Bachelor thesis
Computer Science

Radboud University

Comparing Discretization Methods for Applying Q-learning in Continuous State-Action Space

Author:
Luuk Arts
s4396863

First supervisor/assessor:
Prof. dr. Tom Heskes
`t.heskes@science.ru.nl`

Second assessor:
Prof. dr. ir. Arjen P. de Vries
`A.deVries@cs.ru.nl`

June 30, 2017

Abstract

Q-learning is a learning algorithm that can be used to find optimal solutions for Markov Decision Processes, most commonly in discrete state-action space. One way to extend Q-learning to continuous state-action space, is to discretize the environment in order to reduce the state-action space. In this thesis we evaluate two such discretization methods for their learning speed and solution optimality.

For this research we used Project Malmo, an AI experimentation platform released by Microsoft Research in august 2016. Over the course of this research, we evaluated this platform for its usefulness in implementing, visualizing and analyzing learning algorithms.

Contents

Chapter 1

Introduction

In August last year Microsoft Research released Project Malmo[1], an AI experimentation platform built on top of the popular game Minecraft. This platform provides an accessible way for researchers to program agents to solve diverse tasks within the Minecraft world.

Project Malmo contains an example of a Q-learning agent that can solve navigation problems in discrete state-action space. Since actual players and AI in Minecraft move around in continuous state-action space, it would be more interesting to see the performance of such an agent in continuous state-action space.

So for this research we will extend the discrete Q-learning agent to continuous state-action space using two different discretization methods. The first method discretizes the state-action space by mimicking a fully discrete state-action space, which allows only movement in the north, east, south and west directions. The second method gives more precision in movement by also allowing movement in the north-west, north-east, south-east and south-west directions.

Theoretically the second method allows the agent to move around faster by moving diagonally in one movement rather than in two. In addition, being able to move more freely makes the agent more versatile in environments involving more complex tasks and other AI. This comes at the cost of a doubling in the state-action space however, since we need to keep track of twice the amount of movement directions.

The main goal of this research is then to apply the two discretization methods to a navigation problem within the Project Malmo platform in order to see if this tradeoff has an impact on which method learns faster and which method learns a better performing solution.

As a subgoal of this research, we will be evaluating the Project Malmo platform on its usefulness for implementing, visualizing and analyzing learning algorithms.

Chapter 2

Preliminaries

2.1 Reinforcement Learning

Reinforcement learning[3] is a type of machine learning that, as opposed to other types of machine learning, does not need full knowledge of the environment and of the exact desired behaviour of the agent. Supervised learning for example, requires an external supervisor to provide the agent with learning examples to teach the agent the desired behaviour. When faced with problems with an unknown or complex environment, it becomes increasingly hard to provide the agent with a set of examples that covers all behaviours that are necessary for it to fulfil its task. Reinforcement learning is capable of dealing with these kinds of problems, because it allows algorithms to gain experience through interaction with the environment and to then determine the course of action that gives the best results.

For this to be possible, there needs to be a way to mathematically model the problem. This can be done by representing the problem as a Markov Decision Process (MDP)[3]. MDPs contain a set of environment states S, a set of actions A, a reward function R, which determines the reward that is received when performing a certain action in a certain state, and a probability function P, which determines the probability of moving from state s to state s' after using action a. This can be represented as a tuple

$$< S, A, R, P > . \tag{2.1}$$

The received rewards can be positive, neutral or negative and help the agent determine what actions are good and what actions are bad. In addition to states, actions and rewards, there are also rules that determine which aspects of the environment an agent can observe. This generally consists of the received rewards, but can also contain hints about the immediate environment.

The overall goal of reinforcement learning algorithms in regards to MDPs is to get an accurate internal model of the environment and to then find the sequence of actions that results in the highest possible reward within the MDP.

2.2 Q-learning

Q-learning is an algorithm in the field of reinforcement learning that was proposed in 1989 by Watkins[4]. Implementations of Q-learning contain all concepts that were described in section 2.1, notably a set of states S, a set of actions A and rewards associated with every state transition. The algorithm is environment independent, meaning that it requires no prior knowledge of the states and the associated rewards. Therefore, all knowledge about an environment can be gained through interaction with that environment.

