Multi-Agent Programming Contest 2016: lampe Team Description

Philipp Czerner, Jonathan Pieper

Department of Informatics, Clausthal University of Technology,
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
E-mail: {philipp.czerner, jonathan.pieper}@tu-clausthal.de

Abstract In this paper we describe our participation in the 2016 edition of the Multi-Agent Programming Contest as team 'lampe', which was developed in C++ without the use of any agent-specific technology. It utilizes a centralised approach focusing on planning a job's completion in advance.

1 Introduction

The Multi-Agent Programming Contest (MAPC) is an annual competition with the goal of advancing the field of multi-agent system development and programming. We participated in this year's contest as team 'lampe', taking third place among five participants. Our team consists of two undergraduate students.

We initially designed our system as a course project with the goal of both testing the scenario of the 2016 MAPC and evaluating the use of the C++ platform without any specific agent technology. After deciding to also participate in the contest, we refined our solution and improved its stability. Our main motivation was to test our implementation in a live environment against real opponents.

The library we developed in our course project was originally called 'lampe', which is an acronym for 'Library for Agent Manipulation and Planning Efficiently'. We also used this name for our team.

2 System Design and Implementation

Our design focused on creating a system for this year's MAPC, without using agent-specific technology or architectures typical for agent programming. Instead we employed conventional programming techniques.

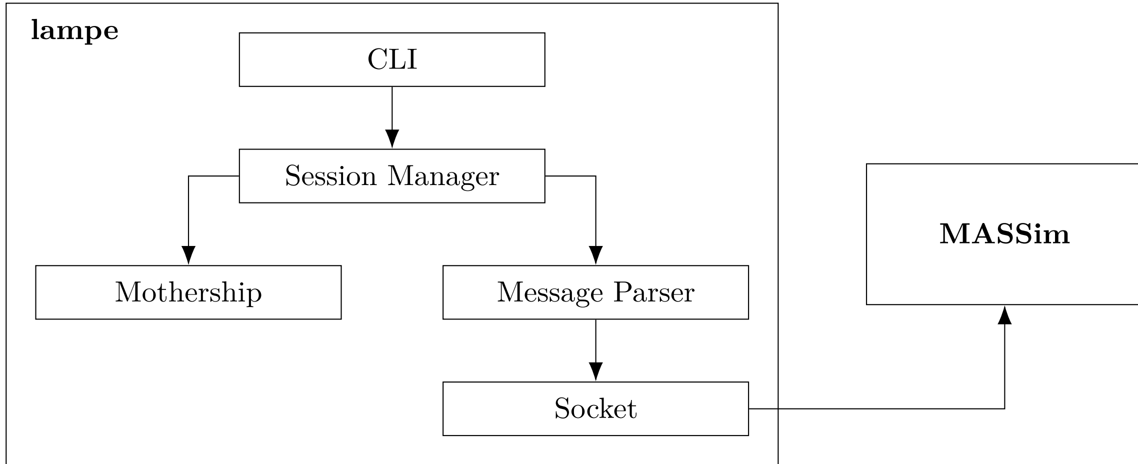

Figure 1: Structure of the framework

2.1 Architecture

We chose to organize our agents in a centralised fashion, meaning that a single entity holds all data and controls the agents. The rest of our system is designed as a framework, which manages the connection with MASSim and parses the messages sent to the entity. Then, the entity processes this information and returns actions for all agents. It communicates with the framework only over this interface; no other data is shared. We call the specific interface implemented by the entity in our framework 'Mothership', as well as the specific implementation of that interface used in the contest.

The general structure of the framework is shown in figure 1. The user configures the application via a command-line interface, which in turn calls the session manager, to handle the necessary state of a connection with MASSim over multiple simulations. The session manager relies on the message subsystem to communicate with MASSim over a TCP/IP socket. It instantiates a Mothership for each simulation and transfers the perceptions and actions between the Mothership and the message layer.

The system is designed to allow for different implementations of the entity using the same interface, making them easily replaceable. We did only implement a single version, which was used in the contest. Its agent architecture is shown in figure 2, with arrows indicating data flow.

Perceptions of all agents are stored in a single location. There is some bookkeeping to infer additional information as well as to update internal state. The data is then used by the job analysis to generate an execution plan for each job, which consist of a series of steps to complete the jobs. The dispatcher uses these to rate the profitability of jobs and select a job to pursue. The steps of the execution plan (called requirements) are then dispatched to the agents and executed.

