G53DIA COURSEWORK 2 REPORT

Alexander Steenson
psyajs@nottingham.ac.uk

4252754

Introduction

In this report, I will be explaining the different architectures considered for the project and the architecture I implemented based on the given task environment for multiple agents. Each algorithm used within the implemented architecture will be explained in detail.

The implementation section shows how each of the algorithms fit within the code and architecture. Evaluation of the performances will be shown and explained.

Finally, discussion about what I found particularly difficult, some problems I encountered and what I would change next time.

Relevant background material

Intelligent agents

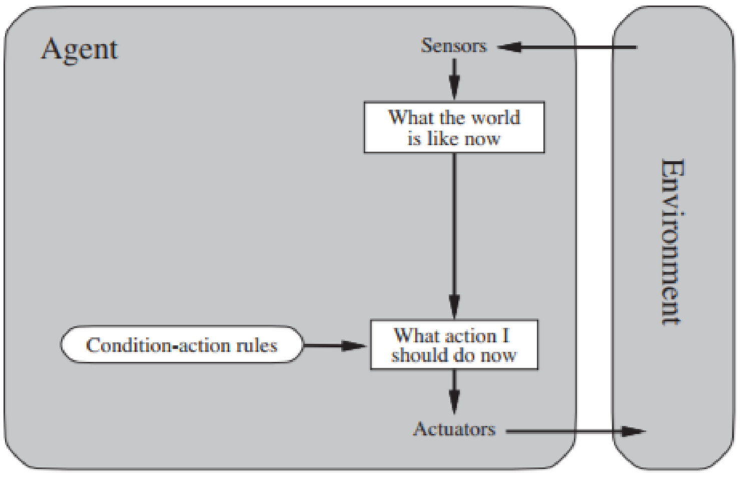
In this report I will be referring to an agent as per Russell and Norvig's definition, "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." (Russell and Norvig, n.d.). This definition is appropriate for this project as it represents the tanker accurately.
I will be referring to multi-agent systems as In an environment where multiple agents co-exist, a multi-agent system is an agent program of a single agent designed as a collection of autonomous sub-agents (Russell and Norvig, n.d.).

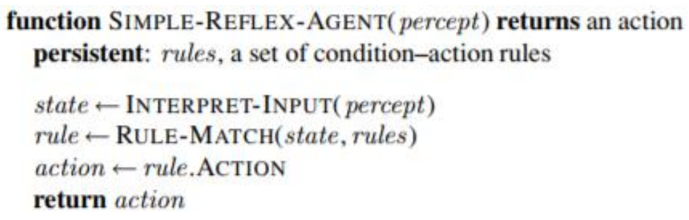
Architectures

There are a few different types of intelligent agent designs each one suited to an environment. The simplest a reflex agent. This agent selects actions based on its current location and sensor readings, ignoring what the agents have previously done and not making and future calculations. These agents occur more frequently in more complex environments that are unpredictable and hard to model. Purely reflexive architectures can cause problems with multiple agent as they have no way of communicating and both agents could try completing the same task.

Reflex agents react based on condition-action rules. These rules outline what the agent should do given a certain condition. For example, if an automated car driving detects a breaking car in front the car should break. Here the rule is 'if car-in-front-is-breaking then initiate-breaking'. All conditions the agent can encounter must then be accounted for otherwise the agent could run into a situation where it doesn't know what to do.

