

Reimplementing a Multi-Agent System in Python

Jørgen Villadsen*, Andreas Schmidt Jensen, Mikko Berggren Ettienne,
Steen Vester, Kenneth Balsiger Andersen, and Andreas Frøsig

Department of Informatics and Mathematical Modelling
Technical University of Denmark

Richard Petersens Plads, Building 321, DK-2800 Kongens Lyngby, Denmark

Abstract. We provide a brief description of our Python-DTU system, including the overall design, the tools and the algorithms that we used in the Multi-Agent Programming Contest 2012, where the scenario was called Agents on Mars like in 2011. Our solution is an improvement of our Python-DTU system from last year. Our team ended in second place after winning at least one match against every opponent and we only lost to the winner of the tournament. We briefly describe our experiments with the Moise organizational model. Finally we propose a few areas of improvement, both with regards to our system and to the contest.

1 Introduction

This paper documents our work with the Python-DTU team which participated in the Multi-Agent Programming Contest 2012 [7]. We also participated in the contest in 2009 and 2010 as the Jason-DTU team [4, 5], where we used the Jason platform [3], but this year we use just the programming language Python as we did in 2011 [6]. See <http://www.imm.dtu.dk/~jv/MAS> for an overview of our activities.

The scenario is based on the scenario from 2011 and has only been changed in a few ways. The most interesting change is the increase in number of agents from 10 to 20 agents per team.

Our focus for the 2012 version of the contest has been on reimplementing the system from 2011. Given that the scenario is very similar to that last year, we decided to look into ways of improving our system. We have been exploring the possibility of implementing an organization for the system using the Moise organizational model [1] as part of a two-student bachelor project.

The paper is organized as follows. In section 2 we discuss some of the ideas we have pursued. In section 3 we describe some of the facilities we have added in the improved system. Section 4 describes in detail our strategies and how the agents commit to goals. Finally, we conclude our work by discussing possible improvements of our system and the contest in section 5.

* Corresponding author: jv@imm.dtu.dk

2 System Analysis and Design

We chose to implement the system using Python as it is very fast and convenient to implement experimental systems in this language. Other useful features of Python are support of multiple programming paradigms, compact code and dynamic typing. We did not use any multi-agent programming languages because we wanted to have complete control of everything in the implementation. Last year we used Python 2 and we decided to upgrade to Python 3.

In order to make sure that our changes during the implementation phase improved our system, all new algorithms and architecture changes were tested against the older versions by comparing the data collected from the new statistics to see if the change made any differences.

2.1 Testing Moise

This year we wanted to try to implement some kind of organization for our system, so we made a substantial test implementation as part of a two-student bachelor project using the Moise organizational model [1]. We chose Moise because we have previous experience using it in combination with the Jason platform [3].

The Moise organizational model [1] is a formalism for organizational multi-agent systems where an organization is divided into three dimensions: structural, functional and deontic specification. The structural specification uses the concepts of *roles*, role *relations* and *groups* to build the individual, social and collective structural levels of an organization. Here, the roles an agent can enact are defined, and it is furthermore defined how roles are linked, e.g. by allowing agents enacting different roles to communicate. The collective level is specified using the notion of groups, in which it is determined which roles are allowed to be enacted and what links exists between agents both within internally in the group and with external agents. The functional specification specifies missions and plans using a so-called *social scheme* which is a goal decomposition tree that has as root the goal of that scheme. The responsibilities for each subgoal in a scheme are distributed in missions, which means that an agent choosing to commit to a mission effectively chooses to commit to the goals of that mission. The subgoals are created using the operators sequence, indicating that a goal is fulfilled when the sequence of subgoals are fulfilled, choice, in which a goal is fulfilled when a single subgoal is achieved, and parallelism, where all subgoals must be fulfilled, but no specific order is required. The deontic specification is the relation between the structural and functional specifications: it specifies on the individual level the permissions and obligations of a role on a mission. It makes it possible to specify that an agent enacting a certain role is obligated (or permitted) to commit to certain missions, and is therefore obligated (or permitted) to commit to the goals of that mission.

We follow the approach of S-Moise⁺, which is an open-source implementation of an organizational middleware that follows the Moise-model [2]. Among other things it consists of a special agent, the *organizational manager*, which maintains consistency in the organization, i.e. by making sure that a single agent cannot

enact two incompatible roles at the same time. This is done by letting the agents communicate with the manager when they want to join a group, enact a role or commit to a mission. If any such event is a violation of the organizational specification, the organizational manager will not allow it.

