The Flisvos-2016 multi-agent system

Evangelos I. Sarmas

Email: eis@sarmas.com

Abstract: This paper presents the workings of the Flisvos-2016 multi-agent system that participated in the multi-agent programming contest MAPC 2016 of Clausthal TU.

Keywords: Flisvos; multi-agent system; multi-agent programming; contest; MAPC.

Reference to this paper should be made as follows: Sarmas, E.I. (2018) 'The Flisvos-2016 multi-agent system', *Int. J. Agent-Oriented Software Engineering*, Vol. 6, No. 1, pp.35–57.

Biographical notes: Evangelos I. Sarmas holds an MSc in Computer Science from Rutgers University, an MBA and a BSc in Physics. He is a professional Computer Scientist and has been Oracle Database Beta tester. He has many interests, mainly in algorithms, artificial intelligence, bioinformatics, mathematics, programming techniques, programming languages and their implementation, databases, operating systems, computer architecture, and software engineering.

1 Introduction

The *Flisvos* multi-agent system will be described as it competed in the MAPC 2016 contest and the following friendly matches. The team did not have an objective to try some specific multi-agent development methodology. The goal was to enjoy participation and achieve a very good result relative to the effort spent. This presentation is for anyone interested to learn how a multi-agent system was designed and implemented in short time and with high success.

1.1 Implementation decisions

The *Flisvos* multi-agent system (the *system*) was written in Python in order to ensure a robust and competitive implementation in short time by a single person working part-time on it. Key factors of robustness and effectiveness are proven stability of the language with almost no bugs, language brevity combined with expressiveness, language capability (supporting also object-oriented and functional programming), and the extensive libraries available. Also, in previous contests one team had used Python successfully. Before committing to Python, a multi-agent programming platform that is open source, stable, currently updated, with good and effective educational material, without many open issues, and with networking and xml processing capabilities at a minimum could not be found. Most agent oriented software gave the impression of being abandoned, not updated, and with no vibrant community. Two well known multi-agent development

platforms had major updates after June 2016 while contest coding was active. Furthermore, an examination of past contest literature and few papers available (Hess and Woller, 2013; Köster et al., 2012; Behrens et al., 2011; Pibil et al., 2011; Jensen and Buch, 2014), open source repositories, and stackoverflow questions showed such multi-agent systems presented a number of issues during development and needed effort to be used effectively. It was also disturbing that systems made with them often consisted of too many source code lines (from 2,000 to 8,000 lines), while some systems made with general programming languages needed less code for similar or better performance, and it was indicated that system performance depended mainly on implementation strategy, algorithms, and total effort spent rather than implementation language. The most important evaluation criterion for this team was the need to deliver a competitive and stable software product in short time and this does not imply anything about the potential or merits of such software, which I find interesting. Since the team participated for the first time and had no experience of usual multi-agent programming environments, it was decided to not experiment and go directly with a general programming language. This also allowed complete control of implementation by not being tied to some specific programming paradigm and the development of a platform for future contests.

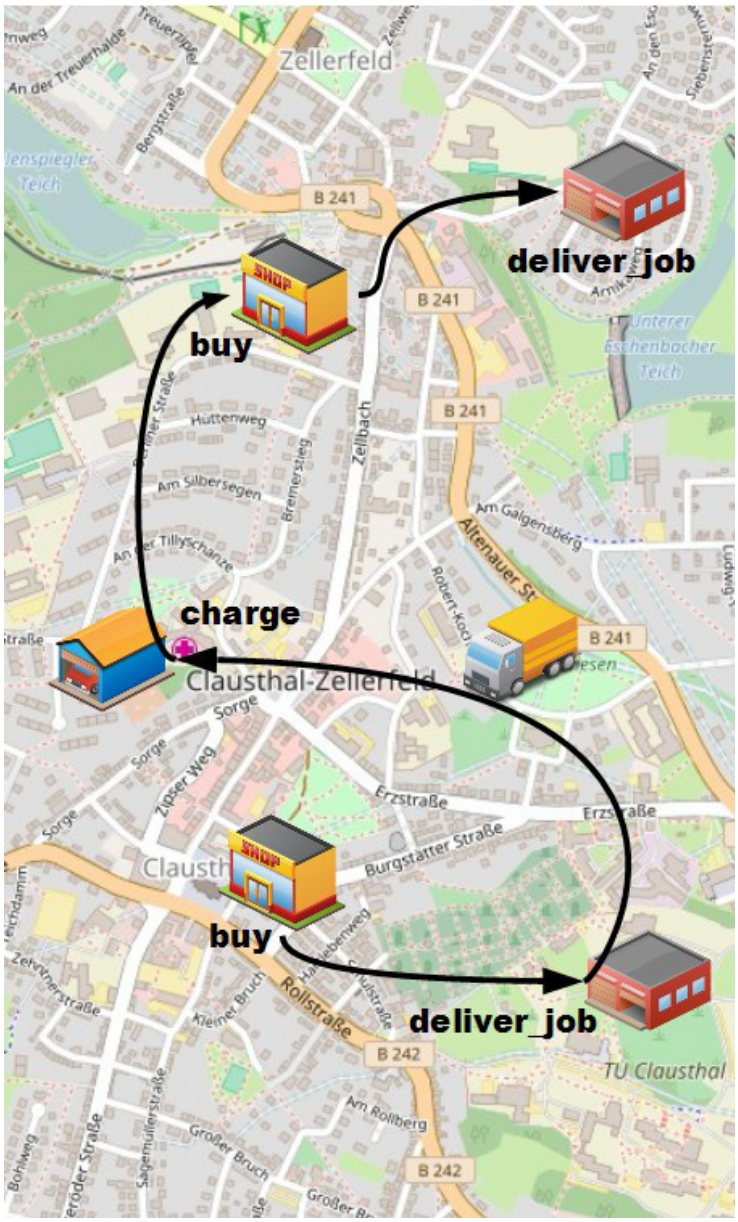
After the selection of Python, a few add-on packages were considered like pyDatalog (pyDatalog, 2016) for logic programming and Intellect (Intellect, 2012) a DSL and rules engine. They were not used because of the early discovery that the contest scenario could be handled easily and competitively with a subset of the available actions, especially excluding the novel *assembly* action. In case *assembly* was later deemed necessary, then a simple but capable enough hierarchical task network (HTN) planner (Nau, 2013) in Python (Pyhop, 2016) would be used.

1.2 Agent actions

Agents will *goto* from facility to facility and occasionally from random places to a facility if they were out of charge and needed service. Each agent, beyond the default *skip* when idle, will *goto* shops, *buy* items, and then *goto* a storage facility and *deliver_job* (partially or fully) for a job. On the move an agent may *goto* a charging station and *charge* and occasionally *call_breakdown_service*. A picture of an agent using the most common actions is shown in Figure 1.