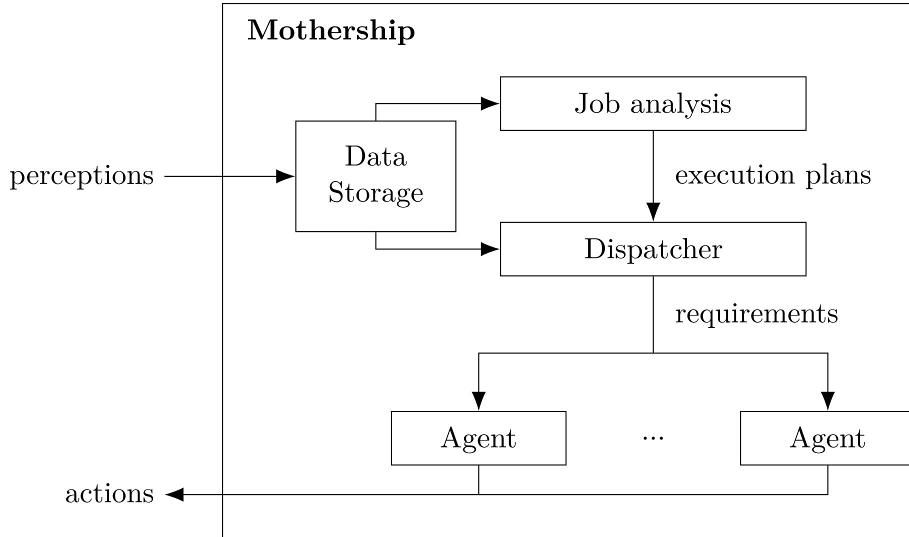

Figure 2: Agent architecture

This architecture is specific to the strategy employed, which is described in detail in section 3.

Due to employing a single entity to control all agents, there is no need for agent communication, nor do the agents need to coordinate their actions. Consequently, our system lacks any components for message passing between agents or similar facilities.

2.2 Implementation

Our system was implemented in C++, without the usage of any agent-specific technology. We chose C++ due to our familiarity with the language as well as the freedom to build our own abstractions on top of the data. While another benefit of C++ is its efficiency and potential for high performance, we did not find any part of our platform to be performance critical and did not perform any optimisations. There are no specific benefits of C++ with regards to our architecture or strategy; the system could easily have been implemented in a comparable, general-purpose programming language.

Choosing C++ required us to implement a framework to handle the communication with the MASSim platform. This framework connects to MASSim using TCP/IP sockets. It converts the transmitted XML into C++ data structures and vice versa. The framework also handles authentication and provides a command-line interface.

The entity implementing the Mothership interface is implemented as a single class containing all simulation data as members and a handful of methods. It is responsible for all agent behaviour. Most parts of our strategy are well suited to being implemented using (long) linear pieces of code organized in a few methods, making further encapsulation unnecessary. In fact, the three main components job analysis, requirement dispatching,

and requirement execution (the agents) are implemented using a single method each.

The entity consists of 882 lines of code. (All counts exclude comments and blanks.) The whole project has 4805 lines. A large part is responsible for communication with MASSim (1200 lines). We wrote some experimental agent code, which was not used during the contest; it amounts to 1027 lines. The code that provides the application, manages the connection to the server and interfaces with the operating system is 1084 lines long. The remaining 612 lines consist of custom containers and general convenience functions.

We decided not to use any advanced C++ features or modelling techniques (such as templates, inheritance or exceptions) for the agents' logic in favour of a straightforward approach. Similarly, we did not distribute our agents over multiple machines for the contest and ran the program inside a single thread.

3 Strategy

Our strategy is based on a central entity controlling the agents, which collects and processes all perceptions and then dispatches requirements to individual agents. Requirements are an abstraction over actions, enabling multi-staged operations, managing the charge of the agent and handling failed actions transparently. (Requirements are discussed in section 3.3 in more detail.) They are small enough to be processable by a single agent. The agents execute their respective requirement, yielding an action that is then returned to the server.

The team tries to complete one job at a time, using exactly the resources needed for the job. No items are bought speculatively; we wait until we find a profitable job and then buy the items required to complete the job. Completing a job may involve multiple steps of buying and assembling items, each possibly requiring multiple agents to cooperate. We deal with this by constructing a dependency tree in advance, which consists of the requirements necessary to complete the job, together with their dependencies.

Thus, we have three components: the job analysis to build the dependency tree, the dispatcher, which executes this tree by assigning requirements to agents, and the finite state machines that execute the requirements, yielding the actions for the agents.

We ignore many aspects of the simulation, such as hidden information (except the prices of items), transfer of items between agents, storage facilities (apart from job deliveries), dump locations, auction jobs, posting jobs, the street grid, and the other team.

Figure 3 shows the general strategy of our team. First, we gather data that is not available globally, but limited to agents near a certain facility. This allows us to estimate the value of jobs. If a profitable job was found, the actions necessary for its completion are dispatched, until the job is either completed or cancelled. We then repeat this process and continue analysing available jobs.