The figures below show a diagram and the pseudocode for a simple reflex agent. (Russell and Norvig, n.d.)





```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Model based reflex agents keep track and constantly update an internal state of the environment is can observe. Using the past observations, current sensor readings, and condition-action rules the agent can determine what action to take. This model is particularly good in an environment that isn't completely observable.

Software design

Environment

The environment can be characterised as partially observable, non-deterministic, sequential, discrete and dynamic. Each agent is only able to see a certain part of the task environment, a 40 by 40 square. Each action an agent performs effects future decisions, for example, if an agent decides to take waste from a station, the next action for any agent wouldn't be to visit that station. The environment is discrete as there are a finite number of states within the fixed grid. The location of the wells, stations, and fuel pumps do not move but tasks are generated randomly throughout the runtime and new stations, wells, and fuel pumps can be discovered.

Architecture Design

Hybrid Design

The design of the architecture must be suited to the task environment to achieve optimal performance. As the environment is partially observable the agent will need to keep a model of the environment to keep track of any states. The environment is also dynamic and new tasks can randomly be generated, and new sections of the map can be discovered. The agent, therefore, needs to take into consideration that there's a better solution available and to abandon its current plan. The agents will need to be able to communicate so they can know what each other's plans are so no two agents attempt the same plan.

The architecture design implemented is a hybrid design. This architecture allows the agents to maintain and update a clear model of the environment, calculate a plan of action and allow the agents to communicate. The reactive layer can focus on the urgency of refuelling and exploring the environment when no tasks are available, whilst the planning layer can calculate the optimal route to reach its goal ensuring no other agent is carrying out the same task.

Reactive Layer

The reactive layer responds to the current state of the agent and doesn't perform any complex calculations. Before the agent has explored any of the environment the planning layer has no goal to work towards and therefore cannot return a suitable course of action. The reactive layer then executes one of its most important behaviours, foraging. Foraging is crucial as the environment is only partially observable, because of this the agent doesn't have a complete model of the environment to generate a plan. The behaviour then makes the agent explore as much as possible, building up a more detailed model of the environment, maximising the number of wells, stations and fuel pumps it can see.

Whilst wondering the reactive layer also monitors the fuel level of the agent. This is essential as fuel is compulsory to move within the environment. If the fuel level gets near the distance to the closest fuel pump it will refuel.

Model Layer

