G53DIA coursework 1 report
Single agent forager

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

Introduction

The problem

In this coursework, the task was to implement an agent that operates in a given environment. In this environment, there are wells and stations scattered around randomly. When a station requires water, the agent will sense this as a task. Stations do not always require water, but generate this requirement randomly. The agent's objective is to collect water from wells and take it to stations that require water. 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 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, an agent's performance is scored with the formula: completed tasks x total delivered water.

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)
- *single agent* environment.

Moreover, there are achievement goals (with differing utilities, and resource bounds) and a maintenance goal, which run in parallel. Lastly, the agent's 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).

The agent

The above classification has several implications on the agent's architecture and implementation. Firstly, since the environment is only partially observable, the state of the environment will have to be stored in memory. Secondly, as the environment is deterministic and the agent's actions are infallible, there is no need to check whether an action has succeeded and planning can be used to predict future states of the environment. This report will describe the hybrid agent that was implemented to solve this problem.

BACKGROUND MATERIAL

Architectures

As in any other field of computing there are many ways to solve this problem. The most defining factor of an agent is the basic architecture that it has. This defines how it will represent the world internally and how it will choose an action based on its percepts. Since each architecture has advantages and disadvantages, this property has to be chosen carefully, so that it can complete the task at hand efficiently. In the following few paragraphs, the three core architectures of agents will be shown.

Reactive

Reactive architectures are the simplest way of controlling an agent. Usually such an agent does not even have a representation of the world; rather it is simply a mapping from percepts to actions. As a consequence, reactive agents are the easiest to program, but they are incapable of complex reasoning. However, reactive agents that add state (or a representation of the environment) are capable of any behaviour. While there are many well-known reactive systems, probably the best known [2] is the subsumption architecture proposed by Brooks in 1986 [3] and further developed in 1991 [4, 5].

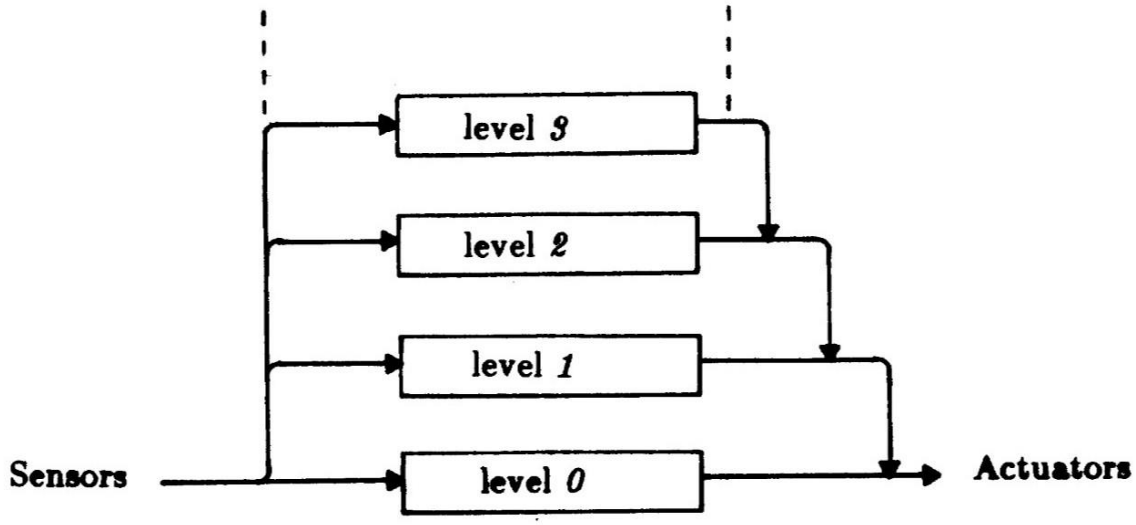


Figure 1 High-level view of the subsumption architecture (from [3])

The architecture's high-level view is shown in Figure 1. With this architecture, Brooks is trying to prove that complex behaviour is not necessarily the cause of a complex control system, but a reflection of a complex environment. This system works as follows. The agent's main capabilities are ordered into layers. Each layer is capable of doing one action and each layer gets all percepts. When a higher-level layer wants to take control of the agent it subsumes the layers below it. A concrete implementation of this architecture is shown by Mataric (1993) [6]. She applied this theory to robots that collect pucks and return them to a base. The subsuming layers in this implementation were dispersion, collision avoidance, flocking, following, aggregation, and homing.