This is a simplified overview; gathering data and estimating the profitability of jobs happens simultaneously whenever there is no active job. It does, however, reflect the observable behaviour of our agents in practice. Usually all necessary information is

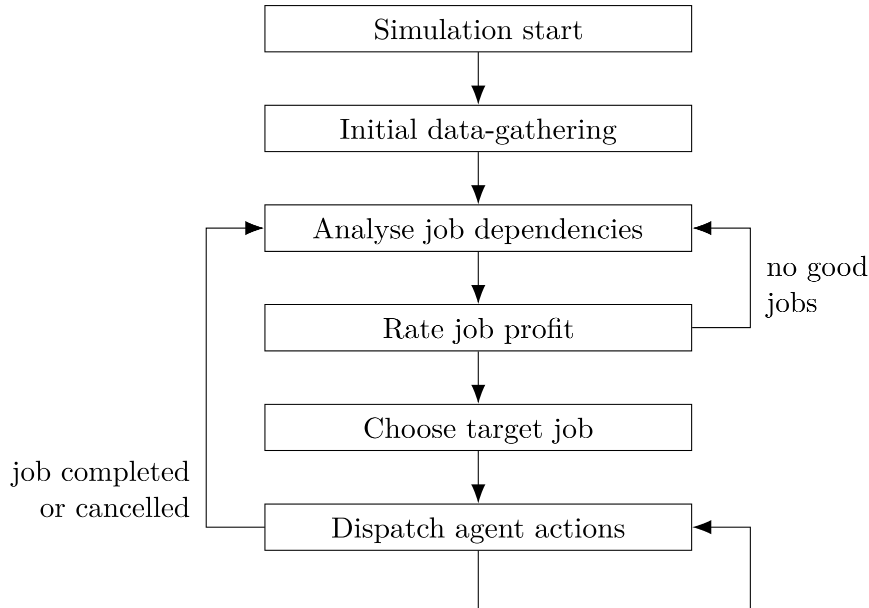

Figure 3: Overview of the strategy

collected at the beginning of a round, and after completing a job another profitable job is found right away.

3.1 Data Gathering

Price and amount of items in stock at the various shops are only visible to agents near those shops. In order to gather this data, agents are sent out to the different shops at the beginning of each simulation. This happens whenever the team is without a job and there are shops left to explore.

The fuel requirements for this are moderate and it does not represent a loss of time, as the search is conducted only until a profitable job has been found. Items with an unknown price can be bought, but are assumed to be highly expensive. Whenever the team has completed (or failed) a job and is waiting for the next one, this search continues, until all shops have been visited once. Since the price of a shop item is static and the amount is assumed to be sufficiently high, this information is used throughout the simulation.

In order to gather this information, every shop is successively assigned to the nearest available agent, excluding trucks due to their low speed. Faster agents are preferred (this holds true for almost every assignment the dispatcher performs).

Most decisions are based solely on the state of the requirements and the most recent perception. We derive knowledge about the state of the world in only two cases, item prices and items delivered for cancelled jobs. The latter is necessary to retrieve the items for a later job.

3.2 Job Analysis

Before any job is selected for execution, it is first screened for profitability and feasibility. First, we generate a list of necessary steps to complete the job. For each item there may be multiple possibilities of acquisition, these include buying, assembling or taking it from an agent's inventory or a storage facility. The latter one is a fallback if the agents do not manage to complete the job, it only retrieves items that have previously been delivered to a now cancelled job.

If an item can either be bought or assembled, in general the latter yields more profit and is preferred. However, in order to avoid problems with completing jobs in time, agents prefer to buy items more than two levels deep in the recipe tree, with the first level being the items requested by the job.

To assemble an item, the routine recursively runs on the ingredients (there can at most be one recipe for an item) and tools needed. There is a separate requirement for each ingredient and tool, and an additional one for crafting the item. To buy an item the routine tries to find the shop offering the lowest price. As the amount of items in stock at shops is quite generous, items are always assumed to be available.

Items not found during the data-gathering phase can be bought, but their price is estimated too high for these jobs to be chosen.

The analysis produces a list of requirements to complete a job. Any requirement may be marked as a dependency for another requirement. They are ordered as the results of a depth-first search of the dependency tree, to help the dispatcher with executing them linearly. Requirements for buying items contain the shop's location, which may later be optimised by the dispatcher based on the position of the agent.

Requirements are the basis to rate the profitability of a job. This takes into account

- the cost of all items that need to be bought,
- the cost of assembling the items at the workshop,
- and a constant for each requirement, roughly estimating the costs of the agents moving around.

The time needed to complete a job is also estimated as a function of the number of requirements and compared to the time left until the job has to be completed. From all completable jobs, the job with the biggest profit margin (which has to be greater than zero) is chosen to be the active job.

3.3 Requirement Dispatching

At any given time there may be at most one active job. This simplifies our implementation, but negatively affects our team's performance, due to a high number of idling agents towards the end of a job (see section 4.2 for a discussion of agent utilization).

The dispatcher holds a list of all requirements necessary to complete the job and dispatches them to the agents until all requirements have been dispatched. When dispatching a requirement, the dispatcher must ensure that the agent is able to complete

it. This turned out to be nontrivial, as there are many conditions to be checked. A deadlock may occur if all agents are assigned requirements that are dependent on a requirement that has not been dispatched. The dispatcher avoids this by assigning the requirements in the order of a depth-first search of the dependency tree and waiting until all dependencies for a requirement have been dispatched before moving on to dependencies of a different requirement.

Only one workshop is ever used, it is chosen to be in the centre of the map.

There are five different types of requirements: Get-Item, Buy-Item, Craft-Item, Craft-Assist and Visit. The former three first acquire the item, and then either assist a subsequent assembly or deliver it to the job's storage facility. Craft-Assist also performs the latter step, but it does not acquire an item first. It is used for tools and items in the inventory of agents that were left over by the previous job. Visit is for the data-gathering phase and causes the agent to move to the facility.

