Multi-Agent Programming Contest 2017: lampe Team Description

Philipp Czerner, Jonathan Pieper

philipp.czerner@nicze.de, jpieper@swbmail.de
Department of Informatics, Clausthal University of Technology

Abstract In this paper we describe our participation in the 2017 edition of the Multi-Agent Programming Contest as team 'lampe'. Our strategy was implemented in C++; it uses a centralised organisation of agents and evaluates different strategies based on an internal simulation of the future game state. Strategies are generated using handwritten heuristics in a randomised fashion, also relying on the internal simulation.

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 [1]. We participated in the 2017 version of this contest as team 'lampe', taking fifth place amongst seven participants. Our team consisted of two undergraduate students.

This is the second time team 'lampe' participates in the MAPC [2]. Our system was originally developed in 2016 as a course project and then further improved upon in advance to that year's contest. For this year, our system makes use of the framework we developed, which handles communication with MASSim and other organisational tasks, but the strategy has been completely rewritten.

The name 'lampe' is an acronym for 'Library for Agent Manipulation and Planning Efficiently' and used to denote both our team and the framework we wrote to implement our strategy.

This paper is structured as follows: Section 2 first introduces the general design of our system and gives a high-level overview of our implementation. We then provide a detailed description of our strategy in section 3, including the various layers and components involved. A brief overview of techniques used for implementation and debugging is given in section 4. Next, section 5 analyses some interesting games from the contest, discusses strengths and weaknesses of our solution and evaluates the choices we made in our

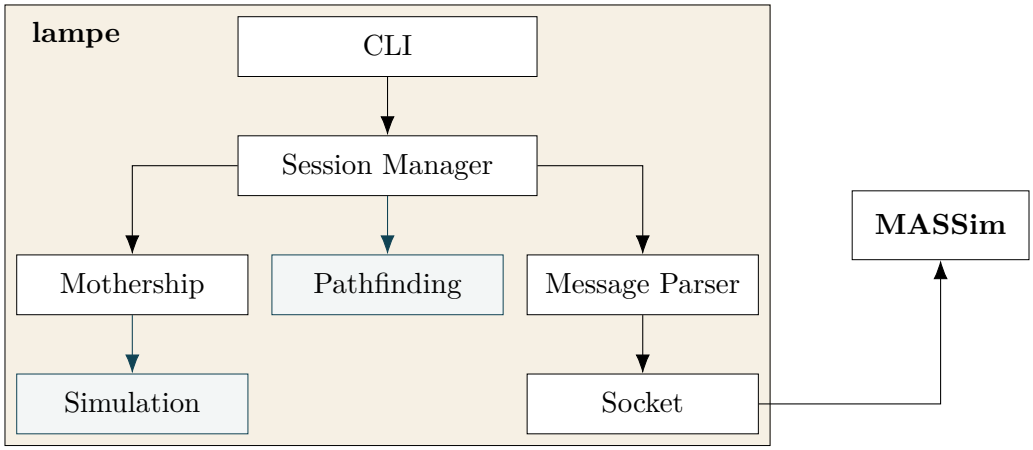


Figure 1: Structure of the framework (adapted from [2]). New parts are highlighted.

implementation. Finally, section 7 comprises our answers to the organisers' questions, giving an overview of our team and its strategy.

2 System design and implementation

In this section we briefly describe the overall architecture of our system, as well as its major components. Then, we give a general overview of our implementation of that architecture.

As our system is based on our work during last year's contest, most of the architecture surrounding the team's strategy was already in place. Consequently, our design work focused on our agents' logic and various supporting systems, such as the internal simulation or exact distance calculation.

We did not use any specific methodology for our software development, and opted against architectures and technologies typically used for agent programming, instead focusing on conventional programming techniques.

2.1 Architecture

We briefly recap our general architecture as used in last year's contest before specifying the changes and additions made.

Figure 1 shows the general structure of our framework. The application is invoked via a command line interface, which initialises the session manager, responsible for managing the connection to MASSim, authentication, loading the maps for pathfinding and controlling the lifecycle of the mothership. The mothership is the central entity controlling all agents, communicating to the session manager via an API. This makes it possible to plug in different motherships, a feature we used mainly for debugging.

The message parser transforms the XML-based messages exchanged with MASSim into our internal data structures and vice versa.

For this year's version, we added subsystems dedicated to pathfinding and the internal simulation. Additionally, we completely replaced our mothership, i.e. the part responsible for our strategy. Pathfinding parses the graphs output by MASSim (which uses the GraphHopper library internally) and provides a routine to quickly find shortest paths. It is imperative that these paths correspond exactly to the ones calculated by MASSim, as they are used to predict travel times.

The internal simulation tries to predict future events based on the current situation. It basically mirrors MASSim's internal workings concerned with advancing the game state. Since the simulation is non-deterministic and the opponents change the state as well, precisely simulating the game is of course impossible, wherefore the internal simulation can only ever be an approximation.

2.2 Implementation

We implemented our system using the C++ programming language. More specifically, we used the C++14 standard with some C++17 extensions that were supported by our compiler. Some features we deliberately abstained from, most notably exceptions and class hierarchies, both for their negative impact on code readability and performance.

Originally we chose C++ for our framework due to our familiarity with the language as well as the freedom in choice of abstractions it offers. With our new strategy, performance becomes an important factor, too. We believe that implementing our strategy in another comparable, general-purpose language such as Java or Python, while not impossible, would incur additional overhead in applying further optimisations, in order to get to an acceptable level of performance.

For communication MASSim's XML protocol is used via a TCP/IP socket. The message parser translates between XML and our data structures, and performs various transformations to make the data easier to use and more compact (e.g. strings are replaced by integers, coordinates are normalized and converted to fixed-point precision).

Our data structures used for representing the game state are unusual in the sense that they are stored contiguously in memory. This makes it difficult to apply structural changes to existing data (such as adding an item to a list), but simplifies copying and storage. Since the former happens very rarely during simulations (it is mostly agents changing positions and items changing amounts), this is a trade-off we are happy to make.

In total we wrote 8512 lines of code (all counts excluding comments and blanks), of which 2396 are implementing the strategy and internal simulation. The code implementing the communication with MASSim is 1051 lines long, and the pathfinding amounts to 996 lines. The rest consists of interaction with the operating system (1167 lines), as well as various utilities, data structures and debug code.

Last year's project consisted of 4805 lines, of which only 882 were controlling the agent behaviour. This illustrates that most of our work has gone into implementing a more complex strategy.

