G53DIA coursework 2 report
Multi-agent forager

Name: Barnabas Forgo
Email: psybf@nottingham.ac.uk
Student ID: 4211949

INTRODUCTION

The problem

In this coursework, the task was to implement a fleet of agents that solve a task in a given environment. In this environment, there are wells and stations scattered around randomly. When a station requires water, an agent can sense this as a task. Stations do not always require water, but generate this requirement randomly. The agents' objective is to collect water from wells and deliver it to stations that require it. To accomplish this task, an agent can choose to execute one of five available *low-level actions*:

- MoveAction: This action simply moves the agent to one of the eight adjacent cells. There is an alternative action that moves the tanker, MoveTowardsAction, which is only syntactic sugar for moving the agent towards a point of interest.
- RefuelAction: Moving takes fuel, which needs to be replenished occasionally. This action can be only executed when the agent is on a FuelPump.
- LoadWaterAction: To deliver water to stations first a tanker must collect water. This action can be only executed when the agent is on a Well. Wells never run out of water, and the tank can be completely refilled with one action.
- DeliverWaterAction: After collecting water, an agent has to deliver it to stations using this action. If there is not enough water in the tank to complete the task, then this action will supply the station with as much water as there is in the tank.

What makes this task difficult is that moving between wells and stations takes fuel and this can only be replenished at the centre of the environment. This effectively bounds the otherwise infinite environment, and limits the amount of tasks that can be completed before refuelling. If the agent runs out of fuel then it will not be able to complete any other movements until the end of the run. Furthermore, the agent cannot see the whole environment; it can only sense cells in a 12-cell (Chebyshev) distance. As a consequence, it will not be able to detect new tasks from stations unless it moves close enough to them. At the end of a run, the fleet's performance is scored with the formula:

$$T = number\ of\ tankers$$
$$c_t = completed\ tasks\ for\ tanker\ t$$
$$d_t = delivered\ water\ for\ tanker\ t$$

$$fleet\ score = \frac{\sum_{t=0}^{T} c_t * \sum_{t=0}^{T} d_t}{\mathrm{T}}$$

According to these criteria, the environment can be classified (based on the classification proposed by Russel & Norvig [1]) as a

- *discrete* – even though the environment is theoretically infinite, in practice it only has some well-defined limited number of states as opposed to continuous environments
- *partially observable* – since the agent can only see in a 12 cell radius
- *dynamic* – because tasks are generated randomly that is out of the agent's control
- *deterministic* – as the result of the agent's actions that modify the state of the environment can be predicted (disregarding the generation of tasks)
- *multi-agent* environment.

Moreover, there are achievement goals (with differing utilities, and resource bounds) and a maintenance goal, which run in parallel. Lastly, agents' actions are infallible, unless it tries to execute an action that does not make sense (e.g. moving towards a point of interest when the agent is already there).

SOFTWARE DESIGN

Organisational structure

Probably the most important design decision in developing multi-agent systems (MAS) is deciding what organisational structure to follow, since this has implications of the design of all agents in the system. One of the key properties of a MAS is whether to have specialised agents that focus on completing one task effectively or to have agents that can accomplish all actions (totipotent). One obvious solution would be to have some discovery agents that just roam around looking for tasks and others which only complete them. The issue with this is that a scout tanker would not accomplish any tasks and therefore have a score of zero. Since the aim of the system is to maximise the score, this would not be very efficient, as it would only add one to the divisor of the score formula but none to the dividend. For this reason, it was decided to use totipotent agents.

The next key decision is how to balance cooperation and competition between agents. Since the goal in this environment is to maximise the score, it would make sense for the individual tankers to maximise their own score, and thus maximising the whole system's score. This means that agents would be very competitive, with almost no focus on explicit cooperation. This also means that most components could be reused from the single agent system that was previously developed, since its task was to optimise its own score. This also implies that individual agents only share goals implicitly and that control is distributed between all agents as opposed to being hierarchical. Lastly, since other agents are not explicitly considered the agents can be implemented in a way that the system can support any number of agents. Then it can be easily determined how many agents are required to get the maximal score.