The actions of *give*/*receive*, which imply a carefully designed rendezvous, were thought useless, as also the *store*/*retrieve* actions. The reason is that it is more efficient in terms of elapsed *steps* for each agent to deliver individually (with optional *charge* or *call_breakdown_service*) than do a *rendezvous/store* in connection with other agents. The *dump* operation was thought of little use as there is a small number of products in the city, and if a product becomes an unused load (because of failing to complete a job), it can be used in a future time against a new job. Same for the *assemble*/*assist_assemble* actions because in all jobs all items needed can be found in at least one shop and in about 50% of jobs the items can even be found in two or more shops. An *assembly* for an item means the loss of much time and money in gathering the required items/tools and doing the assembly process when it can be bought in less time and for less money from shops.

Figure 1 An agent doing two deliveries (see online version for colours)



If we examine the formula *(battery/10) * speed* for each agent type, which gives the maximum range (150 for car, 125 for drone, 140 for motorcycle, 200 for truck), we see it is practically the same and almost spans the diagonal of the simulation maps (which ranges from 144 to 173 steps; and it is also reasonable that the facilities will rarely be

located exactly at the edges of the map). For this reason and because in the planning algorithm we can select the best agent to do an action based on speed/range and capacity, it was decided that all agents be general workers and no specific role assigned to any agent. Occasionally, the system will stop any work for a specific job and the agents working on it will *abort* if the job is not available (either the end time has been reached or the other team finished the job first) or no delivery has been made and a new job has arrived with almost the same items and higher reward, in which case the new job replaces the old one.

The system works only with *priced jobs* reviewing them and selecting only a few to work on concurrently up to a configurable limit (usually 3). These jobs are called *committed* and the system makes all effort to complete them fast. For simplicity *auction jobs* and *bid_for_job* were left as a future addition. At random but almost regular intervals (semi-regular) one agent does *post_job* with lower reward compared to the cost of the job items in order to fool the opponent into working on it.

1.3 Paper structure

In the next sections, I will describe the details of the previous actions and the architecture as it played in the contest starting with an *overview* first, then the *parts/modules*, and finally the central *control/planning logic*. Also, some additional bug fixes and improvements will be described that were not possible to have available on the contest day but were completed one week later for the friendly matches (in fact, it is good luck that a working system was available on the contest day). These changes reflect what was expected to deliver, and though they do have a small impact individually, their sum makes for a much more effective system. The contest code was about 4,200 lines, and the final code was 4,500 lines (line count includes a lot of comments, blank lines and commented versions of code).

2 Overall design

There are 16 threads, each one representing one agent and running the same code. A separate module in each thread handles the network communication and the xml parsing of the incoming percept information. All communication is checked for errors or delays and for the correct format of messages at every step. In case of error, there is automatic attempt to reconnect and do all connection steps from start again (authorisation, etc.).

Generic functions parse parts of the incoming xml document tree and convert each one to a mapping object (called *dictionary* in Python) with nested mappings that mirror the xml fragment. In certain cases extra keys are used to index entries, for example ‘item_name’ is key for each product entry in the mapping of ‘products’. In order to ease significantly the code writing in accessing these mappings, a special enhancement was made to them to access the value of a key as an object attribute (the equivalent of an instance field in languages like Java). For example, instead of *simulation[‘step’]* one can simply write *simulation.step* and it is even possible to specify a long such chain, e.g., *agent.view.simulation.step* where *agent* is the agent thread’s object, *view* is an attribute which holds the recent action-request dictionary, *simulation* is the ‘simulation’ tag in the xml, and *step* is the ‘step’ attribute. All percept information is mapped automatically at

each step and is easily accessible in a unified manner during operation. There are two final mappings, the *sim* for the *sim-start* tree and the *view* for the *request-action* tree.

The agents need to communicate their percepts, and in order to simplify development a shared storage strategy was used. Even if messaging was implemented, the result would still be the same. There is a shared mapping (imaginatively named *shared*) with one entry for each agent which contains a copy of the *sim* and *view* mappings and the *job_op* (the queue of actions to execute – more details in 'the job queue' chapter). There is also a number of common variables that are useful in operation and are shared with the collective name *team* variables.

Each agent thread works in a cycle where it waits for a simulation percept/message, does some operations, and sends an action. Specifically

- on *sim-start*, updates its *sim* entry in the shared dictionary
- on *sim-end*, just prints the score and some statistics
- on *bye*, stops working
- on *request-action*, handles properly the result of the previous action, updates its *view* entry in the shared dictionary, and if it is not the last agent in the critical section that updates the *shared/team variables*, waits at a *barrier*, else executes the *control*/*planning logic*, and then wakes up all other agents waiting at the *barrier*; each agent after exiting the *barrier* reads from *shared* the new *job_op* and does what is appropriate to execute it.

The *control*/*planning logic* is the heart of each agent and generates the actions that the agent will do. It is really the same for all agents and contains no randomised element, so all agents can execute it independently with the same result. This means that each agent can generate its own action and also know what other agents will do. There is *implicit coordination* so that when two or more agents are candidates to execute the same action, then a *simple agent ranking principle* is used to decide which agent performs it (the rank is simply the unique agent id) without costly communication overhead and duplication of agent work. We found later that this idea has been explored previously (Hindriks and Dix, 2013).

In this implementation and for processing efficiency only (so the code runs 1× time instead of 16× times on a low-end personal CPU) the *control/planning logic* is run only by the last agent to enter the critical section and this agent then updates the *shared/team variables* and the *job_op* for all agents. This design is not centralised control really, and it can be easily modified to totally independent agents with a change of a few lines of code only and making a copy of the *shared/team* variables for each thread.

The development had a target of system response time below 1.5 sec and this was a factor in changing or simplifying some algorithms discussed later. Care was taken to handle the situation with slow connections or slow CPU that would cause agents to start at different step numbers (e.g., half agents would start at step 0 and half at step 1) which if not handled can possibly stay the same for the whole simulation and can cause *noAction* events (events caused by slow response to simulation requests). In such a case, the agents that are a step behind are allowed to proceed and the others are blocked until all continue at the same step number.

A considerable amount of the development effort (perhaps 50% but not unreasonable considering the two month time frame of this development) was spent on creating the

infrastructure for the communications, xml parsing, thread logic, and shared storage locking and to ensure it is robust and dependable. A big contribution to a good system performance comes from *stability and robustness in operation* (in fact there is no point in efficiency if there is no reliability).

For the same reason there are *assertions* checked almost everywhere and *defensive programming* is employed. Defensive programming in the sense of the system continuing operation despite getting unforeseen percept input or the system being in states that were not thought of during development. For example, any division is coded to divide by 1 if the divisor is 0, many functions that must return some state or result are coded to always return a closest state or result that is legal. There are also extensive *assertions* (either as *pre-conditions*, *post-conditions* or *invariants*), but they never really signal an exception halting the code. Instead, the error is logged and in some cases recovered, trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes. An example case of inaccurate operation where defensive programming was employed a lot will be described later in 'the jobs db' chapter.

Finally, quite a big amount of logging and statistics are recorded at each step for analysis and tuning of the system operation and certain information is summarised in a useful way and shown live during a contest. A very small indicative part is shown in Figure 2 (the log content will become more obvious later).

