G53DIA
COURSEWORK 1 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. 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

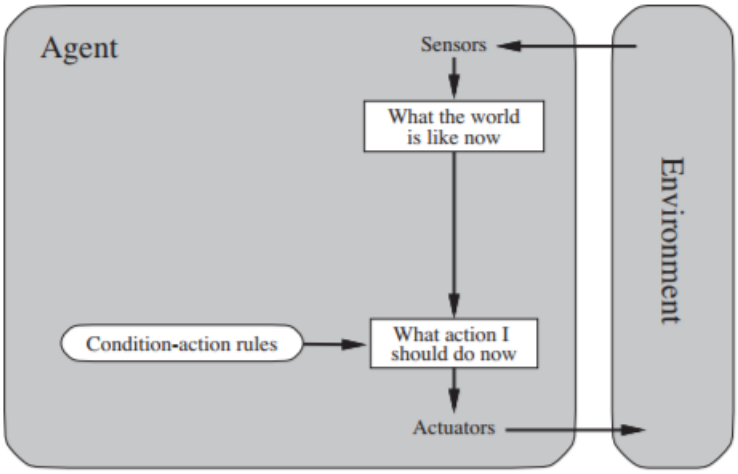
In this report I will be referring to an agent as per Russell and Norvig's definition, "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." (Russell and Norvig, n.d.). This definition is appropriate for this project as it represents the tanker accurately.

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.

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'.

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
```

While simple reflex agents are easy to implement and fast they do have many problems, the limited intelligence can be a serious problem if every condition isn't accounted for. The agent in the self-driving car would run into problems if the environment isn't fully observable. Other models are created to overcome this issue, such as the model – based reflex agent.

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.

Software design

Environment

The task environment generates 'tasks' the agent must complete to generate a score. This score can be considered as a performance measure and therefore you can compare different solutions. An example of a plan for an agent could be, move towards a task without running out of fuel, collect the waste from the station ensuring the tanker can hold all the waste.

The environment can be characterised as partially observable, deterministic, sequential, discrete and dynamic. The agent is only able to see a certain part of the task environment, a 40 by 40 square. Each action the agent performs effects future decisions, for example, if the

agent decides to refuel, the next action wouldn't be to refuel as the agent is holding the maximum amount of fuel. 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 now 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 architecture design implemented is a hybrid design. This architecture allows the agent to maintain and update a clear model of the environment, as well as a current plan. By allowing a plan and urgent responses from the reactive layer without any complex reasoning, the agent can produce an optimal solution. The reactive layer can focus on the urgency of refuelling and exploring the environment when no tasks are available, whilst the planning layer is calculating the optimal route to reach its goal.

The hybrid architecture is split into four sections, the reactive layer, the model layer, the planning layer and the control system.

Reactive Layer

The reactive layer responds to the current state of the agent and doesn't perform any complex calculations. Because of this the goals are of lower level but is critical to good planning. Before the agent has explored any of the environment the planning 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.

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

The model layer keeps an 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 deterministic environment this is critical, otherwise, the planning won't be optimal, and the agent will get lost and not perform as well as it should. Each completed task needs to be updated and marked as complete to prevent the planning layer visiting stations without any tasks.

The model layer is also used to search for objects within the environment. If the reactive layer returns that the tanker needs to refuel the model layer will find the closest fuel pump for the agent to visit.

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. Any action leading from one state to another is a path in the state space, therefore there are multiple different ways a goal can be achieved and for an optimal solution, the best must be selected. Each path is given a cost function g(n) and is used to assess the quality of a path.