Another key feature in MASs is the way agents communicate. Since there are no clear advantages or disadvantages of each communication type this was abstracted away to an interface. The idea here is that in this environment communication is 100% reliable. In fact, tankers are directly sharing some parts of their memory. On the other hand, if only a less reliable means of communication was available in another environment then the communication protocol could be implemented in a different way that uses caching or peer-to-peer data exchange to facilitate reliable data sharing. Therefore, if the interface was implemented using a different concrete communication protocol then the tankers would not have to be changed. The concrete implementation of this interface that was used is a mixture of a server-client model combined with a publish-subscribe pattern. That is some data (e.g. list of wells) is available on demand, others (e.g. a new task is available) are sent directly to tankers. A further advantage of using this abstract communication protocol is that the tankers and the server could all run on different machines and still be able to work together.

The last key property of the system is that the interactions between agents are not predefined as each tanker is the same and they do not control each other directly. Therefore, the emerging system can be characterised as a redundant generalist organisation with a distributed control structure.

Agent design

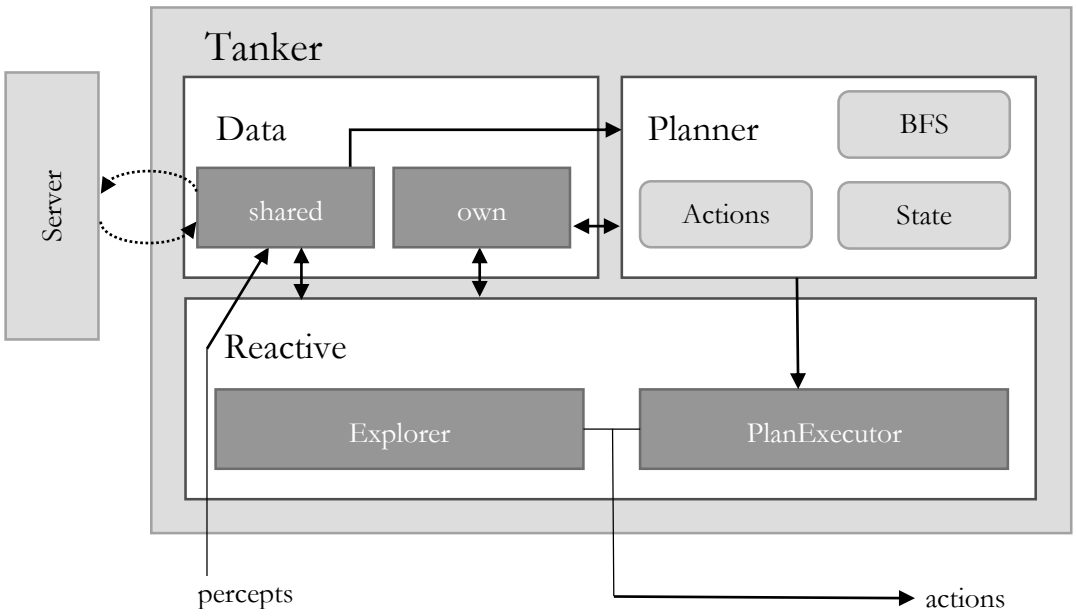
The most important factor in choosing the architecture for the agents is the environment and the task(s) it has to complete. If this property is chosen properly then it can make the task easier. The architecture chosen was a hybrid hierarchical architecture with three layers: a reactive low-level control layer, a planning layer, and a data layer that includes communication with other agents as well. The reason for choosing this architecture is as follows.

As the environment is quite easy to navigate and there are no obstacles, the obvious first choice would be a reactive agent. Furthermore, there are a relatively small number of tasks that a reactive agent could handle. What reactive architectures excel at is quick reactions to changes in the environment (hence the name). However, since the environment is not changing that fast it would not make sense to use a reactive approach. Moreover, there are resource constraints on the tasks, which require some sort of planning; therefore, a reactive agent cannot solve this problem efficiently.