The actual dispatching is based on the type of the requirement. An overview is given in algorithm 1. For each available requirement an appropriate agent is chosen, subject to the following conditions:

- The agent must not already be assigned a requirement (there are exceptions, see below).
- The agent must be able to carry at least one of the target items.
- If the target item is a tool, the agent must be able to use that tool.
- If the target item is a tool, the agent must not be a truck.

Whether an item is a tool or not does not depend on the item, as any item might be used both as material and as tool in different recipes. Instead, this is determined by the job analysis when the recipe is selected.

In general, an agent may only execute one requirement at a time. However, the dispatcher may choose to upgrade an existing requirement. For example, any agent delivering an item to assembly may be upgraded to also execute the assembly.

The dispatcher may also choose to discard a requirement implied by the agent's requirement. An example would be the requirement to provide a tool for an assembly; an agent carrying the tool and participating in the assembly would fulfil this requirement implicitly.

An agent unable to carry the full amount of items may be assigned a partial requirement. This is needed, as for some jobs or recipes not all items *can* by carried at the same time by any one agent. It is also possible to only partially discard a requirement.

It is necessary to keep track of the items in the inventory of an agent. Some may be needed to execute its current requirement or even a future one. The dispatcher marks these items as reserved until a specific requirement has been completed and takes care not to assign requirements that would make use of these items. Assembling items does not respect the reserved items, thus it is possible for an item to be taken out of the inventory of a different agent than intended. This is the reason for the expiration on

Algorithm 1 Dispatching a requirement

```
Input a requirement r = (type, item, amount, is_tool)
Output whether the requirement has been dispatched
for all agents a do
    if is_tool and (a cannot use item or a is truck) then
        continue
    end if
    if type ∈ {Get-Item, Buy-Item, Craft-Item} then
        available := Number of items item that a could carry
    else if type = Craft-Assist then
        available := Number of unreserved items item that a carries
    end if
    if available = 0 then
        continue
    else if available > amount then
        available := amount
    end if
    if a is executing requirement r' then
        if r' implies r then
            Reserve available items item of a
            amount := amount − available
        else if r implies r' then
            Reserve available items item of a
            amount := amount − available
            Assign r to a
        end if
    else
        Reserve available items item of a
        amount := amount − available
        Assign r to a
    end if
    if amount = 0 then
        return true
    end if
end for
return false
```

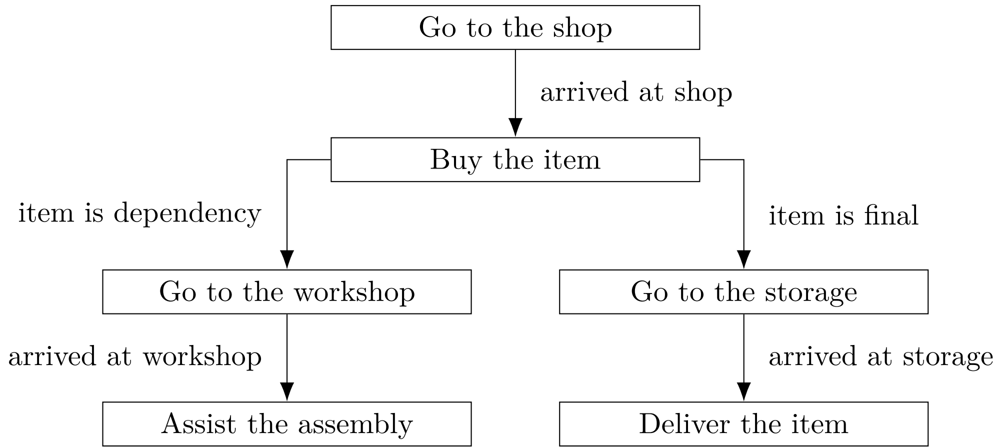

Figure 4: Executing a Buy-Item requirement

requirement completion; when the assembly is done, the items intended to be consumed are no longer reserved and can be used by subsequent requirements.

When dispatching a requirement to buy an item, the corresponding shop may be changed, based on the distance between shop and agent and the item's price in a specific shop.

Completing the job relies on all agents to complete each requirement that has been assigned. Even a single one failing can cause agents to wait on ingredients for an assembly until the job is cancelled. Although an agent should not be able to fail any requirement, programming errors do exist. To eliminate the period of waiting, we implemented a watchdog, which triggers whenever a few rounds have passed without a successful action. This causes the team to abandon the current job and look for a new one.

3.4 Requirement Execution

The agents execute their current requirement via a finite-state machine. The requirements include travelling to the shop or workshop, purchasing or assembling an item, and finally delivering the product to either the storage for the job or the next assembly site. Figure 4 shows an example of the stages necessary to execute a Buy-Item requirement. Each stage of execution may span multiple steps of the simulation. To handle errors, for example due to random failures, each action is tried until it succeeds or the current job is cancelled.