The goal in Q-learning is then to learn what the expected reward will be for every state-action pair, in order to find the path that results in the highest possible reward. Generally the rewards for each state-action pair will be saved in a lookup table[2]. The agent then tries many state-action pairs repeatedly and updates the associated rewards in the lookup table with the newly gained knowledge, using a specific update function. When applied to a terminating state, this update function will simply assign the reward received from transitioning to the final state to the state-action pair corresponding to that final state and the action that was taken to get there. These terminating states can occur when, for example, the goal is found or the agent runs out of time or resources. For all other states, the non-terminating states, the update function for any given state-action pair is

$$Q_n(x, a) = (1 - a_n)Q_{n-1}(x, a) + \alpha_n[r_n + \gamma V_{n-1}(y_n)]. \tag{2.2}$$

Here $Q_n(x, a)$ represents the Q value, or reward, associated with the state-action pair consisting of state x and action a at stage n, given that the agent's learning experience consists of a sequence of distinct stages. The update function then takes the old value for the state-action pair and updates it using the reward received in stage n, r_n, and the maximum future reward the agent can receive taking any action in state y_n, $V_{n-1}(y_n)$, and then assigns this updated value to the Q value of the current state-action pair.

Finally, the update function contains two factors, that each can be assigned a value between 0 and 1. The first one is the learning rate, α_n, which determines the weight of the combination of the received reward and the maximum future reward. A high learning rate means that the algorithm favours this new knowledge over the old Q value. The other factor is the discount factor, γ_n, which determines the influence of future rewards. Here a value close to 0 means the algorithm is more short-term reward oriented, whereas a high value means the algorithm is more long-term reward oriented.

2.3 Project Malmo

Project Malmo is an AI experimentation platform built as a mod on top of the popular game Minecraft. This platform allows researchers to program agents that simulate player behaviour within the complex 3D game environment (fig. 2.1). The platform offers support for multiple areas of research, including robotics, computer vision, planning, multi-agent learning and reinforcement learning. In this research, version 18.0 of the platform was used.

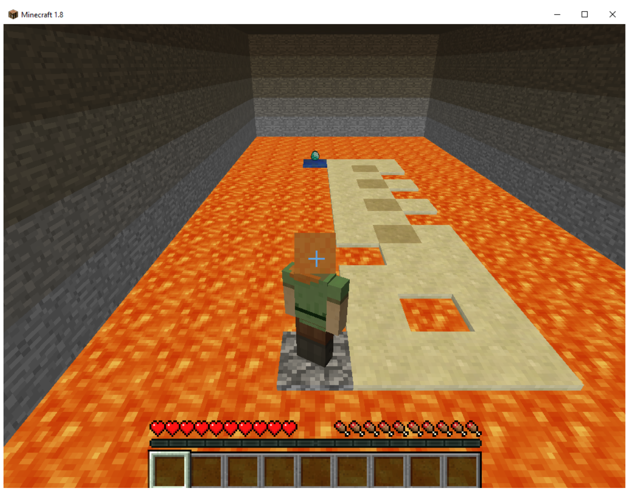


Figure 2.1: An agent in a Minecraft environment.

The game Minecraft is very diverse and offers a great variety of problems for agents to solve, such as survival mode, where the agent has to fight against enemies and gather food to prevent itself from starving, or crafting mode where the goal is to gather and combine smaller items in order to build one bigger item, or simple navigation problems like mazes and obstacle courses. These problems are represented within the game as missions. Missions load a Minecraft world in the Minecraft game and spawn an agent. They can be customized in mission definitions, which are written in XML.

The platform uses client-server interactions in order to interface the user's code with the Minecraft environment. The server (fig. 2.2) simply runs the Minecraft game and sends updates of the current state of the world when they are requested. The client runs the user's code and sends the mission XML and the commands that determine the agent's behaviour. It keeps track of the agent's environment by requesting the world state and updating its internal representation of the environment.