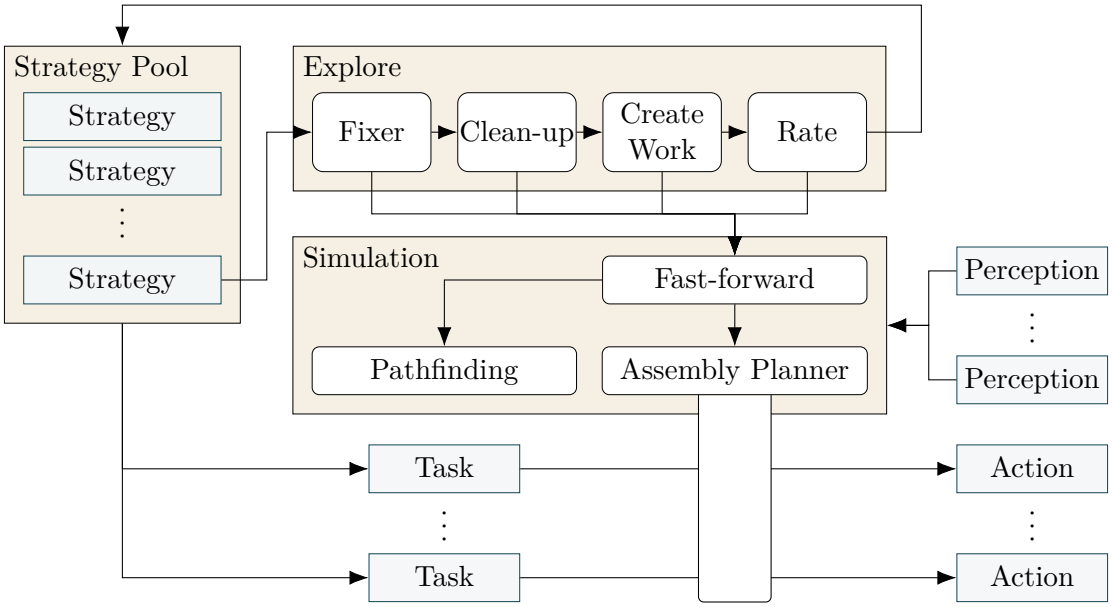


Figure 2: Overview of the general strategy.

3 Strategy

We organise our agents in a centralised fashion, meaning that there is a single entity, which collects and processes the individual perceptions and then dispatches actions for the agents to execute. We call the component responsible for implementing this entity *mothership*.

There are multiple layers of abstraction between the decision-making of the mothership and the actual game. Firstly, there are *tasks*, which are an abstraction over actions. Each tasks comprises an operation which could take multiple actions to complete, and handles failures of individual actions transparently. For example, the buy-item task covers the agent going to a shop (potentially involving multiple goto actions) and then buying the item (a single buy-action, or more if a random failure occurs). For more details, see section 3.2.

A *strategy* assigns each agent a number of tasks, to be completed in order. The maximum number of tasks queued up per agent is limited to 4 in our implementation. Strategies can be executed, resulting in a single action per agent being sent to MASSim. More importantly, we are able to internally simulate the results of a strategy, allowing the mothership to consider multiple possible strategies in advance.

A *situation* contains both a strategy and the current state of the simulation. The latter is mostly equivalent to the contents of the perceptions as received from MASSim, with duplicated data removed. Additionally, there is a small amount of bookkeeping to be done, in order to track information that is not present in the perceptions, most notably the state of delivered items.

The internal simulation of a situation for a certain number of steps—called *fast-forward* in our code—is central to the other high-level operations on strategies. This operation takes a situation as input and returns the situation that would result from executing the strategy for a certain number of steps. It applies all deterministic elements of the simulation, but cannot predict random events or the behaviour of the opposing team.

Most of the work is done in the *fixer*, which takes a strategy and addresses any issues during its execution. For example, an agent might not have enough charge to reach its destination, leaving it unable to move. Running out of charge will be noticed inside fast-forward, causing a charge task to be added to the agent. The fixer is very robust and able to deal (at least in principle) with all problems that can arise during the execution of a strategy. It is responsible for handling the dependencies arising during item assembly. Section 3.3 describes the fixer in more detail.

Other operations on strategies include *clean-up*, which removes superfluous tasks, *create work*, which searches for profitable jobs and adds tasks to complete them (see section 3.4), and *rate*, responsible for rating the expected score in any given situation (see section 3.5).

An overview of our general strategy is provided in figure 2.

The mothership maintains a pool of prospective strategies, that is initialised with the strategy chosen in the previous step. Until the time limit is reached, the mothership then explores the most promising strategy.

To explore a strategy, the mothership first applies the fixer. As the fixer only runs limited number of iterations, a strategy may not necessarily have all its issues resolved after this step. Subsequently, the mothership performs a clean-up of the strategy, as the fixer may create superfluous tasks (such as buying an item which is not needed). This happens when a task is removed while its dependencies remain. As those are difficult to determine ahead of time, it is easier to remove them afterwards in a separate clean-up phase.

If there are no errors present in the strategy, create work is called and tasks to complete profitable jobs are added. Finally, the strategy is rated, to update both its own value, and the expected exploration value of its parent.

After exploring strategies, the mothership selects the best strategy based on the rating of its situation (this does not consider the value of children of the strategy) and executes it. The resulting actions are sent to MASSim. When the next set of perceptions is received, all completed tasks are removed from the strategy, which then initialises the pool for the next phase of exploration.

There is no specific mechanism to encourage the mothership continuing in its previous course of action, so it is possible for it to pick a wildly varying strategy. However, our heuristics generally operate in a way that keeps working tasks unchanged, and the previous strategy is always part of the next pool of strategies. Consequently, any new strategy would have to achieve a higher rating to be adopted.

3.1 Comparison with Iampe 2016

As our strategy has been rewritten from scratch, there are only conceptual similarities.

Similar to last year's strategy, we used an abstraction to represent operations consisting of a sequence of actions. The original version was called 'requirement' and fulfilled a similar role to our current model of tasks, i.e. handling failures of individual actions transparently and grouping sequences of actions, making them easier to reason about.

However, tasks are more light-weight and thus allow finer control over an agent's actions. A task consists of travel to a destination and some kind of action being executed there. A requirement would additionally consider a delivery of the resulting item (if any) to a workshop or storage, involving two more stages of execution.

Furthermore, charging was built-in and handled transparently—an agent would locally decide whether to charge and which charging stations to choose. This feature was dropped in favour of a dedicated charge task, which is managed centrally by the fixer.

Requirements were executed via the use of finite-state-machines, keeping track of the different stages of execution locally in each agent. We have simplified the former part, rendering the execution of tasks stateless, only depending on the information about the game. Tasks are still executed locally, but there is an additional consideration specifically for assemblies. When multiple agents cooperate on an assembly, the assembly planner creates an execution plan for the current step, which the agents then use to synchronize their actions.

Switching from requirements to tasks enables the use of simple building blocks for our strategy, featuring very little local reasoning. Instead, decisions are made centrally, making it easier to take context into account and consider the strategy as a whole.

In more general terms, the biggest difference is in the handling of errors. Last year we were focused on *preventing* problems from occurring, always taking care to only assign requirements to agents that can actually execute them. Ensuring this beforehand was complicated and had to take a multitude of factors into account. Some heuristics had to be especially conservative in order to avoid running into problems later. When errors did happen due to bugs in our programming, our team was often not able to function at all, as a single failed delivery would prevent the completion of a whole job

This year, our strategy revolved around *handling* errors as they happen. However, rather than running into problems in the actual simulation, we use our internal simulation to find these problems in advance. Whereas previously our heuristics had to be quite clever to make accurate guesses about the future, we can now make use of the (simulated) future in our heuristics. These are then much simpler, not having to factor in any prediction.

