Multi-Agent Programming Contest 2016 – The Python-DTU Team

Villadsen, Jørgen; From, Andreas Halkjær; Jacobi, Salvador ; Larsen, Nikolaj Nøkkentved

Published in:
International Journal of Agent-Oriented Software Engineering

Link to article, DOI:
10.1504/IJAOSE.2018.10010604

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Multi-agent programming contest 2016 – the Python-DTU team

Jørgen Villadsen*, Andreas Halkjær From, Salvador Jacobi and Nikolaj Nøkkentved Larsen

DTU Compute,
Technical University of Denmark,
Richard Petersens Plads,
DK-2800 Kongens Lyngby, Denmark
Email: jovi@dtu.dk
Email: s144442@student.dtu.dk
Email: s113423@student.dtu.dk
Email: s123675@student.dtu.dk
*Corresponding author

Abstract: We provide a detailed description of the Python-DTU system, including the overall system design and the tools used in the agent contest.

Keywords: multi-agent systems; MAS programming; contest.

Reference to this paper should be made as follows: Villadsen, J., From, A.H., Jacobi, S. and Larsen, N.N. (2018) 'Multi-agent programming contest 2016 – the Python-DTU team', *Int. J. Agent-Oriented Software Engineering*, Vol. 6, No. 1, pp.86–100.

Biographical notes: Jørgen Villadsen is an Associate Professor in the Algorithms, Logic and Graphs Section, Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU). His research is in logic, computational linguistics and artificial intelligence. He has previously been a researcher at Roskilde University, Danish Defence Research Establishment and Centre for Language Technology and worked as a IT specialist in industry.

Andreas Halkjær From is a BSc student at the Technical University of Denmark (DTU). He has taken courses on algorithms, logic and artificial intelligence at the Department of Applied Mathematics and Computer Science.

Salvador Jacobi is an MSc student at the Technical University of Denmark (DTU). He has taken courses on algorithms, logic and artificial intelligence at the Department of Applied Mathematics and Computer Science.

Nikolaj Nøkkentved Larsen is an MSc student at the Technical University of Denmark (DTU). He has taken courses on algorithms, logic and artificial intelligence at the Department of Applied Mathematics and Computer Science.

1 Introduction

The name of our team is Python-DTU. We participated in the contest in 2009 and 2010 as the Jason-DTU team (Boss et al., 2010; Vester et al., 2011) in 2011 and 2012 as the Python-DTU team (Ettienne et al., 2012; Villadsen et al., 2013b) and in 2013 and 2014 as the GOAL-DTU team (Villadsen et al., 2013a).

We are affiliated with DTU Compute (short for Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU) and located in the greater Copenhagen area).

This paper details how the system was meant to work and we locate and analyse problems in the implementation of the system. The system was developed in Python. We describe the structure of the multi-agent system in Section 2 and the overall strategy in Section 3. Section 4 has a more detailed description of selected parts of the system. We describe the tests and problems in Section 5 and Section 6. Section 7 discusses the changes made during the contest and Section 8 discusses possible future work. We conclude in Section 9.

2 Structure of the multi-agent system

The system was built with a central planner in mind, with some limited free agency given to the agents. The central planner makes an overall plan and the agents are tasked with carrying out the plan. The strategy of the team is to work together and complete one job at a time. This is a simple straight-forward approach that is designed to be able to complete any kind of job. The agents will gather information and the central planner will make a plan for the agents that tell them what they should do. The agents will then figure out what actions to send to the server in order to execute their part of the plan. Sometimes this can be a single command and sometimes it may be more. Once the agents have completed a job, the system will soft-reset and start completion of another job and continue to do so until the simulation ends.

Each agent also has a list of orders. An order is the simplest instructions given to the agents. The orders are custom-defined instructions that the agents then themselves convert to MAPC instructions. An instruction could be ‘buy’ and then a list of what and how many items the agent should buy. These orders are given to it by the central planner. The agent will simply execute the orders chronologically. If it has no more orders, it will skip its turn until it gets new orders. If an order fails, the agent will retry the order until it succeeds. This is often the case in assembly since some agents may need to assemble some different items first or some agents have not yet arrived at the workshop, causing different assist or assemble actions to fail.

