ARTIFICIAL INTELLIGENCE (H) 2017/2018 - ASSESED EXERCISE

Cameron Harper (2136555h) and Jamie Sweeney (2137284s)

Team 30

ABSTRACT

Artificial Intelligence is a very broad subject and intelligent agents can be constructed in many different ways, with extremely varied levels of intelligence. For this reason the analysis and design of an agent are crucial to it's resulting performance, and thus future progressions in the field. In this exercise, we will consider three types of agents that we will analyse, design, and implement, upon evaluation of the results we hope to draw a suitable conclusion for the 3 different types of agents and how their designs effect the performance of the agents. The main method we are evaluating is that of Q-learning - a reinforcement learning algorithm, and trying to prove that an agent can learn from its own experience performing a task. For a basis of comparison, we also implement a random agent to provide a baseline, and an implementation of the A* path finding algorithm has been provided to show the "ideal" result of the problem.

Index Terms— Agent, Malmo, Random Agent, Simple Agent, Q-learning, A*

1. INTRODUCTION

Minecraft is a popular video game which has many different applications, including in the classroom and for scientific analysis. Microsoft purchased Minecraft in 2014[1], and since then has been developing Malmo - providing a platform for creating and running artifically intelligent agents which can be introduced as players of Minecraft[2]. These agents can be taught to perform actions programmatically.

Agents can receive information about their environment through sets of sensors. These sensor inputs can be used to influence the behavior of these agents and through this - in theory - improve their effectiveness of solving the given problem.

Our problem is to create three different agents which would each attempt to solve a maze. The first agent would be simply random - with no access to sensors, which could be used to provide a baseline/random set of solutions to the problem. The second agent will be provided with a full "view" of the environment using which it can calculate the optimal route.

The third, or realistic agent, is meant to intelligently solve the maze. Intelligently, in this sense, means in a way similar to that of a person trying to solve the maze. Given repeated attempts at the same problem, the realistic agent should "learn", and as such achieve better results with each iteration of it running.

By examining both the results of the first agent along with the second, we can compare these to the realistic agent and then show the effectiveness of our solution. We will be using a small sized maze for each of our experiments, with the different agent types running across the same size of maze.

2. ANALYSIS

In order to allow direct comparison, the performance measure must be kept constant for each agent. Positive rewards are given for completing the maze or collecting intermediate rewards, and negative rewards are given for taking actions or for running out of time. This is sufficient for evaluating the agent's ability to navigate the maze efficiently and for finding rewarding paths through the maze.

In addition the available actions that an agent can take will remain the same, with each agent being allowed discrete movements of magnitude 1 in the direction of North, East, South or West.

The environment will be the same for all agents, however each will have different levels of observability and prior knowledge of the state-space. To keep comparisons simple, we will consider an environment that is deterministic, discrete and single-agent. The environment will also be semi-static, although the maze does not change over time, timing does matter in terms of performance.

2.1. Random agent

The random agent's purpose in this project, is to provide a baseline for the simple and realistic agents to beat in order to demonstrate a performance achieved by something other than random chance. For this reason the agents environment must be completely unobservable and unknown i.e no knowledge prior to execution, this includes knowledge on the performance measure. The agent will have no sensors as they are not necessary, given that the environment is unobservable.

2.2. Simple agent

The simple agent will serve to give a us best performance for which to compare the realistic agent to. The highest performance measure for the agent should be the maximum possible result for the specific environment and performance measure. We expect that the final reward achieved by the simple agent will remain constant for each iteration, however we must also be aware that it may change slightly due to unforeseen circumstances or implementation errors, to reduce such a effect we will consider the highest reward for the agent evaluation. To ensure that the actions taken are the best possible, we must allow the environment be fully observable and known to the agent, including all possible rewards or penalties. The agent will therefore have a complete map of the maze with rewards and the exit. The agents sensors will consist of an 'oracle' sensor that can be queried for complete state-space information.