Additionally, we are able to assign tasks optimistically. If any failure occurs as a result of a complex interaction, the simulation will notice this and in order to resolve the issue the full state of the simulation at the time of the error is available to the fixer.

The downside is the necessity of solving problems in an arbitrary strategy with complex interactions between agents, without running into a loop. This makes the implementation of the fixer quite difficult.

Table 1: The different task-types. Each tasks stores data about its location and items. The first operation is always travelling to the target. All operations make use of goto actions.

Name	Parameter	Second operation	Non-goto cctions involved
buy-item		buy item	buy
deliver-item	job id	deliver to job	deliver
retrieve-item		retrieve item	retrieve_delivered
craft-item	assembly id	assemble item	assemble, give, receive
craft-assist	assembly id	assist assembly	assist_assemble, give, receive
charge		charge the battery	charge

3.2 Tasks

Tasks are an abstraction over actions, describing an operation that takes multiple actions to finish. Table 1 gives an overview of the different types of tasks used in lampe. All tasks involve travel to a destination, using a direct path to the target. In particular, they are not taking the agents' charge into account while travelling, as this is considered at a higher level (the fixer, to be precise).

Each task stores its destination, which is always a facility, the type of which depends on the task, e.g. buy-item always targets a shop. Furthermore, all tasks except charge specify type and amount of a requested item. While the action 'assemble' only creates a single item at a time, the task craft-item allows for an arbitrary amount of items to be created. The item parameter of a craft-assist specifies a lower bound on the agent's contribution to the assembly, i.e. the agent has to contribute at least the item (or tool) specified. This models the common case of a single item, helping our heuristics to map out approximate dependencies and detect failures early. When the assist-assembly task is executed, the agent may contribute all items in its inventory, regardless of what is specified in the task.

After arriving at the destination, an action specific to the task is performed. For most tasks (Buy-Item, Deliver-Item, Retrieve-Item and Charge) this action is simply repeated until the requested amount of items has been processed. Assemblies are a bit more complicated, they are described in the next section.

If an agent has no task, or is currently waiting on some other operation to complete, it usually spend its action recharging. Sometimes the mothership decides to place a bid on an auction job, which is then placed by an idle agent. Section 3.6 describes this in detail.

3.2.1 Assemblies

As they involve the synchronisation of multiple agents, extra care is taken to efficiently execute assemblies. For each assembly there is exactly one craft-item task, and any number of craft-assist tasks.

The craft-item task stores an assembly id, which uniquely references the assembly.

Table 2: Types of errors addressed by the fixer and their respective resolution strategy.

Task type	Error description	Resolution strategy
any	out of battery	add charge task
craft-item, craft-assist	assembly failed, missing item	add item to agent, or dispatch a craft-assist
craft-assist	useless assist	remove task
craft-assist	invalid assembly id	remove task
deliver-item	delivery failed, missing item	add item to agent, or move delivery
deliver-item	useless delivery	remove task
deliver-item	invalid job id	remove task
buy/retrieve/craft-item	maximum load reached	reduce the item's amount
buy-item	not in stock	reduce the item's amount

Corresponding craft-assist tasks then use this id to point to the specific assembly they apply to, in order to not confuse different assemblies by the same agent or at the same workshop.

Our agents spend much of their time waiting for all participants of an assembly to arrive. This makes it necessary to optimise the execution of the craft-item and craft-assist tasks, so that this time is used as efficiently as possible.

First, the mothership needs to determine whether it is possible to complete the assembly in the current step, i.e. all agents carrying the required items and tools have arrived at the workshop. If that is the case, the agents assemble the item.

Else, the mothership considers whether an agent can give their items to another agent, and leave before the assembly is completed. There are a number of factors to take into account (e.g. carrying capacity, tools), but this optimisation is often viable (on average about once per two assemblies).

Once that possibility is exhausted as well, the agents at the target workshop are idle, i.e. they recharge or place a bid for an auction job.

When the internal simulation executes the task, we additionally look for possible future agents joining the assembly, and whether they are able to bring the necessary items. If that is not the case, the assembly is impossible to complete and an error is emitted. This serves to detect failed assemblies as soon as possible.

3.3 The fixer

Central to our strategy is the concept of discovering issues using our internal simulation, and then iteratively improving on our strategy to arrive at a working state. This is realised by the fixer, which is both the most complex and most important component of our system.

The fixer uses randomised heuristics that assign possible choices different weights

based on a number of factors. For each viable choice there is a non-zero probability, so that running the fixer indefinitely will eventually discover a solution, if it exists. More importantly, while the heuristics are biased so that most of the time an acceptable outcome is reached, multiple runs of the fixer explore different solutions.

Table 2 lists the possible errors that may arise in a simulation. For most of them, the resolution is trivial. If a task is useless (for example delivering an item to a job twice) or malformed (like a craft-assist referencing an assembly that does not exist), the offending task is simply deleted. These tasks are not results of malfunctioning code, but can arise when other tasks are modified.

When a task cannot be executed due to limits in an agent's storage capacity or a shop's stock, the amount of items is simply reduced to the maximum feasible number, or the task is deleted if not even a single item can be transferred.

Some complications arise when an agent runs out of charge. This situation is generally solved by adding a charge task just before the failing task, minimising the total distance travelled. In rare cases this does not work, as the agent is unable to even reach the nearest charging station, and the charge task is moved back one slot further. If there is no further task in front, the agent is stuck and must call breakdown service.

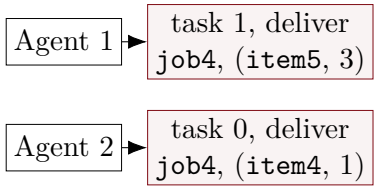
The most complex heuristics are used to deal with missing items, with the following possibilities being considered:

- An agent already holds the item in its inventory and it is not being used by other tasks. This is the preferred method to obtain items.
- The item was previously delivered to a job which can no longer be completed, so that the item can be retrieved from storage. Our heuristics are also biased towards this choice, though not as much.
- Another task to buy this item is already underway, and there is capacity to increase its item amount. This is considered an average choice.
- The item is available for purchase in a shop. There is a slight bias against this method.
- Finally, assembling the item is considered, although this choice has the lowest bias.

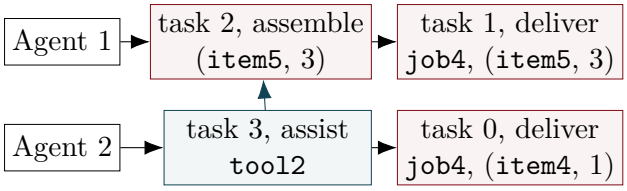
Depending on the type of task that is missing the item (craft-item or deliver-item), we then need to issue a craft-assist to the new agent (if it was not already involved), or a deliver-item task. In the latter case, the original delivery can be deleted.