The agents do have some limited agency over these orders. If an agent is ordered to move to a location, it will first determine if it has enough charge (plus a threshold) to do so. If it does not, it will postpone its order and go charge before it moves to its new location. In addition, when an agent receives an order that requires it to be at a specific location, it will manoeuvre to the location before executing the order it was given.

If an agent receives an order and it is already in the process of completing another order, it will simply queue the new order. This means that agents can potentially receive many orders at a time, yet they still execute them all one after one. This allows the central planner to plan ahead and make plans even though the previous plan is not complete.

The agents are not organised in the system in a specific way. The central planner makes use of the agents as it sees fit. It will simply task the different agents with orders depending on what is needed. Some agents can carry a lot of materials and they can also use specific tools. The central planner will simply order the agents around as their abilities are required.

3 Overall strategy

We have developed our own central planner with a simple hard-coded planning structure. The overall plan consists of four steps: shops, buying, assembly and delivery. The planner will make a plan that tells each agent what to do in each of these steps. First, the planner makes agents go to the shops. This allows the planner to collect the information that each shop holds. With this information, it will find a suitable job. When a job is acquired, the planner will plan buying of the items and tools needed for this job. If there are items in the job that require assembling, it will make a plan to assemble these items. This includes coordinating the use of tools and making agents assist each other. Lastly, the planner will make a simple plan for delivering the items to the correct location.

Because the jobs usually only require a rather small set of products to be delivered, it is often difficult or impossible to utilise all agents for one job. This means that quite a few agents are unused/idle a lot of the time. Despite this, we kept our focus on one job at a time. Dividing the agents into groups and completing more jobs at the same time can always be done later, so while we did indeed want to utilise all agents, it was a low priority.

Because the agents only buy exactly what they need to and they do not buy tools more than once, it was considered unnecessary to dump or store items. Our buy-algorithm makes sure to never overload the agents with items because we buy only exactly the items we need and only buy each tool once. This means that storing items is considered unnecessary and dumping items serves no purpose, thus these two aspects of the simulation were left out.

The agents will wait for the others to catch up if one gets too far ahead. If an agents is supposed to execute an action, say, assist another agent, it will wait for that agent to get to the workshop before it moves on to its next order. That is, it will execute the order that it has, but if it fails (which it does if it is too far ahead), it will simply try to execute the same action again until it succeeds, in which case the other agents have caught up.

4 Detailed description

4.1 Shops

The first step in the system is to send all the agents to shops. This is to give the system information about what shops sell what items and their price. We use a single agent for each shop. Having only one agent at each shop means that we get all the information

from all the shops while only moving as little as possible. This way, we save energy in as many agents as possible and thus both time and points down the line.

4.2 Finding job

The planner will attempt to get a job for the team once there is an agent at each of the shops in the simulation (we assume there are less shops in the simulation than there are agents on a team). Once there, the planner will want to get a job only if there is enough estimated time to complete it and only if we already have at most one job. That is, we have at most two concurrent jobs at any given time.

The estimation about how much time is required for completion of a job is only given as a rough estimate. We wanted the estimate to be high enough so that we could complete a given job despite problems, but also wanted it to be low enough to waste as little time as possible. It would be bad if the estimate was too high because then the agents would stop working despite them being able to complete a job. We chose an estimate of 110 simulation steps. This means that the agents can complete any given job, but would not impose a huge downside if they stop 110 steps before the end of the simulation. In addition, this limit is only imposed if the agents are not already completing a job. This means that the agents will only rarely stop 110 steps before the end but will usually complete a job within this limit and will stop at a lower amount of steps before the end.