While travelling, a subroutine estimates the distance an agent can travel using the current charge. In general the agent tries to charge—in case the current target is out of reach—by choosing a charging station that minimises the total distance travelled (first to the station, then to the target). However, an agent may choose to directly visit a charging station, if it would risk running out of fuel.

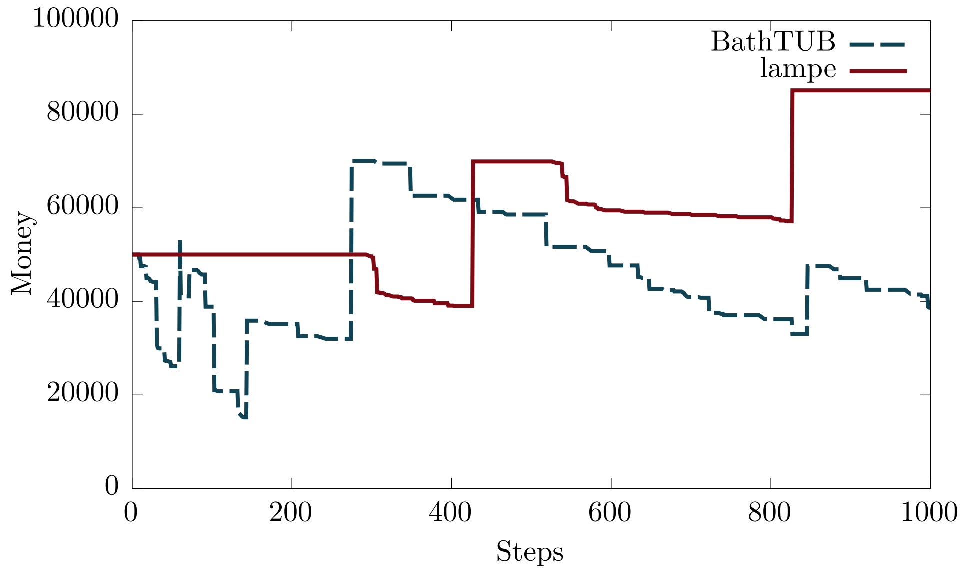

Figure 5: Money during simulation 2 against BathTUB

4 Evaluation

During the first day of the contest, we faced technical difficulties and were not able to perform well. There was a bug inside the dispatcher, causing incorrect behaviour for assembly involving tools. Additionally, our communications layer depended on the socket buffering, which failed in some cases when running over an internet connection. This bug was still present during the first match of day two, affecting most of simulation 1 against team Python-DTU.

4.1 Conservative Heuristics in Match Against BathTUB

In our match against team BathTUB simulation 2 exhibited interesting map generation. Before taking any job, we estimate its potential profit, so that we do not take any jobs when we expect to lose money. Sometimes this causes our agents to idle for extended periods of time. As shown in figure 5, there was little activity until step 300, between steps 430 and 520, and after step 830. (Our agents did move in the first 20 steps, but no agent needed to charge.) Our heuristic is not perfect, as team BathTUB demonstrates by making a profit during the first period.

We completed only 2 jobs in total, compared to the 5 jobs finished by BathTUB. Our approach forms a distinct pattern, consisting of a sharp decline at the beginning, when most items are bought, followed by a long period with little agent activity, until the job is completed. Planning the jobs in advance in a centralised fashion, enables a high degree of efficiency in terms of the money we spend, but it negatively affects our agent utilization.

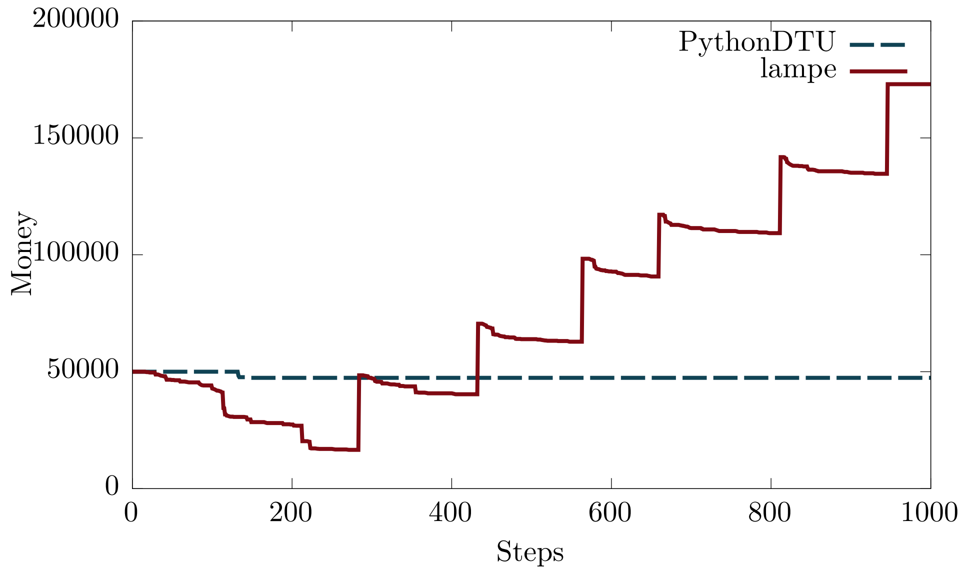

Figure 6: Money during simulation 2 against PythonDTU

4.2 Low Agent Utilization in Match Against PythonDTU

Simulation 2 between team PythonDTU and us is shown in figure 6. The general pattern is similar, but jobs are found more easily and completed in a shorter amount of time, resulting in a higher income. This pattern represents an ideal performance from our agents, without major interruptions.

