G53DIA Assignment 1 Report

Joel Cox psyjc5

Introduction

The task given was to pick up waste located at various stations across an environment, and deliver it to similarly placed wells, without running out of fuel. 10 000 time steps were allowed to deliver as much waste as possible.

Included alongside this report is a spreadsheet log of the testing performed on the agent.

Representation

The environment consists of a global point co-ordinate system, which is inaccessible to the agent. Each Station, Well, and Fuel Pump are only visible to the agent if they are within the view range. This means that if a Station acquires a task, the agent will only see this change if the station is in view.

Global Position

To combat the agent's forgetfulness, an internal representation is created to store all features discovered. Additionally, the global co-ordinates of each feature are calculated on discovery, using the agent's current location. For example, should the agent be at (4,4) and see a feature at relative (6,6), the agent can calculate that the feature's global position is (10,10).

Locations

To more easily record features and their properties, the agent creates a Location object to put it in. This class contains the type of feature, global (x,y) co-ordinates and a copy of the feature's Point for use with framework functions.

Clusters

In addition to Locations, Clusters are used to rank fuel pumps for a late-stage step in the agent's algorithms. Clusters contain a list of Locations as well as a nearby Well location, and the Cluster's centre (x,y) co-ordinates.

Data Structure

All the important data are stored in ArrayLists of the appropriate type, including a list of Tasks and Locations. This implementation is useful for checking if a discovered feature is new or needs updating, as well as identifying the best task to perform next.

Core Methods

In addition to the internal data structure, several helper methods are implemented to make processing the environment easier for the agent.

Code Architecture and Layout

The way the agent is coded is designed to facilitate manipulation of available information as smoothly as possible, which helps when the agent makes decisions. The architecture is split into Getter, Helper, and Core functions

Getter Methods

The following methods simply return an appropriate value:

- **getBestPump** – rates all discovered pumps with regard to nearbyClusters and stations within them.
- **DistanceTo** – returns the distance between the agent's location and target location, or between two given locations.
- **getNearestPump, getNearestStation** – both return the appropriate feature that is nearest to either the agent or a given location.
- **getBestTask, getBestTaskHOME -** both return the "best" task to perform next. The HOME version is used when the agent is in "Home" mode (see Extended Methods) and has ifferent biases for tasks.
- **getNearestWellLocation, getNearestWellLocationHOMETIME** – returns the nearest well location to a station,again with a version for "Home" mode.
- **getLocation, getStation, getStationID, getWell, getPump** – these use a Point to lookup in the Locations list the equivalent location and return the value requested.

Helper Functions

- **IncrementXY** – this method just calculates the agent's global co-ordinates based on the direction its moving. Implemented to de-clutter other parts of the code.
- **CheckIfInRange** – used to tell if a location is within fuel range of the agent. This prevents the agent attempting to reach a task and having to turn around and waste time and fuel.
- **UpdateClusterWells** – updates each Cluster's nearest Well

Core Functions

- **senseAndAct** – the default Tanker action function, returns an action based on the current State. Uses the other functions to decide which action to return.
- **ScanArea** – scans the entire view available to the agent, and records any stations and tasks seen. Updates previously seen locations. Updates Clusters with new stations if appropriate.

- **CustomMoveToward** – this function is a method to facilitate using the infallible move action every move, to improve reliability.
- **Initialise –** sets up variables.
- **CheckModes, UpdateTasks, CheckFuel, VerifyTask –** these functions perform several checks, to determine behaviour.
- **MoveToFuel, MoveAroundAction, LoadWasteAction, DisposeWasteAction –** handle the logic for return actions to move, and to load and unload waste.

Core Logic

Architecture Design

The agent's architecture is hybrid in type; it mainly implements deliberative architecture and logic, but has minor reactive components.

Reactive

The reactive parts include forcing the agent to return and refuel should the fuel drop too low, as well as avoiding tasks that are too low in waste remaining. The agent implements "Reactive with State", as it looks at fuel levels to make "decisions".

The decision is based purely on fuel remaining and distance to the fuel pump. If there is not enough to move and make it back afterwards, the agent is forced to return to the pump.

This reactive component occurs very rarely, notably in the first exploration phase of the agent, and sparsely later on, if the agent should become idle.

Deliberative

The deliberative parts scan the environment, decide whether a task is good enough, whether it is time to load or unload waste, as well as several other functions (see Extended Logic).