While the reactive architecture is very simple conceptually, since it is a direct mapping from percepts to actions it can react to a rapidly changing environment very quickly. However, when it comes to complex problem solving or when multiple solutions exist for a single problem reactive agents fall short. Furthermore, this type of agents requires that all situations need to be programmed in advance, since it cannot use reasoning to develop new solutions.

Deliberative

A deliberative agent is any agent that considers alternative courses of action and chooses one to execute. To achieve this such an agent must have an internal representation of the environment that it inhabits and it must be able to reason about its own actions without executing them. The most widely adopted model for developing deliberative agents is called the Belief-Desire-Intention (BDI) model. First coined by Bratman in 1987 [7], the model was developed to explain future intentions in human reasoning in the field of psychology. However, surprisingly, the book was most influential in intelligent agent design. The algorithm that powers these agents can be simplified to (from [8]):

- based on the agents sensory information the beliefs are updated
- based on the updated beliefs some options for the agent are generated
- the generated options are then filtered producing intentions
- finally a single intention is selected that will be executed

Even though deliberative architectures offer solutions to many shortcomings of reactive agents, it is still not the best solution for all task environments. If an accurate symbolic representation cannot be created of the environment then the reasoning is pointless, as the future state of the world cannot be predicted. Moreover, since planning takes an unpredictable amount of time, if the environment is time constrained then regular planning techniques cannot be used, and therefore a deliberative agent might be inapplicable.

Hybrid

Hybrid architectures try to solve the problems of both reactive and deliberative agents by combining them into a single system. In a hybrid agent, there are one or many layers of both reactive and deliberative layers. The reactive layers' task is to handle simple, low-level actions, while the deliberative layers' task is to sequence the reactive layers' actions. One key problem with hybrid architectures is that the two types of layers' have fundamentally different ways of representing the world and controlling the agent. To alleviate this problem a way of unified control needs to be established between layers. A common solution is to organise the reactive layers below the deliberative ones, such that the reactive layers act as a way of abstracting the problem to a higher level. By using this solution, the deliberative layer can be coded on a higher level without needing to worry about the inaccuracies of sensors and actuators, as that is handled by the lower layers. Such a system is called hierarchical control. A hybrid agent if engineered properly, can fit into most (if not all) environments.

SOFTWARE DESIGN

The most important factor in choosing the architecture for an agent 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 belief (modelling) layer. 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. However, there are resource constraints on the tasks, and the environment is not changing that fast to justify a reactive approach.

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. Since there is no time constraint on how long planning can take this approach would be fine, however the branching factor of the low-level actions (mainly because single cell move actions) is too high for any planning algorithm to handle.

Therefore, the last possible approach, hybrid was chosen. The reason for this is that both deliberative and reactive approaches seemed unable to complete the task efficiently for the above-mentioned reasons. In the following few paragraphs the high-level structure of the solution will be explained.

Architecture

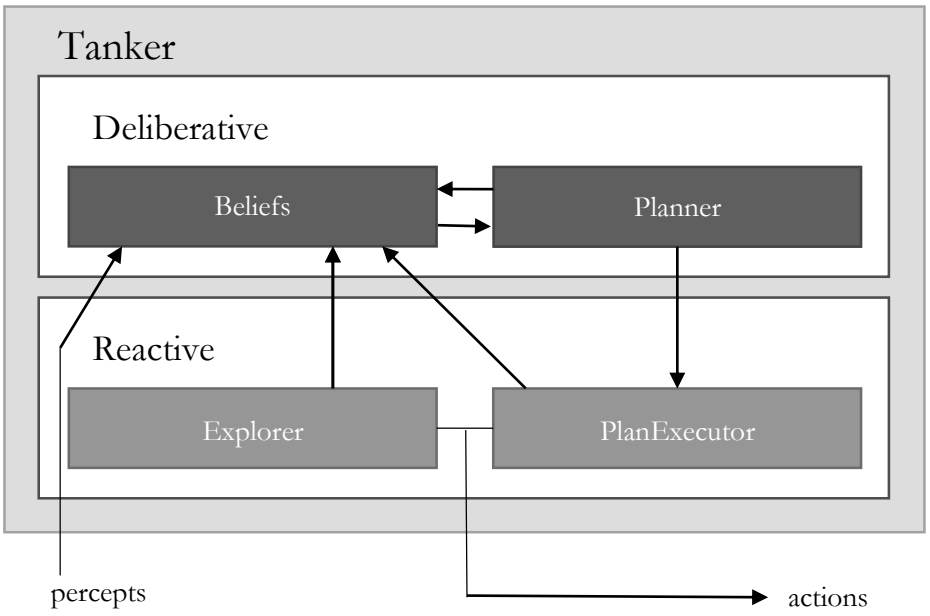


Figure 2 Conceptual architecture of the tanker

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. On every step, the view of the Tanker is pre-processed by the Reactive layer and the information is handed over to the Deliberative layer to update the Beliefs. With this initial exploration about 95% of the environment will be discovered, which includes about 5-10 tasks.