Agents must achieve a certain goal within the Minecraft world, for example reaching a certain coordinate, within a time limit using the actions

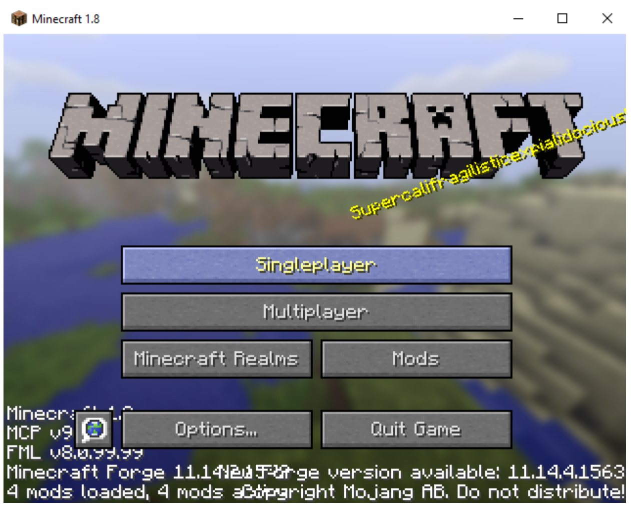

Minecraft 1.8
MINECRAFT
Supercalifragilisticexpialidocious!
Singleplayer
Multiplayer
Minecraft Realms
Mods
Options...
Quit Game
FML v8.0.99.99
Minecraft Forge 11.14 New Forge version available: 11.14.4.1563
4 mods loaded, 4 mods a Copyright Mojang AB. Do not distribute!

Figure 2.2: The Project Malmo server running the Minecraft game.

and the items provided in the mission definition. Each mission definition consists of two main programmable parts; the server handler (section 2.3.1) and the agent handler (section 2.3.2). These define the mission setup and the agent's capabilities.

Reinforcement learning is explicitly supported in these handlers by providing an easy way to programmatically set observation and reward policies in the agent handler. These policies determine when the agent receives rewards and which parts of the environment the agent is allowed to observe. The agent handler also provides several rules for actions, which determine the kinds of actions the agent is allowed to take in order to solve the mission. State management, in the form of dividing the environment in states and determining what constitutes a state transition, is not provided by the platform and must be coded from scratch.

The logic that is used to determine an agent's behaviour must be coded in separate files. This is where learning algorithms are implemented and where data about the observations and rewards that were received by the agent and the actions that were taken in response can be exposed. To support this, Project Malmo offers functionality for various programming languages, including Python, Java, C++ and C#, with functions that expose internal aspects of the platform. This makes it easy to connect self-written code with the Minecraft client.

2.3.1 Server Handler

The server handler determines the time limit of a mission and the kind of world that is created when the mission is loaded. Minecraft worlds consist of blocks with a certain type of material, like stone, wood or lava. The mission handler exposes XML definitions that allow a world to be drawn block by block, but also offers several generators which randomly generate structures. One such structure is the classroom structure, which consists of one or more rooms, separated by a door or by an obstacle. Examples of such obstacles are narrow paths across lava or a door that has to be opened by using a lever. We will be using a customized version of the classroom structure in our experiment.

2.3.2 Agent Handler

The agent handler is the part where all permissions regarding agent behaviour are defined. As mentioned earlier, this includes several concepts used for reinforcement learning.

The first important concept of the agent handler is its use of command permissions to determine which commands the agent can use. Commands are predefined and simulate keyboard and mouse input in order to let the agent mimic player behaviour. There are three categories of commands; movement commands, chat commands and inventory commands. These categories contain all elements related to their domains. For example, inventory commands contain the commands that allow the agent to use, swap, pick up or drop items. The agent handler provides XML definitions that grant permission to use a category of commands. It is also possible to allow or deny individual commands by entering them in the lists for allowed or denied commands, which are also provided as XML definitions.

Besides command permissions, the agent handler also provides XML definitions that determine when rewards are received and which observations are allowed. Examples of events that trigger positive rewards are touching a certain block type, collecting a certain item or reaching a certain position. Negative rewards can be triggered when the mission ends before the goal is reached or when the agent dies.