There are other factors modifying the chances of assigning a task to an agent:

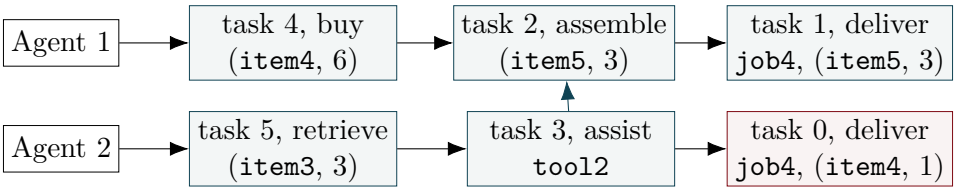
- the agent is idle for an extended period of time at the end of the internal simulation (which only simulates 80 steps into the future)
- there are free slots left for the new task(s) (if there are none, the agent is not considered)
- the agent is able to carry all items for the task



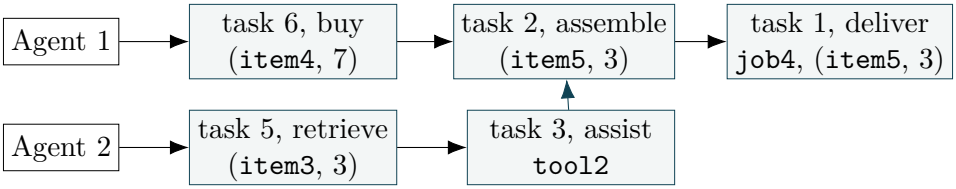
(a) The initial strategy, possibly generated by our create work subroutine. As the agents do not have the required items, both tasks fail.



(b) The fixer first adds task 2 to assemble the required items. As `tool2` is needed, agent 2 is assigned to assist the assembly. They already have this tool in their inventory, and are able to use it.



(c) There are still items needed for task 2, so the fixer adds the tasks to obtain them. Task 1 now works successfully. Note that the assist in task 3 is not limited to the item originally specified, but rather contributes all useful items in the agent's inventory.



(d) Task 0 does not work, as agent 2 is not in possession of `item4`. The fixer notices that agent 1 is already buying this item, and is involved in the delivery. So it increases the amount in task 4 (assigning a new id in the process) and deletes task 0. Similar to assists, deliveries contribute all items in an agent's inventory, so task 1 will also deliver `item4`.

Figure 3: Example showing a possible execution of the fixer.

- the agent is already participating in the assembly

Then, an appropriate facility is chosen, minimising a combination of total distance travelled and price (the latter only for shops). Figure 3 shows an example execution of the fixer, illustrating the above possibilities.

Usually, there are multiple possible slots for the task(s) to be inserted. Always choosing the last possible slot does not work, as one of these tasks may be a prerequisite for a task scheduled earlier on. A converse problem may occur when inserting them in front, possibly causing problems with tasks that were working fine before, some of which may generate items that are needed for a task at hand. So task has to inserted somewhere 'in the middle'.

This situation is further complicated by the fact dependency chains between tasks are

notoriously hard to analyse correctly, as much depends on the state of the simulation at the time. For example, a craft-assist may contribute any item in the agent's inventory to the assembly, if they are needed. That, however, depends on the inventories of the other agents involved in the assembly.

In the end, we ended up with a simple heuristic: insert the task after the last task that completed successfully in the simulation. This ensures that working tasks are not impeded, and any tasks dependent on the inserted one are executed afterwards.

This rule works well together with another property of the fixer, namely that errors in tasks with higher id (task ids are always increasing) are considered first. Essentially, the fixer performs a depth-first traversal of the dependency tree of the tasks, and inserts tasks according to that order. Whenever a task is working, i.e. completed successfully in the last simulation, it will be 'committed' in a sense, in that no tasks are inserted before it (or any of its dependencies, as they must also have been completed successfully).

Besides being a simple rule that resolves dependencies without having to construct them explicitly, this also avoids deadlocks, at least most of the time. The depth-first traversal always causes any dependencies to be placed in front of their tasks, which avoids cyclic dependencies. However, it is possible for a task to fail in more complicated ways (a shop running out of stock, for example), some of them outside of our agents' control. Thus there is the possibility of a working strategy breaking with a step of the actual simulation, or due to various indirect interactions of tasks.

Deadlocks have been observed in practice, albeit very rarely. As a result, our fixer detects cycles in the dependency chain, and moves one of the tasks to resolve the deadlock, updating their task ids in the process.

Sometimes the fixer is not able to find a solution for an error, for example when most agents are occupied and a complex assembly cannot be completed. Therefore, after trying a certain number of times, the fixer gives up on a task and deletes it instead.

3.4 Create work

We only consider jobs that are currently available to generate work, so no work is done speculatively in preparation for future jobs. We tried to add a subroutine that buys tools in the beginning of the simulation, when our agents would otherwise be idle, but it did not improve our performance.

Our heuristic for evaluating a job's profitability is as follows:

$$\alpha + \frac{r + f - \sum c_i(\beta_1 n_{have}(i) + \beta_2 n_{need}(i))}{m}$$

where r is the reward of the job, f the fine, c_i the estimated base cost of item i, $n_{have}(i)$ the amount of item i that is currently stored in the inventories and can be used for the job, $n_{need}(i)$ the difference between $n_{have}(i)$ and the amount needed to complete the job, m the number of assemblies required to complete the job, α is a positive value iff the job has been partially completed, and β_1, β_2 are factors chosen empirically.

As an aside, we estimate the base cost of an item by averaging the prices of all shops. This generally yields values very close to MASSim's internal base value. Furthermore, we

are interested not in the true value of an item, but the price we have to pay to acquire it, for which the actual prices in shops are a better estimation.

We choose a job probabilistically, biased towards jobs with higher profit. If a job is not estimated to be profitable, it is not considered.

Auctions and Missions assigned to our team are considered in exactly the same manner as other jobs, with the virtual reward being equal to the sum of bid and fine. The latter is usually high enough for the mothership to pursue these jobs. (Note that bidding on auctions is completely independent of this process, here only auctions where we have already won the bid are relevant. See section 3.6 for a description of the bidding process.)

In order to create tasks after a job is chosen, we add deliver-item tasks to a random, available agent. All the other logistics are handled by the fixer.

3.5 Rate

The previous operations yield randomised results, which naturally leads to the possibility of a search over strategies by applying them repeatedly. This search is guided by the rate operation, which assesses the value of a certain situation (recall that a situation contains both a strategy and the state of the world at a certain point in time).

We take four things into account:

- The score of our team.
- All items currently in inventories of our agents. This value becomes less important as the game nears completion, and is irrelevant at the end.
- The amount of time the agents are idle after completing their tasks.
- Whether an error occurred during the execution of a task.

This heuristic is run on the situation obtained by internally simulating the execution of our strategy, allowing it to be rather simple. This means that in step 920 we would simulate the strategy until the end of the game (step 1000), at which point items no longer hold any value.

3.6 Auctions

Auctions are a major feature we ignored last year, for which we decided to add at least a simple solution, that does not lose us money but potentially improves our chances.

We rate auctions based on the same heuristic we use for choosing jobs (see section 3.4), which (mostly) ensures that an auction is profitable if we win the bid. There is a minimum profitability we target. For any auction better than that, we place a random bid such that we still meet the minimum.