The plan trees and social schemes of Moise have a large potential, due to the fact that they will make sure that the right amount of agents will work together toward the best goal. We have chosen to only plan for a single subgoal for each agent, because of the very dynamic nature and the size of the map and number of agents. This makes the plans sufficiently small for the agents to coordinate themselves using direct communication, which makes the plan trees unnecessary.

It might be possible to split the agents into smaller groups to perform more coordinated plans, like finding the opponent's zones etc., but we did not have the time to try to implement groups. In the end we decided not to use Moise as we found that the benefits did not outweigh the needed effort to get the computation under the time limit, due to the quite large communication overhead of the organizational manager.

2.2 Agent behaviour

Our resulting system is a decentralized solution with a focus on time performance. The communication between the agents relies on shared data structures as this is a very fast way to communicate for the agents. The `Runner` class which coordinates communication is described in more detail in section 3.3.

Instead of letting the agents find goals based on their own knowledge alone they use the distributed knowledge of the entire team. This does add some communication which in some cases is unnecessary but in most cases the extra knowledge will produce better goals for the agents.

In each step each agent will find its preferred goals autonomously and assign each of them a *benefit* based on its own desires (i.e. the type of agent), how many steps are needed to reach the location and so on. In order to make sure that multiple agents will not commit to the same goal they communicate in order to find the most suitable agent for each goal. This is done using our auction-based agreement algorithm which will be discussed in more detail in section 4.3.

The agents in this contest are situated in an inaccessible environment which means that the world state can change without the agents noticing from step to step, e.g. if the opponent's agents move outside our agents' visibility range. Hence our agents should be very reactive to observable changes in the environment.

The agents are only proactive in a few situations. The most important one being the communication between a disabled agent and a repairer. They use their shared knowledge in order to decide which of the agents should take the last step and who should stay, so that they eventually are standing on the same vertex instead of simply switching positions. This is implemented by considering the current energy for each agent.

Some of our agents also attempt to be proactive by for example parrying if an opponent saboteur is on the same vertex. Furthermore, repairers will repair wounded agents since they are likely to be attacked again.

2.3 Random generation of the map

Last year all maps had one high-valued area, indicated by numbers on the vertices, as seen in figure 1. For this setting we developed an algorithm which places the agents in defensive positions inside the area in order to defend it. For more information we refer to the paper about our system from 2011 [6].

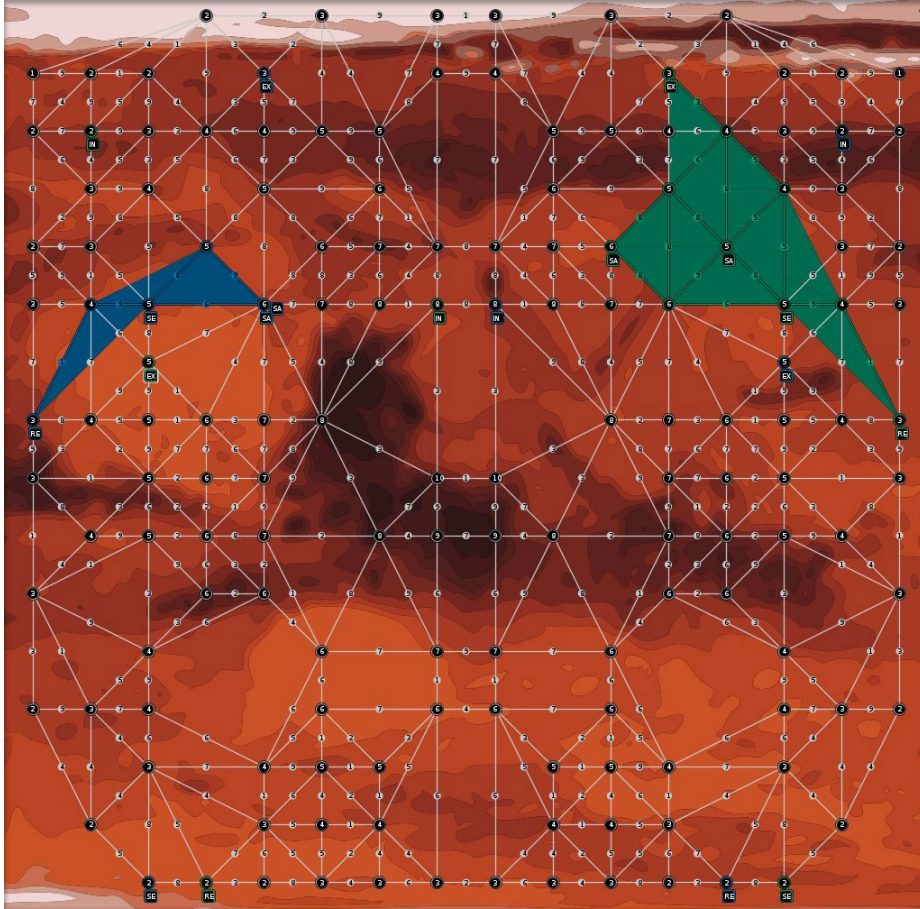


Fig. 1. An example of a map in the MAPC 2011.