If the planner has decided to do a job, it will figure out what job it wants. It considers both auction jobs and priced jobs, but in a naive way. First of all, it compares all the priced jobs available and chooses the one with the highest reward. It makes sure to not choose jobs that our team has posted themselves. A posted job is a job that our own team has created. We post a job so that it is available for both teams. A posted job can be either a priced job or an auction job. Thereafter, it considers all the auction jobs (again it makes sure to exclude auctions posted by our own team). There it finds the job with the highest max bid and chooses it. The reason it only compares max bid is that we only make one bid – the maxbid (see next subsection). If we would bid lower than that, then we should instead consider a priced job as one of them might have a higher reward.

Lastly, it compares the best priced job and the best auction job and chooses the one with the highest reward and chooses that as the chosen job. The planner then finds the first agent that is not doing anything (i.e., waiting for new orders) and tasks it with buying/bidding on the job.

Once a job has been chosen and the agent has bought it (or won the bid for the auction job), we do not change. Focusing on one job at a time makes the system simpler and also reduces possible waste of resources.

4.2.1 Bidding on jobs

In the section above, it is said that we only make a maxbid on a job. This is not always the case. In our decision about what to bid, we knew that the other teams would likely want to maxbid as well. So in order to make sure that we get the maxbid, we instead bid 2 points below maxbid. The two points is such a small difference that it is essentially the same bid, but if the enemy team makes a direct maxbid, we will outbid them instead. However, if the enemy team makes a lower bid than this, we will just drop it and not bid on it again. This is because of the volatility of the auction jobs. We do not exactly know what the payout will be since we can keep bidding. Instead, we like to stick to priced jobs

since their reward is much more reliable (unless the enemy team completes it first of course).

4.3 Buying

Once there is an agent at every shop, we are ready to buy. We have assumed that there will always be shops less than or equal to the amount of agents, so this strategy will fail on more complex maps, but it has worked for this contest. The system will figure out what items are needed for the job and break them down into base items and tools. It will figure out exactly what and how many tools and consumed items are needed. Once done, it will make a shopping list about what agents should buy what items at what shops.

The system will first figure out what tools are needed for assembly. It will find all the tools needed for assembly of all the items required by a certain job and add the highest amount of tools to the list. That is, if an item requires 1 of item3 and another requires 2 of item3, it will add a total of 2 item3 to the needed tools list so we can assemble both products that require this tool since tools are not consumed.

Once it knows what tools are needed, it will make a so-called shopping list for these tools. It will figure out what agents should buy what tools. This is based on what tools the different agent types can use and who can buy the tool the cheapest. If there are four agents that can use a specific tool, it finds the agent that can buy the tool cheapest and tasks that agent with purchasing (and later using) that tool. At this point, it also checks if the needed tools are already in the agent's inventory. If they are, we do not buy them since there is no need to.

Next, it figures out how many base items are needed. A base item, in this case, is an item that is not assembled. If one of the items required for the job is assembled, it will break the item down in the materials required to assemble it (and further break those down if they need assembly too). The result is a table of the total amount of base items required to complete the job.

Once it knows what items are required, it will create a shopping list for these items. It finds the agents that have enough space to carry all of one item and from that list of agents, it chooses a random one.

Now that the planner has made a shopping list for tools and one for items, the agents will buy all the materials needed to complete the job. The planner then combines these shopping lists and then transforms them into a plan. When it transforms it into a plan, it will simply change the shopping list into a set of actions for each agent that they need to execute to buy the items they are assigned. If the agent is in a shop, it will simply buy the items on its shopping list. If the agent is not in a shop, it will first figure out how to move to a shop that sells its specified item and then buy the item once it arrives.

The planner also takes assembled tools into account. In that case, it will make the agent that needs the tools buy the base items, so when the item is constructed it is placed into the inventory of the agent that needs to use the tool for further assembly.

4.4 Assembly

In order to make assembly a little simpler, we make all agents go to the same workshop. This ensures that all products can be assembled in sequence with no other actions in between. It may in some cases be faster to assemble some items at one workshop and some other items at another workshop, but that complicates the process a lot. The fact

that all agents are in the same workshop also means that a single agent can take all the products to the delivery location, rather than two or more (or one agent would have to go to multiple locations).