As many teams in the last contest (including ourselves) chose not to implement betting on auctions at all, we try to detect whether the other team has placed a bid at some point in the game. If that is not the case, we always bid the maximum allowed amount.

After determining to place a bid on an auction, this bid is placed by an otherwise idle agent, of which there usually is no shortage. If we win the auction, create work will consider it when creating new tasks.

Note that this part of our strategy did not work correctly during the contest, so that we always bid the maximum amount.

4 Implementation details

This section describes how our strategy was actually implemented and some techniques we used for debugging.

Most of our time is spent executing the internal simulation, so we had to make sure that it is reasonably efficient. To this end, the simulation does not proceed step-by-step. Instead, it is event-based, with the events always being some agent waking up and processing its task.

Whenever a task is considered, we execute as much as possible at once and schedule an event to continue after the appropriate duration has elapsed. For example, if an agent is assigned a buy-item task, the simulation would first calculate the duration of travelling to the shop, immediately move the agent and update its charge, and then let it sleep for that amount of time.

The simulation jumps forward in time until the next agent wakes up (i.e. the next event), and only that agent is then processed. In other words, our internal simulation operates one level of abstraction higher than MASSim, as it executes tasks rather than actions.

This implementation gets a bit more complicated when assemblies are involved. The execution of the assembly is handled entirely by the assembling agent. They are woken up when an assisting agent arrives at the workshop.

Our data structures, which are described in more detail below, are inefficient when making structural changes (such as adding an item to a list). When running the internal simulation, these changes are mostly confined to adding items to an agent's inventory. Removing items can easily be done by replacing them with an invalid item id. The other places where items may be added are shops (due to restock) and jobs, both of which have a small list of possible items for which a slot is reserved. This is not possible for agents, as they may carry arbitrary items. In practice, however, they rarely exceed 7 items at once, which is why we make sure that agents have some buffer in the form of invalid items in their inventory. When adding items, our routine uses this buffer if possible. Should the limit of 7 items be exceeded, these changes are accumulated during an iteration and then applied at once.

Pathfinding is heavily used in the internal simulation, as most tasks involve travelling. We are able to estimate the time of travel exactly (not considering random failures) almost always, only rarely do our results differ from MASSim by a few steps. There is much redundancy in the paths in multiple runs of the internal simulation, so we keep a cache of distances between agents and facilities (which are the only valid targets).

To debug our program, we mostly relied on liberal use of assertions and print statements.

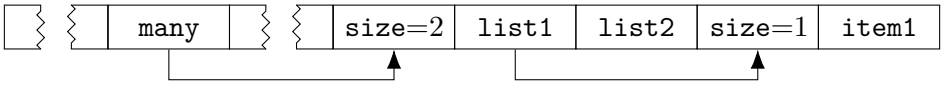


Figure 4: Example illustrating the layout of our data structures in memory. `many` is an array of arrays of items, while `list1` and `list2` are arrays of items. The arrows indicate the offset of an array's data.

Our containers do bound-checking in debugging builds, which catches most cases that would otherwise result in invalid memory accesses or corruption. Finding out that a certain check has failed is not very useful on its own, so we implemented the printing of stacktraces using operating system facilities. Once in a while the stacktrace does not contain useful information, in which case we use a debugger to investigate further.

Printing nested data structures is not easy in C++, as there is a lack of good facilities for introspection of data types. Still, we implemented subroutines for formatted, human-readable output of our data structures. As these procedures already have the structural information, it was quite simple to also implement a structural diff. This helped in investigating differences between our internal simulation and MASSim.

While our heuristics are randomised, our pseuodo-random number generator is always initialised with the same seed, so that subsequent runs produce the same results. As MASSim is deterministic as well, there was no problem in reproducing errors during testing.

For our data structures we only use simple unsorted arrays, most operations involving a linear search. Some of these arrays are nested. We store them contiguously in memory, using offsets to point to the start of an array (see Figure 4 for an example). Increasing the size of a list would require moving all subsequent bytes and adjusting some offsets. This is an expensive operation, but it is rarely needed.

5 Evaluation

In this section we will evaluate our performance by analysing two matches against teams Flisvos and Jason-DTU, respectively, who share second place in the tournament. We feel that these two simulations are the most interesting and highlight the characteristics of our strategy well.

5.1 Exploiting auction jobs against Flisvos

Figure 5 shows the score during the game against team Flisvos. Looking at the graph, one may assume that both teams were evenly matched and we managed to win by a slim margin in the end. However, a more detailed analysis reveals that the two teams had different sources of income and the final score was mostly a result of favourable randomness in the simulation's generation.

Flisvos completed 54 jobs during the simulation, consisting of 51 normal jobs and 3 missions, as shown in table 3. We finished only 34 jobs, with a split of 14 normal

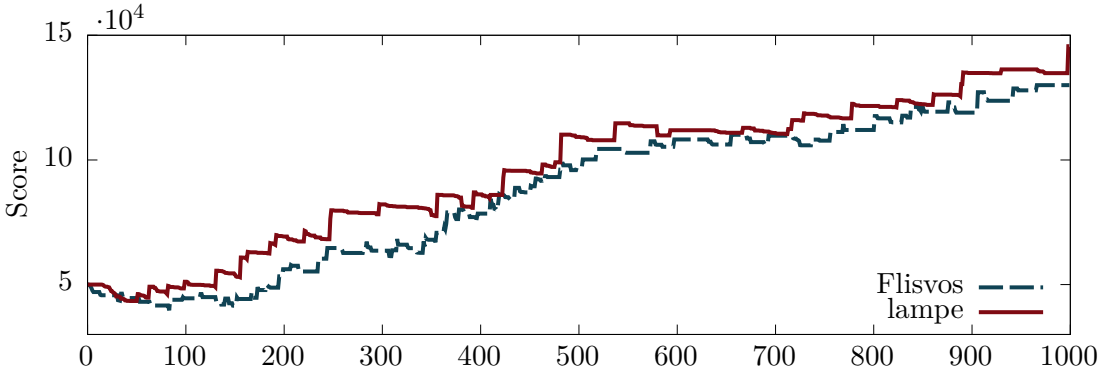


Figure 5: Score during simulation 2 against Flisvos, with time in steps. lampe ends with a score of 145621, Flisvos with 129985.