Figure 2 Log output

```
rewards = 183044, cost = 178142 (jobs b/c/s = 152489/9800/2500, explore c/s = 9353/4000)
balance = 4902, money = 54902 (seed = 50000)
explores = 60, per_step = 0.2
jobs QoS: system = 37 => selection = 36 97% => committed = 16 44% => completed = 12 75%
job_id11922894544191326 61, e: 465, r: 6848, needed: 1, 1, states: ON_BOARD,3
job_id5810586341057386157, e: 405, r: 11040, needed: 0, 0, states: ON_BOARD,4
job_id12491364915845501 89, e: 517, r: 5122, needed: 0, 0, states: ON_BOARD,3
```

('b/c/s' stand for 'buy/charge/service', 'e' for end_time, 'r' for reward)

3 The parts/modules

3.1 The job queue

Each agent has a queue of actions to do. This is called *job_op* (in singular for historic development reasons but is really a queue containing many actions). This queue is simply a custom list with a few specialised methods, such as getting a string representation of the whole queue content, appending to the queue, getting the current and the next action. There is also a special *drop* method used when a decision has been made to drop the current sequence of actions that itself calls a *on_action_drop* method on the current action and gets a new queue to replace the entire queue. It is assumed that the current action knows best what has to be done to stop itself gracefully and efficiently and this may need a whole new sequence of actions. The *job_op* may be empty (or null). A typical log output showing the action queues of all agents is shown in Figure 3.

Figure 3 Agent action queues

flisvos1 => [flisvos1 (shop3) buy: item10, 1 => job_id5962675194271705896, flisvos1 (shop3) goto => charging0 (56 steps NOCACHE) job_id5962675194271705896, flisvos1 (charging0) charge (7 steps) job_id5962675194271705896, flisvos1 (charging0) goto => storage0 (36 steps NOCACHE) job_id5962675194271705896, flisvos1 (storage0) deliver_job => job_id5962675194271705896]

flisvos2 => [flisvos2 (shop1) buy: item0, 1 => job_id5962675194271705896, flisvos2 (shop1) buy: item8, 1 => job_id5962675194271705896, flisvos2 (shop1) goto => charging0 (22 steps NOCACHE) job_id5962675194271705896, flisvos2 (charging0) charge (4 steps) job_id5962675194271705896, flisvos2 (charging0) goto => storage0 (36 steps NOCACHE) job_id5962675194271705896, flisvos2 (storage0) deliver_job => job_id5962675194271705896]

flisvos3 => [flisvos3 service (25 steps) job_id7259785893873253724, flisvos3 (none) goto => charging0 (6 steps NOCACHE) job_id7259785893873253724, flisvos3 (charging0) charge (7 steps) job_id7259785893873253724, flisvos3 (charging0) goto => storage0 (23 steps CACHE) job_id7259785893873253724, flisvos3 (storage0) deliver_job => job_id7259785893873253724]

...

flisvos15 => [flisvos15 (storage0) goto => charging0 (47 steps CACHE_R) , flisvos15 (charging0) charge (11 steps) , flisvos15 (charging0) goto => shop6 (47 steps CACHE_R)]

flisvos16 => [flisvos16 (storage1) goto => charging1 (37 steps NOCACHE) , flisvos16 (charging1) charge (6 steps) , flisvos16 (charging1) goto => shop2 (62 steps NOCACHE)]

a typical log output during a step showing the job_op(s) of all agents (the log has been edited slightly to remove logging information from the start of each log line that is not relevant)

All actions descend from a template action. There are methods to get a string representation of the action, send the action, reset the action state so that the action can be sent again (useful in cases of *failed_random*), and to indicate if the action has to be repeated on every step until completion. There are also methods to check if the action is still *active* or has *completed successfully* or has *failed*, to update *shared/team* variables when an action completes successfully or fails, and to return a new sequence of actions to execute if the current action has to be stopped before completion (*on_action_drop*).

The following action classes are defined for each action the system uses:

- *goto*: it updates information needed by the *steps cache system* (see 'the steps calculations and goto generation' chapter) and updates the *shop inventory* when the agent enters a shop. The goto is *failed* if the battery charge becomes 0 (and only if the agent has not arrived at the destination). The *on_action_drop* sequence is a single *abort* action.

  The *goto* object is the only one that is not meant to be instantiated and appended to an action queue directly. Instead, there is a special method that generates the *goto* object and appends it to an action queue. This method may also generate extra *goto* objects for actions to goto to/from *charging stations* if needed. More about this in 'the steps calculations and goto generation' chapter.

- *service/charge:* the *on_action_drop* sequence is a new *service/charge* action (the reason for this is that the system drops *jobs* not *actions*, which means that all action queues that relate to the job to be dropped have to be dropped, but an ongoing *service/charge action* should continue).
- *abort/buy/post_job*
- *deliver_job:* it checks the items to be delivered in *jobs db* (the database of job items – see 'the jobs db' chapter) in connection with the percept last action result and decides if this is a partial delivery or the job is completed (i.e., all items delivered) and if yes, it also drops all other system actions that are going on for this job. For reasons to be explained in 'the jobs db' chapter, a job may sometimes be completed earlier than expected by the system, so it is necessary to always check the last percept action result and also to drop any other active action queues associated with this job.

3.2 The steps calculations and goto generation

It is obvious that a competitive system must be able to estimate with some accuracy the time it takes to move from one place to another. Because I am not versed with the specific map format of the simulation and mapping software in general, an approximation method was used. All estimations are in units of simulation *steps*.

No special map information is needed except for the map boundaries (lat/lon of the map square) and the true map centre (calculated from the map boundaries).

Initially a direct estimate is made assuming plane geometry and using the Euclidean distance. The calculation is done in simulation cells (one cell is the distance measure in the simulation world) adjusted in size by the speed of the agent so the result is *steps* of simulation. If the agent is a drone, then the direct estimate is used as is. If the agent is any other vehicle, then the estimate is multiplied by a *route factor* (or *detour index*) which should be ~ 1.4 (Boscoe et al., 2012) but experience in this contest showed a best value of 1.67. This is best in the sense that a few times the true distance is more than the estimated and most times is about the same or less and was determined after a series of test simulations. Exact accuracy is not needed for this application, an error of 10%–20% is tolerable. The important thing is to make the right decision to use or not use an intermediate charging station when going to a destination.

As the simulation progresses, and for gotos between facilities only, the number of steps actually used is stored in a *cache*. The *cache* has a composite key containing the *agent type* (e.g., car, truck, etc., because the number of steps depends on agent speed) and the *directional move*, i.e., for a goto A → B there is one entry and another entry for a goto B → A (this is because tests showed that often the distance is different in the reverse direction; probably there are many roads connecting two points and some may be one directional only). The *cache* is first checked in the direction of the goto we are interested. If no entry is found, then it is checked in the reverse direction and finally if not found again, then the estimate described before is used.