An example of observations that can be allowed are grid observations, which look at the block the agent is standing on and all surrounding blocks to determine their type. This can then be used to see if the agent is standing in front of obstacles like lava, a door or a wall. Other observation types include inventory observations, which determine what items the agent possesses, observations of nearby entities, including enemies, and so-called full stats observations, which contain information about the status of the agent, including the coordinates of its current position, its health and whether or not the agent is currently alive.

Finally, the agent handler offers XML definitions which determine what events trigger the end of a mission. Examples of triggers are reaching the specified time limit, reaching a specified maximum amount of commands, or when the agent reaches a certain position, touches a certain block type or collects a certain item.

Chapter 3

Research

3.1 Experiment

3.1.1 Setup

In this experiment we gathered data on the performance of two agents, each of which used Q-learning in combination with one of the two different discretization methods that we want to evaluate to solve the same mission.

This data gathering process contained two main phases: training and evaluating. In the training phase we trained each agent on a mission. In each of these training sessions the agent was run for a number of repeats until a solution path was found. The rewards for discovered state-action pairs that were learned in these repeats were saved in a lookup table as described in section 2.2. After every 10 repeats this lookup table was exported to a file.

When all lookup tables had been exported, the evaluation phase began. In this phase we tested the agents 10 times for their final lookup tables. No exploration was allowed in these runs, so that the results were fully determined by the knowledge represented in these lookup tables. The data we gathered in each of these test runs was whether or not the agent succeeds in finding its goal and if so, how many actions were taken before succeeding and how long the run lasted in seconds. The solution paths of the agents were then compared to each other by the success ratio and by how optimal the found solutions were in terms of duration and actions taken.

To evaluate the learning process, we took the lookup tables that were exported after every 10 runs and tested them again. Each of these tests was then scored by how much progress towards the goal was achieved in terms of the amount of cleared obstacles. In these runs exploration was allowed, to simulate the actual training phase of the agent.

3.1.2 Mission

The mission we used in this experiment contained an environment with two rooms with obstacles in the form of walls and lava bridges (figs. 3.1 and 3.2).

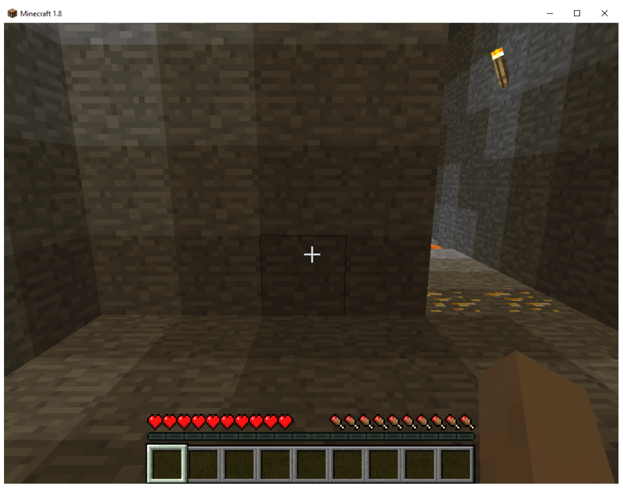


Figure 3.1: The first room of our mission, containing a wall obstacle and one subgoal, marked with yellow ore.

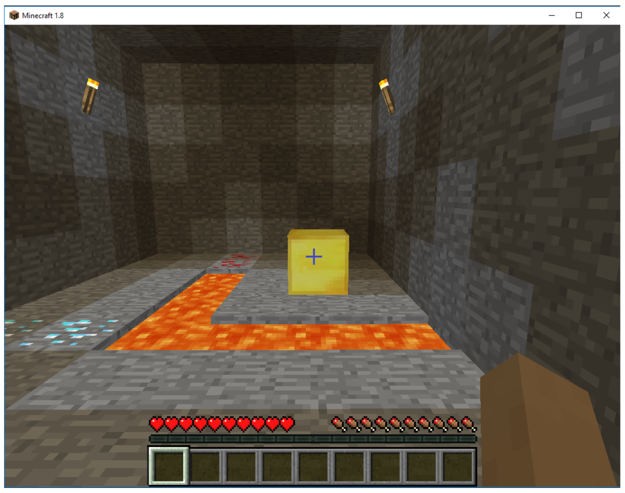


Figure 3.2: The second room of our mission, containing two lava bridge obstacles, two subgoals, marked with blue and red ore, and the final goal in the form of a gold block.