The central planner also made the plan for buying items and tools, which means that it knows what agents have been ordered to buy what. This allows the planner to figure out what the inventory of the different agents looks like after the buying orders have been completed.

4.4.1 Coordinating assisting and assembly

First of all, the planner will create a list of required instructions in order to create each item. These are special instructions that denote how many of a specific item is used in order to create a product. That is, there is an instruction for each required item for each recipe. These instructions consist only of consumed items, i.e., they exclude tools. This list is recursive so it includes all items that the agents have bought as well as intermediate products since they both need to be assembled and then used in another recipe.

We consider an example to make it clearer. Assume that we have the following recipes:

- item1:
  2× item3
  3× item4
  tools:
  1× item5
- item2:
  3 × item3
  tools:
  1× item5

Let's also assume that the job we are completing requires the following:

- 2× item1
- 1× item2

Assuming that the planner has already made a plan for buying these items, it will make the above-mentioned instruction-list. The list will end up looking like this

```
[
  [(("item3", 4), ("item1", 2)),
   (("item4", 6), ("item1", 2))],
  [(("item3", 3), ("item2", 1))]
]
```

Each sub-list corresponds to one type of product (first list for item1 and second list for item2). The list contains tuples of ((consumed name, consumed amount), (produced name, produced amount)). This keeps track of what items, and how many need to be assembled into what items and how many. This system is useful for complex recipes that

either require many items or where intermediate items need to be assembled that then has to be further assembled into the final products.

With this list of instructions, the planner will then distribute them among the agents that have items in their inventories. If an agent has an item that is used in an instruction and it has enough of that item, it can fulfil the instruction. If not all of one item is consumed in that instruction, there is one or more instructions left in the list that will consume the rest of the items. Assuming the instruction fits, the planner assigns this instruction to the agent. It will then remove these items from the agents' inventory (from its own book keeping, not the real agent). The planner will assign all instructions this way.

Now that the planner knows what agents will assemble what products, it will coordinate assisting of items and tools. When coordinating tools, it will loop through each agent's instructions. For each instruction, it will find the needed tools for this instruction. Even though this instruction is specific for an agent, other tools may be needed to assemble the product. The planner will then look through the inventories of the other agents and figure out if they have the tool needed for assembly of this product.

If they do, that other agent will assist. If not, the planner will look through the assembly list again and see if this other agent is assembling some products and then check if one of those products is a tool that is needed for the assembly. If it finds such an instruction, it assigns the assist order. That is, if the needed tools do not exist yet, but are created soon, then the system will still assign the assist order because the agent will soon be in possession of the tool it needs to assist with. The planner then executes the same strategy again, but for items this time.

At this point, the agents know what they are assembling themselves and what tools and items they need to use to assist other agents. Next, each agent will sort all the created assemble and assist actions. It will prioritise assembly of tools in order to have as many tools as possible available as soon as possible. Of course, if a tool is assembled from items that require further assembly, it will try to assemble those items first. If those items also require assembled tools, the planner will prioritise those first. At some point, there will be an assembled tool that does not require other assembled items, and the planner will start from there. The same strategy is used for items so it assembles intermediate products before the final products. In addition, it will sort the assembly and assists of a product at the same time in the list of actions. This means that assembly and assist actions are appended to the agents' actions at the same time. That is, if all agents have the same amount of actions, they will execute assembly of a product at the same time. If an agent has fewer actions, it will simply execute its assemble/assist until it succeeds, effectively waiting for the other agents to catch up.

4.5 Delivery

As mentioned above, all agents go to the same workshop for assembly. Once all the products are assembled, it will find the fastest agent located in the workshop that can carry all the products at the same time. This means that it only makes one trip. This usually ends up being the motorcycle or sometimes the car or drone. It is rare (but not unthinkable) that the truck is required to deliver the products. This means that our system cannot handle jobs with products that exceed a total of 3,000 weight since no agent can

carry all the products at the same time. We do assume that in a worst case scenario, the truck is able to carry all of the products since it has the largest inventory. However, during the competition, we saw another team posting jobs that required an absurd amount of products. In the event that our team chose to complete that job, we would have a problem since there is no agent that can carry that much. However, we would like to avoid jobs posted by other teams anyway since they are likely meant to confuse us. Once an agent is chosen for delivery, all the other agents will give the items to the chosen agent and the agent will then receive the items and deliver them.