This year the map generation algorithm has been updated to create more than one high-valued area. An example of this can be seen on figure 2, where the size of a vertex represents its value. In some cases this lead to situations where our agents would protect a single good area even though it would be better to make smaller groups and have control over several areas. Therefore our previous solution would only be effective in special cases, so we have implemented a

new algorithm which takes multiple areas into consideration. The new solution is actually much simpler and it works well for both maps with multiple areas and maps with a single, high-valued area. In section 4.2 we describe the main properties of this algorithm.

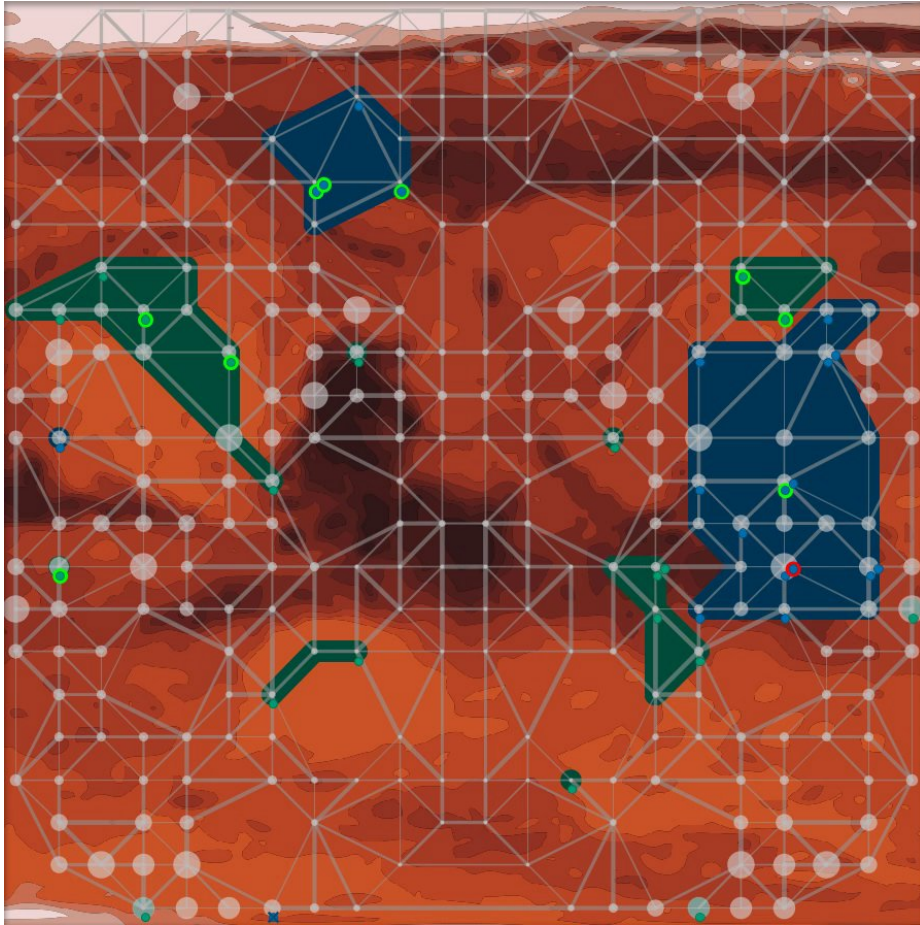


Fig. 2. An example of a map in the MAPC 2012.

3 Software Architecture

The software architecture, including the auction-based agreement approach, is thoroughly described in the paper about our system from 2011 [6] and will only be described briefly here. The rest of this section will describe a few minor facilities added this year.

3.1 Considerations

The competition is built on the Java MASSim-platform and EISMASSim framework which makes it easy to implement a system quickly without spending time on server communication and protocols. However, we did not utilize this framework but chose to implement our system in Python exclusively to have better control and complete knowledge about the implementation. Another solution based on EISMASSim, ActiveMQ and the Java implementation of Python, called Jython, was implemented as well. This solution was discarded due to performance issues. We also considered using a multi-agent framework such as Jason, but due to prior experiences, we thought that the benefits were outweighed by the increased complexity and thus chose to implement our own framework. 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 has quite effective for this setting.