This mission environment has a total size of 6 by 14 blocks. The agent was spawned in the first room on a fixed starting position at the front wall. The goal of the mission was reached upon touching the gold block, which was placed on a fixed position at the end of the second room. Alternatively, the mission could end when a time limit of 1000 seconds was reached. Subgoals were placed at every obstacle to help the agent find the obstacles, as is common practice in Project Malmo missions. These subgoals were represented as blocks of colored ore, with a yellow, blue or a red color. Positive rewards were associated with touching subgoals and the final goal and negative rewards with touching lava. This mission was loosely modelled after missions provided by the Project Malmo platform, but adapted to our specific needs.

3.1.3 Agents

Discretization Methods

In this experiment we tested two agents, each using a different discretization method. For the first agent, the state space was discretized using discretization method 1, where we limited the turn directions of the agent to the north, east, south and west directions, or to 0, 90, 180 and 270 degree turns. Prior to this discretization the agent could turn to any degree within a 360 degree circle. For the second agent we used discretization method 2, which allowed the agent to turn in the north-west, north-east, south-east and south-west directions on top of the directions allowed in discretization method 1, thereby doubling the amount of directions the agent could turn to. Another step that was taken to discretize the state space was to limit the possible X- and Z-coordinates to whole numbers, meaning states correspond to the 1 by 1 blocks in the Minecraft world, rather than every possible combination of X- and Z-coordinates the agent could be standing on.

For both discretization methods the states were then defined as a combination of the X-coordinate, the Z-coordinate and the yaw, meaning the degree of rotation of the agent. So taking the first method as an example, for every valid combination of X- and Z-coordinates, there were then 4 possible states, corresponding to the north, east, south and west directions of the yaw. Because the X- and Z-coordinates were tied to blocks in the Minecraft world after performing the discretization, this meant that the amount of possible states was 4 times the amount of blocks in the mission. For the second method there were 8 possible states for every valid combination of X- and Z-coordinates, meaning the added turn directions doubled the state space.

Action Selection

The actions the agent could take were continuous actions, with the exception of the turn action which was discretized, meaning the agent turned instantly to the desired yaw degree. The full list of possible actions was as follows: move forwards, move backwards, strafe right, strafe left, turn right and turn left.

New actions were chosen whenever the agent entered a new state, or when an action resulted in no state change, which happened when the agent was not moving or turning. The chosen action was determined by the ϵ-greedy strategy, where generally the action that offered the best possible known reward in the current state was chosen, but there was an ϵ chance every time an action was chosen of selecting a random action instead, in order to encourage exploration.

3.1.4 Data Visualization

For the visualization of the lookup tables, we extended an example provided by Project Malmo for their discrete Q-learning agent. It consists of a simple grid of X by Z rectangles (fig. 3.3), where X and Z represent the width and depth of the mission environment.

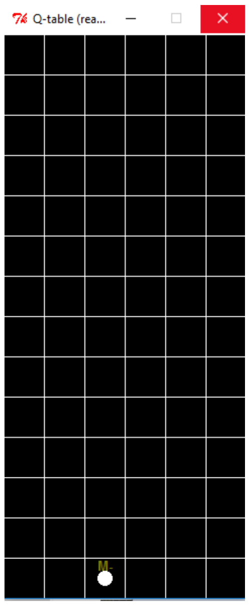
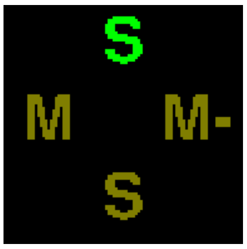


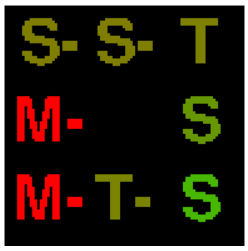
Figure 3.3: Grid representation of the mission environment.

Each of the rectangles in the grid represents a block in the Minecraft world, and for our discretization methods also the X- and Z-coordinate part of our states. The white circle represents the agent. To also incorporate the yaw part of the state, we fill the rectangles in using single state visualizations (fig. 3.4).