Action Selection

1. The agent starts by scanning the environment for data.
2. Afterwards it tries to find the best task nearby.
3. If one is found, it starts to move towards it. Otherwise it moves in a diagonal to find tasks.
4. When it arrives it loads the waste.
5. If there's room for more waste it goes searching again (step 2.).
6. Otherwise it checks if it can make it to the nearest well with current fuel.
7. If it can, it does, otherwise it goes to refuel.
8. After refuelling it finds a well to unload at.
9. Then it starts again.

The agent scans the environment every single step, not just at step 1. This is to allow for discovery when travelling, which can lead to finding closer Fuel Pumps and Wells, or even better Tasks.

Subsumption Architecture

The agent's architecture is similar to a subsumption model; certain actions have priority and override others. A good example of this in the agent is once a task has been found, if the task is out of range, the agent will move directly to the nearest pump to refuel, without considering other behaviours. Other behaviours are more subtly prioritised, with key overriding behaviours including fuel, early exploration, home mode, and cluster exploitation.

Extended Logic

In addition to the Core Logic of the agent, extra logic is used to determine actions.

Exploration

To avoid getting stuck in environments with limited Fuel Pumps, the agent starts off by moving in all four diagonals are far as it can. The aim of this is to find as many Fuel Pumps as possible, which will help prevent the agent becoming "tethered" to a single Pump, which would restrict the efficiency of the agent. If at the end of the initial search phase, only a single fuel pump is found, the agent attempts to locate a second by going in the four non-diagonals just in case it can find another pump.

This mode is also used should the agent need a well, but have none in range.

Emergent Exploration

At several points during the agent's run, it checks values against other possible values in minimisation tasks. These checks usually use the "<=" operator, instead of the "<" operator, as the "<" can exclude potential paths of added exploration, during the intended exploitation steps. This is because these checks are in order of when the agent discovered each feature, and so it is more likely to have explored near the beginning of the list, and less likely to have explored near the end of the list. Using the "<=" operator encourages the agent to stray from its current location, for instance in the determining of which Fuel Pump to use, and can result in the agent exploring more than if the "<" operator were used. This is useful for expanding the agent's potential working space.

Efficiency

The agent checks if it can make journeys before it attempts to make them. This saves a lot of time and fuel, preventing the agent from effectively spending double time on a journey. It also prevents looping a journey, never reaching it, before having to refuel.

The agent also ignores tasks with less than a threshold waste remaining. This is to prevent wasting time on small tasks that barely increase score.

Biases

In certain methods, such as getBestTask and getNearestWell, biases are used to try to improve efficiency.

Tasks are biased by how close they are to other stations, as well as how close they are to Pumps. This aims to allow the agent to smoothly “chain” load actions after each other. Tasks are also biased by how much waste is remaining in the station, with larger amounts being considered better than lower amounts.

Wells are biased by how close they are to pumps for the same reason.

If a task is in progress, the agent will consider the closest Fuel Pump to be that which is closest to the target station, if it is possible to reach it without running out of fuel.

Exploitation

The stations in the environment have a 0.1% chance each tick to acquire a task. The chances of a station not having acquired a task after a certain amount of steps go down as the step count goes up. The agent exploits this by implementing a “Home” mode. After ~3000 steps, the chances of a station not having a task lower to about 10%, which means that in theory, any stations seen near the beginning of the agent’s journeys are likely to have a task available.

Home Mode

In Home mode, the agent implements different biases, in an effort to steer itself toward its starting location. When without a task, the agent moves directly toward the start location. Home mode triggers every 4000 steps, to exploit the probabilities of task acquisition.

Home mode biases replace station biases with a similarly weighted bias toward the start location.

Cluster Implementation

After ~2500 steps, towards the end of the agent’s available time steps, the agent ranks all known Pumps by how many stations are nearby and how close they are. The “best” pump is then travelled to, and then normal behaviour resumes.

The original aim of this was to spend the later stages at a Pump with lots of available tasks, and more stations lead to more available tasks, however, testing showed that attempting this multiple times helped to improve scores.

This logic only happens if the best pump is within a single fuel tank distance. Chaining together a path to pumps was trialled, but it was found to be too inefficient. As such, it appeared that exploiting high density pumps if nearby improved scores more than without, but going too far away wasted too much time.

Rejected Logic

Several pieces of logic were explored, but were ultimately rejected for lowering efficiency too drastically. Whilst no longer implemented, they are documented as they were important to the development of the agent.