4.6 Reset

The way we made the system complete more jobs was to simply reset the system after completion of a job. This reset was only a soft one. When the reset occurs, the overall plan is reset. That is, the agents will then again move to shops, buy, etc. The agents kept their items and the previous information about the shops is still there. But this reset would make the agents go to the shops again, the planner would find a job to get and plan buying, assembly and delivery again. We put a small threshold in to make sure that we did not end up with leftover materials. Since buying materials costs points and we want to maximise our score, there is no reason to buy items if there is not enough time to complete the job. So after delivery of a job, the system only resets if there is enough (estimated of course) steps left to complete the job. If the agents are already in the process of completing a job when this threshold is passed, they will continue the job since it would otherwise be a waste of materials.

4.6.1 Planning ahead

When a delivery has been planned, the planner can just continue and plan the completion of the next job right away. If the agent that is making the delivery is used in this new plan, the other agents will simply wait for this agent when they reach a point where they cannot continue without it. This usually happens in assembly.

4.7 Movement

In order to allow for fast calculations, we use Euclidean distance in distance calculations. This makes sure that the system is not too slow to respond. It does introduce some minor errors in the system, but we introduced an approximation that allows for these errors to be rare.

The agents will keep track of their own charge and how much is required for moving to a specific location. In order to avoid agents running out of charge, the agents will figure out if they are always within reach of a charge-station, that is, they can get to a charging station without running dry. If an agent cannot reach its destination with its current charge plus a threshold, it will go to a charging station, charge, and then move to its destination. The threshold is a constant. This means that if an agent moves to its destination, it still has enough charge to move to the nearest charging station.

The size of the constant was chosen based on the size of the maps. We wanted it to be short enough for the agents not to require charge all the time, but at the same time large enough to make sure that agents can get to a charging station before it breaks down. We

used the existing maps and analysed their size and the distance from the location furthest away from the charging stations. Assuming other possible maps have almost the same size, this approximation should hold up in the majority of the cases.

The fact that the threshold is a constant and our distance is Euclidean (and not 100% precise) means that it is possible for agents to run dry and require breakdown service. But because of the price of the breakdown service, we would like to avoid it. We have seen rare cases where agents break down, but if that happens, then another agent would be able to take its place instead. This works because we have idle agents that can take its place. We have not seen more than one break down in a simulation, so we assume it will never happen.

4.8 Posting jobs

Sometimes an agent is not part of the group of agents that is currently solving a job. Thus, this agent can use its time to post a job. Posting a job can potentially confuse the other team and make them pick this job. The agent itself will figure out if it has nothing to do and will then post a job instead with a small chance. The chance is small in order to not flood the server with jobs.

The types of jobs can vary depending on strategy. Our idea was to post jobs that would provide a negative profit if chosen. The reward of the job will be low and will require only a small amount of materials. While the job may not be attractive because of the reward, it may be of interest to other teams because of the low requirements of items. Depending on the strategy for choosing jobs of the other team, they may be interested in easy jobs. There is no guarantee that the other team will pick up our jobs, but since we only use idle agents to post jobs, we are not hurting out own job-completion in the process.

Naturally, we do not want to pick up our own jobs. The central planner makes sure to not pick up the jobs that we have posted.

5 Tests

During the development of the system, we decided to create some tests for the system. These tests were not meant to test the system in a simulation, but only test small individual parts of the program. They were made to test correct functionality under different circumstances. We did not use any automatic test tool.

Our test environment consisted of a test simulation, that is, a world with shops, workshops, prices and so on. The tests would then input this test data as needed into the parts of the system we wanted to test. We could thus set up expectations for these parts of the system and they would then alert us of any errors in the output as we made changes.

