ARTIFICIAL INTELLIGENCE 4 2017/2018 – ASSESSED EXCERCISE

Marko Caklovic (2349514C), Igor Drozhilkin (2353454D), Apoorva Tamaskar (2349061T)

University of Glasgow
Team 17

ABSTRACT

The purpose of this assessment was to create three different agents which would navigate various environments in the popular videogame "Minecraft". Each agent had different constraints on its behavior, and different access to sensor information about the environment. In the end, we describe the implementation of each agent, why we made the choices we did, and how the performance of the agent compares to one another.

Index Terms— Malmo, A*, Q-Learning, AI Assessment

1. INTRODUCTION

The maze solver problem is a simplified version of navigation problem. Developments in efficient maze solving have helped develop solutions for other navigation problems. Solutions for the maze solver problem can be used as a form of navigation to achieve the shortest path for:

1) Emergency medical response
2) Disaster response operations
3) To meet urgent requirements constrained by physical terrain

"Maze solving problem involves determining the path of a mobile robot from its initial position to its destination while traversing through environment consisting of obstacles. In addition, the robot must follow the best possible path among various possible paths present in the maze. Applications of such autonomous vehicles range from simple tasks like robots employed in industries to carry goods through factories, office buildings and other workspaces to dangerous or difficult to reach areas like bomb sniffing, finding humans in wreckage, etc." [1]

In this assessment, our team was tasked with creating three different agents to navigate randomly generated mazes in the Minecraft. We used the Malmo library to write code which interfaces with Minecraft. The agents we had to write were the *Random* Agent, the *Simple* Agent, and the *Realistic* Agent.

The *Random* Agent had no information about its environment, had no way to assess its own performance, and in general was expected to perform randomly. For this agent, we used the code given to use in the assignment template without improving it. Consequently, we will spend very little time in this paper talking about the *Random* Agent.

The *Simple* Agent had almost perfect information about the environment. It always knew the full layout of the maze it was expected to solve, including the start and end points. The only thing missing from the information available to the *Simple* Agent was the location of the intermediate rewards. In the end, we used the A* algorithm to analyze the information about the environment, and compute an optimal solution to each maze.

The *Realistic* Agent had very little information about the environment, but was still expected to perform in a rational manner. The *Realistic* Agent only knew its position, and the contents of the environment around it in a radius of "1". Notably, the *Realistic* Agent could be run multiple times on the same maze, in order to learn from each attempt to solve the maze. We used the Q-Learning algorithm to train the agent to solve the maze in a close-to-optimal manner in very little time.

2. ANALYSIS

We defined the execution of an agent to be successful if the agent reached the goal in 60 seconds. We had options to keep the agents discrete or continuous or absolute, we decided to keep the agents discrete for the simplicity of the implementation. As per the exercise specifications we have kept the time required to carry out an action as 0.2 seconds.

2.1. Assumptions and restrictions on the agent

2.1.1. The Random Agent

The biggest restriction on the *Random* Agent is that it has no access to any sensors. The action taken by the agent is selected uniformly at random (forward / backward / left / right). To classify the agent and the environment, we see that at any given state, the agent takes actions uniformly at

random. Because of this, we can conclude that the agent is stochastic and episodic. The *Random* Agent has no access to sensors and hence the environment is unobservable. As the start state, goal and the rest of the environment do not vary during runtime, then we can conclude that the task is in a static environment. As we know the possible actions and the goal state which gives us the stopping condition, we classify the environment as known. As we are not concerned with any other aspects of the environment, this is a single agent environment. Hence, we can conclude that this agent is a simple Reflex agent as the agent selects actions on basis of current percept, ignoring rest of the percept history.

2.1.2 The Simple Agent

The *Simple* Agent has perfect knowledge of the entire environment, as well as a heuristic function which lets it assign a score to any state in the environment. The action taken by the agent is selected based on its current state, and the feedback from the heuristic function. The agent will either move forward / backward / left / right, and will not select any invalid actions. To classify the agent and the environment, we see that the agent is given a heuristic function, hence at any given state the agent has a fixed action which depends on the current state it is in, so we conclude that the agent is deterministic and sequential. At the start of execution, the agent has access to the full map of the maze. This map does not change throughout the entire run, so the environment is fully observable and static. As we know the possible actions and the goal state which gives us the stopping condition, we classify the environment as known. As we are not concerned with any other aspects of the environment, this is a single agent environment. As knowing about the current state is enough to decide what to do, the agent needs some sort of goal information which in this case is the heuristic function. Hence, we classify the agent as a Goal based agent.