Last year we used a decentralized solution where the agents shared their percepts through a shared data structures but each kept their own copy of the graph representing the environment. The increase in the number of agents and the size of the maps for this year's competition, forced us to rethink and reimplement the percept sharing. To efficiently handle the increased amount of information, all agents share a single instance of the graph. To avoid deadlocks, percepts that lead to updates in this graph are handled with synchronized queues which allow safe exchange of data between multiple threads.

3.2 Testing using flags

A lot of testing was required for verifying that our system was improved compared to our previous system, so we needed an easy way to select which algorithms to use. In order to be able to run several instances of the program, we decided to create program arguments, or flags, for the system. In the beginning we had a configuration file in which we set flags. This was not a very practical way to do it as we had to have multiple configuration files in order to run more instances of the program. These flags make it possible to specify which algorithms the system should use. The help page for our multi-agent system where the different flags are described is shown below:

```
$ python ./bagent.py -h
usage: bagent.py [-h] [-b] [-d] [-a] [-w] [-l] [-v {0,1,2}] {a,b,Python-DTU}

positional arguments:
  {a,b,Python-DTU}      agent name prefix

optional arguments:
  -h, --help            show this help message and exit
  -b, --buy             make the agents shop for upgrades
  -d, --dummy          dummy agents
```

```
-a, --attack          do attack
-w, --weak_opp       attack EXP and INS in the start of the simulation
-l, --load_pickle    load vertices from pickled data
-v {0,1,2}, --verbosity {0,1,2}
```

The flags are used to start multiple instances of the system using different strategies. For example we can test whether it is better to use our buying strategy by starting the server and then start two instances of the system where the flag `-b` was passed to one of them. This was used to test whether it was beneficial to use our heuristics, but as we found that this was not the case we have removed them from the system.

3.3 Code structure and files

We briefly describe the main classes and files:

global_vars.py: We have all our global variables in this file. They are mainly used to make the implementation more dynamic and easier to maintain.

comm.py: This is the file where we have implemented the `Agent` class and the procedures used to communicate with the server. The `Communicator` class is implemented as processes such that all the agents can send and receive messages at the same time. The logic of the agents are implemented in the `util.py` and `algorithms.py` files.

bagent.py: This is where the main program is started and where the flags are parsed. It is also in this file that our `Runner` class is implemented. The `Runner` class starts and lets the agents do their calculations in a sequential fashion.

algorithms.py: Most interesting of our algorithms are implemented in this file, including:

- The *greedy zone control* which will be discussed in section 4.2.
- The *get goals algorithm* called by each agent. This algorithm is discussed in more detail in the paper about our system from 2011 [6].
- The *best-first search* used by each agent in order to find specialized goals according to their type.

util.py: We have implemented our graph representation of the map in this file. The file also includes a timer which was used to find bottlenecks in our code.

4 Strategies, Details and Statistics

In the competition each step of each achievement is exponentially harder to reach than the previous, thus our agents need a way to change their goals as the simulation progresses. We describe our strategy for getting achievements in section 4.1 and our zone control strategy in section 4.2. We describe how the agents decide what to do in section 4.3 and finally how communication works in section 4.4.

4.1 Getting achievements

In the beginning every agent will work towards achieving as many type specific goals as possible in a more or less disorganized fashion, e.g. the inspector will inspect every opponent it sees.

We do this to achieve as many achievements as possible as fast as possible. We tried implementing different heuristics to improve the first part of the strategy. We considered the following heuristics:

Survey heuristic: The agents always survey the vertex with the most outgoing edges if the steps needed to reach the vertex are the same (figure 3). The idea is to get survey achievements faster, but it turned out that even though we got the first few achievements faster, the last ones were achieved a lot later using this heuristic, so we did not use it.

Probe heuristic: The agents probe the vertex with the highest valued neighbours (figure 4). This worked very well in the scenario from 2011, but in the 2012 scenario it can be more beneficial to first find a lot of potentially high valued areas which can be probed later. This can be achieved using a random walk, which will reduce the time in each area increasing the chance that the agents might find more areas in less steps. We chose not to use the probe heuristic since a random walk was more successful.

Attack vulnerable opponents: This heuristic is only applied in the first 80 steps (a simulation has 750 steps). We prefer to attack agents that cannot parry, as this will get us more successful attacks. Furthermore, as added value this will also lead to fewer successful parries for the opponent. This turned out to give us a slight advantage in the beginning of the simulation, so we chose to use it.