The next candidate therefore is a purely deliberative system, since the completion of tasks needs to be ordered which can be accomplished by a planner. When the branching factor of the state space is high, a purely deliberative system might fail as it would spend too much time planning, and by the time this finishes, the environment might change. However, this is not the case in this environment as there is no time constraint, since the environment will not change until the agent makes a move. For these reasons, a deliberative architecture would be an appropriate choice; yet, the design was taken one step further.

The reason why a hybrid architecture was used is that if actions were fallible in the environment the reactive layer would make sure that the plan produced by the planner is correctly executed, and therefore no explicit planning would be required for cases when actions fail.

Agent architecture



The figure above shows the basic structure of the solution. Even though the figure might be complicated at first, the underlying concept is quite simple.

Every run starts with a predefined exploration route. This is handled by the Explorer part of the reactive layer. To stop multiple tankers from going on the same route, the server keeps track of which exploration route has been visited. On every step, the view of the Tanker is pre-processed by the Reactive layer and the information is uploaded to the server. With this initial exploration about 95% of the environment will be discovered, which usually includes a few tasks as well.

After the initial exploration has finished a plan is generated using the Planner layer. The Planner uses Breadth-First-Search (BFS) to find the best sequence of actions to execute. The search tree that the algorithm traverses consists of two components: State/FutureState nodes and Action edges. State is a mutable data container class that stores all relevant information of the tanker. The root of the search tree is always the current State. A state can be expanded to FutureStates by applying Actions (one of Refuel, FillTank, CompleteTask, or EmptyTank). These high-level Actions consist of moving towards a point of interest and executing a basic action. For example, the Refuel Action consists of moving to the FuelPump and executing a RefuelAction.

The search algorithm expands this abstract search tree until the next Refuel, i.e. it enumerates all possible actions between two Refuels. To choose the best sequence, a scoring function is used. This scoring function uses an objective and heuristic score to calculate the utility of a plan. The objective score rewards completing tasks and the heuristic one rewards non-scoring actions (e.g. refilling the tank before Refuelling). After all possible sequences of actions have been enumerated and the best one has been chosen, the best plan is handed over to the PlanExecutor.

When the PlanExecutor accepts a plan it parses it and publishes the intention of completing these tasks. This effectively makes the tasks in the plan invisible to other tankers, therefore preventing any interference. When the plan has been locked in, the PlanExecutor translates the high-level actions to individual low-level actions; i.e. a CompleteTask Action would be translated to some number of MoveTowardsActions finished by a DeliverWaterAction. While the plan is being executed, the environment is continually parsed by the Reactive layer and thus the server and other agents are always up-to-date.

If new information is encountered (new well or task) by any agent the server fires an event which is received by all tankers. If the event is in the tanker's range (i.e. if it discards the current plan and moves to the event location it can still get back to the FuelPump), then the Planner recomputes the plan and hands it down to the PlanExecutor. If the new plan is better, then the PlanExecutor replaces the current plan by releasing the locks that were held for the previous plan and acquiring locks for the new tasks. A better plan is defined as either having a relatively high score increase or the plan that goes further from the fuel source, and thus exploring a bigger area. This makes the agent commit to tasks that are on the edge of the map.

When there are no more tasks available to the agent then it switches to 'reexploration' mode. When switched to this mode the tanker requests the next free exploration path from the server, locks it for itself, and starts exploring on that path. The next exploration path is produced by analysing the least recently visited stations, clustering them, and returning the exploration path that corresponds to the cluster with the most least-recently-visited stations. If other tankers are reexploring as well, then these paths are filtered out and the first free one is returned. Since there are only four clusters, there can be only four tankers in the fleet. If this constraint is removed then the system can support any number of tankers. During reexploration, the agent does not collect water or complete tasks.