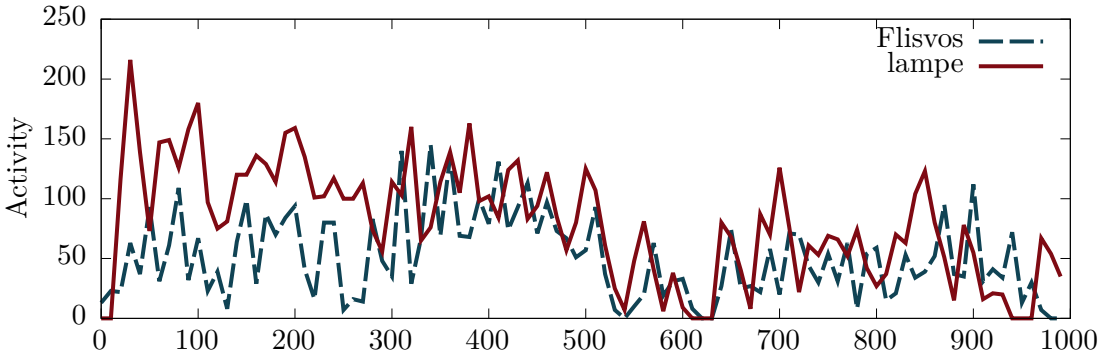


Figure 6: Activity during simulation 2 against Flisvos, with time in steps. For each 10-step interval, the number of successful actions is shown, not counting 'skip', 'abort' and 'recharge'.

jobs, 18 auctions and 2 missions. Indeed, not a single bid was placed by team Flisvos, leading to the assumption that they did not implement the handling of auction jobs. Our agents took full advantage, and placed bids on 22 auction jobs uncontested. As we did not detect any bids from our opponent, each bid was placed for the maximum amount. Consequently, the auctions we completed had an average reward of 6148, which almost doubles the average amount Flisvos received, which is 3390.

Even with that advantage, the game was extremely close and we won by just a few points, indicating superior efficiency of Flisvos with regards to normal jobs. To investigate this, we try to determine whether a job has been attempted by a team. We consider both (possibly partial) deliveries and moving an agent with all required items to the storage as attempts on a normal job. (For auctions and missions only deliveries count.) The former is well suited to our agents, as they often use partial deliveries to complete a job, while the latter is necessary for Flisvos, because they only deliver the full amount at once.

This heuristic is far from perfect, as two jobs may happen to require similar items,

Table 3: Various statistics for simulation 2 against Flisvos. A job is considered attempted if a partial delivery occurs or an agent with appropriate items goes to the storage (only for normal jobs). This yields some false positives. Contented refers to jobs attempted by both teams, excluding auctions and missions. Completion time is counted relative to the job's beginning. For completion time and the rewards, the mean value is shown.

Description	Team	All	Contended	Normal	Auctions	Missions
Number of completions	Flisvos	54	27	51	-	3
	lampe	34	6	14	18	2
Number of attempts	Flisvos	60	33	57	-	3
	lampe	65	33	41	21	3
Completion time	Flisvos	34	34	34	-	38
	lampe	44	37	35	48	72
Reward (completed)	Flisvos	3390	3361	3395	-	3304
	lampe	4628	3106	2774	6148	3925
Reward (attempted)	Flisvos	3362	3315	3365	-	3304
	lampe	4115	3315	3161	6093	3304

such that items intended for the completion of one are also counted as an attempt for another. However, considering that Flisvos completed all but 6 of their attempted jobs, this error seems to be small.

If both teams attempt a job, we call it contended. In total there were 33 contended jobs, 27 of which were completed by Flisvos compared to 6 by lampe. This points to what we believe is the key point concerning Flisvos' advantage in normal jobs: Our team is not able to complete jobs as quickly as other teams, and thus looses out on the most profitable jobs.

This effect can be seen in multiple statistics. On average, our team takes 46 steps to complete non-contended jobs, but only 37 for contended ones. Flisvos takes 34-35 steps for both, indicating that we managed to complete some contended jobs by being quicker than Flisvos. As the average reward for those jobs was slightly lower than that of an average contended job, team Flisvos maybe did not place a high priority on them. Indeed, we see that for completion of normal jobs, our average reward is 2774, which is decidedly lower than Flisvos' 3395.

In total, Flisvos completed 183060 points worth of jobs, which is 16% *more* than our combined rewards of 157352. In total, Flisvos spent 41344 points more during the simulation. We conclude that our team operated at a higher efficiency in general, which may have led to our decreased ability in quickly completing jobs.

Another interesting thing to note here is that our team was able to prioritise between started jobs, as for both auctions and missions the average reward of completed jobs was higher than that of attempted ones, with both being uncontested by the other team. This indicates that our mothership decided to forgo the pursuit of some of these jobs,

seeing better opportunities elsewhere.

Figure 6 shows the game from a different angle. We measure the level of activity of a team by counting the number of successful actions for each 10-step interval, excluding actions 'skip', 'abort' and 'recharge'. The maximum number of actions is 280, as each of the 28 agents can take one action for all 10 steps.

In last year's edition, our agents showed a distinct pattern of activity characterised by a burst of activity at the beginning of a new job, followed by a long period of inactivity waiting for a small number of agents to complete the job. We noted this as a major disadvantage, because most of our agents possible actions are spent idling. Hence we are happy to report that our new strategy does not suffer from similar problems. In fact, the general level of activity is quite similar for both teams. In total we performed 7954 actions, 57% more than Flisvos at 5075.

Right at the beginning of the game, our agents are waiting for the first jobs to appear. While our implementation supports buying items speculatively, our agents performed worse during testing when this was enabled. This is a possible avenue of improvement for the future.

Then, until step 280 we are considerably more active than team Flisvos. We believe that during this initial phase, with no stockpiles of items having been acquired yet, our optimisation procedure is able to utilise all agents to a high degree, coordinating the multiple assemblies that have to happen more efficiently than our opponents. Indeed, half of the contended jobs we completed were done in the first 100 steps (at steps 63, 82 and 99).

Between steps 280 and 960 both teams show a very similar patter of activity. Some of our actions are due to auction jobs, which are ignored by team Flisvos, but especially during steps 520-670 they are almost identical. We think that this is caused by there being only a small number of highly profitable jobs, which are pursued by both teams at the same time. As many of the items are left over from previous jobs, which were not completed, our level of activity decreases, compared to the beginning.

After step 960, activity of team Flisvos vanishes, maybe to avoid starting a job that cannot be completed in time. Our strategy makes no pre-emptive considerations in this direction, if a job completes in time in our internal simulation, it is carried out regardless. Thus our agents manage to complete an auction job at step 998 with a reward of 10805, which was very significant in this game. (We won by a margin of only 15636 points.)

In conclusion, Flisvos played very well, beating us soundly in completing normal jobs for most of the game with the exception of the first 100 steps, which were quite balanced. We were lucky to exploit highly profitable auction jobs in this simulation and thus managed to win. The map generation in the other simulations did not exhibit this many highly profitable auctions, and Flisvos won both with a considerable margin. Still, our team was able to operate efficiently, even when only able to complete about half of the jobs we attempted.

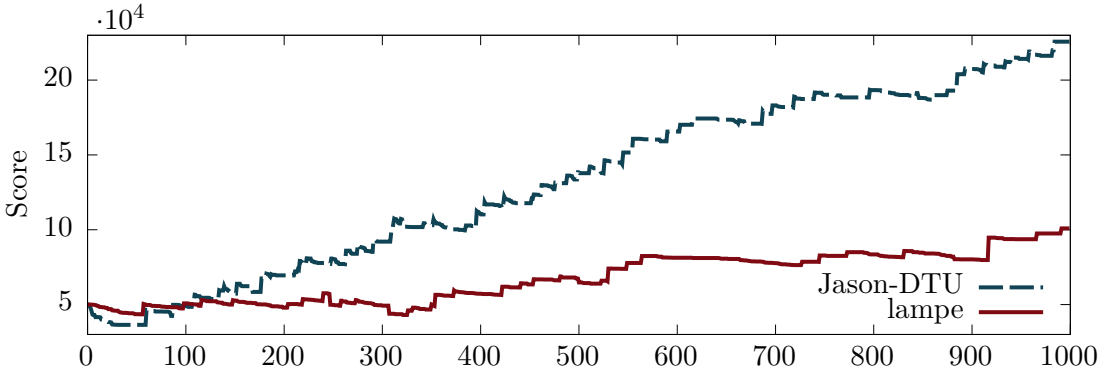


Figure 7: Score during simulation 3 against Jason-DTU, with time in steps. lampe ends with a score of 100880, Jason-DTU with 225755.