Figure 7 shows the number of successfully completed actions (excluding skip and abort actions) in ten steps wide intervals. For comparison, the activities in simulation 3 between teams Flisvos-2016 and PUCRS are also shown. There is some noise originating from actions without effect, such as charging an agent with a full battery.

Our agents completed six jobs in total, at steps 284, 433, 564, 660, 812, and 946. Following each job, there is a peak in activity, as the agents start to collect the items for the next one. Even in this phase, we reach an agent utilization of only 80 actions per ten steps, which is 50% of the theoretical maximum. (There are 16 agents per team, with a maximum of one action per step, which yields an upper bound of 160 actions per ten steps.) After the initial peak, our agent utilization declines until the job is completed.

Team Python-DTU did not perform many meaningful actions in this match, after their initial peak at step 120 the activity consists mostly of noise.

In the match between Flisvos-2016 and PUCRS the teams achieved a score of 350240 and 761920, respectively. PUCRS' agents were generally more active than ours, having both higher peak activity and less idle periods. In total our agents executed 3337 actions in our match and the agents of team PUCRS did 5254 in theirs. Flisvos-2016, surprisingly, executed only 1901 actions in total. This points to a high degree of efficiency of their agents, considering their score in this match and overall performance during the contest.

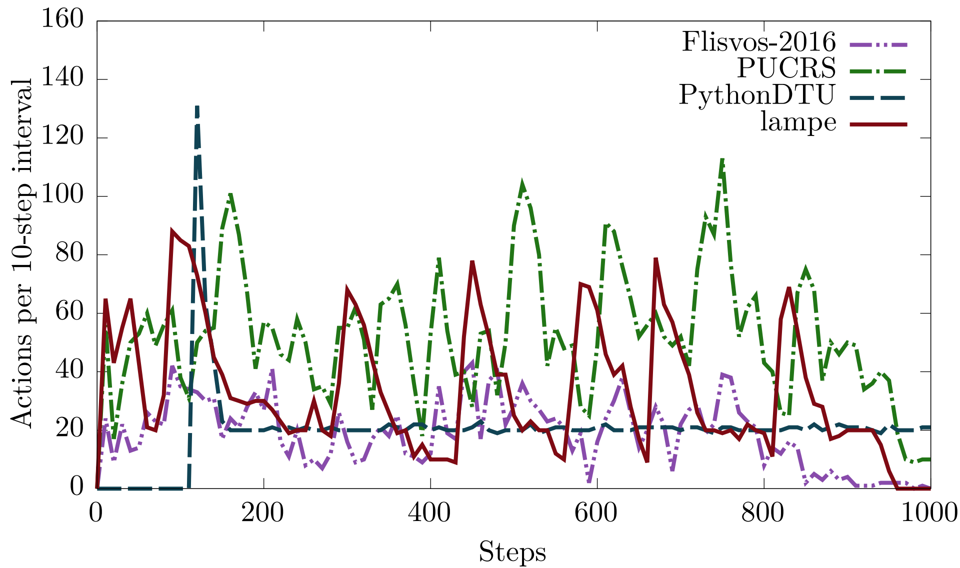

Figure 7: Activity of different teams

The number of actions of both teams PUCRS and Flisvos-2016 is more consistent over time than ours, they do not show the periods of inactivity exhibited by our team towards the completion of a job. The reason for this may be a more decentralised strategy that does not require the synchronisation of all agents at once.

4.3 Technical Difficulties in Match Against PUCRS

Figure 8 shows simulation 1 against team PUCRS. Our agents completed 8 jobs in total, finishing with a score of 63776, compared to 24 jobs finished by PUCRS with a score of 73549. This simulation had low rewards per job, resulting in comparatively low scores for both teams.

The pattern of our team's money is similar to the other simulations, in that there is a steep decline at the beginning of a job, followed by a stable period until the job is completed. However, there are additional declines without having completed a job, for example in steps 468 and 887. These are caused by a bug, preventing our agents from cooperatively assembling items involving tools.

When there is no progress in completing a job, meaning that no successful actions (apart from abort and skip) were completed, the watchdog triggers and the active job is discarded, causing the team to look for a new job. This allows our agents to quickly recover from a deadlock scenario. The job analysis takes the existing inventory of agents and items delivered to cancelled jobs into account when selecting the next job.