In these single state visualizations the positions of the symbols represent a yaw degree, with the top position being at 0 degrees, the right position at 90 degrees, the bottom position at 180 degrees and the left position at 270 degrees. The specific meanings of the symbols can be found in the legend. The different colors represent different reward values, where the color green



(a) Method 1.



(b) Method 2.

Figure 3.4: Single state visualizations for both discretization methods.

M:	Move forwards	$M-$:	Move backwards
S:	Strafe right	$S-$:	Strafe left
T:	Turn right	$T-$:	Turn left

represents a positive reward, red represents a negative reward and yellow represents a zero reward. Brighter colors represent bigger reward values.

3.2 Results

In this section we show the results we obtained by following the procedure described in section 3.1.1. These results represent an example of typical behaviour of Q-learning agents in combination with our discretization methods when they are run in Project Malmo.

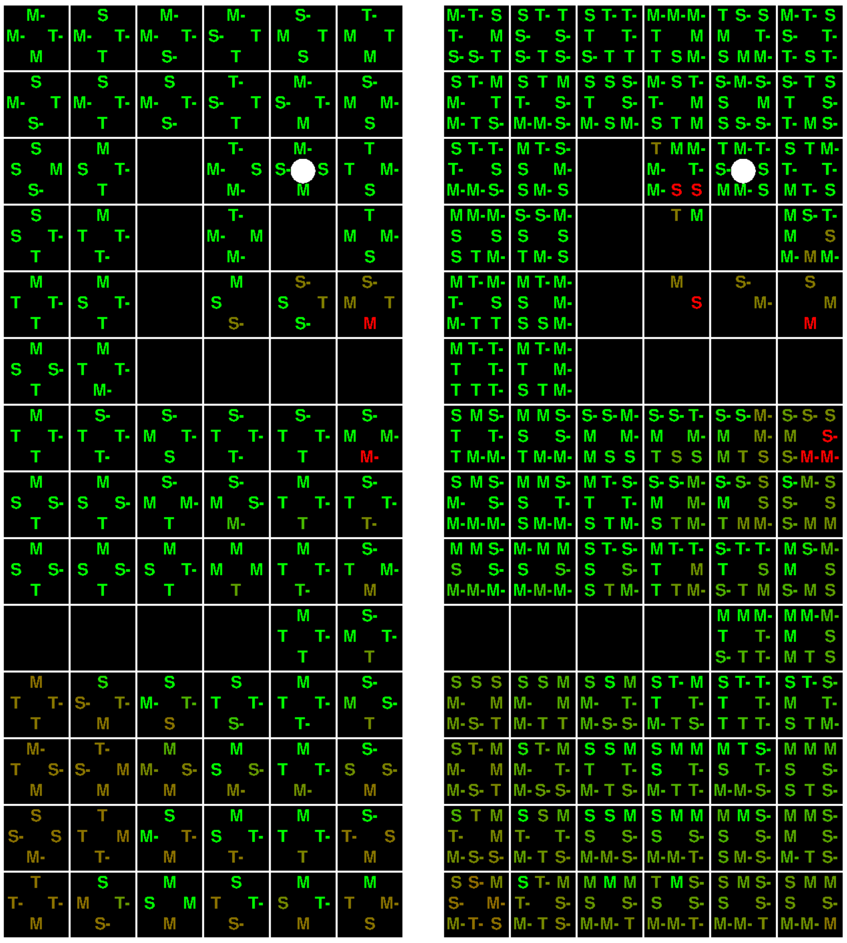
We defined two agents and two discretization methods, along with a mission. For this section we will name the agent that runs the first discretization method Agent 1 and the agent that runs the second discretization method Agent 2. We let the two agents run the mission until they found a solution path, resulting in two lookup tables (fig. 3.5) and a set of intermediate lookup tables that were exported after every 10 runs.

For each of our intermediate lookup tables, we will show what a typical training run with the knowledge contained within that lookup table looked like. For this we had the agent perform another training run with each intermediate lookup table and we looked at the agent's progress towards the goal over time. This progress was determined by the number of received rewards, where the first three received rewards meant that an obstacle towards the goal had been cleared and the fourth reward meant that the goal had been reached.

