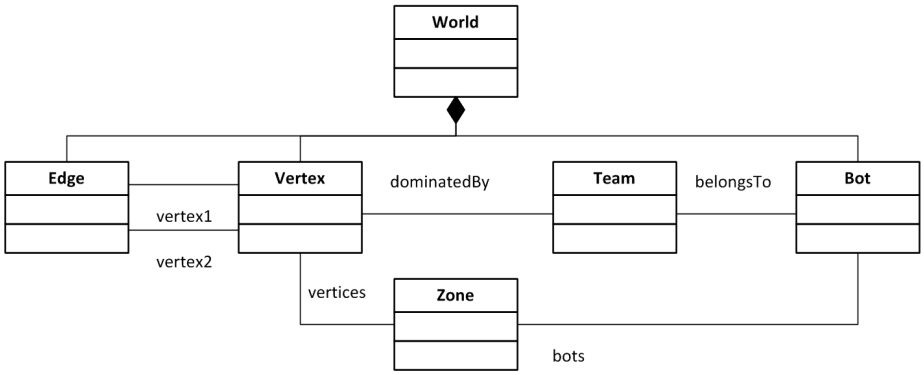


Fig. 1. TUB domain model (first version)

then assigned simple and basic capabilities to the roles. As many of them were identical in each role, we created the generalised role of the *Mars invader*, which is a collection the capabilities that all roles share, such as surveying, charging and moving. All other roles inherit from the invader role and add special capabilities such as probing, introspecting, and so on.

The role model was subject to many iterations. In Figure 2, an intermediate version of the role model is shown that is very close to the final role model. In principle, every contest agent in this role model could take every role (the *ContestAgentRole*), but during this contest the roles are static properties given by the contest server to every agent in the team. Common capabilities (*goto, survey, buy, recharge*) are implemented to the *DefaultDecisionBean* component. Special capabilities (*probe, inspect, attack, repair*) are implemented in the corresponding role specific component (e.g. *ExplorerDecisionBean* or *SaboteurDecisionBean*). Every agent instance has a specialization of the *ServerCommunicationBean* component that the credentials for authentication. Finally, every agent is instantiated once on the *ContestNode*, which provides the infrastructure for acquaintance and inter-agent communication. The role model has been generated with the help of the *AgentWorldEditor (AWE)*, which is part of the JIAC V tool suite *Toolipse*. The AWE generates configurations for all agents and agent nodes that are used by the JIAC V runtime at startup.

The UI prototype is a simple visualisation of the world graph. The problem here was that we could not find a solution to draw the graph in a repetable way during preparation. As a workaround we patched the contest server to send the coordinates that project the graph to a grid as used by the monitor tool. The next step is *implement*ing the simple and basic behaviours and then *evaluate* their function. After several iterations, until the basic actions can be reliably achieved by the agent, more complex capabilities are added,

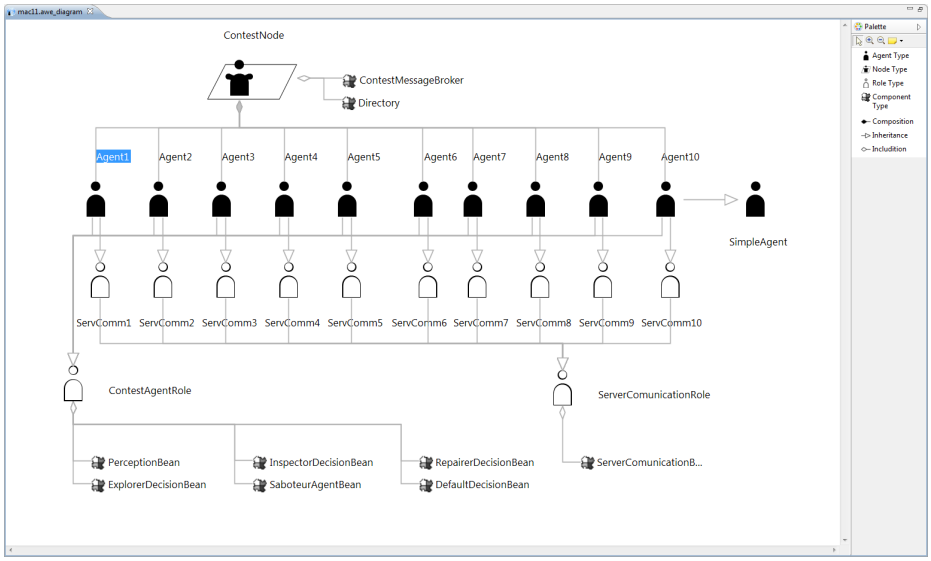


Fig. 2. TUB role model

such as finding the most promising node to occupy or calculate the shortest or fastest path to an arbitrary node, and so on.