During this simulation, our recovery features proved to be quite effective, as our agents managed to turn a profit even while unable to complete the more complex jobs. However, the aforementioned problems remain and team PUCRS' agents were able to earn a higher

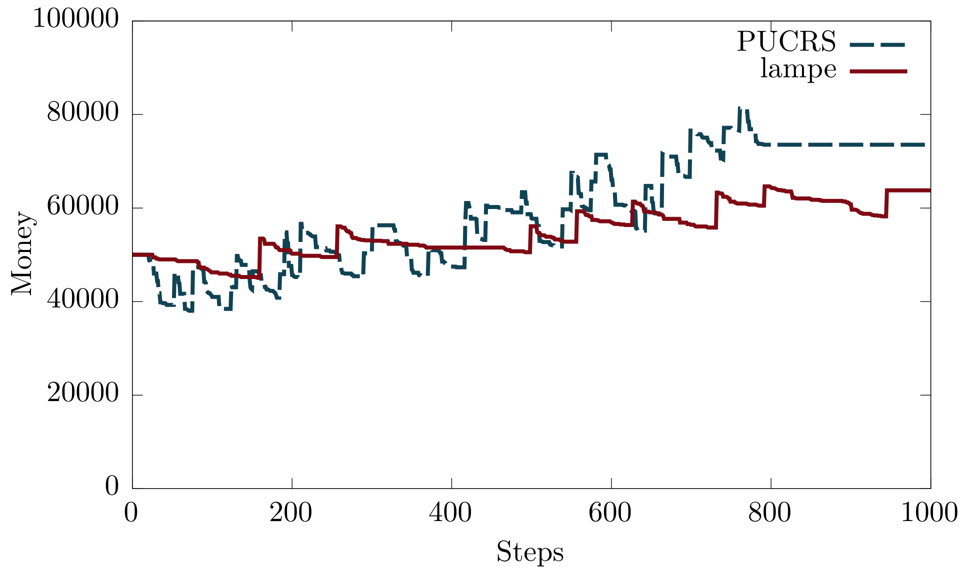

Figure 8: Money during simulation 1 against PUCRS

score, even without completing any jobs in the last 239 steps. We believe their key advantages to be a higher agent utilization, and a focus on smaller jobs.

On average, we earned 1722 points per job, while PUCRS gained 981 per job. They were able to complete three times as many jobs, leading to the final score. Our strategy focuses on being able to complete any job, even those involving many agents and multiple stages of assembly. We are less efficient in completing smaller jobs due to the overhead imposed by having only one active job at a time. This leads us to prioritise longer jobs with higher rewards. It is necessary to synchronise our agents with regards to cooperative assembly of items, limiting our agent utilization significantly.

4.4 Other Problems During the Contest

Our team is able to complete jobs quite consistently, but on some maps we run into problems with our routing. The heuristic responsible for charging the agents is based on rectangular distance to the target. In general, a heuristic can either overestimate the actual distance, or underestimate it. The former causes excessive charging (reducing the efficiency of travelling), and may result in loops, where an agent can never reach a target, always returning to the charging station. The latter results in agents without charge, needing to call breakdown service, which has severe penalties regarding both time and money. Our heuristic exhibits both, but it does tend to overestimate distance.

Another problem is our disregard of locality, both in estimating expenses and in planning routes. Ideally the job analysis would consider the location of shops, workshops and storage facilities and adjust its estimate of the job's cost and duration. For the dispatcher, most of our efforts have focused on selecting any agent that will complete the

requirement. Little time has been invested in dispatching to the most suitable of such agents, including the rather obvious consideration of the agent's position.

Loops between one charging station and a target were common enough that we detect them and order the agent concerned to proceed to the target regardless of charge. Having an agent running out of charge is preferable to having loops, as the latter prevents any progress in completing the job. Unfortunately, loops involving two charging stations are not detected and have been observed in some of the matches (e.g. simulation 1 against team BathTUB).

4.5 Design and Implementation Choices

Our architecture is centralised, which eliminates the need to communicate between agents and simplifies agent synchronization, enabling a sequential implementation. We believe this to be well suited to conventional programming languages. There are drawbacks in not being able to distribute the system across multiple machines or cores, which would provide performance benefits. However, we did not experience any performance problems during either the contest or testing.

Our architecture was built specifically for our strategy in this year's contest and is not reusable. There are further problems in using a centralised strategy, which were discussed in the previous sections.

The system was developed in C++ without the usage of any agent-specific technology. The communication with MASSim was implemented from the ground up, using the XML-based protocol over TCP/IP sockets. Other approaches would be the utilization of an existing platform from C++ or the use of an agent-oriented programming language.

Using an existing platform has the benefit of not having to write code that interfaces with the MASSim platform and being able to focus on the logic controlling the agents. Code interfacing with MASSim makes up most of our codebase when counting lines (as discussed in section 2) and we spent roughly half of our programming time developing it, suggesting a large potential for time savings in this area. It is also worth noting that the bug in our communications layer mentioned at the beginning of this section, which prevented us from playing in two simulations, could have been avoided by using a stable framework.

However, there would still be a need to wrap or convert the data structures into a format, that can easily be read and written by C++ code, which makes up most of our current non-agent code. Furthermore, there is additional complexity in crossing language boundaries.

There are no specific benefits of C++ with regards to our architecture or strategy; the system could easily have been implemented in a comparable, general-purpose programming language. C++ in particular offers static typechecking, large freedom in the choice of abstractions and a potential for high performance.

