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,leos.molas,emm.montenegro,fsisul,jmtorresluc}@gmail.com,
{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) [2] 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 [8] [9] [3] via defeasible logic programming (DeLP [6]) in a BDI based agent [1], 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 made several changes and additions necessary and precluded the use of design methodologies specific to multi-agent systems. Nevertheless, our approach was more than

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

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 [10], and is explained in detail in further sections.

The agents' behaviour 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[7]. Although the initial plan was to run tests against other agent teams prior to the competition, time constraints made this impossible.

3 Software Architecture

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 [6] was used for the deliberative

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

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.

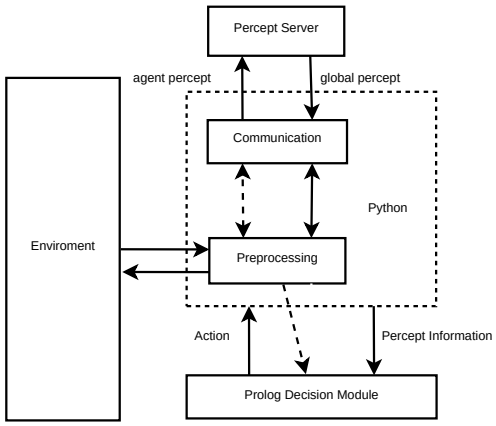

Fig. 1. Agent architecture in a flow chart-like diagram. Dashed arrows represent process flow, solid lines represent data flow.

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

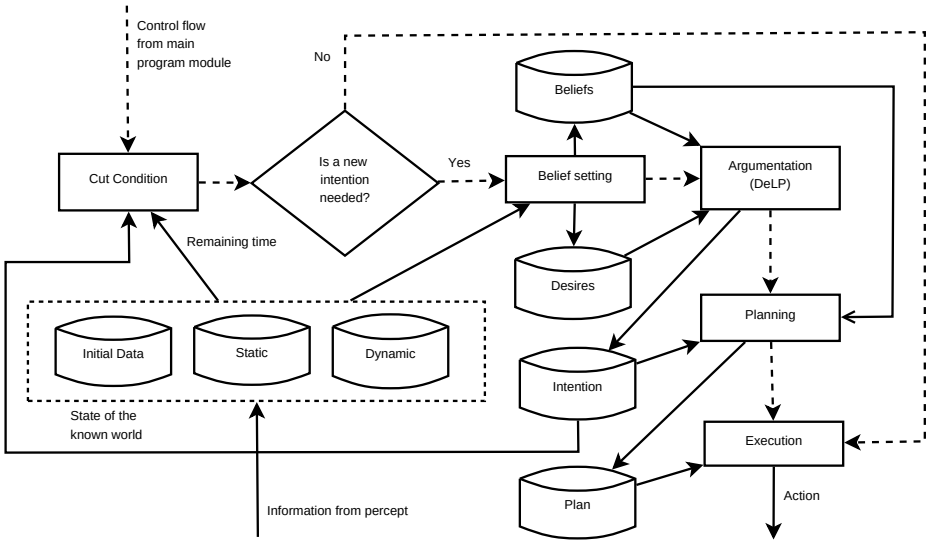

Fig. 2. Architecture of the Prolog decision module

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.

On startup, an agent is passed the information needed to authenticate with the MASSim server and some configuration options such as network addresses and ports, whether the PS and the argumentation framework are to be used or not, and logging verbosity levels. 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.

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.

If so configured, the agent will then send the percept to the PS and await the global percept containing the remainig information perceived by the team. The global percept is merged with it's 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 3.2 and 3.2 show example percepts after parsing, before being sent to the percept server.

```
{ 'surveyed_edges' : [ ],
  'vis_verts'      : [ { 'name': 'vertex65',
                         'team': 'none' },
                       ...
                       { 'name': 'vertex141',
                         'team': 'B' }
                     ],
  '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.

```
{ 'total_time':          2000L,
  'last_step_score':     '20',
  'strength':            '0',
  'money':               '12',
  'last_action':         'recharge',
  'zone_score':          '0',
  'timestamp':           '1323732915832',
  'energy':              '11',
  'max_energy_disabled': '16',
  'max_health':          '6',
  'step':                '7',
  'score':               '160',
  'deadline':            '1323732917832',
  'vis_range':           '1',
  'last_action_result':  'successful',
  'health':              '6',
  'achievements':        ['proved5'],
  'type':                'request-action',
  'id':                  '8',
  'max_energy':          '16' }
```

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

Belief revision Once the remaining information is received from the percept server, the data is then asserted into the agent's knowledge 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 commited 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[11] [5], a defeasible logic programming language that uses argumentation [4] 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.

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[6], knowledge is represented using facts, strict rules and defeasible rules. Facts and strict rules are ground literals representing firm information that can not 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.

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` is an argument that supports or defeats the argument for the Self Defense 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.