Our system is completely decentralised. Every agent is capable of taking every role and doing everything (depending on the role). Also, the Zoning can be done be every agent, but we have decided to let only one agent calculate the Zoning if necessary. This agent is selected among and by interested agents that want to know where to position in the zone, using a simple voting protocol. The result is then calculated by the selected agent and shared with the other interested agents.

Regarding the communication strategy of our team, we follow our successful approach used from 2007 to 2009 (e.g. in [Heßler et al., 2009]) to distribute all perceptions and intentions among all other agents, where we could reach an appreciable enhancement of the team performance. We know that this approach does not scale very well as the number of perceptions and intentions sent around is $2n * (n - 1)$ per cycle, but within a small agent team it works perfectly.

When it comes to coordination aspects we distinguish between explicit and implicit coordination. Implicit coordination can be achieved when the agents share their intentions. This notion of intention is often misunderstood when discussing the approach in the agent community. The intention in our case more often reflects a perform or achievement goal than the action that the agent has decided to execute. Taking the intentions of other agents into account, the actual agent can adopt the intention when it has a better

utility or even dismiss its own decisions in case other agents will perform better. We have built only a few explicit role-based coordination strategies into our agents, e.g. among explorers and between two roles, e.g. the collaboration between inspector and saboteur, or the unhealthy agent requesting the nearest repairer.

We have implemented general agent attributes such as autonomy, proactiveness and reactiveness as follows: JIAC V agents have their own thread of control and decide and act autonomously, i.e. each agent decides for itself, what general action it should take (self-protection, role-specific). We see the agents with low health level proactively seeking the repairer's help using a simple request, whereas probing or surveying has been implemented as a simple reactive behaviour: if the node is unprobed then probe.

For testing the team performance we have developed another internal team. Our sparring dummies employed simple swarm algorithms for coordination and were used for testing the real team.

3 Software Architecture

We have used the JIAC V agent framework to implement the contest multi-agent system (MAS) of our TUB team. Every agent consists of a number of components: memory, server communication, inter-agent communication, perception processing and one component per role.

We used the memory of the agents to store the world model for the contest, i.e. the graph, in a central place for each agent. All agents forward their perceptions to all other agents via the communication. Thus each agent has a complete model of the world as it is currently known to the team. This model can then be accessed by the role specific components, the zoning calculation or accessed for debugging.

The server connection and the processing the of the server messages are handled by a dedicated component that receives the perceptions, processes and stores them in the agents memory, and sends the action of the agent back to the server each turn.

As the JIAC V communication mechanism allows the agents to register for arbitrary messageboxes and groups, we could implement the inter-agent communication in a very efficient way. Messages to individual agents can simply be send to the corresponding message box, messages to the team are send to a group address via multicast. Both types of messages can be send with a simple method call.

For our agent researchers the contest is always an excellent reliability benchmarking of the framework, and also a test case for teaching principles of agent programming. We left out the declarative agent language because we have experienced longer training periods and fewer usable code. The Eclipse IDE with several useful plugins is the main develoment tool although some developers prefer a simple editor with syntax highlighting and a separate build system.

The system can be distributed over several machines if available, without changing any line of code, even at runtime. This is one of the features of the JIAC agent framework [Hirsch et al., 2009] that is usually used for MAS (self-)administration [Kaiser et al., 2010,Thiele et al., 2009]. However, we did not use this feature due to a lack of available hardware during the contest.

As far as algorithms are concerned we used the opportunity to implement two different pathfinding algorithms: Bellman-Ford and Dijkstra. The zone calculation algorithm has been adopted from the server source code and is used for Zoning. Synchronization: reasoning is triggered by new perceptions. The best action for the current situation is calculated. We tried to have the reasoning cycle complete before deadline.

4 Strategies, Details and Statistics

We intended to have a twofold stategy for our agent team: First, the single agent follows a simple, straightforward achievement collection strategy, e.g. if a node is not yet probed, the agent has the the capability to probe and is situated on the unprobed node then it will try to probe until the probe action has been successfully achieved. And second, the agent follows an algorithm that we haved called “Zoning”, i.e. two or more agents try to create and extend a zone in order to achieve the maximum zone score gain, i.e. to occupy the best zone (relative to given current positions). We try to localy optimize zonescore in the next move. Thus the zone is not defended as such. If an agent has to move, the best possible neighbour is chosen. Secondary strategy: to attack and repair as good as possible.