Agent-oriented programming languages are designed to work well in multi-agent systems, such as the scenario used in the MAPC, whereas C++ is a general purpose language, without features specific to agent programming. As a result, our agent code is verbose and cumbersome to write. However, writing very explicit agent code can be beneficial, in

that it is easy to reason about the behaviour of code.

5 Conclusion

Our participation in the 2016 MAPC has been a valuable source of experience in developing a C++ application within the context of multi-agent systems. We felt that C++ was quite suitable with regards to our approach, allowing us to build our own abstractions and be explicit about all operations during our applications execution.

Our architecture enabled us to handle complex jobs having multiple levels of dependencies. It did suffer from low agent utilization, and much data with a potential for optimisation is not taken into account. The underlying routines handling the travelling of agents are not able to work consistently on all maps, causing additional problems.

Overall we are quite happy with our results, especially considering that this is our first time participating in the MAPC. The technical problems were unfortunate, but at least we did not run into any memory issues. Our system was competitive once we got it to work, but did not achieve the effectiveness of teams PUCRS and Flisvos.

6 Team overview: short answers

6.1 Participants and their background

What was your motivation to participate in the contest?
We wanted to evaluate our implementation regarding both its stability and competitiveness.

What is the history of your group? (course project, thesis, ...)
The system was developed within a course project.

What is your field of research? Which work therein is related?
Does not apply.

6.2 The cold hard facts

How much time did you invest in the contest (for programming, organizing your group, other)?
About 250 person hours were spent, about 200 hours to write code and 50 hours to write documentation.

How many lines of code did you produce for your final agent team?
The code base consists of 4805 lines of C++ code (excluding comments and blanks), of which 882 are describing agent behaviour.

How many people were involved?
Two people worked on the project.

When did you start working on your agents?
April 2016

6.3 Strategies and details

What is the main strategy of your agent team?
All decisions are made by a central entity. A job is chosen based on expected profitability, then a dependency tree is calculated and executed.

How does the team work together? (coordination, information sharing, ...)
Everything is centralised and planned in advance, this includes actions requiring multiple agents.

What are critical components of your team?
The three main components are the job analysis, which creates the dependency tree, the dispatcher, responsible for selecting new jobs and assigning tasks to the agents, and the finite state machines executing those tasks.

Can your agents change their behaviour during runtime? If so, what triggers the changes?
In the downtime between jobs the dispatcher will have the agents visit shops, to learn the prices of items.

Did you make changes to the team during the contest?
We fixed bugs that cropped up during the matches.

How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?
There is a central entity collecting the perceptions of all agents and assigning actions to each of them.

Is most of your agents' behaviour emergent on an individual or team level?
Only the travelling (including charging) happens on an individual level.

If your agents perform some planning, how many steps do they plan ahead?
The agents plan the execution of one job, which generally takes around 200 steps to complete.

If you have a perceive-think-act cycle, how is it synchronized with the server?
We react to messages sent by the server.

6.4 Scenario specifics

How do your agents decide which jobs to fulfil?
The dependency tree for each job is used to estimate the costs of completing a job. Whenever there is no active job, the agents choose the most profitable job that does not yield a loss.

Do your agents make use of less used scenario aspects (e.g. dumping items, putting items in a storage)?
No, we did not use these features.

Do you have different strategies for the different roles?
No, the dispatcher favours faster agents and trucks are excluded from certain tasks, but mostly the logic is the same for all agents.

Do your agents form ad-hoc teams for each job?
No, there is only one team and one job.

What do your agents do when they do not pursue any job?
They visit shops that have not been visited before to learn the prices of their items.

6.5 And the moral of it is . . .

What did you learn from participating in the contest?
We learned a lot about developing MAS related abstractions in C++.

What are the strong and weak points of your team?
Our team is able to complete even very complex jobs with a few dozen intermediate tasks. It is also quite good at ignoring unprofitable jobs. There are still some problems with the pathfinding and keeping all agents occupied.

How viable were your chosen programming language, methodology, tools, and algorithms?
C++ leaves the programmer a large degree of freedom in the choice of abstractions. It allowed us to express our ideas in a straightforward way and made it easy to reason about the behaviour of code.

Did you encounter new problems during the contest?
We encountered errors related to the interaction with the other team as well as a problem when running over an internet connection.

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

The other teams were better at utilizing the agents' time, this is something our agents were certainly lacking.

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

Important short-term improvements would be fixing the pathfinding and the ability to complete multiple jobs at once. In the long term, we would improve the dispatcher to better anticipate future events and remove some of the simplifications that were made during its design.

Which aspect of your team cost you the most time?

Most of the work went towards the dispatcher. Assigning tasks to the agents is subject to many constraints and our strategy is not able to compensate for dispatching errors well.

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

We feel that an increased bonus to assembly of items would make the scenario more interesting due to a higher need for agent cooperation.

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

Once the bugs were worked out, our team was able to generate profit very consistently over the course of a match. We did, however, suffer from low agent utilization due to our centralised approach, which allowed the other teams to complete many more jobs over the same time period.