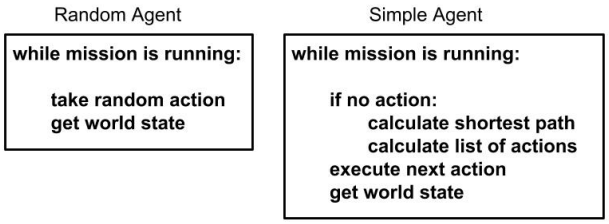

Fig. 1. Pseudo-code of the random and simple agent designs

2.3. Realistic agent

The realistic agent will demonstrate a performance in the case a scenario that is more comparative to a real life problem, the agent will have no prior knowledge of the state-space or performance rewards. However it will be able to observe the environment and it traverses it and retain this information for future use. In order of the agent to be able to collect information on the environment, it must be given appropriate sensors. It will have a 'oracle' sensor similar to the simple agent, however with a radius of one around the agent. This will allow the agent to explore and observe its surroundings until it has a complete map of the state-space. To construct a more realistic scenario, we will consider actions that are subject to noise, resulting in an unwanted action being taken 10%. Ideally, each time our agent progresses, the score would increase every time, however in order for the agent to properly explore the state-space sometimes it must take paths that are less than optimal so that it can discover all available paths and ensure that it is not missing a potentially optimal solution.

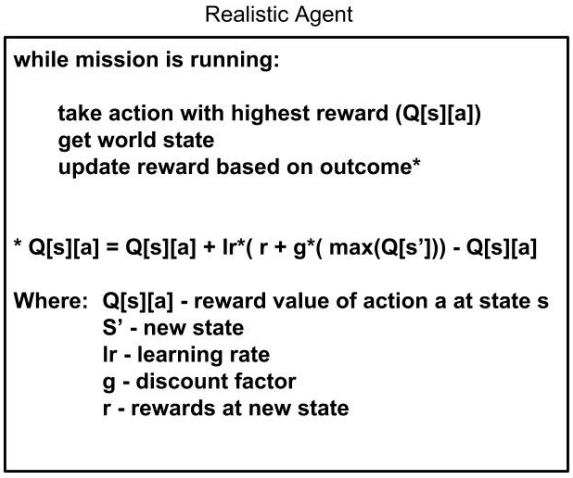

Fig. 2. Pseudo-code of the realistic agent design

3. DESIGN

3.1. Random agent

The design of the random agent should be as simple as possible, as it requires no sensory input or knowledge (either built-in or learned) and the actions taken should be completely random. For this reason we can just let our agent generate a random positive integer to represent the action number, out of the total number of actions. This process can be repeated until the agent runs out of time or manages to complete the maze.

3.2. Simple agent

In order for the simple agent to find an optimal solution, it must use it's state-space knowledege in combination with an appropriate shortest path algorithm. As have have a complete map of the state space, we can construct a graph of the state space with blocks being represented as nodes, and available movements at those blocks as edges between those nodes. With this graph we can use one of the many informed search algorithms that are complete in finite space.

We decided to use A-star (A*) as it has been shown consider a small amount of nodes compared to similar search and hill-climbing strategies[3]. Once a shortest path has been computed using A*, all that is left for the agent is converting this path into a sequence of actions and executing them one by one until the exit is reached.

3.3. Realistic agent

Designing the realistic agent requires more thought, as the sensory information observed by the agent is very limited. As the agent an only see within a radius of one block around itself, to effectively solve the maze and perform well, it must maintain some form of knowledge of the environment that can be constructed as it continues to learn the required features of the state-space.

The main required properties of this agent are the ability to explore the environment and maintain a internal model of the environment and the ability to use this model to compute which action should be taken at the agents state. These requirements most closely match those of an online agent that undergoes a 'learning phase' where it will explore and learn the state-space. As this knowledge becomes more complete, the agent will narrow down the paths it takes until an optimal solution is found and taken every iteration.