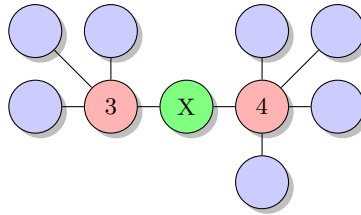


Fig. 3. Illustration of the heuristic values our agents would get trying to survey, standing on the green vertex. The vertex to the left has a heuristic value of 3 because it has three outgoing edges, whereas the one on the right has a slightly better heuristic value of 4.

After a certain number of steps the agents will proceed to the zone control part of our strategy. The sentinel is the only agent surveying after step 30. The explorers keep probing until step 150 and will probe in our target area for the next 50 steps to make sure we control as many vertices as possible. Afterwards

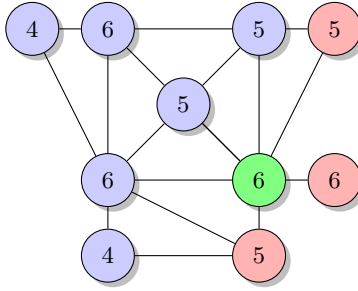


Fig. 4. Illustration of the heuristic values our agents would get trying to probe, standing on the green vertex where the blue ones are owned with the given value. The heuristic value of the red vertices are calculated by taking the mean of the known neighbouring vertices.

they will follow the zone control strategy. All other agents begin zone control after step 150.

4.2 Zone control

The zone control part of our strategy uses a very simple, but surprisingly effective, greedy algorithm. The algorithm works by first choosing the node with the highest value, and then by choosing a potential neighbour node. The potential value of choosing that node is then calculated as the value of the node plus the sum of all the neighbours which, according to the graph coloring algorithm [7], will be owned if the potential node is chosen. For each agent, the algorithm will choose the best node according to some parts of the graph coloring algorithm. If a vertex has not been probed the algorithm will use the value 1. This way we take some of the area coloring algorithm from the contest into consideration and as it is an inaccessible environment this is the best we could achieve.

This algorithm will to some extent choose the optimal area or several areas which are still fairly easy to maintain, even though our choices are limited by our (partial) knowledge of the map and the missing parts of the area coloring algorithm.

During the zone control part every type of agent has a specific job.

- *Repairers and saboteurs* do not directly participate in the zone control, instead they are trying to defend and maintain the zone.
- *Inspectors* keep inspecting from their given expand node, because the opponents might have bought something which we need to make a counter move against.
- *Explorers* will probe unprobed vertices within the target zone. When all vertices are probed they are assigned a vertex by the zone control strategy.
- The *sentinels* will stay on a vertex assigned by the zone control strategy and will parry if some of the opponent’s saboteurs move to the sentinels position.

The last important change in the state of mind of the agents is that after step 150 the saboteurs start buying. They buy exactly enough extra health so that they will not get disabled by a single attack from an opponent saboteur that has not upgraded his strength. Furthermore we buy enough strength to disable any opponent saboteur in a single attack by buying strength for all our saboteurs every time we inspect the opponent saboteurs and find that it has more health than all other inspected saboteurs. This buying strategy is chosen in hope of dominating the map which will make it possible to gain control of the zone we want. The advantage is that we only try to out-buy in one specific field, thus we are unlikely to use all our achievement points. As this is a quite aggressive buying strategy we had to wait to step 150 to have enough achievement points to execute it.

4.3 Making decisions

The agents need a consistent way of figuring out what to do. We do this by letting every agent find the nearest goals according to their type. They do this by using a modified best-first search (BFS) which returns a set of goals. To make sure that every agent always has at least one goal the BFS returns as many goals as we have agents. This is a very agent-centered procedure meaning the agents simply commit to the goal with the highest benefit, instead of coordinating any bigger schemes. However, since the goals are more or less dependent on each other there is some implicit coordination. For example the repairers will often follow the saboteurs as these search for opponents and thus more often will share a vertex with an opponent saboteur and get disabled.

To decide which goal to pursue the agents use an auction algorithm. Every agent can bid on the goals they want to commit to and will eventually be assigned the one they are best suited for. This results in a good solution, which however might not be optimal. For further details we refer to the paper about our system from 2011 [6].

Even though our planner calculates a few turns ahead the agents recalculate every turn. We do this to adapt to newly discovered obstacles and facts, such as an opponent saboteur or the fact that the agent has been disabled. The agents will not end up walking back and forth as their previous goal will now be one step closer, thus the benefit of the goal has increased. If another goal becomes more valuable it means that it is a better goal than the one the agent was pursuing, thus changing the commitment makes sense, so we do not lose anything on recalculating each turn.