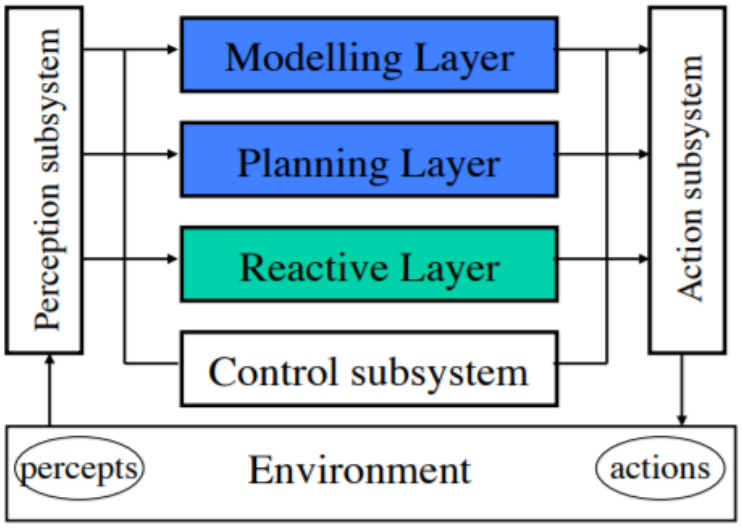
A plan starts with the goal state and each pre-condition produces a sub goal. These sub-goals are backwards chained until all preconditions are met to produce a list of actions. The pre-conditions for an action are requirements that the agent must fulfil to be able to complete the action. For example, 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. This pre-condition ensures the agent doesn't run out of fuel and can continue completing other generated goals in the task environment, and stops the agent trying to refuel just after its refuelled. The effect occurred for the same action would be, the agent is carrying the maximum amount of fuel. The search starts at the goal state at checks the pre-conditions. If the agents' current state meets the pre-conditions the action will be carried out to complete the goal. If the agent doesn't meet the conditions to complete the goal, for example, the agent may want to collect waste from a station but can't carry the total amount, the search will find an action which effect satisfies the pre-

condition. The pre-condition is then checked for that action and continues until all pre-conditions have been satisfied. The pre-conditions can be written with a conjunction of literals forming a description of a set of states. For example, the goal hasEnoughFuel ∧ hasEnoughWasteStorage describes those states where the agent has enough fuel to get to the goal and enough space available to take the waste.

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

The foraging algorithm works by allowing the agent to explore the largest area as possible in the least amount of time steps. It does this by moving in diamonds in each direction from the fuel pump. It can move a distance far enough away from the starting fuel pump that if it finds a new fuel pump, it will go there to refuel, but if it doesn't it'll make it back to the starting fuel pump. This allows the agent to search a larger distance rather than just around

the starting location. The agent moves in diamonds to minimize the number of cells it has already seen and maximizing the discovery area. With foraging techniques such as spiralling, once the agent has done a lap, half of its observable view it has already discovered.

Backwards search and plan generation

Backwards search isn't effective without a good heuristic function (Russell and Norvig, n.d.). I have implemented a greedy heuristic for my planning. While greedy search isn't optimal, unlike A*, it runs in a quicker time and is a good heuristic for this problem. Agent architectures are often scrutinized for the time needed to calculate an optimal plan as they run in real time. Given the task environment, however, timesteps aren't used until an action is performed giving the agent an infinite amount of time to calculate an optimal plan.

Backwards search has a lower branching factor than forward searching for an unknown goal as it can prune more possible states when searching. This increases the runtime and space complexity of the search.

The greedy heuristic calculates the largest amount of points the agent can acquire per timestep, this task then becomes the goal state for the backwards search.

The pre-condition is determined and the state that satisfies the precondition becomes the new current state. The agent simulates moving to the location by tracking how much fuel it would have after visiting that location and how much waste. The search recursively then finds the next location for the agent to visit. Once all the pre-conditions for each action has been satisfied the positions of the objects are added to a linked list.

When the agent needs to drop waste off at a well a suitable well to visit needs to be calculated. To do this the agent uses a shortest line algorithm to find the closest well from its current position to the goal. Each well is given a score based on the distance from the agent to the well, then from the well to the goal. The well that returns the smallest score is selected. This gives the optimal path for the agent to take when dropping off waste.

Plan Comparison

Once a plan is generated it is compared to the current plan. Each plan is given a score based on how many points the goal gets per distance moved, much like selecting a goal state. The plan to generate the highest score is taken. Often the plans will be equal and when the current plan is completed the newly generated plan is automatically used.
Comparing plans is important and an agent needs to be decisive in the actions it is taking.

Updating Locations

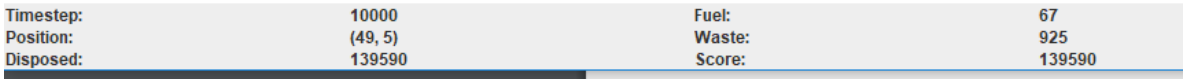
To monitor the environment and keep an internal representation of each object, the model layer updates locations by adding newly discovered locations to array lists. There is a dedicated array list for each object the agent can encounter. The observable environment is scanned for each time step, every 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.