2.1.3 The Realistic Agent

The *Realistic* Agent has only limited sensors with which to observe the environment, and can only observe its immediate neighboring states. The intended action depends on the training of the agent, which depends on the utility function and the Bellman equation. The agent is noisy, so there is a chance that for any intended action, a different random action could be executed instead. To classify the agent and the environment, we see that the agent keeps on improving the best action to take at any state until the reward it gets after completing the task converges. The current action taken depends on the previous action taken by the agent, and due to existence of noise in the agent we classify it as stochastic and sequential. At any given state, we can only observe the immediate neighbours of the state we are in, but we also know that the maze given to does not change, so the environment is partially observable and static. As we know do not have access to the full map of the environment, we classify the environment as unknown. As we are not concerned with any other aspects of the environment, this is a single agent environment. As searching for the goal by itself is not enough, we define a utility function which helps us measure our performance in each state we are in, and to choose our next action. This is the performance measure of the agent. Hence, we classify this agent as a Utility based agent.

3. METHOD / DESIGN

In the *Random* Agent, we haven't implemented any special techniques, we simply reused the given code. By implemented specification agent randomly chooses an action (without any noise) to take and then executes it. Either it reaches the goal or it fails due to timeout.

For the *Simple* Agent, we used A* (A- star) search algorithm to find an optimal solution of the problem. In this section of the assignment we were allowed to use the **informed search** strategy [2]. This strategy uses specific knowledge of a problem.

In our current problem, we are allowed to use prior knowledge – the full layout of the maze.

The general representation of A* search is
$f(n) = g(n) + h(n)$ where*:*
$g(n)$ - the path cost from the start node to node *n*
$h(n)$ - heuristic function, estimated cost of the cheapest path from node n to the goal.
$f(n)$ – estimated cost of the cheapest solution through node *n* [3].

Thus, we are trying first to travel to the node with the lowest value of $g(n) + h(n)$ possible. The strategy is reasonable (**optimal**) provided that $h(n)$ satisfies two conditions: **Admissibility and Consistency** [4]**.** An **admissible** heuristic $h(n)$ **never overestimates** the cost to reach the goal [5].
A heuristic $h(n)$ is consistent if, for every node *n* and every successor *n'* of *n* generated by any action *a,* the estimated cost of reaching goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n':*

$$h(n) \leq c(n, a, n') + h(n')$$

This is one of the forms of general **triangle inequality** [6]. In our case we are using the graph-search version of A* since our maze can be represented as an undirected graph. This version of A* search is **optimal** when heuristic $h(n)$ is **consistent** [7].

The time complexity of the algorithm depends on the chosen heuristic. Worst case scenario is exponential $O(b^d)$, where d is the shortest path to the goal, and b is **a branching factor** (number of successors per state on average). When it comes to the heuristic function, we have to be really cautious. Good

heuristics are the ones with low possible effective branching factor.

In the *Realistic* Agent, we used the Q-Learning algorithm. It is a Temporal-difference (TD) method which learns a utility-action representation instead of learning utilities [8]. That means we should map states to actions directly without any model required for learning or for action selection. Q-values are related to utility values as follows:

$$U(s) = \max_a Q(s, a)$$

where $Q(s, a)$ is value of doing action *a* in state *s*.
The equation for TD Q-Learning is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

The value of this equation is calculated each time action *a* is executed in state *s* that leads to state *s'* [9]. Since Q-learning uses the best Q-value it is considered an off-policy algorithm. That means it doesn't follow any specific policy, and despite this fact it still able to find an optimal solution. It is however considered to be a trade-off for this type of TD methods, they tend to be slower than on-policy methods.

We decided to implement Q-Learning algorithm because it assumes no previous knowledge of environment. It backs up the best Q-values by exploring the maze thus eventually finding an optimal solution.

4. IMPLEMENTATION

4.1. Random Agent Implementation

In the implementation of the assessed exercise, we relied heavily on the template code provided to us, as well at the AIMA Python examples and Malmo Python Examples.

For the *Random* Agent, we elected to keep the code in the exercise template. We experimented a bit and tried to add patterns to the random behavior, in hopes of somehow exploiting biases in the generated mazes to make the random agent solve them more consistently, but ultimately, we were unsuccessful. Thus, the code for the *Random* Agent is almost identical to the code given in the exercise template. The only difference is that we changed the agent's timeout from 0.5 seconds to 0.2 seconds, to give it more time to execute actions in the given 60 second timeframe of the missions.