Our tests mainly focused on the central planner. The functionality in the individual agents was not very complex compared to the central planner, thus we focused our testing there.

Some of our tests relied on previous tests. For example, in order to test assembling, we would have to have the agents buy some items first (we cannot just place items in the agents inventory since the central planner directly requires the agents to have buy orders planned). Thus, we would have to make sure that the test testing the buying part of the system succeeded before we could test assembly.

There were parts of the central planner that went un-tested, but these were simpler parts, and knowing that there were for example no errors in our tests, could help us determine the point of origin for the error much quicker, even if it was in un-tested code.

All in all the tests helped develop our system faster and more reliably.

6 What went wrong?

During the competition, we discovered a new bug in the system. This bug caused agents to move to different workshops. The system assumed that the coordination works, so we had no failsafe in place to fix this problem because we assumed it would never happen. This meant that the agents were standing in different workshops trying to assemble and assist. This lead to a dead-lock of our system. This bug went undiscovered in our testing, likely because the job that our team normally completed was taken or completed by the other team. This meant our team would have to take a different job that had the unfortunate properties of creating this bug.

Our system was constructed as a sort of waterfall. The system would plan buying items and tools, and the assembly system would use that information to plan assembly and delivery would use assembly information to plan delivery. This meant that if one bug occurred in the system, it would trickle down and cause further bugs down the line. We had constant problems in the buying and assembling and while we tried our best to fix these issues and make it work, we sadly never did. This meant that our system was not working very well and we did not complete a lot of jobs during the competition. This was by far our biggest problem. The above bug with different workshops only happened once during the competition, but because of the other problems mentioned here, our agents sadly ran into a dead-lock all the time.

6.1 Our team versus other teams

Because of the above-mentioned problems, our team did not fare well in the contest. We never completed many jobs and if teams did, we were often left in the dust. However, there were a few situations where that turned out in our favour. Sometimes a team would buy materials for more points than us. Then both teams would get stuck somewhere, leaving us in the lead. There was also an example of a team that completed jobs, but kept buying materials at an expensive price (or bought too many), costing the team more points than they gained, which also allowed our team to get in the lead.

7 Changes made during the contest

Our system about posting jobs when an agent is idle did not work very well, so we tried to make a small fix for it during the contest. We applied this fix, and it worked perhaps too well. Because there is between 5 and 10 idle agents on our team at any given point that meant that we posted that many jobs per step in the simulation. With a simulation of 1,000 steps, that is 5,000–10,000 jobs. It turned out that the server had trouble handling this many jobs and we ended up slowing the server down. Seeing as this was unintended, we quickly reduced the amount of jobs we post to about 1 or 2 per 10 steps (down from 50–100).

8 Possible future work

The first obvious point of improvement is to fix the bugs in the central planner when it plans buying and assembly. These bugs caused many deadlocks in our system throughout the contest and fixing them would allow our agents to complete jobs.

Many agents were left unused. A good idea would be to put the unused agents to work. We thought about trying to split the agents up into two groups of eight agents (or four groups of 4) and have them work independently of each other. This will ensure that all agents are used, but this introduces new problems and possible optimisations as well as requiring some information-sharing between the groups (unless we keep the central planner idea, but instead of making one plan for everyone, it makes a plan for each group).

Because our team only takes into account profit, there are a few pitfalls that could hamper the team. Our strategy worked fine in testing, but during the tournament we saw a team post jobs with absurd payouts, but also required an absurd amount of materials. While these jobs may have had the highest payout, picking them up would have been a mistake since our team is not equipped to complete a job of that size. Fixing this problem so we do not fall into this trap is a good idea.

9 Conclusions

Developing in Python gives us the freedom of a regular programming language, but also requires that we keep track of possible goals and beliefs manually, whereas an agent-oriented programming language like Jason or GOAL will help keep track of that.

The use of Python allowed us to easily create the architecture for the central planner. It did make representation of knowledge in agents more difficult that compared to e.g., GOAL, but since that was only used to a limited degree, it was not a big downside. Python had the built-in data structure ‘dictionary’ which we used extensively when dealing with items and their amounts.