Explorers try to explore (probe, survey) the whole map, closest unknown node first. If the other explorer is already underway, the explorer chooses the next one. When all nodes/edges are explored the explorers contribute to Zoning. Exploitation is done by using the best direct neighbours for the next zone (MinMaX-style). Repairers repair the closest damaged bot. in Case no bot is damaged they do some Zoning. Saboteurs sabotage the closest active opponent. If all opponents are inactive they do Zoning. Sentinels always build zones. And, finally, inspectors inspect the closest opponent. If the other inspector is already underway, the inspector chooses next opponent. When all opponents are inspected, the inspectors contribute to Zoning. We have implemented the decision between actions as a state machine for gamestates (Exploring, Repairing, Zoning, etc.).

In general, all roles follow some simple rules: they evade when the opponent saboteurs come close. If an agent is damaged it approaches the closest repairer. They always recharge when possible and when adjacent to non-surveyed edges they help in surveying.

The “Zoning” algorithm works as follows:

- All agents that decide to participate in the zoning publish their intention to do so with a multicast message to all other agents.

- One agent (usually the first to publish its intention) volunteers to perform the zoning calculations.
- After a certain deadline has passed (usually about 30% of the time available for the turn), the zoning is calculated for all agents that intended to participate.
- The current zone score is calculated.
- For all participating agents, all possible moves and the corresponding new zone scores are calculated.
- The combination of moves that lead to the best zone score for this turn is selected.
- All participating agents are informed, which move they should execute.
- Each agent decides, whether it is "safe" to execute the move, or if it is better to avoid e.g. opposing attackers.

The decision to have only one agent perform the zoning calculation was made, because of the complexity of the calculation. As the calculation is rather expensive, it seemed more appropriate to switch to a hierarchical approach. However, the decision about the actual move is still left to the individual agent, as it does have to take the positions and possible moves of opposing agents into account. However, if we had incorporated this into the calculations of the zoning algorithm, it might have improved the performance of the zoning.

5 Conclusion

The TUB agent team solved the problem of exploring and exploiting the Mars resources. We used the contest to improve our skills in teaching agent programming principles, i.e. the two researchers of the team had chosen a hands-off approach when it came to implementation. We also appreciate the new Mars scenario and higher scenario complexity.

References

[Behrens et al., 2010] Behrens, T., Dastani, M., Dix, J., Köster, M., and Nov'ak, P. (2010). The multi-agent programming contest from 2005-2010: From collecting gold to herding cows. *Annals of Mathematics and Artificial Intelligence*, 59:277–311.

[Heßler et al., 2009] Heßler, A., Küster, T., Niemann, O., Sljivar, A., and Matallaoui, A. (2009). Cows and Fences: JIAC V - AC'09 Team Description. In Dix, J., Fisher, M., and Novák, P., editors, *Proceedings of the 10th International Workshop on Computational Logic in Multi-Agent Systems 2009*, volume IfI-09-08 of *IfI Technical Report Series*. Clausthal University of Technology.

[Hirsch et al., 2009] Hirsch, B., Konnerth, T., and Heßler, A. (2009). Merging agents and services — the JIAC agent platform. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 159–185. Springer.

[Kaiser et al., 2010] Kaiser, S., Burkhardt, M., and Tonn, J. (2010). Drag-and-drop migration: An example of mapping user actions to agent infrastructures. In *Proceedings of 1st International Workshop on Infrastructures and Tools for Multiagent Systems (ITMAS 2010)*, Toronto, Canada.

[Thiele et al., 2009] Thiele, A., Kaiser, S., Konnerth, T., and Hirsch, B. (2009). MAMS service framework. In *SOCASE '09: Proceedings of The Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE) Workshop*, Lecture Notes in Computer Science (LNCS). Springer.