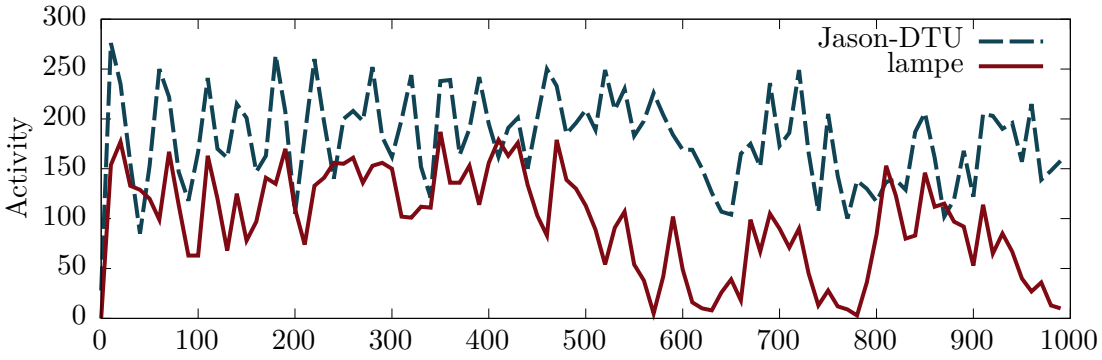


Figure 8: Activity during simulation 3 against Jason-DTU, with time in steps. For each 10-step interval, the number of successful actions is shown, not counting 'skip', 'abort' and 'recharge'.

5.2 Technical difficulties against Jason-DTU

The score in simulation 3 against team Jason-DTU is shown in Figure 7. This game was not close, with Jason-DTU steadily increasing their lead throughout. We see that our team was not able to consistently complete jobs, for example there is a large period of inactivity between step 564 and 744.

Looking at the activity graph in figure 8 we note that once again both teams have similar periods of activity. However, Jason-DTU is consistently more active, they complete 18045 actions in total, almost doubling ours at 9560.

Unlike team Flisvos, Jason-DTU placed bids on auctions, with a total of 23 bids placed in this simulation (compared to our 21). This should cause our mothership to switch to the slightly more intelligent logic, where each bid is random with the restriction that the job still has to be profitable. This would line up well against our opponent's strategy, which always bids the maximum reward minus 2. Sadly, our detection logic was not

Table 4: Statistics for simulation 3 against Jason-DTU. See table 3 for a description.

Description	Team	All	Cont.	Normal	Auctions	Missions
Number of completions	Jason-DTU	62	30	39	18	5
	lampe	30	9	22	3	5
Number of attempts	Jason-DTU	71	39	48	18	5
	lampe	61	39	52	4	5
Completion time	Jason-DTU	53	53	56	48	44
	lampe	48	47	42	70	58
Reward (completed)	Jason-DTU	5175	5139	5097	5940	3032
	lampe	4132	4359	3768	8637	3032
Reward (attempted)	Jason-DTU	5071	4959	4958	5940	3032
	lampe	4746	4959	4559	9320	3032

working correctly, and we only bid according to our default behaviour, which always chooses the maximum amount.

Hence any auction which had both teams bidding was always won by Jason-DTU. As shown in table 4, we only attempted 4 auctions compared to Jason-DTU's 18. Interestingly, we won 6 bids and Jason-DTU won 21, so both teams chose not to pursue some auctions despite the fines this would incur.

For some auctions we were the only bidder. These had a comparatively high reward, so we speculate that Jason-DTU might have considered them impossible to complete in time. Our agents completed 3 out of 4 attempted auctions. The one left unfinished also had a high reward (11369), indicating that it was dropped due to infeasibility.

Normal jobs were also mostly won by Jason-DTU, they completed 30 of the 39 contented jobs first. We see a similar pattern as in the previous game, in that our completion time for auctions and missions is much higher than for contended jobs. Additionally, the rewards of jobs we completed were much lower on average.

Jason-DTU completed their jobs mostly using a single delivery (24 partial deliveries to our 114), which is also similar to Flisvos. We believe that to be an advantage when considering completion times.

5.3 Problems with our strategy

As seen in the two games analysed, we were less suited to compete with other teams for jobs. This, we believe, is mostly due to our team not completing jobs fast enough. Our heuristics do not consider the other team when assessing the completion of a job, which leads to inaccurate predictions of our internal simulation.

For example, if an opponent has an agent with the required items on its way to the storage, our simulation would not predict it completing the job. So our own strategy may very well be optimised under the assumption that completing the job 50 steps from now would yield the same reward.

The subsystem for detecting bids of opposing agents did not work correctly in the contest, probably due to a simple bug in our implementation. As a result, we were unable to test our betting strategy against other teams and lost out on many lucrative auction jobs during the games.

Our agents employ a search over possible strategies, with the best one being chosen. This means that errors in our lower-level routines sometimes show themselves only through indirect means, as the affected strategies are then not chosen. For example, we noticed some jobs being abandoned despite having a large reward (or fine), and being partially completed. We thus suspect bugs in our subroutines, especially the fixer.

Some features of the simulation we did not take into account, most notably resource nodes. Implementing resource nodes would not be trivial, as we would then have to reason about the value of exploring the map. However, it poses no fundamental problem with our strategy.

5.4 Design and implementation choices

We make use of a centralised architecture in our strategy, with the mothership collecting all perceptions and making all major decisions. Similar to last year, this fits our strategy very well. We are focused on optimising a combined allocation of tasks for all agents. Making all decisions centrally allows us to take interactions between different agents into account without additional difficulties.

Choosing C++ as programming language comes with several advantages. We are able to choose the representation of our data structures to suit our needs. For example, the game-related data is stored contiguously in memory, making copying trivial (a single `memcpy` suffices), but structural modifications (such as adding an item to a list) are more complex. As we run multiple internal simulations, resetting the state of the game each time, copying of the whole game state is a common operation.

As we run a randomised search of possible strategies, the feasibility of our approach is dependent upon our application's performance. C++ is well known to have the potential for high performance, and it offers fine-grained control over data structures and other low-level implementation details. We believe that implementing our solution in a managed language such as Java or Python would have incurred a significant amount of time spent on further optimisations, resulting in more complex code. An implementation in agent-programming languages such as Jason or Pyson may not be feasible at all.