The centralisation of the planner has both pros and cons. On the one hand, it makes data exchanging easier since all agents only have to send their data to the central planner and they do not have to coordinate with each other. However, the central planner also provided an easy way to make a plan since it was all knowing in that we did not have to take into account the planning with incomplete knowledge.

The overall architecture is easy to understand and most of the code is split into different modules, making maintenance fairly easy. There are some modules that are linked together in the sense that changing one requires changing the others as well. However, most of the modules are ‘black boxes’ in that they just need to respond correctly and the internal working can then be changed at will without affecting the rest of the system.

During this project, we learned a lot about how to apply different techniques and strategies about multi-agent systems that we had learned throughout our years at university.

The project was a great way to get a hands-on approach with actual problem solving. Trying to figure out an initial strategy was interesting. While we did have big problems in the system, we still managed to create our strategy that would function on a theoretical level.

10 Multi-agent programming contest 2016: retrospection

10.1 Team overview: short answers

10.1.1 Participants and their background

What was your motivation to participate in the contest?

> Our motivation to participate in the contest was to increase our experience with many of the practical, implementation and theoretical aspects and challenges of constructing a multi-agent system.

What is the history of your group? (course project, thesis, …)

> We have both had courses in artificial intelligence and multi-agent systems before, so we already had the knowledge required to go into the contest and design a multi-agent system.

What is your field of research? Which work therein is related?

> We are from the Department of Applied Mathematics and Computer Science at the Technical University of Denmark (DTU). We are part of AlgoLoG, the Algorithms, Logic and Graphs section, which is responsible for the Efficient and Intelligent Software study line of the MSc in Computer Science and Engineering program.

10.1.2 The cold hard facts

How much time did you invest in the contest (for programming, organising your group, other)?

> We used about 300 hours in total. We had both expressed interest in working with multi-agent systems so getting the group together was a quick job. Organizing and designing the group/project took of course some time, but the vast majority was used on programming.

How many lines of code did you produce for your final agent team?

> In total we wrote 2,149 lines of code for the system.

How many people were involved?

> Our project was built upon an already existing system made by one person. Our group of two then continued the work on that project.

When did you start working on your agents?

> We started August/September 2015.

10.1.3 Strategies and details

Many of these questions are answered in detail in the main report.

What is the main strategy of your agent team?

> We focus on completing one job at a time. We buy items, assemble the required products, deliver them and repeat.

How does the team work together? (coordination, information sharing, …)

> The system is built with a central planner. All agents report their percepts to the central planner so it has all the current information. This planner then makes a plan that tells all the agents what to do.

What are critical components of your team?

> The central planner is obviously critical since it does 90% of the computation.

Can your agents change their behaviour during runtime? If so, what triggers the changes?

> The overall strategy of the team cannot change. The planner will of course always try to execute this plan as best it can which will lead to slight variations, but there are no alternate strategies.

Did you make changes to the team during the contest?

> Yes. We found that our agents weren't posting jobs, so we fixed that bug during the contest. However, we then posted too many jobs and ended up almost crashing the server. We then reduced the amount of jobs we posted by a lot.

How do you organise your agents? Do you use e.g., hierarchies? Is your organisation implicit or explicit?

> The agents aren't organized as such. The central planner will simply use the agents it needs for a given job. Some agents are given priority over others in certain situations (where carrying capacity or speed might be good), but otherwise no.

Is most of your agents' behaviour emergent on an individual or team level?

> Most of the behaviour is on a team level since the planner will coordinate everything between the agents such as buying, assembling and assisting. The agents do have some limited individuality in that they keep track of charge themselves and must plan their own routes to their destinations.

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

> The planner is able to plan buying and assembling once it has acquired a job. Once the items are assembled it can plan delivery and already start planning for the next job. The amount of steps this include varies a lot depending on the job and the agents positions.

If you have a perceive-think-act cycle, how is it synchronised with the server?

> We do not have one.

10.1.4 Scenario specifics

How do your agents decide which jobs to fulfil?