Quick Movement

Quick movement aimed to exploit the cheaper fuel cost of MoveToward actions. The agent would attempt to move using 1 fuel, and if it failed, would use 2 to move without failure, and this would repeat to hopefully chain several “lucky” moves together to be more efficient. The maths behind this are:

Normal Move: 100 moves at 2 fuel = 200 fuel.

Quick Move: 50% chance to use 3 fuel with Quick Move, so 50 moves cost 50 (success), and the other 50 cost 150 (1+2 per failed move) = 200.

Theoretically the average cost should be the same between the two types of move, but Quick Move would have a chance to chain together cheaper movements.

The maths unfortunately falls down due to time-step inefficiency. Quick Move can take 2 steps to move, whereas Normal movement only ever takes 1.

Ultimately, Quick Move failed to improve the average score of the agent, and in many cases reduced it substantially.

Waste Efficiency

A bias for waste amount over distance to task was implemented to cost tasks based on how much score per movement the agent would get. However, it seemed to either have no improvement, or a reduction in the points scored, so this was abandoned.

Optimisation

When optimising, changes were scored against previous versions on the same seed, ensuring the environment would be the same. This helped to ensure that higher scores were not a result of a particularly lucky environment.

When implementing several changes, they would sometimes result in more extreme results, with some seeds giving much higher than before, and others giving much lower than before. Changes that instead increased the average without introducing these extremes were ranked much better than changes that didn’t.

To prevent over-fitting the agent to certain set of seeds, a further set of seeds were used to ensure the average remained constant.

To record these changes and their effects, a thorough spreadsheet was used to determine which changes were the most impactful.

Version	SCORE		0	1	2
1.00		114387			
1.01		112524			
1.02	Better Exporation			111960	124550
1.03	Better Waste Pickup			112493	121578
1.04	Raised min waste to 200 from 100		133369	120219	130080
1.04b	Tests if well is in fuel range			115977	126542
1.04c	Lowered min waste to 150 from 200			110311	130328
1.05a	0.5 weight on next station		125254	44151	89059
1.05b	0.2 weight		132572	87572	123625
1.05c	0.1 weight		135010	87572	117684
1.06	MoveAction attempts the half roll once				
1.07	Check if in range for well		136997	115977	126542
1.08	BestTask finds <= instead of strict <		127639	131806	129546
1.09	Added Check for disposal time to avoid corner case of looping.		143024	100960	130907
1.09b	With <= instead of < for BestTask		135470	102885	143582
1.09c	As 1.09b but with <= for well location		143657	121426	130724

Figure 1: A small snippet of the Logs spreadsheet, included with the agent source code.

Something that happened frequently whilst optimising was encountering bugs that would drastically reduce performance in certain environment seeds. Solving these bugs then reduced the average performance across all seeds. This is likely due to the agent no longer being able to be “lucky”, instead bringing the different scores closer together for a lower, but more reliable score.

Hyper Parameters

Several parameters, including weights and biases, are used to help guide the agent toward an optimal action. These parameters are tuned slightly using trial and error, but would benefit from further tuning via AI learning methods such as a neural network.

The Parametric Curve

The agent’s scores are able to be plotted on a graph, however with multiple parameters, it is very difficult to do so manually. One issue is that by changing a single weight, or bias, the others also get effectively modified. For example, if there are 3 weights, all valued 1, and one weight is changed to be 2, it is equivalent to changing both of the other weights to 0.5.
This leads to an issue when tuning parameters, as weights may be tuned to a local maximum, dependent on the current ratios.

Reflection

The agent performs well on average, across different seeds for the environment. This can be put down to several reasons:

No limit on processing time: the agent spends a lot of real-world time deliberating over what to do, as it scans the viewable environment every time-step. In an environment where real-world time is limited, the agent would perform much worse, and would have to be optimised to account for this.

The ability to tell how much waste is remaining at a station: Knowing this allows the agent to better identify good tasks to go to, and not waste time on tasks which do not contribute much to the score.

Conclusion

Overall, the agent performs well. It currently averages a score of approximately 134 000. The range is reliably between 110 000 and 160 000 with outliers either side of these. The outliers are the product of the agent either being unable to efficiently exploit “unlucky” seeds, or the inverse for lower / higher scores.

Although improvement to the average score has slowed, I believe that with better tuned parameters, the agent could reach at least 140 000 average score, without implementing any additional logic. However, due to the amount of parameters, tuning would require AI learning methods to be achievable in a reasonable amount of time. For this reason, further tuning has not been pursued.