4.2. Simple Agent Implementation

For the *Simple* Agent, we opted to reuse the code for A* from Lab 2, which relies heavily on the AIMA Python libraries. We began by acquiring the grid of the current scenario from *agent* object. Note that because we get the grid from the *agent* directly, there is no need for *AgentHelper* to run and map the grid for us beforehand. Consequently, we removed the code to run *AgentHelper* which was present in the assignment template given to us.

Once we acquired the *grid* array from the *agent*, we parsed it into a 2-dimensional array, which represented the maze. We then translated this 2-dimensional array into an *UndirectedGraph* object, which is a class defined in the AIMA Python *search* library. Next, we used the *UndirectedGraph* object to initialize a *GraphProblem*, which contains information about the graph that needs to be traversed, and what the start and end states are. The *GraphProblem* was then passed to the *astar_search* function given in Lab 2, which then computed and returned an optimal solution. Note that we chose not to write a heuristic function, and thus the default heuristic function was used. The default heuristic function scores a node by its distance from the solution node.

Once we received a solution from the *astar_search* function, all we had to do is iterate across it, and translate every node transition into an equivalent Malmo action. After every action in the solution is executed, we will have arrived at the goal state, and the scenario is then terminated.

One thing to note is that in each scenario, there were intermediate rewards present. Technically we could have written a heuristic function which accounted for these intermediate rewards, and even picked them up in some cases to improve our score.

However, the problem is that the reward offered by the intermediate rewards is in the range of '12' to '30', while the cost to move from one node to the next is '-6'. This means that if the agent must move away from the solution path to pick up a reward, in the majority of cases the reward it gains will be nullified by the cost of reaching it. Another problem is that the location of these intermediate rewards was not visible in the *grid* that we got from the *agent*, and in order to locate these rewards we would have to write extensive code in *AgentHelper*.

Thus, there is little point in picking up these intermediate rewards, or coding a heuristic function which takes them into account. Because of this, we believe that the default heuristic function for A* is optimal, and will always generate the optimal solution.

4.3. Realistic Agent Implementation

To implement the *Realistic* Agent, we relied on Q-Learning code from the Malmo Python Examples located in *tabular_q_learning.py*. We adapted the code from that file to work with our *Realistic* Agent code in Malmo. One notable change we made is that for each state, we assembled a set of *potential actions* which deliberately did not contain any

actions which could bump into a wall. We did this, because the *Realistic* agent is able to observe its surroundings in a radius of one, and thus is able to tell if it is next to a wall or not. We assumed that this would save time training the agent, because it would not spend many action cycles bumping into walls, and learning how not to bump into said walls.

For every scenario, we save to a file the *Q-table* of the agent after each action it takes, so that the agent would be able to retain all the knowledge it learned from one run to the next. These files are located in the *q_tables_dir* folder.

After our initial implementation of the Q-Learning algorithm, we did additional experimentation and tuning to the *alpha*, *gamma*, and *epsilon* parameters of the algorithm. We found that the optimal parameters were *1.0* for *alpha*, *0.51* for *gamma*, and *0.01* for *epsilon*. These parameters allowed the *Realistic* Agent to converge on a solution in 30 iterations 91% of the time.

To tune these parameters, we first extracted our Q-Learning code into a separate file called *q_learning_optimize.py*. We then removed all dependencies on Malmo, and wrote code which would emulate the behavior of Malmo. This allowed us to quickly send actions to our emulated Malmo, which had none of the overhead of the real Malmo. This in turn let our Q-Learning code execute much faster.

We then ran our *Simple* Agent 100 times, and for each mission we saved the mission grid to the *grids* folder. We could then use these saved mission grids as test scenarios for our Q-Learning optimization code.

Finally, we ran the Q-Learning code in *q_learning_optimize.py* with various different parameters for *alpha*, *gamma*, and *epsilon*. For each combination of these parameters, we ran each of the 100 saved scenarios 50 times. To be able to get acceptable performance, we were forced to use the PyPy Python interpreter [10], which had about 20x better performance than the default Python 2.7 interpreter.

For each scenario, we recorded the performance of each iteration. In the end, we were able to get the average reward for every combination of *alpha*/*gamma*/*epsilon*, along with the average time to convergence for each scenario. Charts of some of the results are presented in Appendix B.

From this data, we were able to deduce that the optimal value for *alpha* was 1.0, for *gamma* was 0.5, and for *epsilon* was 0.01. With these parameters, the agent was able to converge at its best solution the fastest. We also found that with these parameters, in 91% of cases the agent was able to converge on a solution in 30 iterations.