The *cache* should be able to provide very accurate distances soon, independent of the map. However, it was not as helpful as expected because it was not filled completely, i.e., not all agents did all possible routes in the simulation time. This was recognised early, but unfortunately no workaround was devised. In retrospect, an easy solution would be for the system to use the cache entry available from any agent role, except the drone, and

adjust the cache value. For example, if there is an entry for a route from a car (speed 3) and we want to estimate the steps for a truck (speed 2), then the 3/2 of the value should be used. Sometimes simple solutions easily escape the mind.

There is a procedure that generates the goto actions needed in order to move from one facility to another and it is named *creat* (inspired by the well known Unix system call). This procedure may add extra *gotos* to and from an intermediate charging station.

There are two strategies in deciding how to use charging stations. One is obviously to use them only as needed in the middle of the way to a facility. The other, and the one chosen, is to use them pre-emptively, i.e., even if not needed to arrive at the destination, so that on arrival there is always enough left charge to go to a nearby charging station on the way to the next facility. This approach is safer in the sense that there is less danger to run out of charge. Both approaches have the same total cost in the long run. A middle approach is to use a direct path always if the destination is a storage facility where we deliver. Unfortunately, this was not explored adequately and we cannot know how valid it is, but it has merit over the long run argument because the delivery part is the most critical part in the job process and must finish as soon as possible (after delivery the agent is free usually and not committed to a job so it can call breakdown service; a job commitment starts often in a shop where items are bought).

creat uses two helper procedures.

1 One selects a suitable charging station next to a destination. It first tries to find the *nearest charging station* from all the charging stations in the rectangle bounded by the destination and the centre of the map. The logic of this heuristic is to find a charging station that is on the way to most of the next destinations after this destination. If this fails, then the nearest of all charging stations is used.

2 The other tries to find the *best charging station* from source to destination. Best is the one with the least total *steps* (including *charging time* steps; and in a tie the one nearest to start is chosen). In order to get a correct result, routes where there is enough charge to reach the *charging station* are examined first and then routes where there is not enough charge. For example, assume the battery has charge for 30 steps and *charging station1* is 20 steps from start and 50 steps from destination for a total of 70 steps while *charging station2* is 35 steps from start and 20 steps from destination for a total of 55 steps, then *charging station1* must be selected over *charging station2* even though it leads to more total *steps* because we cannot reach *charging station2* without calling breakdown service.

creat will always return a result even if the plan is not realistic and will not consider plans with more than one charging stations. The reason is both for reduced complexity and that in the majority of world maps one intermediate charging station is enough. If the plan is not feasible, the *steps* will reflect the true distance and can be used for comparison purposes with other plans. Also, if an unworkable plan is chosen it means there is no available workable plan with less *steps* and the comparison is still valid.

In summary, the steps calculation system is considered adequate. Exact accuracy is not needed much but rather comparative accuracy, i.e., the ability to say that an agent will get faster to one place than another agent. This is done well enough, except in the cases of the *cache effectiveness* (as described before) and the *delivery to storage* (when we should go straight to the storage facility for delivery without an intermediate charging station if possible), where both did cost in jobs finished first by the opponent.

3.3 The jobs db

The *jobs db* contains all necessary information for each job the system decides to work on. These jobs are also called *committed* because the system is committed to finish them as fast as possible. For each job it contains the *pieces* of items it has to deliver. A *piece* is simply a pair of *(item name, item amount)* that is the responsibility of a single specific *agent* to deliver. Each *piece* has a state: *ON_BOARD* if it is on board the agent (the piece may be already on board if it is a leftover of a previous job that was not completed in time or becomes on board after a buy action), *BUY* if it is about to be bought, *DELIVERED* if it has been delivered. The states for *ON_BOARD* and *BUY* imply that there is an active *job_op* with actions to deliver the *on_board* item or *buy* a new item, make it *on_board*, and then deliver it. Each *piece* amount is either the same amount as the job specifies for the item or smaller.

Each *jobs db* entry is a hierarchical object structure where at the top is the job data like *end step*, *reward*, and *storage* facility to deliver. This contains *items* with *name*, *amount*, and *delivered amount*. Each item contains *pieces* with *agent name*, *amount* (of piece), and *state*. For each job there is extra computed information which is used in the control/planning algorithms: the number of *distinct items* and the *total amount* that is *needed* to complete the job. A printout of the structure is shown in Figure 4.

Figure 4 Jobs db printout

```
=== jobs_db ====
id: job_id5514686329131958312, end = 814, reward = 18562, storage = storage3; needed = 0, 0
  item: name = item11, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 172 => agent = flisvos11, amount = 1, state = BUY
  item: name = item15, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 169 => agent = flisvos9, amount = 1, state = BUY
  item: name = item16, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 170 => agent = flisvos9, amount = 1, state = BUY
  item: name = item2, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 173 => agent = flisvos16, amount = 1, state = ON_BOARD
  item: name = item8, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 171 => agent = flisvos1, amount = 1, state = BUY
id: job_id2950680989017713956, end = 907, reward = 18215, storage = storage6; needed = 0, 0
  item: name = item11, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 166 => agent = flisvos4, amount = 1, state = ON_BOARD
  item: name = item15, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 167 => agent = flisvos14, amount = 1, state = ON_BOARD
  item: name = item5, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 165 => agent = flisvos4, amount = 1, state = ON_BOARD
id: job_id6334214708760569607, end = 841, reward = 21396, storage = storage4; needed = 0, 0
  item: name = item14, amount = 1, delivered = 1; total = 1, needed = 0
    piece id: 163 => agent = flisvos2, amount = 1, state = DELIVERED
  item: name = item5, amount = 1, delivered = 0; total = 1, needed = 0
    piece id: 168 => agent = flisvos10, amount = 1, state = ON_BOARD
  item: name = item6, amount = 1, delivered = 1; total = 1, needed = 0
    piece id: 161 => agent = flisvos5, amount = 1, state = DELIVERED
  item: name = item7, amount = 1, delivered = 1; total = 1, needed = 0
    piece id: 162 => agent = flisvos6, amount = 1, state = DELIVERED
```

An important thing must be noted about this structure and its use in practice. It does not represent all the items an agent has *on_board* but only those needed for the job as *planned by the system*. This can have a very interesting consequence in some delivery scenarios. An agent may hold a piece for a job which is not planned by the system and is

not listed in this structure; e.g., many agents may have on board *item0*, but only *agent1* is chosen to deliver and only its piece *(item0, 3)* is listed in the structure. Now it is possible that a second agent (*agent2*) that holds an unlisted piece *(item0, 1)* also holds a listed piece for the same job (e.g., a partial piece for item2) and so will do delivery for the same job too. This may happen for various reasons, one being that the system prefers to assign for delivery complete pieces (i.e., items at their entire amount) rather than partial pieces (of course if there was no whole piece available on any agent, then the system would take as many partial pieces as needed in order to complete the job). If the second agent (*agent2*) delivers first, it also delivers the unlisted piece which is hidden from the system and is not tracked at all. Then when the first agent (*agent1*) delivers, only the missing part of the piece is delivered really which is *(item0, 2)* and the job is completed. The system does not track all *on_board* pieces and does not adjust the pieces listed in the structure – it could list *(item0, 1)* for *agent2* and *(item0, 2)* for *agent1* – so in this example it has a wrong view on each delivery.