For our implementation of such an agent, we will use Q learning, which is a reinforcement learning technique that continuously builds a model of state- action possibilities along with a computed reward value for each action[4].. This will allow out agent to observe the rewards received for each action at each state and eventually compute an optimal solution based on previous iterations of the maze.

4. IMPLEMENTATION

The designs above were implemented using python 3.6.3 and the Malmo platform (0.31.0), built on top of Minecraft.

4.1. Random agent

The random agent was a very simple implementation, consisting of a loop that iterates until the mission finishes. While inside the loop, the agent will pick one of the available actions by random choice and execute that action, taking 200ms to complete.

Agent Helper

```
get list of blocks from oracle sensor

convert to 2D array

for each point in array:
        teleport to point
        add rewards to point data structure

pass data structure to simple agent
```

Fig. 3. Pseudo-code for agent helper.

4.2. Simple agent

For the simple agent, an oracle sensor was implemented by using a helper agent, which would build a representation of the state space as an undirected graph, using the python networkx library which contains classes used for graphs.

The helper agent also found the locations of the intermediate rewards, by teleporting to each location and checking for rewards. However due to the game mechanics of Minecraft, rewards will be collecting from nearby blocks, making it possible for the agent to collect 9 rewards in one location and have no way of knowing which location they came from. For this reason we were unable to effectively use the reward locations in the path generation algorithm. One possible way around this would be reloading the mission for every single location and cross-reference all the results to get a complete map, however this would require a large amount of modification to our agent helper, and we decided against it.

When the agent is initially run, it already has the complete map of the state-space, so it can compute a path straight away. We used the A* search algorithm that was provided by networkx with an initially empty heuristic function. The heuristic function could later be modified to incorporate the intermediate rewards. After a path is computed, the agent will then execute the actions sequentially until the mission finishes.

4.3. Realistic agent

Q learning operates by choosing an action for its current state based on a reward value which is computed each time it takes that action, this data is held in some central model that can be queried and updated while the agent runs. To implement this, we used a dictionary to hold key/value pairs, with the keys representing a state and its corresponding values representing the rewards value for each action. As dictionary keys must be unique, we can use the simple key of e.g "1_9" to be the key for the block at x=1 and z=9. The value for each of these dictionary elements can be an array of size 4, one element for each possible action.

When the agent is determining it's next move, it looks at the dictionary values for the current state and takes the action with the maximum rewards value. Upon taking this action the agent will then update the reward value for it using a combination of: the previous value, the reward received, the learning rate, and the discount factor.

The learning rate and the discount factor are key constants in determining the agents progress rate and how it 'weighs up' the different rewards received. The learning rate defines how new information is compared to old information and its value is crucial to agent performance, a very high learning will cause the agent to overwrite old information with new information immediately, whereas a rate that is too low will cause the agent to never learn or progress at all. After the process of trial and error we found that a value of 0.4 seemed to work best for us.

The discount factor determines how rewarding the agent considers future rewards to immediate rewards, a value of 0.95 was used in this case, as future rewards i.e the exit, are extremely important for getting a high performance measure.

5. EVALUATION

As we can expect, the random agent fares very poorly in the mission, yielding an average cumulative reward of -3115. We can attribute this to the fact that all the actions are completely random and the chances that a successful path will be generated are extremely slim. It follows that this agent is an effective implementation of a worst-case agent that can be used to evaluate the performance of the more intelligent agents.

The simple agent also performs exactly as expected, with it taking the same path every time for the identical maps. Although the agent does not take into account the intermediate rewards, this does not effect the final path computed, as any other path will result in a smaller reward. The agent managed to achieve a solution that gives a reward value of 1341, which we will assume is the optimal path, as a more rewarding path could not be found through human inspection.