SOFTWARE IMPLEMENTATION

Data layer

Server

This class is only a wrapper for storing and accessing the current knowledge. The following data structures were used to store the information:

- *HashMap for wells and full map storage* – since the only important part about wells is their location therefore it makes sense to create an unordered map of the pair (Position, Well). The complete internal representation of the environment is stored only for debugging purposes.
- *LinkedHashMap for stations* – this data structure was used to store stations as it can be iterated in access order, i.e. the first will be the least recently visited station. This property of the data structure was used to provide some heuristics on what reexploration path to take next.
- *PriorityQueue of PlannableTasks* – PlannableTasks were implemented to provide a way to reason about tasks, also containing their Position. An idea that was discarded was to complete tasks starting with the one that requires the most water. This data structure stores these tasks in that order and the biggest task can be polled in constant time. Even though this feature is not utilised in the final implementation the data structure was kept, as it is not that costly to keep the tasks ordered.
- *ArrayList of data change Listeners* – a simple list was kept to notify tankers of new tasks and wells.

Planner

The Planner's job is to create a vector of high-level Actions that is most optimal for the tanker to execute, according to the current knowledge. The following planning algorithms were considered to complete this task:

- *A** - first developed by Hart et al. in 1968 [2] the algorithm is mainly used for finding shortest paths between two points. The algorithm utilises a heuristic function that optimises node expansion in the search tree. The algorithm was to be altered so that instead of finding shortest paths it would maximise the score of a plan. This idea was discarded though, as an admissible heuristic function could not be established.
- *A* with bounded costs (ABC)* – developed by Logan in 1998 [3-5], this algorithm is similar to A* (as the name implies), as it is used for route planning. However, the difference lies in that it is not optimising a single scoring function, but rather it tries to satisfy a set of ordered constraints. This algorithm was not implemented either, as it was deemed too complicated to implement.
- *Multiobjective A** - developed by Mandow and Pérez in 2005 [6], similarly to ABC, this algorithm is capable of optimising multiple scoring functions at the same time. Again, the implementation of this algorithm was out of reach for this task.
- *BFS* – this classical algorithm was the first to produce promising results. At first, it seemed that it would be impossible to use this method, as the search space seemed too large. However, after making a few careful adjustments to the algorithm made it produce great results in an acceptable timeframe. This solution, however probably would not be feasible in a real-life situation, since the time it takes to search a whole search tree might take too long in a time-bounded environment. However, this was not a constraining factor in this task.

The search always starts with the current State. This State is constructed by collecting all the relevant information that is required for the planning process, such as the fuel level, water level, number of completed tasks, the amount of water delivered so far, current position and tasks. The server only returns tasks that are available to the tanker.

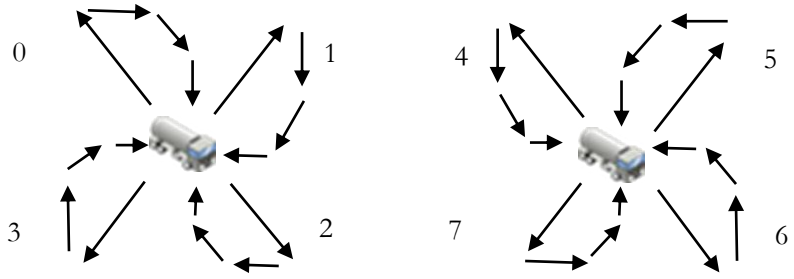
To construct the full graph from the root State the following expansion technique was used. Whenever a State expansion happens, it is checked whether the resulting FutureState is not too far from the FuelPump. It is only added to the list of child States if it passes this check. First, the State is expanded with all FutureStates where an extra task is completed. If the water is not enough to complete the task then it is added as an EmptyTank Action. Then, the State is expanded with FillTank Actions that take the Tanker to Wells if the water level is less than the maximum. Lastly, if the fuel level is less than the maximum the state is expanded with a Refuel Action. However, if the State is reached by Refuelling the expansion does not produce any child States, as the search's goal is to make an optimal tour. One issue with this approach is that the Tanker will not refuel if it passes by on the FuelPump, as it is completing other tasks. However, this was one of the optimisations that radically reduced the search space.