We wil also show the solution paths found in the final lookup tables for each of the agents and analyze them over 10 evaluation runs based on the following criteria:

- Success ratio
- Average actions taken

- Average duration in seconds



(a) Agent 1.

(b) Agent 2.

Figure 3.5: Visualization of the final lookup tables using the method described in section 3.1.4.

3.2.1 Agent 1

Learning Process

We visualized the learning process of the first agent in a graph (fig. 3.6). This graph shows that the agent reached the second subgoal for the first time after 20 runs, and reached it consistently after 110 runs. The reason for the inconsistency between runs 20 and 110 is that the agent died to lava for a lot of these runs. After run 110 most of the ways in which the agent could die between the first and the second obstacle were known and could be prevented.

The graph also shows that subgoal 3 and the goal are reached at the same time at run 280. What this means is that while over the course of the runs before that point, the obstacle before subgoal 3 had been cleared enough times to learn the path from subgoal 3 to the goal, the agent took some more time to find a full path between subgoal 2 and subgoal 3 that reliably cleared the obstacle.

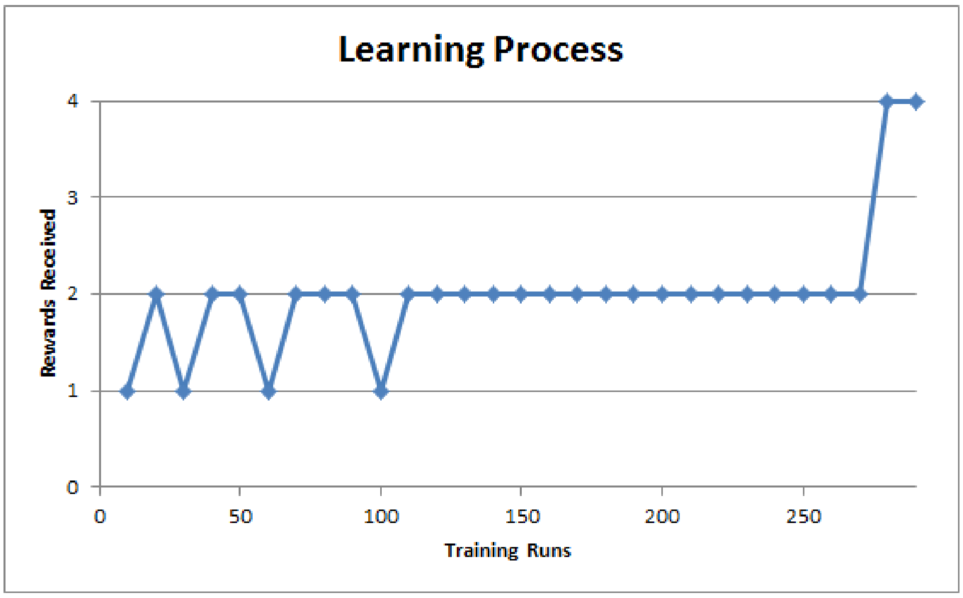


Figure 3.6: The progress of agent 1 towards the goal over time, in terms of the amount of positive rewards it received in every tenth run.

Solution Path

For agent 1 it took 289 runs in total to find a deterministic solution path from the starting point to the goal (fig. 3.7).

Across the 10 evaluation runs the agent scored very consistently, with the goal being reached in every run, using the same amount of actions and with a a difference of two seconds, or about 10% of the whole duration, between the shortest and longest of the runs.

The average performance across the 10 evaluation runs is described in table 3.1.