```
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`

For the coloring algorithm, we added several improvements, for both optimization and correctness. In essence, since we only had an incomplete version of the full map in every step, we added the concept of "fog of war" to the agents, assuming always in a pessimistic way.

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.

Finally, for this last algorithm too, we added an important optimization that allowed us to call it several times, with the virtual cost of only one call. It was done using memoization, and a more thoughtful invocation.

Initial plans were 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 performance issues. Having the choice 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.

To accomplish this, all agents follow the same concept. Every agent is concerned with the formation and expansion of zones, beyond its role. The desicion-taking process is responsible for calculating and selecting the most beneficial intention, which may be focused in the zone conquering (if possible), or not. This selection process is based on many factors, such as 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.

For example, a repairer agent that is part of a zone will attempt to keep or expand the zone by moving to a better node (in terms of zone coloring), unless a teammate needs to be repaired.

Agents coordination is achived in an implicit way. This is, the information shared consists only of the perception received, not including neither preprocessed beliefs, nor control variables. The agents do not communicate their intentions nor plans, so any coordination that they may exhibit is accomplished implicitly.

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

Zone conquering. The exploration of the map is done gradually, as a result of the reasoning proccess. The actions related to the exploration (probe, survey) are weight along with all other actions and are selected when it is considered important. This is seen to a greater extent during the inicial steps of the simulations, when the team lacks of knowledge about the map and other kind of actions are unnecesary. Agents make no assumptions about the map topology.

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. This is achieved by checking the color of the current node (received in the perception), and if 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 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. If the expansion intention explained is selected and carried on, 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. Sabouteurs 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 posibbly 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 behaviour 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 proccess 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' behaviour. 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 [12]. What we tried to minimize was 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 requieres. This may sound excesive, 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.

Communication. Some functionality provided by the *eismassim* library was reimplemented in a connection library 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 percepts and send the difference back to each agent. After receiving the joint percept, the agents enter a belief setting phase, and later an argumentative phase.

4.3 Agents' organization

As explained before, all agents operate in the same way. The desicion-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 behaviour. The only organization that they have is the proper given by the environment, which is the roles.

Refering 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 ammount 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.

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.

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.

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.

5.2 Our thoughts about possible optimizations to the contest

Many optimizations ocurred to us for the scenario, and the contest in general. For example, more information for the nodes, including something useful for a directed search (i.e., absolute coordinates), would help in the implementation of an A* search, that would decrease the execution times.

Strategically, the early dominion 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, and it's in the middle of the map.

More informing feedback from the server would be appreciated, specially involving errors. This is important for a better and quicker detection of bugs involving communication, i.e. problems with the connection, files sent.

It also 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. Many of us reimplemented the eismassim module, so we are vulnerable to many errors that were difficult to foresee. Early testing would help with that, and detecting infrastructure issues, such as network problems.

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

Bibliography

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

[2] 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*, 59:277–311, 2010. ISSN 1012-2443.

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

[4] Philippe Besnard, Sylvie Doutre, and Anthony Hunter, editors. *Computational Models of Argument: Proceedings of COMMA 2008, Toulouse, France, May 28-30, 2008*, volume 172 of *Frontiers in Artificial Intelligence and Applications*, 2008. IOS Press. ISBN 978-1-58603-859-5.

[5] Edgardo Ferretti, Marcelo Errecalde, Alejandro Javier García, and Guillermo Ricardo Simari. Decision rules and arguments in defeasible decision making. In Besnard et al. [4], pages 171–182. ISBN 978-1-58603-859-5.

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

[7] Daniel Hölzgen, Thomas Vengels, Patrick Krümpelmann, Matthias Thimm, and Gabriele Kern-Isberner. Argonauts: a working system for motivated cooperative agents. *Ann. Math. Artif. Intell.*, 61(4):309–332, 2011.

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

[9] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387981969, 9780387981963.

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

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

[12] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

6 Short Answers

6.1 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.

6.2 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. However, development was conducted using a simplified XP (Extreme Programming) methodology.

Question 2. 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?

The solution follows a decentralised 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' behaviour 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 centralised system in disguise?

The team is a truly multi-agent system, and has no centralised 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[7]. Although the initial plan was to run tests against other agent teams prior to the competition, time constraints made this impossible.

6.3 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 peceptions, 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.

6.4 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 desicion-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 proccess.

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. Sabouteurs 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 posibbly 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 analysed, 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 proccess.

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 behaviour 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. That is, the agents' reasoning proccess 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' behaviour.

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 desicion-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 behaviour.

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 requieres. This may sound excesive, 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.

6.5 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 ammount 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.