However, it is not wrong entirely. Both agents would still need to deliver and the total delivered amount is correct. The agent actions to deliver are correct too. Adjusting the structure to reflect more correctly the delivery situation would need extra processing steps in already complicated procedures and would offer no increased benefit. It is a rare case of having *a not so correct data structure that is effectively correct* for the job. It is also a case were defensive programming was used a lot in manipulating this database.

3.4 The jobs SPEC system

The *SPEC* system (from *SPEC*ification) evaluates all new posted jobs and permanently ignores those that are not considered worthwhile. The correct operation of the *SPEC* system is fundamental to the success of the team. Because the *SPEC* system uses the total buy cost of all the items as a major factor in its selection, it gets active only when all shops have been visited and all item prices are known (item prices never change during the simulation). From then on, each new job is checked and a min, max, and estimated buy cost is calculated for all the items of the job. The estimated cost is a *three-point or PERT estimation* [Pinedo and Chao, (1999), chapter 4.3], i.e.,

$$\frac{\text{min\_value} + \text{max\_value} + 4 \times \text{most\_likely\_value}}{6}$$

where most_likely_value is currently selected as a configurable preset percentage point between min and max value. In practice there is consistently small difference between minimum and maximum cost and this estimate is a very good approximation of the true buy cost in operation. Of course the final buy cost can be much lower if existing items on board agents are used, but this would be *extraordinary profit* in accounting terms. Finally the job is ignored if any of the following is true.

- *steps remaining* less than a limit (configurable and usually 63–120)
- *estimated cost* greater than a limit (configurable as a percentage of reward and usually 70%–90%)
- *reward – estimated cost* less than a limit (configurable and usually 2,000); this limit is thought of as a minimum of the *charge/service* costs and some profit remaining

- *job has the encoded characteristics of a cookie job* (see CC_op procedure in 'the control/planning logic' chapter).

4 The control/planning logic

4.1 The loop

The *control/planning logic* is the control of each agent activity, and part of it is responsible for making job plans. It performs at each step the following operations in Figure 5.

Figure 5 The loop

```
jobs_db.cleanup
jobs_db.check
fac_agents_update

jobs_spec_update

drop_jobs

complete_jobs
new_jobs
explore_jobs

service

CC_op
```

For the *jobs db, cleanup* runs mainly to handle cases when a job has expired and *check* does consistency check (and just reports any errors it finds).

fac_agents_update tracks which agents are located in each facility. It is used together with the tracking of which agents are going to each facility (which is maintained by the *goto* action class) to know which agents are *covering* each facility. We are mainly interested about shops in the algorithms, and a shop is *covered* by an agent if that agent is already inside the shop or is going to the shop.

jobs_spec_update updates the jobs SPEC system.

drop_jobs makes the decision and drops a committed job in *jobs db* if doing this is considered to be beneficial. It was added after the contest and just in time for the subsequent friendly matches.

complete_jobs plans and schedules new actions using inactive agents in order to do new partial or full deliveries with intended target to *complete committed jobs*.

new_jobs selects *new jobs* to commit to if the number of *committed jobs* has not reached the limit and plans and schedules the first actions for them using inactive agents (which actions may even complete these new jobs).

explore_jobs plans and schedules new *explores* using inactive agents. An *explore* is simply a goto to a shop with desired target to have as many shops as possible *covered* by agents at all time (it also helps that in the contest the number of shops is roughly half the number of agents) and giving preference to most promising shops so that a *complete_jobs/new_jobs* will schedule a new job delivery as fast and as completely as possible.

service calls a battery recharge procedure for any inactive agent that has stalled (i.e., has no battery charge) and only if the remaining simulation steps are more than a configurable limit (usually 200). Evidence from the matches has shown that the service fee is trivial compared to other costs, and so this limit was rather high and should be less (perhaps about 50). The recharge procedure will schedule a *call_breakdown_service* or a normal *charge* action if the agent happens to be inside a charging station. *service* is not the only procedure that can schedule a *call_breakdown_service*/*charge action*; this may happen in any of the *complete_jobs*/*new_jobs*/*explore_jobs* procedures as part of their action planning if needed and with no limitation on the number of remaining steps.

CC_op (or curious cookie job op) is a procedure that at regular random intervals (this means random intervals within a specific configurable range, usually 7 to 20 steps) posts a job with negative balance (cost of buying is less than reward) with the hope that the opponent team works on it and its performance is sabotaged. In order to speed up its development it relies on the *SPEC* database of posted jobs (by the simulation system and the opponent too) and selects the highest cost one which it posts again with the same items but with lower reward, a fixed destination storage, and a new number of steps to finish. The reward is a random number at around 43%–50% (configurable) of the buy cost of the items, the destination storage is the first storage facility always, the number of steps is a random number in a certain and big enough range of steps (configurable and usually 110–330 steps). The reward and number of steps are specifically selected to have certain properties that flag that these are fake jobs and signal the SPEC system to not consider them (so that screening is a bit faster and as an extra precaution that we do not get ourselves our own 'trick cookie'). It is not believed that *CC_op* really tricked any opponent (at least this is my impression). The reason for this is that (beyond good calculations by the opponent) the selected job is the highest cost one among those posted which may be a fake job by the opponent with unusually huge number of items or other unrealistic parameters and so easily detectable too. More work should be done so that these jobs are hard to complete but realistic, generated autonomously, and as much similar to those posted by the simulation system and with higher reward, to almost 100% of the buy cost or a bit more, with the motive to make the opponent spend a lot of time with little benefit. The current reward factor of 50% was chosen so that in case the opponent completes the job and after paying, the total of our money is still a bit more than the opponent's. This *accounting-style logic* was short-sighted. It is better in the end if we trick the opponent to spend effort and time on a job with little benefit and difficult to complete, thus allowing us to work on high benefit jobs.

The order of the operations is important. For example, *drop_jobs* may remove a *committed* job making space for *new_jobs* later to commit to a better job. Even though *drop_jobs* knows which job is better than the dropped one, it does not do any job scheduling/action planning. Based on the principle of *a single piece of code to do a specific operation*, the *new_jobs* procedure is expected to find the same better job and schedule it. Another example of order importance is that *complete_jobs* uses the best inactive agents to complete *committed jobs* (our system priority); the remaining inactive agents will be considered for *new_jobs* and the last remaining ones for *explore_jobs*.

In the next chapters the detailed workings of *drop_jobs*, *explore_jobs* and *complete_jobs*/*new_jobs* will be given. These call a common procedure to get the jobs to consider and the items needed to complete them with option to return all *new jobs* or all *committed jobs* (in which case, if there have been partial deliveries, it returns the items that have not been delivered completely yet, since the system can handle *partial amount*

deliveries of items too) or both. For *new jobs*, only jobs approved by the *SPEC* system are considered, and furthermore, an extra condition is to return jobs where all items are available in at least a configurable number of shops (usually 1, but could be 2 or more for increased chance of speedier completion).