5. EVALUATION / TEST

To evaluate and test each agent type, we ran it against 100 different mission seeds. We then analyzed what the average time was for the agent to reach the goal state, whether it reached the goal state at all, and on average how much reward it accumulated. Finally, we compare the results of all three agents, and discuss caveats to be kept in mind when evaluating the comparison. We discuss the evaluation methodology and results below.

5.1. Random Agent Evaluation

To evaluate the *Random* Agent, we simply ran it on 100 different scenarios, and recorded its performance for each scenario. We then analyzed the performance to find that the *Random* Agent managed to find the solution 49% of the time. When it did find a solution, on average it took the *Random* Agent 33 seconds reach the goal state. Across the 100 scenarios, the average reward accumulated by the *Random* Agent was -1115.

5.2. Simple Agent Evaluation

We evaluated the *Simple* Agent much in the same way we evaluated the *Random* Agent. The agent was run for 100 different scenarios, and was able to find the solution 100% of the time. This is no surprise, given that the agent has available to it a complete representation of the environment, and thus should be able to find a path to the goal state every time (unless the goal is unreachable, but that never happened). On average, the *Simple* Agent took 2.4 seconds to reach the goal state, with an average reward of 931.

5.3. Realistic Agent Evaluation

Since the *Realistic* Agent needs about 30 training runs on a scenario in order to converge on a solution for that scenario, evaluating the *Realistic* Agent was a bit different than evaluating the other agents.

We first ran the *Realistic* Agent 30 times on 100 different scenarios, to train it on those scenarios. During the training, we logged whether it found a solution, the time to the solution, and the average reward collected. We found that during training the agent reached the solution 91 percent of the time, with an average time to goal state of 7.4 seconds, and an average reward collected of 909.

After training the *Realistic* Agent, we ran it on each of the 100 different scenarios once, this time with the *training* flag set to false. We collected the performance statistics and found that the trained *Realistic* Agent reached the reward 96% of the time, with an average time to reward of 3.4 seconds, and the average reward collected was 853.

A table of our results can be found below (Table 1):

	Random Agent	Simple Agent	Realistic Agent while training (no noise)	Realistic Agent after training (with 10% noise)
% of scenarios where goal was found	49%	100%	91%	96%
Average time to reach goal state	33 seconds	2.4 seconds	7.4 seconds	3.4 seconds
Average reward collected	-1115	931	909	853

Table 1: Comparison of the average performance of all agents across 100 different mission scenarios

6. DISCUSSION

The findings show that the trained *Realistic* Agent performed surprisingly well in comparison to the *Simple* Agent, and that the *Random* Agent's performance is terrible compared to all other agent types.

It might be possible to increase the performance of the *Simple* Agent by using *AgentHelper* to map out the location of each intermediate reward, and to pass those locations to the *Simple* Agent. The heuristic function of the *Simple* Agent would also have to be modified to account for the intermediate rewards. However, given how little reward is given for collecting the intermediate rewards (less than 30) it is unlikely that this optimization would significantly change the final reward in most scenarios.

We believe that these results could have been better, if the noisy transition model was not active in the non-training *Realistic* Agent. The problem with the noisy transition model is that it sometimes caused the agent to transition to a state where the Q-Values for that state caused it to enter a sort of loop, for which it was hard to break out of. Consequently, in some scenarios it failed to solve the maze at all.

Finally, it would be interesting to remove the optimization in the Q-Learning algorithm which prunes actions which result in bumping into a wall from the action set, and then observing the performance of *Realistic* Agent. It would be interesting to see if the *Realistic* Agent takes a longer time to converge to a solution because of this, and if the final performance of the trained *Realistic* Agent changes at all.

7. CONCLUSION

In this assessment, we implemented three different agent types to navigate mazes in Minecraft. The *Random* Agent is a reflex agent which behaves randomly and has no knowledge of its environment. The *Simple* Agent is a goal based agent with perfect knowledge of the environment. The *Realistic* Agent was a utility based agent with highly limited knowledge of its environment. Ultimately, we showed that the Random Agent was able to solve the mission scenarios in just under half the time, while the *Simple* Agent was able to solve every scenario, and *Realistic* Agent (after training) was able to solve almost every mission scenario.