When new States are generated, it is compared with the current best State (initialised with the starting State). If the new State's score is better, then it is updated to be the new best State. States are scored based on the sum of an objective and a heuristic scoring function. The objective scoring function is the same as the simulator uses. The heuristic scoring function adds a small score increase for completing tasks that do not have immediate objective score rewards (e.g. refilling the tank before refuelling). What this accomplishes in practice is that the search focuses on completing tasks, but if there is something that the tanker can do with a small path alteration, it will do it.

Reactive layer

Explorer

For the Planner to work some sort of initial knowledge needs to be established. For this purpose, a static path exploration was implemented. When the single-agent system was developed, the following two paths were the best performing static exploration paths.



exploration reexploration

Initial exploration happens on the paths labelled with 0-3. As mentioned before the server keeps track of which paths have been visited and thus no two tankers can explore on the same path. When tankers run out of tasks, they request a new exploration path from the server. The server then analyses the LinkedHashMap (LHM) of stations to find the exploration path that might yield the most undiscovered tasks. Since a LHM iterates in last accessed order, therefore the first element will be the Station that has been visited least recently. As Stations have some low chance to generate a Task randomly on every tick (until they generate one), the least recently visited station (LRVS) has the highest chance of having a task. Using this information the server takes the Positions from the LHM and calculates an approximate of which path each station belongs to (quadrant). Then it calculates how many of the first five LRVS are there in each quadrant. The one with the most LRVSs is then returned. When the tanker receives this, it locks the received path and its complement (4's complement is 0, 3's is 7, etc.). The complement is locked as it shares a big portion of the explored area with the complemented path. If the complement was not locked, the system could support up to eight tankers.

PlanExecutor

The PlanExecutor is the main arbiter of allocating tasks to tankers. When the Reactive layer asks the PlanExecutor to produce an action, it does one of three things.

The first one is when there is no plan, or the current one is complete. In this case, it simply generates a new one using the Planner. Once the plan has been generated, it announces that this tanker has the intention of completing the tasks that are in the plan. This makes these tasks invisible to other agents.

The second case is when a plan has been computed, but it has not been completed. In this case, the PlanExecutor simply translates the current high-level Action into a low-level one.

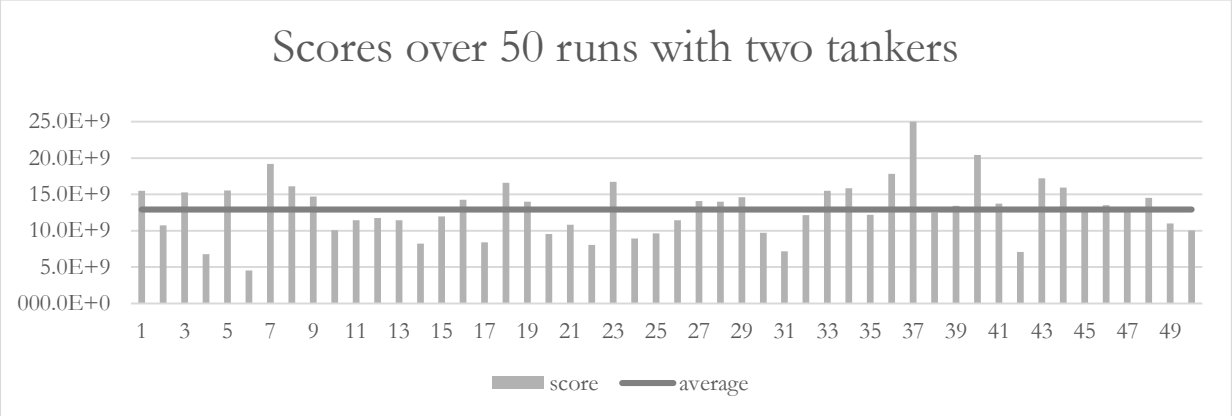
The last case is when the tanker has received an event that in turn triggered a replanning. In this case, the PlanExecutor generates a new plan again. When that happens the new and old plan is compared. If the new plan does not improve on the score dramatically, then the plan that takes the Tanker further will be taken. This combines exploration with task completion decreasing the need for reexplorations, which tend to decrease the overall score. When a new plan replaces the old one, the locks that were held for the old tasks are released and new locks are acquired for the tasks in the fresh plan. This appears to other tankers as if new tasks were discovered. This behaviour is similar to a contract net protocol, as the agent that has just released the task cannot complete the task anymore and requires help. However, there is no bidding; the task is reallocated on a first-come, first-served basis.