4.2 drop_jobs

drop_jobs examines all *new jobs* against those *committed jobs* in *jobs db* for which there has not been any delivery yet and if some items of a new job are a *subset* of the items of a committed job (subset in the sense that every item of it is an item of the committed job and with lower or same amount) so that these items can be completed immediately, then drops this job if all the following conditions hold too.

- *the number of items* is less than those of the *committed* job and at most a configurable limit (essentially the same limit used in *new_jobs*)
- the items are available in a shop
- the *new reward* is at least greater than the current reward by a configurable factor (usually 1.20 for the whole job multiplied by 1.30 for each item not completed)
- the *absolute difference in rewards* is at least greater than a configurable limit (usually 2,000)
- the *new job ends later than the committed job* and the steps available are at least a configurable limit (usually 63)

The criteria really mimic a lot those used in the *new_jobs* procedure and the *SPEC* system and try to assert that the new job can be finished with higher reward and no more risk than the committed job. It is expected that the *new_jobs* procedure which will run later will select the new job and schedule it. A flaw of this logic is that it does not consider the location of the agents and the time needed to complete the current job vs. the time needed for the new job, but in any case in such a dynamic and fluid simulation world it might be more beneficial to delay a bit and deliver for a new job with higher benefit as much as it is risky to loose the current reward.

A good *drop_jobs* is essential to good performance, but evaluation of the implementation at the matches showed that it kicked in rarely and had minimal benefit. In order for *drop_jobs* to be more effective, it should also have the capability to just drop a job even if a new job replacement is not available (or a new separate procedure be used for this functionality). This is the case when a partial delivery is needed to finish the job and the *end time* is soon approaching and no agent is yet scheduled for this delivery or the agents scheduled cannot really make it until the *end time*. In this case, a precious slot in the *committed jobs* database is held for many steps (perhaps 30–50) and this prohibits the system from working profitably on a new job.

4.3 explore_jobs

explore_jobs runs for all *available agents* (not busy) and all available shops (shops which are not *covered* already by a minimum configurable limit of agents, usually 2).

A list is created with one entry for each available shop with the *total distinct items* that can be completed and the *total item amount* and *volume* of all items of all jobs

(*committed* and *new*) that can be bought. For a shop that has been visited by agents the latest inventory is used. For shops that have never been visited an estimation is made as a configurable percentage of the amount of all items (usually 50% for the *total amount* and 40% for the *distinct items*). This list is sorted so that shops that have highest *total distinct items, amount and volume for committed jobs* are first and then follow those for *new jobs*. The first shop in this list is considered to be explored by available agents. Then this shop is removed and the loop is repeated for the next shop in the list until there are no more shops or no more available agents.

Within the loop for each shop, certain key factors are computed for each available agent and an agent is chosen with the lowest value for all factors (i.e., the factors are simply a composite sort key). The factors are, from first to last

- *percentage of available agent capacity that has to be used for items needed for committed jobs*, this percentage is not used exactly as is but is rounded to a multiple of a factor (usually 25%) so that all agents are sorted in buckets/classes of capacity usage (it is wrong to choose entirely on small differences in capacity)
- *0/1 if 'not in shop'/'in a shop' already*, thus giving preference to agents not in a shop
- *amount of available capacity* (the negative value is used so higher available capacity sorts lower; added on day 2 of contest)
- *ratio of steps to go to the shop/available battery charge*, this expresses in effect the number of *steps* as a percentage of the *range* of the agent (added on day 2 of contest, on day 1 just the number of steps was used).

If the agent chosen happens to be already in this shop, then this shop is ignored. An extra check is done that the agent is not in any shop or if it is in another shop, then it is not a *single* one (alone) and there are items in the shop under evaluation to complete *committed jobs*. If the agent passes the check, it is scheduled to go to this new shop and is removed from the list of available agents; if not, no more agents are considered.

As is easily seen the key factors are: if the agent *is in shop or not*, if there is *real estimated benefit* in visiting this shop (benefit in items towards completing *committed jobs*), and *agent capacity*. This configuration tries to setup agents able to carry the highest load for *committed jobs* first and minimise unnecessary moves, especially moves where two or more agents exchange their position within their shops, because if a new good job arrives and there is no available agent to buy items (because it is moving), then there is delay in completing the job which is the worst thing in the performance measure of the system. Agent range is really the last factor because this is an exploration step but certainly cannot be ignored in the case of a *committed job* when an agent is needed to move to a shop to buy items and complete it.

Some tweaking in the order and choice of factors could be considered of course, such as specifying the range before available capacity (since in practice most items are needed in small quantity and since after first selection based on capacity percentage and the *in shop* flag, then range begins to matter a lot) and also expressing it in classes (or buckets of value, e.g., range 0–9 as value 0, range 10–19 as value 1, etc.). Unfortunately, there was no time to test effectively such variations. A correct test would have team A to compete against B and team B change each time the factors and compare the results. All tests would need to start with all agents at the same position and all items, shop inventories, and facility locations be the same every time. But it was not known how to

setup this, and the limited development time excluded this fine tuning exploration. Instead, tests were run where team A and B run the same code and the logs were examined for the agent distribution on shops by both teams (with target of making it even and matched to load capacity) and the number of total explores done (with target of minimising them). The tests could never be exhaustive of course and were guided by human intuition.

4.4 complete_jobs/new_jobs

Both call a procedure *core_setup_jobs* with a flag that selects an initial list of jobs to be only *committed jobs* or *new jobs* (for *new jobs* it runs only if the current number of *committed jobs* is up to a configurable limit, usually 3).

core_setup_jobs runs for all available agents (not busy) a preparation step for each job and item and finds out

- which agents have already *on board* the item needed for the job and only if the agent can goto to the delivery storage facility within the steps left for the job
- which agents inside a shop can *buy* the item and only if the agent can goto to the delivery storage facility within the steps left for the job.

These lists have an entry for each agent and item and are sorted in order by:

- *multiples of steps to go to the delivery facility* (usually 10) so the agents are considered in classes and not absolutely by exact number of steps (accuracy is not that good for such exactness and the exact smallest number of steps should not be the sole factor in choosing an agent to deliver jobs) *Note: 1*
- *amount* of item towards the job (negative number is used so that it sorts lower)
- *cost per item* (or 0 in the case of items on board)
- *agent name*.

Note: 1: it was added later in the friendly matches. In the contest this was not available and the sort was only by *cost and amount of item* first (in this order and the amount was the true value and not the negative value). This order was chosen so that the small amount items on board are consumed first and agents may not get full on leftover items (since the *dump* action was not implemented), but it may lead to more than one agents work on the same item and contest experience proved that it was not really needed much as the item amounts were small and there were few leftover items anyway.