Short Answers

1.1: Our motivation was to employ and evaluate the JIAC V framework. Furthermore we wanted to teach agent oriented principles to the students.
1.2: The team was developed in a project course for master students.
1.3: The name of the team is "TUB".
1.4: We had 2 bachelor students working on the team who were supervised by to agent researchers.
1.5: We come from the field of Agent oriented technology.
2.1: We used the JIAC methodology.
2.2: The solution is based on sharing all knowledge between all agents, thus allowing them to come to identical solutions about the best course of action while still having decentralized decisions. However, for the zoning calculation, we use a centralized approach, as these calculations are rather expensive to compute.
2.3: The communication works in two steps. In the first step, all agents share their perceptions with all other agents via multicast messages. In the second step, once an agent has committed to a course of action, it informs all other agents about his actions. If actions collide (i.e. two agents try to probe the same node), one of the agents (usually the one that was slower to publish its intention) selects another action. Thus for each cycle we have one Multicast message (n-1 individual messages) and one normal message per agent, resulting in 2n*(n-1) messages for n agents.
2.4: Each agent decides autonomously about its course of action. It reacts to the actions of other agents and corrects its decisions if collisions occur.
2.5: As the agents make their decisions autonomously and do not rely on a central instance for coordination, we regard it to be a true decentralized system.
2.6: We invested approximately 640 hours of work.
2.7: We tested our team in the training matches that were organized before the actual tournament started.
3.1: Our agents were implemented with the JIAC V framework which is Java based.
3.2: We did not use the multi-agent programming language JADL that comes with the JIAC V framework, as we did not have enough time to train the bachelor students in this language. Furthermore, we have made the experience, that most work on the contest requires work on the algorithms, rather that work on "agent" problems such as coordination.
3.3: For the communication and coordination of the agents, we used the appropriate JIAC V concepts. The individual functionalities for the roles of the agents were implemented in dedicated components for each role.
3.4: Most of the work was done in Java with help of the Eclipse IDE. The Java implementations relied on the JIAC V framework. It took the bachelor students approximately two to three weeks to become familiar with this framework (they were already familiar with Java and Eclipse).
3.5: As a runtime platform we used JIAC V.

3.6: No features were missing.
3.7: Infrastructure and communication services were readily available. Furthermore, the creation and configuration of the agents was simple and efficient.
3.8: We tried the Bellman-Ford and Dijkstra algorithms for path finding. The final solution however was based on the A-star algorithm, as the other algorithms proved to be to costly to be calculated by all agents.
3.9: We did not distributed our agents across different machines, as our one server was more than capable to handle ten agents.
3.10: The decision making was triggered by the receiving of perceptions and was finished before the timeout for each cycle.
3.11: The most complex problem of the contest for us was the balancing of repair- and attack-actions. Furthermore the zoning algorithm for calculating the optimal placement of the agents proved to be rather complex when opposing agents were involved.
3.12: We did write approximately 8000 lines of code including comments.
4.1: Our agents try to optimize achievement and zoning points.
4.2: The agents share their perceptions and intentions.
4.3: The agents try to probe and survey all nodes and edges of the graph. The results are propagated to all agents.
4.4: We have implemented our own connection to the server and our own parser for the perceptions.
4.5: Each role is implemented in a dedicated component for the agents which is later configured into an agent. The explorers try to explore the whole graph as fast as possible. The repairers try to keep all teammates alive. The attackers try to disable the closest opposing agents. The inspectors make one pass to inspect all opposing agents in order to get the achievement points. All agents that have no role specific tasks left try to build a maximal zone.
4.6: For our zoning algorithm, all agents that want to participate in a zone communicate this. Then the resulting zones for all possible moves of these agents are calculated and the best zone is selected, resulting in the agents to execute the appropriate moves. The zone score is calculated based on the known values of the nodes. Unknown nodes are valued with one point.
4.7: We do not explicitly attack or defend zones. Our attackers simply attack the closest opposing agents.
4.8: Explorers contribute to zoning when they have finished the exploration. Repairers contribute to zoning if no teammate needs repairs. Attackers contribute to zoning if all opponents are disabled. Inspectors contribute to zoning of all opponents have been inspected. Repairers and Attackers may return to their default behavior if it is applicable again.
4.9: The final team uses the A-star algorithm.
4.10: The attackers buy attack-power and health.
4.11: We try to maximize the earned achievement points.

4.12: Each agent has a central fact based (based on Linda like tuple space). The content of this fact base constitutes the mental state of the agent.
4.13: The agents communicate via messages that are equivalent to inform speechacts. They communicate their perceptions and intentions.
4.14: The organization of our agents is decentralized and role-based.
4.15: Individual behavior of the agents is programmed. Team based behavior is emergent.
4.16: Our agents do not plan ahead.
5.1: We underestimated the potential of aggressive attackers. Furthermore, algorithms with a high computational cost like the Dijkstra algorithm are applicable, but are too costly if all agents calculate them at the same time. This could be delegated to a specialized agent in future contests — the so called path finder.
5.2: Our zoning algorithm worked good. However, we did not find the optimal strategy for our attackers and repairers.
5.3: The development of our team worked fine.
5.4: The organization of the contest was very good. The scenario was also good.
5.5: Although we implemented two different teams during development for testing purposes, we underestimated the effectiveness of aggressive play. During the training matches we tried to improve our attackers, but were unable to make them truly competitive with the winning team.
5.6:
5.7: The balance of achievement points and aggressive play styles can be modified in order to give the contest a different character. This is however not so much of an optimization. It rather is a way to keep the contest interesting.