Reactive

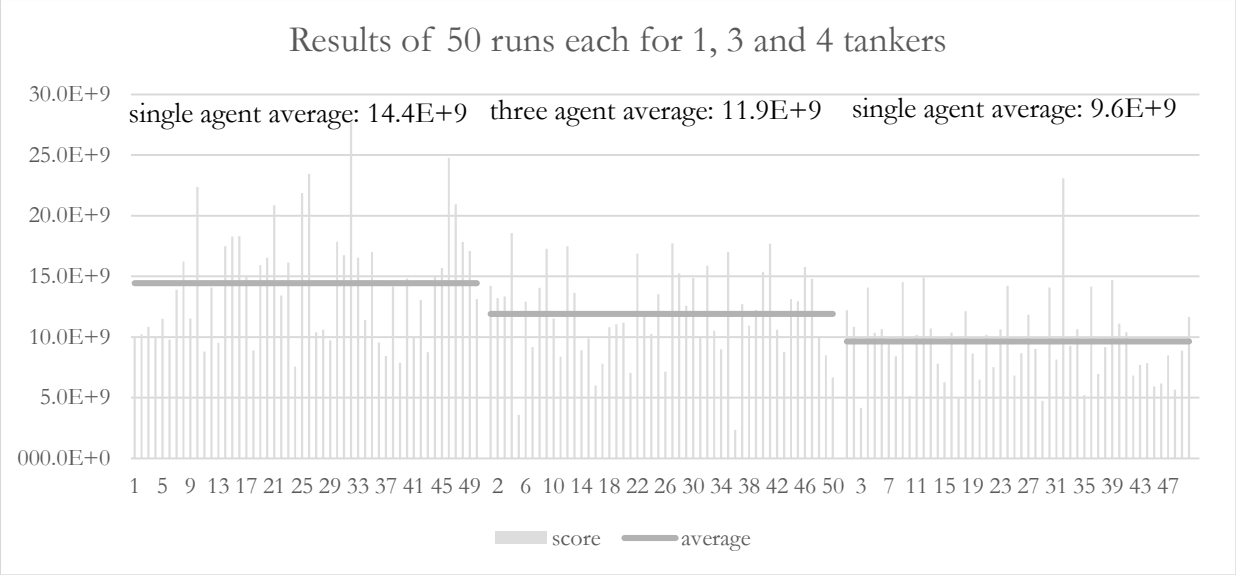
This class is the glue between all above components. It chooses when to use Actions from the Explorer or the PlanExecutor and it takes care of inferring the Tanker's position so that the Server's data can be kept up-to-date.

EVALUATION

To summarise, it can be said that the solution proposed here performs very well. Experiments show that the current implementation scores 12.9 billion points with two agents. The results are shown below.