A new delivery task for an agent starts always in a shop if it has to buy an item or anywhere if all the items are on board already. Starting with the initial list of jobs, a loop runs corresponding to the pseudocode in Figure 6 and with an exhaustive search finds for each job the best plan to complete it (partially or fully), then creates a *priority list* of the jobs from which a top-priority job is chosen to *commit to*, and the loop repeats with the remaining jobs until the limit of *committed jobs* is reached.

Figure 6 Core_setup_jobs

```
while list-of-jobs is not empty
  for each job in list-of-jobs
    for each item-name and item-amount in job

      for each agent that has the item-name on board and is still available
        use-amount ← amount that can be used towards item-amount
        item-amount ← item-amount - use-amount
        add use-amount to on_board-list
        if item-amount == 0 then an item was completed, exit for each agent... loop
      if all items completed then job was completed, exit for each item... loop
      if item-amount == 0 then continue with next item in for each item… loop

      for each agent that can buy the item-name and is still available
        use-amount ← amount that can be used towards item-amount and within avail. capacity   Note:1
        item-amount ← item-amount - use-amount
        add use-amount to buy-list
        add agent to list-of-buyer-agents
        if item-amount == 0 then an item was completed, exit for each agent... loop
      if all items completed then job was completed, exit for each item... loop
      if item-amount == 0 then continue with next item in for each item… loop
      if item-amount > 0 (still not completed) and
            the item is not available in sufficient quantity in a shop
            (or it is available and there is an agent in this shop who is in the list-of-buyer-agents)
        then add this item in a list of unavailable items

    if there are unavailable items
      then remove this job from list-of-jobs, continue with next job in for each job… loop   Note:2

    if both on_board and buy lists are empty
      then remove this job from list-of-jobs, continue with next job in for each job… loop

    if running for new jobs and the number of items not completed > a configurable limit (usually 1 or 2)
      then remove this job from list-of-jobs, continue with next job in for each job… loop

  if list-of-jobs is not empty
    then a new priority-list is made with one entry for each job and sort key
      MAX_INT if the job is completed else number of newly completed_items
      reward - buy_cost   Note:3
      end_step of job
    the highest priority job is selected and an attempt is made to schedule a plan (details in text)

  if jobs_db has reached the limit number of entries (usually 3) then return
```

Notes: *Note: 1:* the logic to get the partial amount that fits within the agent's available capacity was not available during the contest (the item was just not considered at all).
Note:2: available on day 2; in retrospect, this step should be done only for *new jobs* since for *committed jobs* it could severely restrict their chance to complete them faster or at all.
Note:3: available on day 2.

The *priority list* gives preference first to jobs that complete immediately or have the highest numbers of *completed items,* then to those of *higher benefit (reward – buy cost)*, and then to jobs with *more steps available until completion*. The highest-priority entry (the first one in the sort order) is selected as a job to be scheduled. Specifically for each

agent in the *on_board* and *buy* lists of this job a *new piece* entry is made in the *jobs db*, a *buy* action is queued for agents buying, a *service*/*charge* action is queued in case the agent has no charge, and then a *goto task* (by *creat*) to the delivery facility is queued (all these are skipped if the agent happens to be already there, e.g., for a previous delivery). Finally, the agent is removed from the list of available agents and the job is removed from the list of jobs too.

The agents used in the schedule are not available for later jobs which may not be scheduled and thus get delayed. So the selection order of jobs is important. In no way does the system do a search (exhaustive or A* or any other graph search) for a best overall plan considering all possibilities for job orders and agents that can be used (e.g., it cannot use a single agent for many jobs or it may use more agents than necessary in rare situations). For a single job the algorithm is able to schedule many agents each with many items or even a single agent with all the items (with a mixture of on board items and items to buy) in the fastest and most reliable way (reliable in the sense that there is less risk of delayed completion).

Unfortunately, there was a shameful bug in the implementation and during the contest the *priority list* consisted of the same values for all jobs (the values of the last job), so no real ordering was done. The situation was saved from being disastrous by the fact that usually there were few potential jobs competing at the same time. Nevertheless, this bug had an impact and this was proved by the greatly improved performance in subsequent friendly matches when it had been fixed.

5 Conclusions

The system described was finished in two months by one person and had great performance in the contest and an unbeatable performance in the following friendly matches after the bug fixes and improvements discussed before.

If we try to describe it using multi-agent terminology, we can say that it is a *goal/utility-based* multi-agent system [Russell and Norvig, (2009), section 2.4; Dix, 2012] utilising a *perceive-think-act* control loop updating the state of the agent's world and embodying heuristics and decision algorithms to generate a whole plan of many actions or a single action.

The planning of the system is embodied in *explore_jobs* and *core_setup_jobs*. Both work together. The first *explores shops* and attaches agents to shops to be ready for their next job. The second schedules *job delivery action sequences* when it is appropriate using effectively those agents in shops. An extra benefit of using this double setup of exploration and job scheduling is that the exact inventory of the shop is known and correct decisions using availability of items can be made.

Both use a *heuristic-type selection algorithm* enhanced with a *search algorithm* for a single job in *core_setup_jobs*. The heuristic is in the form of a priority queue with a specifically designed priority order that mirrors the performance measure. The performance measure is money, but since the team competes in the scale of time for a limited number of jobs (not only number of jobs available but also number of high benefit jobs it can complete successfully), it is important to make a good selection of jobs to work on and complete them fast. Therefore, the performance measure includes all these criteria in an appropriate manifestation in all priority queues referred before and in the job selection process.

The above resemble a lot the machinery of heuristics of *scheduling* algorithms especially those used in job shop scheduling [Pinedo and Chao, (1999), chapters 3, 5]. Indeed, the contest scenario is mostly a scheduling problem (minus the *assembly* action). This machinery uses a *composite dispatching rule (or priority rule)* as used in *explore_jobs* and a combination of an *exact search solution* for a smaller sub-problem and a *composite dispatch rule* for the greater problem as used in *core_setup_jobs*.

Finally, it was learned that domain knowledge is very important in designing a multi-agent system and that most work is careful software engineering.

Acknowledgements

I wish to thank the organisers and sponsors for a very interesting contest and I am eager to see the next year revisions and hopefully participate again.

References

Behrens, T., Dix, J., Köster, M. and Schlesinger, F. (2011) *Multi-Agent Programming Contest 2011 Edition Evaluation and Team Descriptions* [online] https://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi1202behrens.pdf (accessed January 2017).

Boscoe, F.P., Henry, K.A. and Zdeb, M.S. (2012) 'A nationwide comparison of driving distance versus straight-line distance to hospitals', *The Professional Geographer*, Vol. 64, No. 2, pp.188–196, doi: 10.1080/00330124.2011.583586 [online] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3835347 (accessed November 2016).

Dix, J. (2012) *Lecture Notes: Multiagent Systems I*, Clausthal TU [online] https://www.in.tu-clausthal.de/uploads/media/Multiagent_Systems_I-2012.pdf (accessed November 2016).

Hess, A.V. and Woller, O.G. (2013) *Multi-Agent Systems and Agent-Oriented Programming*, BSc Thesis, Technical University of Denmark, DTU, Lyngby [online] http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6608/pdf/imm6608.pdf (accessed January 2017).