Each agent keeps an internal and ongoing representation of the environment, keeping track of all stations, wells and fuel pumps along with their position. The agent can observe the environment around it, using that knowledge and the agents' position, the model can calculate the object's position. Each action the agent makes the model layer tracks to keep an accurate location of where the agent is. For example, if the agent moves west the model layer will decrement the agent’s x axis position by one. In a non-deterministic environment, it is critical to keep track of the agents position accurately, otherwise, the planning won’t be optimal, and the agent will get lost and not perform as well as it should. The environment is non-deterministic as an action could fail, it is therefore critical to keep track of if an action was successful or not.

Deliberative Layer

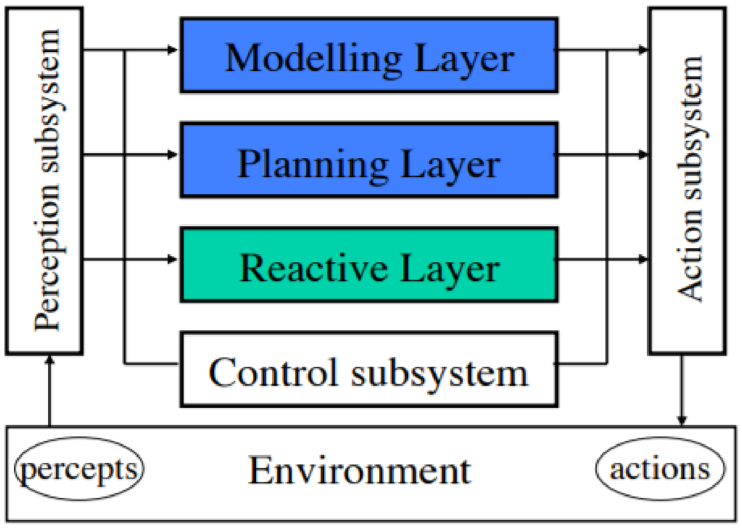
The deliberative layer is key to an optimal solution. From the model layer, the agent has a complete description of what the agent has observed so far, a set of states and a set of actions it can perform, a set of operators. Using this knowledge, the deliberative layer can calculate a goal and search the set space for a course of action on how to achieve the goal.

There’s almost an infinite number of possible plans an agent can follow, therefore the best plan must be calculated for each agent.
A plan starts with the goal state and each pre-condition produces a sub goal. These sub-goals are backwards chained until all preconditions are met to produce a list of actions. The pre-conditions for an action are requirements that the agent must fulfil to be able to complete the action. For example, a pre-condition for going to a fuel pump to refuel could be, does the agent have enough fuel to make it to the pump and the agent does not have the maximum amount of fuel.
Each station with an available task is evaluated. Each task will be backwards chained to produce a list of actions. These plans will be compared with each other to calculate the best plan the agent can take.

Control system

Both the reactive layer and planning layer are running concurrently, independently and producing outputs. The agent must evaluate and decide which of these outputs it should execute. This is the roll of the control system, it uses a set of control rules to censor actions generated by the control layers. For example, a control rule used it to prevent the reactive layer foraging if the deliberative layer has found a suitable plan.

The implemented hierarchical control works like a Touring Machine, as a single layer controls the percepts and decides an output. The architecture works in a decentralised control manner without interfering with each other producing an output.



Algorithms implemented

Foraging

Multiple agents exploring in a straight line away from each other can collectively explore a larger area than a single tanker exploring in a fan pattern.
This makes the foraging algorithm very simple to implement. Each agent generates a random direction and explores that direction until they need to refuel again, when another random direction is generated.

Backwards search and plan generation

Backwards search isn't effective without a good heuristic function (Russell and Norvig, n.d.). Each station with an available task, that another agent isn't currently completing is evaluated as a potential goal. The distance between the agent and the task measured and is used as h(n), the heuristic estimated cost from vertex n to the goal. The required preconditions the agent must satisfy to complete the task such as, refuelling and dumping waste, is calculated. If the agent has 25 fuel remaining and the task is 30 timesteps away the agent must refuel. The best fuel pump must then be selected using an A* search. Every fuel pump the agent has seen is classified as a node. Each node is the given a score, f(n) = g(n) + h(n), where g(n) is the distance between the agent and the fuel pump and h(n) is the distance calculated above. All the nodes are added into a priority queue sorting them from lowest value to the highest. The head of the queue is then expanded, and the search continues backwards searching.

