Implementing a Multi-Agent System in Python with an Auction-Based Agreement Approach

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 algorithm 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) [1,6] 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 [7] and JIAC [8] 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 team since we used the Jason platform and its agent-oriented programming language AgentSpeak [4,5]. We performed well but for the contest this year, with the new and more complex scenario, we decided to focus on an auction-based

* Corresponding author: jv@imm.dtu.dk

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 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 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.

2.1 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 [2,3] and achieves this with some compromises while still satisfying every point to some extent.

2.2 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

$$net\,value_{ij} = benefit_{ij} - p_j$$

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 1 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 [2]. 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.

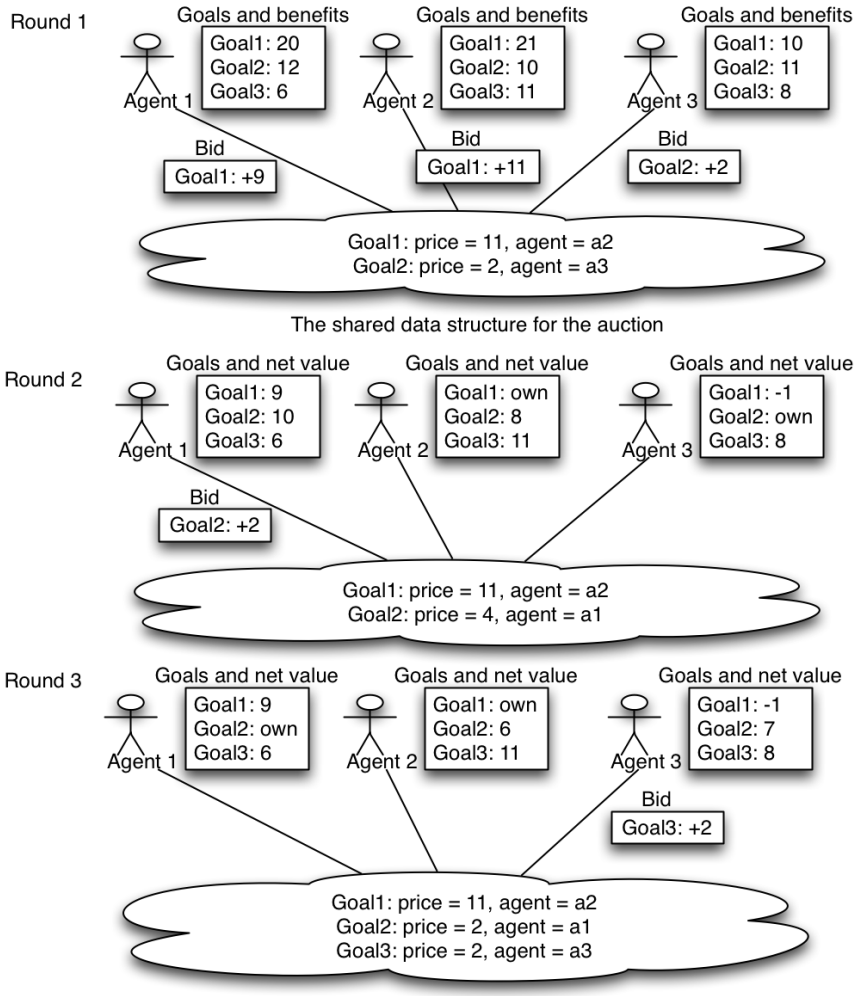

Fig. 1. An example of an auction between three agents. In round 1 the auction data structure is empty before the bidding and thus goal benefits are equal to goal net values. Each agent places a bid on its preferred goal. The bids are calculated as the difference between the two best goals plus ϵ. The 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).

3 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 [9] 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 2.1. 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 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.

3.1 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.

3.2 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 4.

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.

4 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 3.2. The strategy is greedy as agents does not consider subsequent goals when deciding on a set of goals.

4.1 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. [1]. 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. [1]. 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.

4.2 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 discussed in [10] taking path length, the number of vertices on the path and the agents' recharge rates into account.

4.3 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 as discussed in [10] 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.

4.4 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 2.1. 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.

5 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

1. 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.
2. 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.
3. D.P. Bertsekas, and D.A. Castanon. *Parallel synchronous and asynchronous implementations of the auction algorithm*, Parallel Computing 17, 707–732, 1991.
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, 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, 2011.

6. 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.
7. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*, Wiley 2007.
8. 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.
9. 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.
10. Mikko Berggren Ettienne, Steen Vester, and Jørgen Villadsen. *Implementing a Multi-Agent System in Python*, In Multi-Agent Programming Contest 2011, Technical Report, to appear.