4.4 Communication

Communication and sharing of information is extremely important in any multi-agent system. In our system every percept received by the agents are stored in a shared data structure so that all agents have access to the complete distributed knowledge of the team at all times.

Actual communication in our system only happens when the agents are deciding what to do. When they are figuring out what to do the auction-based agreement algorithm is used on conflicting goals and thus two agents will never pursue the same goal.

5 Conclusion

In the process of reimplementing and improving the Python-DTU multi-agent system we have analysed the changes to the competition and used our findings to design and implement better algorithms for the increasingly complex tasks. We have considered imposing an explicit organization upon the agents, and for this purpose we experimented with the Moise organizational model. While it had some advantages, such as the being able to ensure that the right amount of agents work together toward a certain goal using by use of roles and plan trees, we decided not to use Moise in the final version of our system, as its benefits did not outweigh the communication overhead caused by the organizational manager in the organizational middleware, S-moise⁺.

All improvements to the algorithms are quite simple, but are nevertheless effective at reaching their goals. The simplicity and specialized approach is probably one of our strengths, as it makes it easy to implement special cases when certain improvements of the algorithms were necessary. Having aggressive saboteurs was also an advantage as it led to the opponents being disabled often, which in turn gave us a larger zone score. Our greatest weakness was that our uncompromising attempt to have the strongest saboteurs could be countered by buying enough health on a single saboteur to make us use most of our achievement points for improving all of our saboteurs. This could lead to a large difference in step score gained from achievement points each step.

The many advanced programming constructs in Python, e.g. lambda functions, list comprehensions and filters made it possible to implement algorithms very efficiently.

One thing we have noticed during the competition is that it does not seem to pay off to buy anything other than health and strength. This meant that a lot of teams had more or less the same strategies. We think it could be interesting if many kinds of strategies could be sufficiently effective so that we might see the teams following different strategies. One idea could be to introduce ranged attacks which could be achievable through upgrades and should be limited by visibility range. This could allow for some other strategies, since the agents need to figure out where to hit the opponent a few steps in the future and how to avoid getting hit themselves. Furthermore, the teams will need to use their inspectors even more to find out whether or not to avoid possible ranged attacks from the opponent.

References

1. Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Bossier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In Guilherme Bittencourt, and Geber Ramalho (Eds.): SBIA '02, LNCS 2507, 118-128, Springer 2002.
2. Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Bossier. S-moise⁺: A Middleware for Developing Organised Multi-Agent Systems. In Bossier et. al. (Eds.): COIN 2005, LNIA 3913, 64-78, Springer 2006.
3. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. Programming Multi-Agent Systems in AgentSpeak Using *Jason*. John Wiley & Sons, 2007.
4. Niklas Skamriis Boss, Andreas Schmidt Jensen, and Jørgen Villadsen. Building Multi-Agent Systems Using *Jason*. *Annals of Mathematics and Artificial Intelligence*, 59: 373-388, Springer 2010.
5. 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, Springer 2011.
6. Mikko Berggren Ettienne, Steen Vester, and Jørgen Villadsen. Implementing a Multi-Agent System in Python with an Auction-Based Agreement Approach. In Louise A. Dennis, Olivier Boissier, and Rafael H. Bordini (Eds.): ProMAS 2011, LNCS 7217, 185-196, Springer 2012.
7. Tristan Behrens, Michael Köster, Federico Schlesinger, Jürgen Dix, and Jomi Hübner. Multi-Agent Programming Contest — Scenario Description — 2012 Edition. Available online: <http://www.multiagentcontest.org/>, 2012.

Short Answers

A Introduction

1. What was the motivation to participate in the contest?
A: We find the contest very interesting for both research and teaching, cf. <http://www.imm.dtu.dk/~jv/MAS> for an overview of our activities.
2. What is the (brief) history of the team? (MAS course project, thesis evaluation, ...)
A: The team consists of both researchers/students from previous years and students taking a variant of the course 02295 Advanced Topics in Computer Science.
3. What is the name of your team?
A: The name of our team is Python-DTU where DTU is the short for the Technical University of Denmark. We started as the Jason-DTU team a few years ago.
4. How many developers and designers did you have? At what level of education are your team members?
A: We are 6 computer scientists: associate professor Jørgen Villadsen (PhD), Andreas Schmidt Jensen (PhD student), Mikko Berggren Ettienne (MSc student), Steen Vester (MSc student), Kenneth Balsiger Andersen (BSc student) and Andreas Frøsig (BSc student)
5. From which field of research do you come from? Which work is related?
A: Our field of research is AI with an emphasis on algorithms and logic (our section is called Algolog).