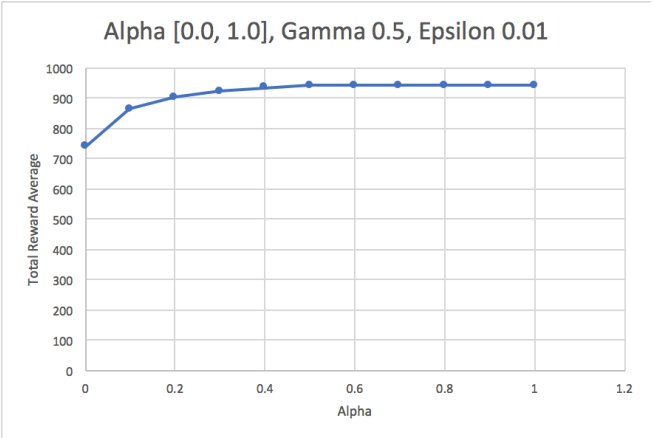
APPENDIX A

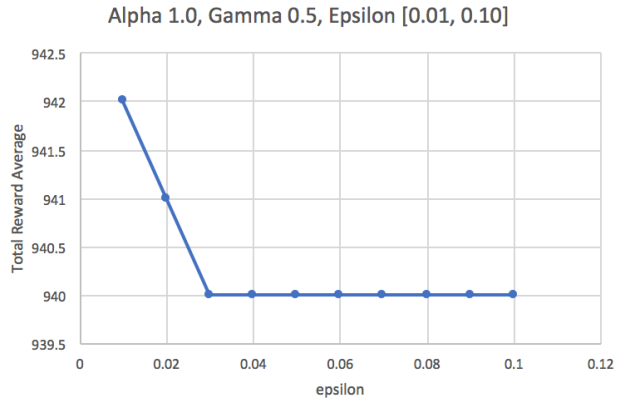
Apoorva Tamaskar – Contributed to writing the initial A* code, evaluating different algorithms to use for the *Simple* Agent, and evaluating their performance. Also wrote the Introduction and Analysis sections of the paper.

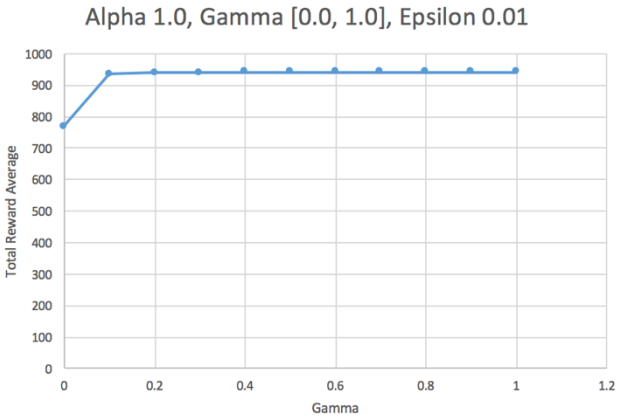
Igor Drozhilkin – Contributed to writing initial Q-Learning code, and evaluating different algorithms for every scenario type. Wrote the Method / Design section of the paper.

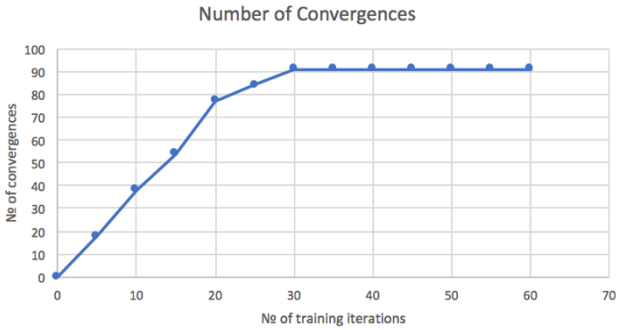
Marko Caklovic – Contributed to integrating A* into Malmo, and optimizing Q-Learning code to find best parameters. Also collected performance statistics on all algorithms. Wrote the Evaluation and Implementation part of the paper.

APPENDIX B









REFERENCES

[1] Omkar Kathe, Varsha Turkar, Girish Gudaye and Apoorv Jagtap "Maze solving robot using image processing" Bombay Section Symposium (IBSS), 2015 IEEE, IEEE, Mumbai, India 10-11 Sept. 2015

[2], [3], [4], [5], [6], [7] S. Russel, P. Norvig, *Artificial Intelligence. A modern Approach Third Edition.* Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England. 2016, pp. 93-95

[8], [9] S. Russel, P. Norvig, *Artificial Intelligence. A Modern Approach Third Edition.* Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England. 2016, pp. 843-845

[10] PyPy, alternative implementation of the Python language.
https://pypy.org/