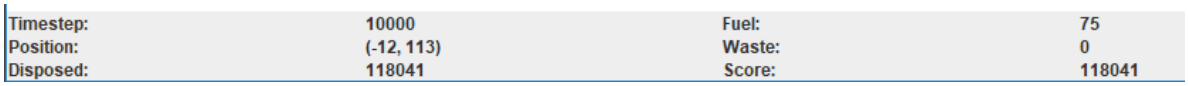
For each time step, the stations will be polled to see if they have generated a new task. If a station has a task the task array list will be checked to see if it already contains the task. If it doesn't the task will be added.

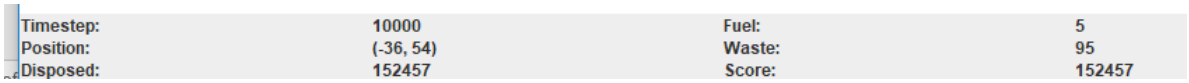
Evaluation, discussion and conclusion

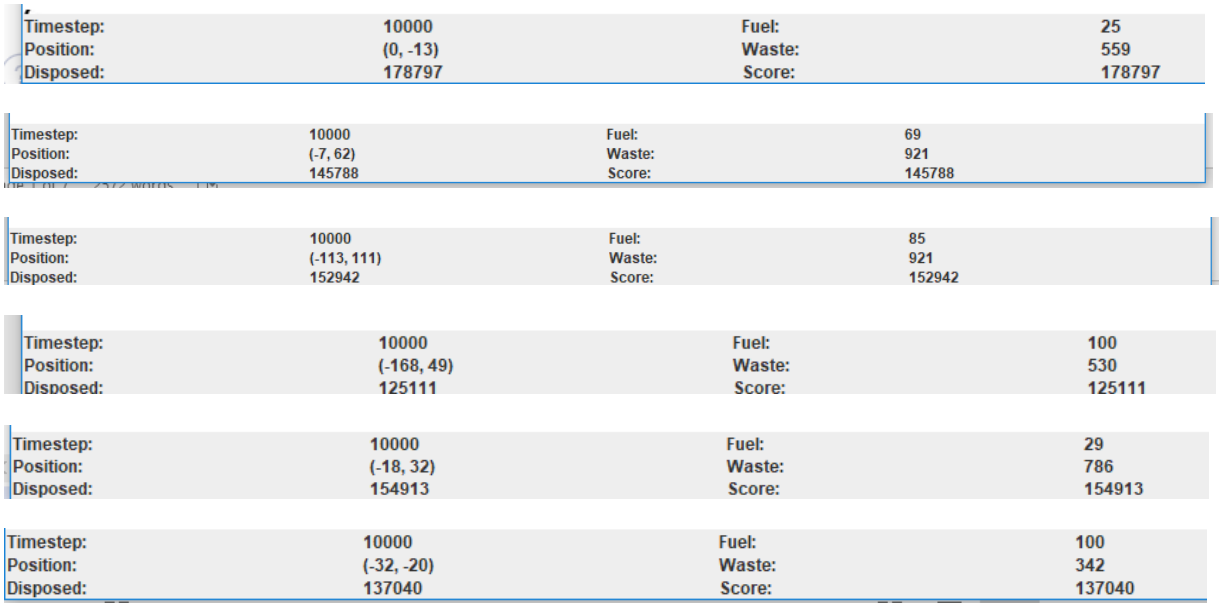
On average the agent scored 144823 and completed an average of 259 tasks. Below is a table displaying each score and number of tasks visited by the agent over ten runs. Below are screenshots of all the completed runs.

Run	Score	Number of tasks completed
1	139590	247
2	118041	215
3	142560	264
4	152457	271
5	178797	302
6	145788	263
7	153942	267
8	125111	241
9	154913	279
10	137040	245
Average	144823	259









This performance will be significantly better than other simpler agents such as a reactive agent or a model based reactive agent. A purely reactive agent would go for the closest task every time not taking into consideration if it can carry the required amount or if it is the best task to visit. It has a high chance of running out of fuel as it could wander away from a fuel pump and not have enough fuel to get back. A model based reactive agent wouldn't run out of fuel but also wouldn't plan the best route to take.

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 two fallible actions when moving. This means there is a very slim chance of three fallible actions occurring when the agent is moving and running out of fuel.

Another problem I ran into is that greedy search isn't complete, this can cause looping in the planning and potentially causing a stack overflow error. A try-catch statement fixed this issue.

If I had more time I would have simulated the actions in the backtracking as schemas instead of scripting them in. As I wrote this specifically to solve this problem and not to re-use the code for other problems the current implementation is sufficient.

References

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

Brian Logan. Lecture slides, University of Nottingham.