The overall distance needed to travel to complete the task is recorded for each task. Each task is then given a score based on the distance the agent needs to travel to complete the task and the size of the task, distanceTravelled / taskSize. This gets how many points the agent can acquire per timestep. The task with the highest ratio is selected for the agent to complete.

Updating Locations

To monitor the environment and keep an internal representation of each object, the model layer updates locations by adding newly discovered locations to array lists. There is a dedicated array list for each object the agent can encounter. The observable environment is scanned for each time step, every object it can see the algorithm will determine what type of object is it. The object will be checked to see if it has already been added to its corresponding array list if not, it'll be appended to the end. For example, if the agent discovers a fuel pump, the fuel pump array list will be scanned to see if the fuel pump has already been added, if it hasn't, it'll be added.

For each time step, the stations will be polled to see if they have generated a new task. If a station has a task the task array list will be checked to see if it already contains the task. If it doesn't the task will be added.

Plan Coordination

This class is essential to good planning with multiple agents. This class allows the agents to communicate and deliberate knowing what the other agents have planned so they can do something different.
The class has two functions, addPlan and checkTaskIsSafe. The class also holds an array list of plan models. This stores the plan and tanker ID for each agent.
When an agent is deliberating what plan it should follow before it calculates a score for a task it checks if another agent is currently working towards it, it can do this with checkTaskIsSafe. The task and ID if the tanker is passed in and the arrayList is checked, if another tanker is currently completing the task the function will return false, otherwise true.

Once the agent has calculated a valid plan, other agents need to know not to work towards the same task. addPlan adds the plan of the agent to the arrayList. If the agent has an old plan in the list it will be overwritten.

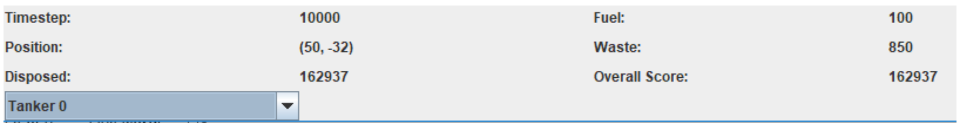
Evaluation, discussion and conclusion

Below is a table of the average score over 10 runs for 1, 2, 3 and 5 tankers.
The results get progressively worse the more tankers are included. This makes sense because although there are more tankers to take waste, one or two tankers end up taking the biggest and best tasks leaving the smaller ones for the other tankers causing them to not do as well. To overcome this problem, I could focus on separating the tankers further away from each other.
I have also observed that a run does particularly poorly when there is a lack of fuel pumps around as the tanker can not explore as far and not pick up as many tasks.