After the initial exploration has finished a plan is generated using the Planner layer. A plan consists of several high-level steps that end with a refuel action. This uses the current beliefs and a Breadth-First-Search (BFS) algorithm to construct all possible sequences of actions that can be executed. The actions that the planner uses are a bundle of low-level actions; i.e. move to a station and do complete a task. It scores these plans using an objective and heuristic scoring function. After all possible sequences have been enumerated, it chooses the best plan, and hands it over to the PlanExecutor. Therefore, the planner makes the most optimal return tour from and to the fuel pump.

After the initial Plan has been created, the PlanExecutor translates the high-level actions to individual low-level actions; i.e. the above high-level task would be a number of move towards actions finished by a deliver water action. While the plan is being executed, the environment is continually parsed by the Reactive layer and thus Beliefs always contain the most up-to-date information.

If new information is encountered (new well or task) the Planner recomputes the plan and hands it down to the PlanExecutor. If the new plan is better, then the PlanExecutor accepts it and discards the previous plan. 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 the Deliberative layer hands the control back to the Reactive Explorer. This is called a 'reexploration'. This task is executed on a fixed path much like the initial exploration. However, the reexploration path is designed so that it explores the remaining 5% of the map. In addition, each reexploration alternates between the exploration and reexploration paths. The difference between initial and reexploration is that the latter stops when there are enough tasks for the tanker to plan new tours.

SOFTWARE IMPLEMENTATION

Deliberative layer

Beliefs

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. 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 seen station. The idea here was to run reexploration between stations in least recently visited order, however this idea was not implemented, but since storing station in this special data structure does not have any additional costs it was kept unchanged.
- *PriorityQueue of PlannableTasks* – firstly, PlannableTasks were implemented to provide a way to reason about tasks, also containing their Position. Secondly, another 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.

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 [9] 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 [10-12], 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 [13], 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 a feasible 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.

To use BFS a search graph had to be created. This was done by making a State class that contains all the information about the tanker that actions can modify (fuel, water, number of completed tasks, delivered water). The States act as nodes in the graph and Actions were added that transform one State to another. The Actions can be one of Refuel, FillTank, CompleteTask, EmptyTank. Each Action can be applied to a State that returns a new FutureState that represents the world after executing the Action.

To construct the full graph the following expansion technique was used. Whenever a State expansion happens, it is checked whether the resulting state 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

Explore

For the Planner to work some sort of initial knowledge needs to be established. For this purpose, a static path exploration was implemented. The following paths were considered:

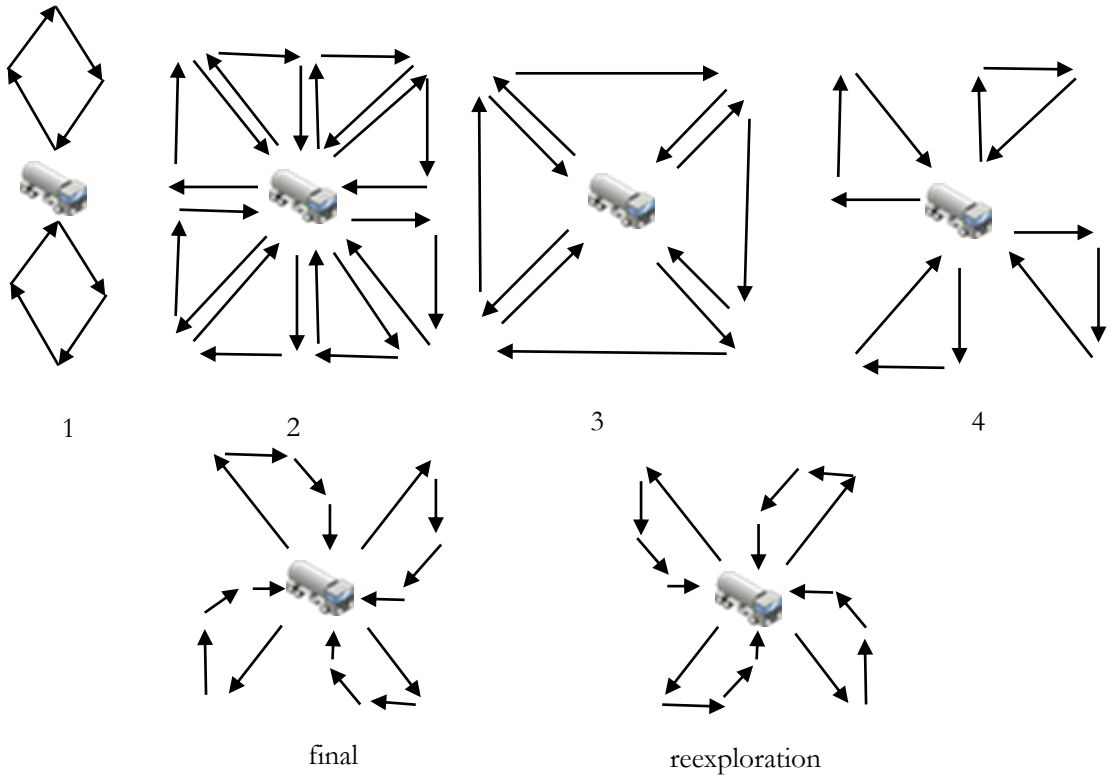


Figure 3 Exploration paths

1. This path was promising, as it had the highest discovery efficiency (discovered cells / steps it took) however it left huge spaces unexplored, and it had discovered many cells outside of the Tanker's range.
2. "The full stack path" was a naïve path that had discovered more than 90% of the usable are, but its efficiency was excessively low for practical usage.
3. This path was another obvious choice, but its long straight sub-paths rendered its efficiency low.
4. "The half stack path" was fairly efficient, but the trade-off was that it had left large spaces unexplored.
5. The final path was a mere experiment at first, but the results were surprising. The idea was to visit the corners of the map (the hardest to reach parts of the map) and then guide the Tanker back from there, going along the edges as long as possible. This resulted in discovering 95% of the usable map with a surprisingly high efficiency.
6. The reexploration path is simply the mirror of the fifth path. Combined with the last path it discovers the full map, and since it starts from the opposite corner, it makes it statistically easier to find new tasks on this path.

PlanExecutor

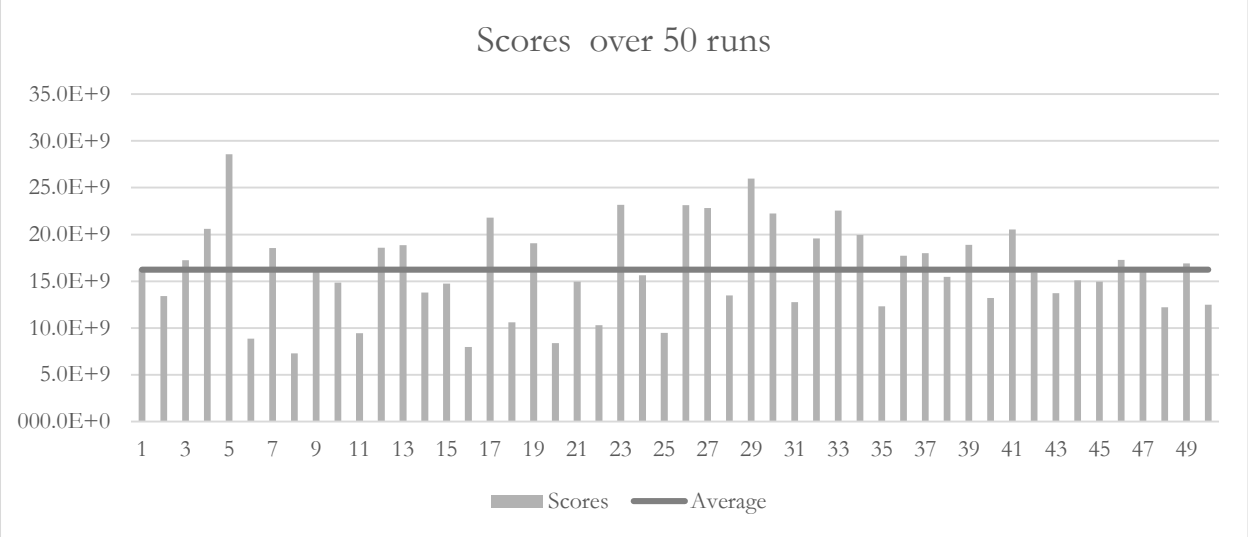
The PlanExecutor simply takes a computed plan from the Planner and translates the high-level Actions described above into low-level ones that the library can understand. It continues doing so until new information about the environment requires the Planner to compute a new plan. When that happens the 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.