Hindriks, K.V. and Dix, J. (2013) 'GOAL: a multi-agent programming language applied to an exploration game', in Shehory, O. and Sturm, A. (Eds.): *Research Directions Agent-Oriented Software Engineering*, pp.112–136 [online] https://multiagentcontest.org/publications/AppliedGOAL.pdf (accessed January 2017).

Intellect (2012) [online] https://pypi.python.org/pypi/Intellect/ (accessed November 2016).

Jensen, R. and Buch, B. (2014) *Multi-Agent Programming in Jason*, BSc Thesis, Technical University of Denmark, DTU, Lyngby [online] http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6798/pdf/imm6798.pdf (accessed January 2017).

Köster, M., Schlesinger, F. and Dix, J. (2012*) The Multi-Agent Programming Contest 2012 Edition Evaluation and Team Descriptions* [online] https://www.in.tu-clausthal.de/fileadmin/homes/techreports/ifi1301koester.pdf (accessed January 2017).

Nau, D. (2013) 'Game Applications of HTN Planning with State Variables', Keynote talk at *ICAPS Workshop on Planning in Games 2013* [online] http://www.cs.umd.edu/~nau/papers/nau2013game.pdf (accessed November 2016).

Pibil, R., Novak, P., Brom, C. and Gemrot, J. (2011) 'Notes on pragmatic agent-programming with Jason', *Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011)*, pp.55–70 [online] http://artemis.ms.mff.cuni.cz/main/papers/jason-lessons-revised-final_01.pdf (accessed January 2017).

Pinedo, M. and Chao, X. (1999) *Operations Scheduling with Applications in Manufacturing and Services*, Irwin/McGraw-Hill, Singapore.

pyDatalog (2016) [online] https://sites.google.com/site/pydatalog/ (accessed November 2016).

Pyhop (2016) [online] https://bitbucket.org/dananau/pyhop/ (accessed November 2016).
Russell, S.J. and Norvig, P. (2009) *Artificial Intelligence: A Modern Approach*, 3rd ed., Pearson.

Appendix

Questionnaire by the organisers

1.1 Participants and their background

What was your motivation to participate in the contest?

> I had participated in the past in similar ACM Queue ICPC challenges, which always excited me, and while I was researching about Clausthal TU in Wikipedia and saw a mention of this contest I made a decision on a whim to participate!

What is the history of your group?

> The group was new.

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

> I am not a professional researcher (unfortunately) though I keep in close touch with advancements in Computer Science and Mathematics and was once IEEE Computer Referee. I have many interests and AI is one of them.

1.2 The cold hard facts

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

> A rough estimate is 250 man hours.

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

> 4,200 for contest days and 4,500 for friendly matches.

How many people were involved?

> One.

When did you start working on your agents?

> The first create date of project files is 20/6/2016 and the first backup was on 28/6/2016.

1.3 Strategies and details

What is the main strategy of your agent team?

> It's twofold: to select jobs and complete them as fast as possible using inactive agents holding items or stationed in a shop where they can buy items, and then to spread any remaining inactive agents over shops so that they are ready for future jobs.

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

> There is no specific coordination programmed rather common code (as a common mind) is run that makes decisions for all agents. There is implicit coordination by predicting what other agents will do and arbitrating agent work by their rank.

What are critical components of your team?

> The good working of the code implementing the strategy expressed before.

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

> There is no specific programmed or reflex behaviour except when there is a planned sequence of actions in order to deliver towards a job. This can be dropped when it is decided that a new job has more benefit to work on it than the current job.

Did you make changes to the team during the contest?

> Yes, there were some changes from first to second day. They were not effective (mainly because of a serious bug that was not noticed).

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

> There are no hierarchies. All agents do the same tasks.

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

> Most of the behaviour is emergent on a team level.

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

> Planning is for a task. The longest is a delivery task when inside a shop with *buy* items, *call_breakdown_service*, *goto* charging station, *charge*, *goto* delivery station, *deliver_job*, for a total of 9 actions (for 4 buys) where each one takes one or many simulation steps.

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

> Perceive starts with *request-action* and its percept payload and then follow think and act within the allowed time limit; and the cycle repeats.

1.4 Scenario specifics

How do your agents decide which jobs to fulfil?

> The decision is made using a few criteria based on efficacy (how complete the job can be delivered), benefit (reward – buy cost), and number of available steps.

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

> No.

Do you have different strategies for the different roles?

> No, all vehicle roles function the same.

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

> No, there is a single team really. The fact that at certain times a few agents work for the same job does not constitute an ad-hoc team really.

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

> They either stay idle inside a shop or they move to a shop with job potential that matches better their load capacity and in a way that all agents are evenly spread over all shops.

1.5 And the moral of it is …

What did you learn from participating in the contest?

> I verified many software engineering teachings and how difficult it is to properly make a 4,000+ line code work right and updated knowledge on many algorithms. I also confirmed for the nth time that there is always a competitor with similar or same strategy.

What are the strong and weak points of your team?

> The strong point is that it is efficient and adaptable. The weak point is that some design/configuration decisions depend on the simulation environment. This is a weak point but not an inadequacy as an agent system is designed for a specific environment.

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

> The chosen language was perfect as it enabled quick development and code refactoring. The fact that it is slower than other common languages was no problem as it is still fast enough and when it got slow it forced a reviewing and improving of the algorithms. The methodology was good as it could handle the scenario well.

Did you encounter new problems during the contest?

> None.

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

> Yes, definitely! It's different when playing, the mind is more productive.

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

> The one week improvements have already been implemented. The one year improvements have also been described at specific places. The main design would rather stay the same.

Which aspect of your team cost you the most time?

> The initial setup of threaded agents, xml message handling, networking code, and locking took precious time at the start of the project and later with various code refactorings. The same for the jobs database. Also, all these amounted to about a quarter of the total code.

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

> For the contest: friendly matches before and after the contest, it should be stated more specifically what is allowed/forbidden for the agent implementations and the matches, e.g., restarts, remote control, use of web-services, etc.
>
> For the scenario: *a single feature relating to a specific research area* should be promoted and the rest of the scenario should stay exactly the same (the idea is to promote a specific research but not make participation very costly for teams).
>
> Really some random ideas: require totally independent agents and require communication through some protocol of peer-to-peer communication (or localised broadcast only; this is a capability needed much in the future, e.g., connected cars), I would like to see the concept of cooperation between opponent teams promoted and rewarded too (yes! but I had no time to think of a good scenario enhancement for this purpose).

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

> For this team, it's the cumulative result of a robust implementation and a good overall design that is matched to the simulation environment (a design utilising not the best but good and effective solutions). For the other teams, I cannot really reason about their performance, but excluding the element of randomness/luck (which is strong indeed), there is a feeling that some performed worse because the code was not reliable or because they did not judge well the simulation environment and they did not tune their agents (i.e., their agents were not able to select the best actions, e.g., unnecessary use of assembly). Some teams performed better because they were faster in completing many of the same jobs as this team, and so they had good or better effective planning.