Number of tankers	Average score over 10 runs	Highest Score	Lowest Score
1	135269	162937	96131
2	122949	146146	84661
3	111886	134229	79365
5	102843	115714	85788

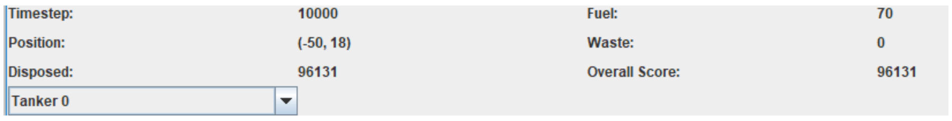
Screenshots of the highest and lowest scores:

1 Tanker



Timestep:	10000	Fuel:	100
Position:	(50, -32)	Waste:	850
Disposed:	162937	Overall Score:	162937

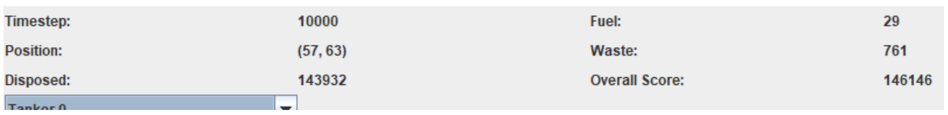
Tanker 0



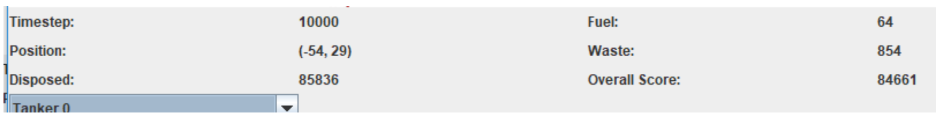
Timestep:	10000	Fuel:	70
Position:	(-50, 18)	Waste:	0
Disposed:	96131	Overall Score:	96131

Tanker 0

2 Tankers

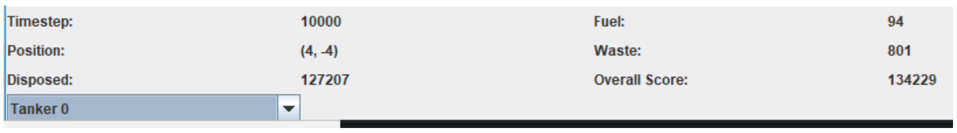


Timestep:	10000	Fuel:	29
Position:	(57, 63)	Waste:	761
Disposed:	143932	Overall Score:	146146

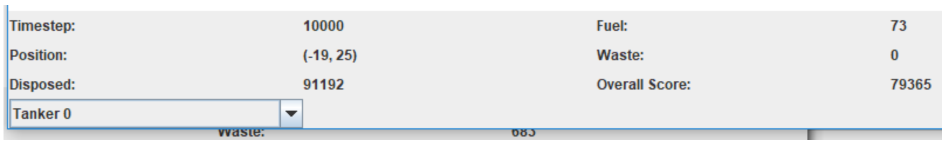


Timestep:	10000	Fuel:	64
Position:	(-54, 29)	Waste:	854
Disposed:	85836	Overall Score:	84661

3 Tankers

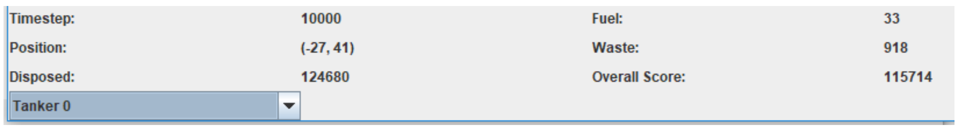


Timestep:	10000	Fuel:	94
Position:	(4, -4)	Waste:	801
Disposed:	127207	Overall Score:	134229
Tanker 0			

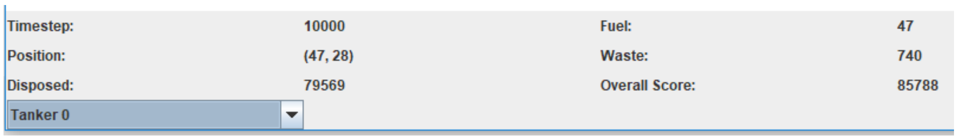


Timestep:	10000	Fuel:	73
Position:	(-19, 25)	Waste:	0
Disposed:	91192	Overall Score:	79365
Tanker 0			
	Waste:	683	

5 Tankers



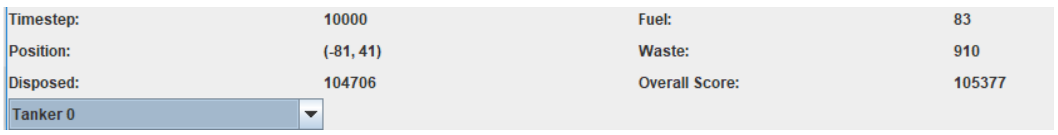
Timestep:	10000	Fuel:	33
Position:	(-27, 41)	Waste:	918
Disposed:	124680	Overall Score:	115714
Tanker 0			



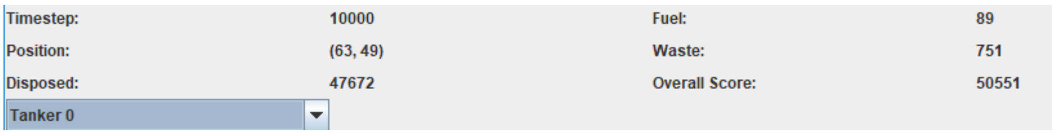
Timestep:	10000	Fuel:	47
Position:	(47, 28)	Waste:	740
Disposed:	79569	Overall Score:	85788
Tanker 0			

This performance will be significantly better than other simpler agents such as a reactive agent or a model based reactive agent.
For comparison I created a model based reactive agent, the agents didn't perform any complex calculations to generate a plan. It simply visited the nearest task, refuelled when needed at the closest fuel pump and dumped waste when needed. The agents didn't communicate so at times multiple agents were trying to visit the same task and resulted in a race. The average score for the model based reactive architecture for 3 tankers over 10 runs was 86728 with the highest score being 105377 and the lowest being 50551. This architecture clearly didn't perform as well as the hybrid architecture.
You can find the code for the model based reactive agent in the class MainTankerReactive, in the submitted code.

Screenshots of the highest and lowest scores for the model based reactive architecture:



Timestep:	10000	Fuel:	83
Position:	(-81, 41)	Waste:	910
Disposed:	104706	Overall Score:	105377
Tanker 0			



Timestep:	10000	Fuel:	89
Position:	(63, 49)	Waste:	751
Disposed:	47672	Overall Score:	50551
Tanker 0			

The implemented architecture is suited to the environment as it is dynamic and partially observable. The architecture benefits from modelling the environment, reacting to unseen states and planning an optimal solution.
If the task environment was completely observable, then the reactive and model layer wouldn’t be necessary as the agent could perfectly plan the optimal route every time.

I ran into a couple of problems when designing the agent, one of the problems was the fallible action. This causes my initial plan to potentially run out of fuel. I then had to consider the possibility of a fallible action. I currently allow for four fallible actions when moving. This means there is a very slim chance of five fallible actions occurring when the agent is moving and running out of fuel.

References

Russell, S. and Norvig, P. (n.d.). Artificial intelligence.

Brian Logan. Lecture slides, University of Nottingham.