B System Analysis and Design

1. Did you use multi-agent programming languages? Please justify your answer.
A: No, we used plain Python.
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.
A: We did not use any specific multi-agent system methodology, since we wanted to take a more direct and simple approach.
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?
A: We use an agent-centered system in which each agent first calculates their own goals, then they bid on the different goals and finally they use an auction algorithm to figure out which goals to pursue.
4. What is the communication strategy and how complex is it?
A: The communication strategy is very simple. The agents only talk to each other at the auction, and they share data using global data structures.

5. How are the following agent features considered/implemented: *autonomy*, *proactiveness*, *reactiveness*?
- A: We have implemented the features as follows:
- We have some degree of autonomy as each agent finds his own goals, but not complete autonomy as the goals are distributed using an auction algorithm.
 - The agents are proactive in certain situations. For instance, some of our agents will parry if a saboteur is located at the same node as the agent, and we also repair wounded agents since they will probably be attacked again later.
 - The agents are situated in an inaccessible environment, so we have chosen that our agents recalculate their goals at each turn and take new percepts into consideration, thus we have a high degree of reactivity.
6. Is the team a truly **multi**-agent system or rather a centralised system in disguise?
- A: The agents have direct access to a common dataset, so one could argue that it is a centralised system in disguise, but actually the agents themselves are communicating in order to figure out what to do. Furthermore, since all the agents find goals and decide which to pursue, the team is a true multi-agent system.
7. How much time (man hours) have you invested (approximately) for implementing your team?
- A: We initially expected that we would have to invest approximately 200 man hours, but when the tournament started we had invested approximately 300 man hours.
8. Did you discuss the design and strategies of you agent team with other developers? To which extend did you test your agents playing with other teams.
- A: We did not discuss strategies with other teams, but we used the test matches as an attempt to lure any good strategies.

C Software Architecture

1. Which programming language did you use to implement the multi-agent system?
- A: Python 3.
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?
- A: Since we used plain Python there was no obvious relation between agent architecture in the design phase and the programming code. We used classes to represent agents.
3. Which development platforms and tools are used? How much time did you invest in learning those?

- A: We have used *Mercurial* to version control our code and *gedit* and *vim* to implement the code. We have not used any time learning this as we had already used it before.
4. Which runtime platforms and tools (e.g. Jade, AgentScape, simply Java, ...) are used? How much time did you invest in learning those?
- A: We have only used Python with a few special packages for argument parsing etc.
5. What features were missing in your language choice that would have facilitated your development task?
- A: Some sort of precompiling for debugging purposes could have speeded up the implementation.
6. Which algorithms are used/implemented?
- A: We have implemented the following algorithms:
- Auction house
 - Best-first search
 - Greedy zone control area algorithm
7. How did you distribute the agents on several machines? And if you did not please justify why.
- A: We used processes to allow the implementation to use all cores of the machine, which boosted the performance. We did not need several machines, since this was adequate for fulfilling the time limit criteria.
8. To which extend is the reasoning of your agents synchronized with the receive-percepts/send-action cycle?
- A: The agents synchronize every time they have handled all percepts and as they start the auction house algorithm.
9. What part of the development was most difficult/complex? What kind of problems have you found and how are they solved?
- A: Developing heuristics for different tasks proved to be quite hard. We used a lot of time on testing different implementations against each other to solve this.
10. How many lines of code did you write for your software?
- A: We have written 1438 lines of code in Python including blank lines. This count includes various test functionalities.

D Strategies, Details and Statistics

1. What is the main strategy of your team?
- A: The main strategy of our team is split into three parts. First we want to get as many random achievements as possible, then at step 30 we partially tries to take control of a high valued area, while explorers keep on probing more or less randomly. Finally, as we reach step 200 every agent is focused on expanding.
2. How does the overall team work together? (coordination, information sharing, ...)