For our solution we did not spend much time on optimisation, as we are able to search about a 100 different strategies per step (depending on circumstances) and improving the underlying heuristics proved more valuable.

Then there is our existing framework, which we could adapt easily to accommodate the changes for this year's contest. Hence we were able to focus our efforts on implementing our strategy, and did not need to work on communication with MASSim and the operating system.

Other benefits we considered are static typechecking, manual memory management and mature toolchains for developing C++.

6 Conclusion

Our participation in the 2017 MAPC was an opportunity to try out a completely different approach, and a valuable learning experience regarding the implementation of a complex strategy with multiple layers of abstractions.

We are quite happy with our results, considering the small amount of time in which our strategy was developed. While only ranking fifth out of seven is not great, we were able to provide some interesting games. Even when playing against stronger opponents our strategy was able to keep functioning, most of time still turning a profit.

Analysing the games revealed multiple weak points of our team, most importantly the slow completion of jobs. We also suspect some flaws in our internal heuristics used by the fixer.

7 Team overview: short answers

7.1 Participants and their background

What was your motivation to participate in the contest?
We wanted to try out a more complex strategy based on internal simulation of the game state, using the framework we built in last year's participation to handle communication with MASSim and interaction with the operating system.

What is the history of your group? (course project, thesis, ...)
Our framework was originally developed in a course project in 2016, as was a basic strategy. We then improved on that project and participated in the 2016 edition of the MAPC. This year, our participation makes use of our framework, but the strategy has been completely rewritten.

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

7.2 The cold hard facts

How much time did you invest in the contest (for programming, organizing your group, other)?
There were about 200 hours invested, almost entirely into programming.

How many lines of code did you produce for your final agent team?
In total there were 8512 lines of C++ code (all counts excluding comments and blanks), of which 2396 are implementing the strategy and internal simulation (see section 2.2 for more details).

How many people were involved?
Two people were involved.

When did you start working on your agents?

The original framework was developed in starting in April 2016. For this year, preliminary work begun in March 2017, although the strategy was being worked on since late August.

7.3 Strategies and details

What is the main strategy of your agent team?

Our agents are controlled by a central entity. Strategies are evaluated based on an internal simulation of the game state, and iteratively improved upon using randomised, hand-written heuristics. The tree of strategies is explored and the most promising strategy chosen.

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

As the team is centralised and all agent are controlled by a single entity, there is no need to share information or communicate. The heuristics generating the strategy ensure that actions involving multiple agents are coordinated appropriately.

What are critical components of your team?

The most important component is the internal simulation, which mirrors MASSim. This allows the agents to reason about future events and consider the outcomes of various strategies. An important part is our pathfinding, to precisely estimate the distances between locations. Most of the work generating strategies is done by the fixer, which simulates the execution of a strategy and addresses any issues that crop up.

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

There are no explicit state changes in the behaviour of our agents, except for the betting strategy on auction jobs (described in section 3.6).

Did you have to make changes to the team during the contest?

We fixed a few minor bugs on the first day, but made no changes apart from that.

How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?

The agents are organised in a centralised fashion, with a single entity controlling all agents.

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

The strategy is planned centrally and assigns tasks to all agents, so all behaviour emerges on a team level.

If your agents perform some planning, how many steps do they plan ahead?
The agents plan up to 4 tasks ahead. This may involve at most 80 simulation steps, with a value of 30 being a typical duration of an error-free strategy.

If you have a perceive-think-act cycle, how is it synchronized with the server?
The synchronisation is based on the blocking behaviour of the TCP socket we use for communication with MASSim. As our program is implemented sequentially, no further synchronisation is necessary.

7.4 Scenario specifics

How do your agents decide which jobs to complete?
The heuristic responsible for generating new tasks rates the profitability of jobs based on estimated item values. Jobs are selected probabilistically, with higher weight given to jobs with higher estimated profit.

Do you have different strategies for the different roles?
There are no explicit considerations of different roles.

Do your agents form ad-hoc teams for each job?
No, each individual assembly is considered separately.

What do your agents do when they do not pursue any job?
They recharge their batteries whenever they have no other actions to execute. Additionally, they may place a bet for an auction job, which are prepared by the centralised planning whenever there are profitable auction jobs.

How did you go about debugging your system?
We relied mostly on `printf`-style debugging combined with liberal use of assertions and printing of stack traces. In rare cases of memory corruption we employed a debugger. More details are given in section 4.

What were prominent questions you would have asked your system during development? (i.e. "why did you just do X?")
Mostly we investigated discrepancies between our internal simulation and MASSim, as well as problems with the fixer, such as not being able to converge on a working strategy, or making questionable decision while moving tasks around.

7.5 And the moral of it is ...

What did you learn from participating in the contest?
We learned a lot about developing robust heuristics based on an internal simulation of the future.

What are the strong and weak points of your team?

Our team is able to both plan ahead and alter its strategy on-the-fly based on incoming information, working around failures of individual tasks. However, it cannot compete with the handwritten and highly specialised strategy of team BusyBeaver and is less efficient overall than the strategies of teams Flisvos 2017, Jason-DTU, and SMART-JaCaMo, taking too long to complete jobs.

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

C++ remained viable as a programming language for the MAPC, especially as we had our framework from last year to build on. It allows us a large degree of freedom in designing our own specialised abstractions and data structures. Additionally, our strategy is viable only due to the performance of our application, which would be much more difficult to achieve in languages like Java, Python or any of the agent-oriented languages used by some of the other teams, if not outright impossible.

Did you encounter new problems during the contest?

We encountered various minor problems, as well as multiple issues with the internal simulation and other strategy, which impacted our performance quite severely.

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

Our team is more focused on completing jobs efficiently than being the first to complete a job.

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

If we had a week, we would implement additional features of the simulation, such as resource nodes and maybe parallelise our implementation, but mostly focus on fixing all the bugs we encountered. Additionally we would incorporate a simple approximation to take completion of jobs by our opponents into account. In a year, we could have a close look at our heuristics and improve them across the board, e.g. by optimising locality of strategies.

Which aspect of your team cost you the most time?

Implementing the fixer was the most time-consuming, as there are many different ways for tasks to go wrong and influence each other.

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

This year's team BusyBeaver exploited a combination of properties, enabling it to adopt a latency-based strategy focused on completing *all* profitable jobs before the other team. We would suggest reducing the speed of agents with regards to the map size significantly, to ensure that it is profitable for a team to spread out across

the map. This may require an increase in the number of facilities to balance out. Another possibility would be to raise the difficulty of assembling items, by raising their price, the required amounts, the number of item types or their volume.

Why did your team perform as it did? Why did the other teams perform better/worse than you did?
Our team was able to execute a competitive strategy, but our implementation was ill-suited to opponents focusing on fast completion of jobs. Hence our overall performance was quite bad, but we still managed to deliver a few close games.

References

[1] Ahlbrecht, T., Fiekas, N., Dix, J.: Multi-agent programming contest 2016. Int. J. Agent-Oriented Softw. Eng. (in press)

[2] Czerner, P., Pieper, J.: Multi-agent programming contest 2016: lampe team description. Int. J. Agent-Oriented Softw. Eng. (in press)