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 Deliberative layer can keep the Beliefs up-to-date.

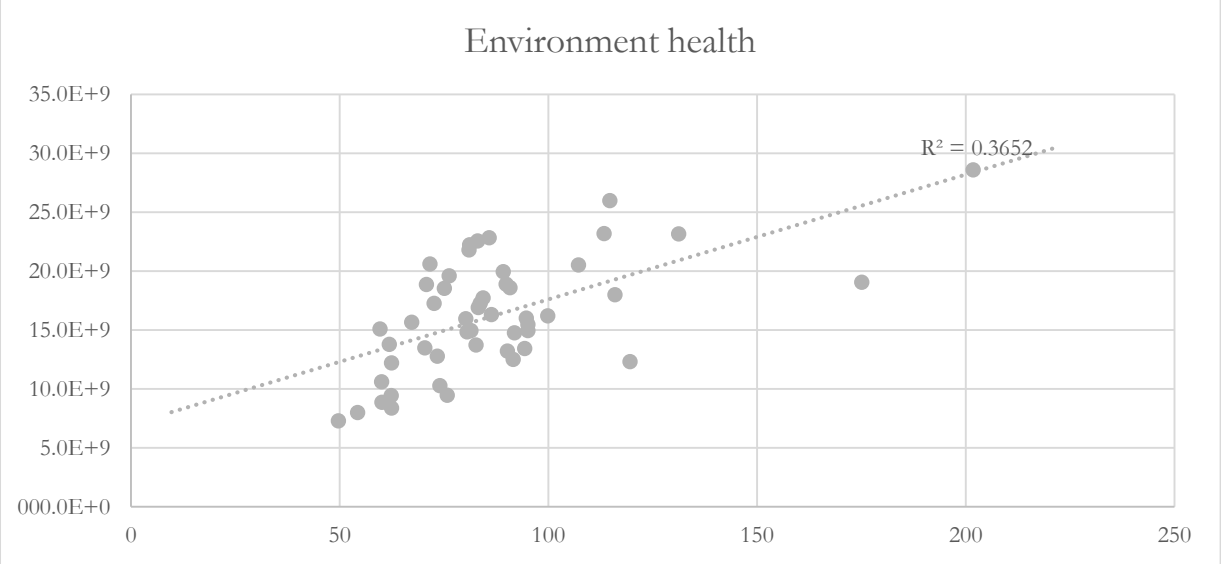
EVALUATION

Overall, it can be said that the solution proposed here performs very well. Experiments show that the current implementation scores 16.2 billion points. The results are shown below.



This is the result of many optimisations. At first when only the BFS was implemented the average score was around 14 billion. Then when reexploration was optimised to stop when there are enough tasks the average had risen to about 15 billion. Then when the initial plan selection was implemented, the score further increased to 15.5 billion. After tweaking the threshold for accepting new plans, the above presented score of 16.2 billion points was achieved, with a maximum of 28.6 billion points.

Although these scores are considerably high, 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 Tanker. On the X-axis, it shows (average required water / reexploration times); on the Y-axis, it shows the tanker's final score. Although it is a weak correlation, it can be seen that if the average task requirement is high, and the tasks are in the right place the tanker can perform very well. While if the offered tasks are far away from the centre and have a low requirement it will perform very poorly.

Conclusion

To summarise it can be said that the agent presented here is performing very well in the given task environment. However, if it was placed in any other environment it would fail, as its search algorithm and percept parsing is thoroughly 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, the reexploration path should not be always static but should revisit the least recently visited stations, as intended originally, etc. However, given the time constraints of this assignment, this agent 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] M. Wooldridge, *An introduction to multiagent systems*: John Wiley & Sons, 2009.

[3] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of,* vol. 2, no. 1, pp. 14-23, 1986.

[4] R. A. Brooks, "Intelligence without representation," *Artificial intelligence,* vol. 47, no. 1, pp. 139-159, 1991.

[5] R. A. Brooks, "Intelligence without reason," The artificial life route to artificial intelligence: Building embodied, situated agents, pp. 25-81, 1995.

[6] M. J. Mataric, "Designing emergent behaviors: From local interactions to collective intelligence." pp. 432-441.

[7] M. Bratman, "Intention, plans, and practical reason," 1987.

[8] I. Vlahavas, and D. Vrakas, *Intelligent techniques for planning*: IGI Global, 2005.

[9] 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.

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

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

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

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