We can see - figure 3 - that the realistic agent performs better than expected. Initially, this was true for most test runs - this is most likely due to the fact that the agent will explore a larger area during it's first iteration, and therefore has a higher chance of finding the exit accidentally. From this point on, the agent tends to perform worse for a few iterations until it begins to level off around the 4th or 5th iteration. Over the course of 10 iterations the agent will achieve a best solution yielding 1197 reward points, with an average reward of 333 points.

Figure 4 shows the realistic agent operating over 50 iterations, we can see that as the agent progresses further, the margin of error gets smaller and smaller. Note that then sharp drop in reward around iteration 30 was due to a stall in Malmo and can be discounted. We can imagine that this range of possible final rewards will decrease until the agent converges on

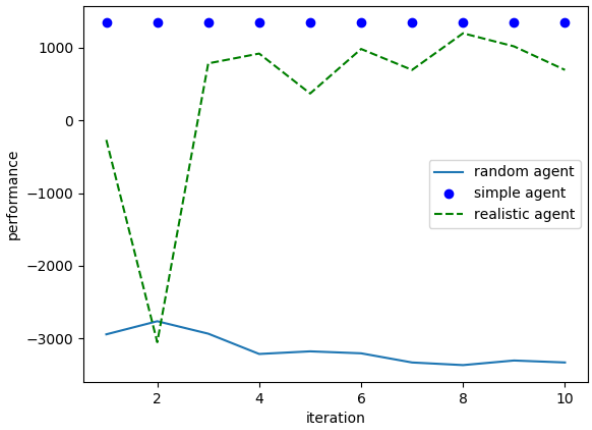

Fig. 4. All agents compared over a cycle of 10 runs

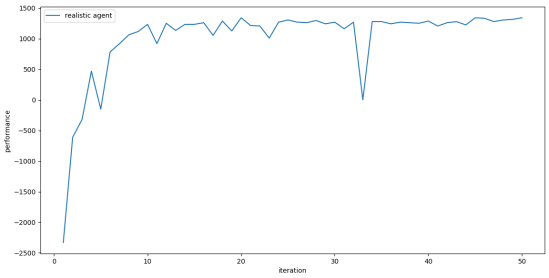

Fig. 5. The progression of the realistic agent over 50 iterations

a optimal path (except for cases where noisy actions result in penalties)

Agent	Max Reward	Avg Reward
Random	"-2762"	"-3115"
Simple	"1341"	"1341"
Realistic	"1197"	"333"

6. CONCLUSION

Following evaluation of the results, it is clear we have demonstrated a model of agent "learning" - in which a computer program has effectively taught itself information about its environment. While not reaching the same performance of the simple agent (as the simple agent was meant to model a perfect score), after a few iterations the realistic agent was able to produce results which much outperformed the random agent. Q-learning - specifically the Markov Decision Process theory - has proven in this case to be an efficient and effective way of solving a simple problem such as a maze (without access to a full picture of the problem, e.g. our Simple Agent) and instead using limited sensors combined with a set of learned information.

7. REFERENCES

[1] "Microsoft agrees to aquire creator of minecraft," Wall Street Journal, 2014.

[2] Tim Hutton David Bignell Matthew Johnson, Katja Hofmann, "The malmo platform for artificial intelligence experimentation," Microsoft.

[3] Mr. Girish P Potdar and Dr.R C Thool, "Comparison of various heuristic search techniques for finding shortest path," Microsoft.

[4] Leemon Bair et al., "Residual algorithms: Reinforcement learning with function approximation," ICML, 1995, pp. 30–37.

8. APPENDIX A

The above analysis and design were achieved using a method similar to paired programming, with a focus on discussion and a determining advantages/ disadvantages for each possibility. Once both team members were happy with the design, we split the implementation section up into small task, usually at the range of an hour per task to keep changes at a minimum. After most of the implementation was done, we returned to our pair programing model as it seemed to reduce oversights and allow us to communicate more effectively. I highly recommend this system and will continue to use it for future projects as I believe it was crucial in maintaining 'on the same page' and greatly improved our ability to compare information and plan ahead.