> The central planner will find the job with the highest profit. It will send agents to the shops and compare the prices of products so it can calculate the potential profit of a job.

Do your agents make use of less used scenario aspects (e.g., dumping items, putting items in a storage)?

> No.

Do you have different strategies for the different roles?

> No. The only difference is that the planner might use different agents for different parts of a job. E.g., when delivering products, a fast agent is preferred.

Do your agents form ad-hoc teams for each job?

> Yes. The central planner will find a set of agents that can best complete a given job. It doesn't take into account pre-determined teams or if an agent was active on a previous job.

What do your agents do when they do not pursue any job?

> They will either be in a shop to provide the planner with information about stock and prices, or they will be posting bogus jobs to the server to confuse the enemy team.

10.1.5 And the moral of it is ...

What did you learn from participating in the contest?

> We learned about the actual problems one can face in a multi-agent system when applied to a real scenario. We learned about different strategies and counter-strategies.

What are the strong and weak points of your team?

> Technically and ideally, we can complete any job. Doesn't matter how complex the assembly is. Our weak point is definitely the apparent bugs in the system that caused our agents not to work most of the time.

How viable were your chosen programming language, methodology, tools, and algorithms?

> Python is a powerful language with many tools. We were able to respond very quickly and never timed out even though we had complex algorithms in our system.

Did you encounter new problems during the contest?

> Yes we encountered a new bug. Our agents were supposed to move to the workshop to assemble products, but one of our agents got wrong information somehow and moved to the wrong workshop. This meant that the agents could not assemble the products and complete the job.

Did playing against other agent teams bring about new insights on your own agents?

> Yes. Other teams were never using the workshops and instead just completing jobs that required no assembling. This showed us that our strategy was possibly very slow, however, the other team would limit the jobs they could complete because of this.

What would you improve if you wanted to participate in the same contest a week from now (or next year)?

> First of all, we would fix our bugs. Second, we would like to split the agents up into two teams of 8 (or four teams of 4) and complete more jobs simultaneously.

Which aspect of your team cost you the most time?

> Programming the buying and assembling algorithms took the most time since they were the most complex.

What can be improved regarding the contest/scenario for next year?

> It would be nice to give a bigger incentive to use the less-used scenario aspects such as storing or dumping. They were hardly used this year because they seem unnecessary in most situations.

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

> The bugs in our system caused a dead-lock most of the time, which sadly meant we often just stopped working. Some other teams performed worse because they used too many points of charging / buying items and not getting enough profit.

Acknowledgements

Thanks to Per Friis for IT support and also thanks to Mikko Berggren Ettienne and Steen Vester for the Python code which we have used as a starting point. Also thanks to Oliver Fleckenstein and Helge Hatteland for comments on a draft.

References

Boss, N.S., Jensen, A.S. and Villadsen, J. (2010) 'Building multi-agent systems using Jason', *Annals of Mathematics and Artificial Intelligence*, Vol. 59, Nos. 3–4, pp.373–388, Springer.

Ettienne, M.B., Vester, S. and Villadsen, J. (2012) 'Implementing a multi-agent system in Python with an auction-based agreement approach', *Lecture Notes in Computer Science*, Vol. 7217, pp.185–196, Springer.

Vester, S., Boss, N.S., Jensen, A.S. and Villadsen, J. (2011) 'Improving multi-agent systems using Jason', *Annals of Mathematics and Artificial Intelligence*, Vol. 61, No. 4, pp.297–307, Springer.

Villadsen, J., Jensen, A.S., Christensen, N.C., Hess, A.V., Johnsen, J.B., Woller, O.G. and Ørum, P.B. (2013a) 'Engineering a multi-agent system in GOAL', *Lecture Notes in Computer Science*, Vol. 8245, pp.329–338, Springer.

Villadsen, J., Jensen, A.S., Ettienne, M.B., Vester, S., Andersen, K.B. and Frøsig, A. (2013b) 'Reimplementing a multi-agent system in Python', *Lecture Notes in Computer Science*, Vol. 7837, pp.205–216, Springer.