- A: All percepts are shared in a global datastructure and the goals are found individually by each agent, and if they conflict with any of the other agents goals then the auction algorithm is used.
3. How do your agents analyze the topology of the map? And how do they exploit their findings?
- A: We simply save the map as a graph of vertices with a set of neighbouring vertices. The percepts are stored in a shared data structure including opponents specifications, types etc. Every time an agent perceives something from the environment, it will update the shared data structures.
4. How do your agents communicate with the server?
- A: We communicate through a socket, where we handle strings of XML received from and sent to the server.
5. How do you implement the roles of the agents? Which strategies do the different roles implement?
- A: Each agent has a tag telling what type it is, and the goals they create depend on this tag.
6. How do you find good zones? How do you estimate the value of zones?
- A: Using a greedy algorithm considering common neighbours combined with the graph coloring algorithm used by the server.
7. How do you conquer zones? How do you defend zones if attacked? Do you attack zones?
- A: We choose a zone to conquer judged by the sum of the values of the vertices in the zone. We have a greedy algorithm that takes most of the vertices we are going to own into account. Our saboteurs defend a zone by attacking opponents near it. If no opponents are near, the saboteurs will move towards a location where we last spotted an enemy in hope of finding some opponents and attack their zone.
8. Can your agents change their behavior during runtime? If so, what triggers the changes?
- A: They change as described in our strategy which is a hard coded number of steps.
9. What algorithm(s) do you use for agent path planning?
- A: We use a best-first search algorithm to decide which goals to commit to.
10. How do you make use of the buying-mechanism?
- A: We make sure that our Saboteurs are strong enough to disable opponent saboteurs in a single attack by buying as much strength as they have health. Besides this we buy a single health point in case the opponents do not buy strength so that they cannot disable us in a single attack.
11. How important are achievements for your overall strategy?
- A: Our entire strategy is build up around getting as many achievements as possible, thus it is extremely important.
12. Do your agents have an explicit mental state?
- A: They are very active until they start controlling a zone at which point they go into a defensive state of mind.
13. How do your agents communicate? And what do they communicate?

- A: Using a shared data structure in which they all percepts are shared with every agent.
14. How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?
- A: We do not organize them explicitly, but as saboteurs often gets disabled the repairers will follow and repair them. In the zone control part the agents will often organize themselves, but only as this gives more zone points.
15. Is most of you agents' behaviour emergent on an individual or team level?
- A: The agents decide autonomously which goals they want to pursue, and use an auction-based agreement algorithm to solve conflicting goals between agents on the team level.
16. If you agents perform some planning, how many steps do they plan ahead.
- A: We only plan as far as the 4 nearest goals for each agent and we recalculate at each turn so we only plan a few steps ahead.
17. If you have a perceive-think-act cycle, how is it synchronized with the server?
- A: For each turn we have a busy-wait until we have received the percepts and we send as soon as we have an action.

E Conclusion

1. What have you learned from the participation in the contest?
- A: We have learned to implement a multi-agent system and all that it includes. We have been experimenting with a lot of algorithms and found that the simple ones are often better as they are fast which are useful in a fast changing world. We also learned that small tweaks and fine tuning in a multi-agent system really changes the behaviour and outcome a lot.
2. Which are the strong and weak points of the team?
- A: Our team are very good at getting a lot of achievements very fast. We also defend our area pretty good. Our downside is that we almost only consider what we can see, this means that in some situations the opponent could have a large area and we would just let them. We ended up losing because the opponents had a strategy to use as little achievement points as possible in order to make us use them all. In the long run this made us lose even though we had a better area. We never back down from a saboteur which in many cases would be good, as their saboteur would be occupied and we would not get disabled thus the repairer does not have to walk a long distance.
3. How suitable was the chosen programming language, methodology, tools, and algorithms?
- A: It was a pretty good programming language though you had to get used to indent-based programming and no precompiling. It could have been interesting to try a multi-agent programming language to really model a system. In the end it seemed like our algorithms were quite effective even though they were very simple.
4. What can be improved in the context for next year?

- A: We could implement some of the missing strategies discussed above, for instance avoiding saboteurs or implementing a better attack strategy, e.g. prioritizing opponents based on current health (by inspection). We could also look into our buy strategies and adding an upper limit if needed.
5. Why did your team perform as it did? Why did the other teams perform better/worse than you did.
- A: Our strategy worked pretty well which we anticipated as we played quite defensive. We lost because one of the opponents figured out our buying algorithm in the test-phase and implemented a clever counter strategy.
6. Which other research fields might be interested in the Multi-Agent Programming Contest?
- A: Algorithms, logic, game theory and AI.
7. How can the current scenario be optimized? How would those optimization pay off?
- A: It does not seem like it pays off to buy anything but health and strength. A reasonable change could be making visibility range more useful by using it as a parameter in the actions. For example this could mean that the saboteurs would get ranged attacks or inspectors could inspect all opponents inside the visibility range. This would 'force' a lot of new strategies to come into play, e.g. avoiding opponents artillery strikes etc.