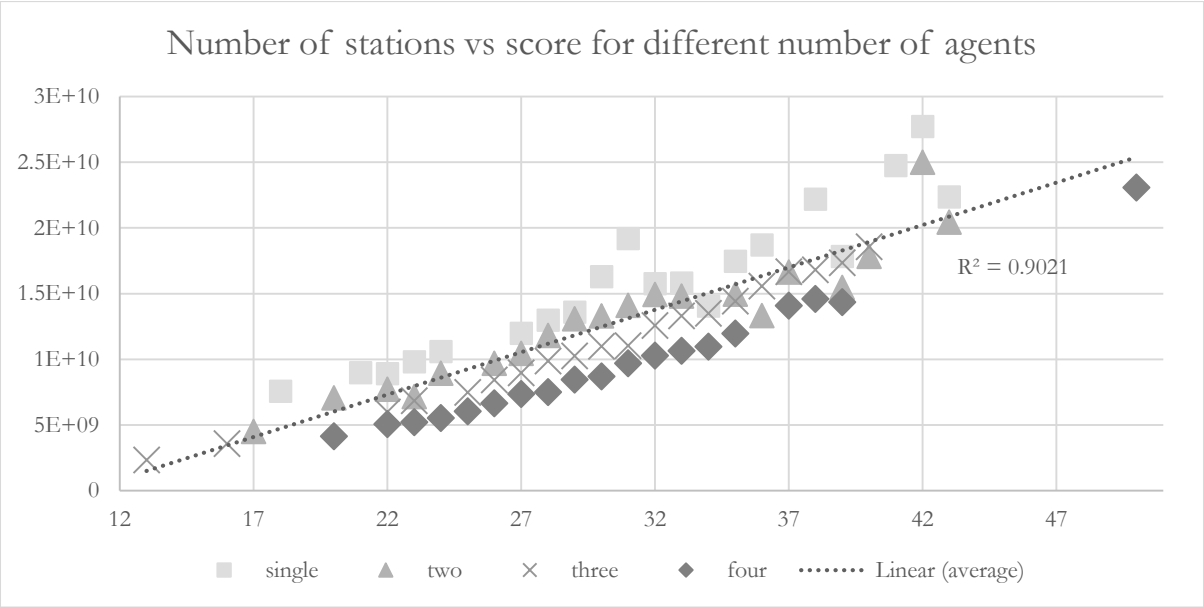
Even though, intuition might dictate that more agents would have a better score this is not the case. The results of the experiments with 1, 3 and 4 agents are shown below.



It is easy to see that the number of agents is inversely proportional with respect to the score. Although this is very counter-intuitive, it can be explained very easily. Suppose that there is a task environment where 2000 tasks are offered over a run, which collectively require one million water. If this was equally split between two agents then, each would complete 1000 tasks and deliver half a million score. Such a fleet would score 1 billion. However if these tasks were split over four agents (500 tasks, quarter million delivered) the score would only add up to 500 million.

As a conclusion, it can be said that even though more agents might complete more tasks and deliver more water, with this scoring system they are not efficient enough to justify having more than two (or in fact more than one).

Although these scores are can be considered good, the implementation is not perfect, since it often performs below 10 billion points. However, this is not necessarily the agent's fault.



The chart above shows, how the environment changes the score of the Fleet. On the X-axis, it shows the number of stations; on the Y-axis, it shows the results of a fleet of a specified size where each data point is the average of some (1-5) runs that had stations corresponding to the X-value. It is very clear that the fleet's score is only dependent on the number of stations in an environment. This is especially true for the

four-tanker fleet, where the trend has an R^2 of 0.98 (not shown here). This is very easy to explain as if there are more stations then there will be more tasks offered.

CONCLUSION

To summarise it can be said that the MAS presented here is performing very well in the given task environment. However, if it was placed in any other environment it would fail, as the search algorithm and communication protocol was designed and optimised to exploit this specific environment. This exploitation includes the lengthy search process, the way knowledge is stored and updated, etc. Furthermore, this implementation cannot be considered complete, as there are many ways it can be improved. For example, the search algorithm could be replaced with a guided search, the heuristic for accepting new plans could be improved, some constraints could be relaxed when nearing the end of the run, etc. However, given the time constraints of this assignment, this MAS implementation can be considered to be performing very well.

REFERENCES

[1] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs,* vol. 25, pp. 27, 1995.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on,* vol. 4, no. 2, pp. 100-107, 1968.

[3] B. Logan, "Route planning with ordered constraints," *COGNITIVE SCIENCE RESEARCH PAPERS-UNIVERSITY OF BIRMINGHAM CSRP*, 1998.

[4] B. Logan, and N. Alechina, "A* with bounded costs." pp. 444-449.

[5] N. Alechina, and B. Logan, "State space search with prioritised soft constraints," *Applied Intelligence,* vol. 14, no. 3, pp. 263-272, 2001.

[6] L. Mandow, and J. P. De la Cruz, "A New Approach to Multiobjective A* Search." pp. 218-223.