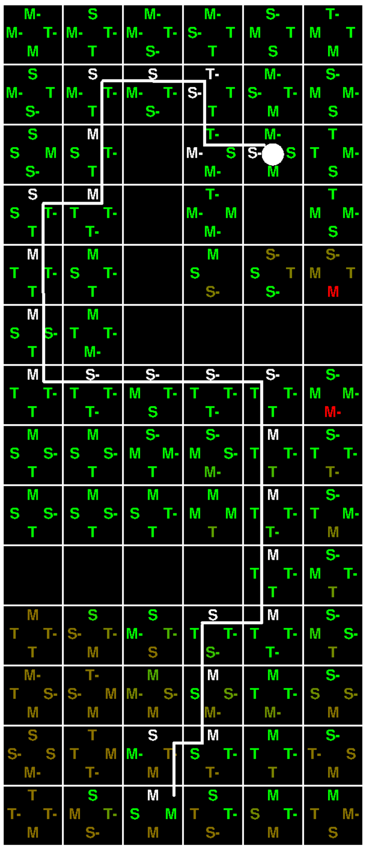


Figure 3.7: The solution path (in white) found by agent 1 over the course of 286 runs.

Succes Ratio (%)	100
Average Actions Taken	25
Average Duration (s)	22.4

Table 3.1: The average performance of agent 1's solution path over 10 evaluation runs.

3.2.2 Agent 2

Learning Process

Like we did for agent 1, the learning process of agent 2 has been visualized in a graph (fig. 3.8). This graph shows that the agent reaches the first subgoal reliably from run 10 onwards. It took the agent 80 runs to find a path from the starting point to subgoal 2. While learning the path towards subgoal 3 and the goal, the agent was still dying to lava in some runs, as shown at run 190 and run 300 in the graph. The relatively high state-action space of the second discretization method makes it hard for the agent to learn all the ways in which it can die, meaning it still randomly died all the way up until the solution path was found.

The agent finds the closed path towards subgoal 3 and the final goal relatively close after each other. In run 320 it reaches subgoal 3 and it takes another 40 runs to finalize the solution path.

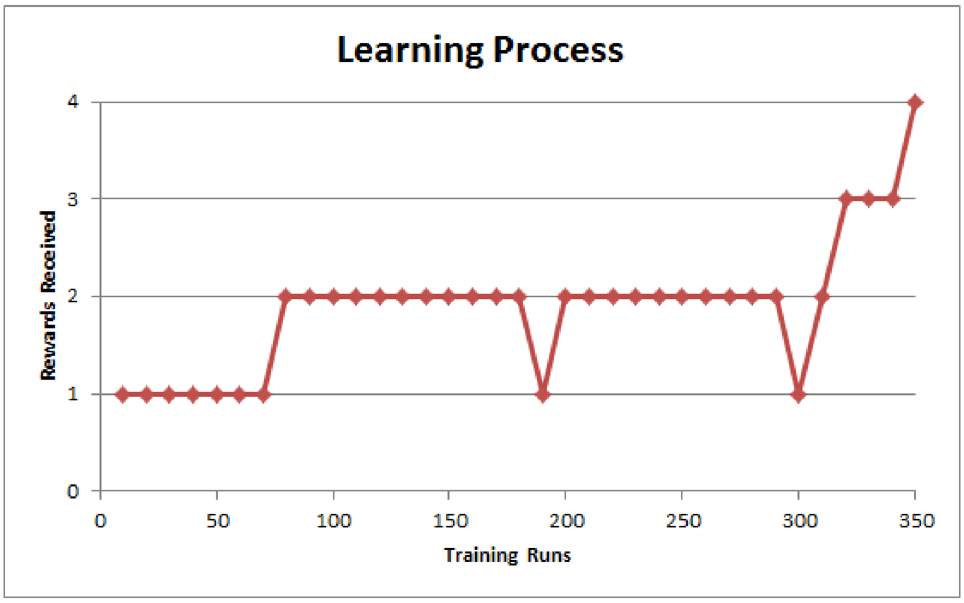


Figure 3.8: The progress of agent 2 towards the goal over time, in terms of the amount of positive rewards it received in every tenth run.

Solution Path

It took the second agent 346 runs to find a deterministic solution path from the starting point to the goal (fig. 3.9). Its performance for the 10 evaluation runs was not very consistent, with the goal being found in 6 of the 10 runs, different amounts of actions taken in all but 2 of the runs and a difference of 4 seconds, or about 14% of the total duration, between the durations of the longest and the shortest run.

The average performance across the 10 evaluation runs can be seen in table 3.2.

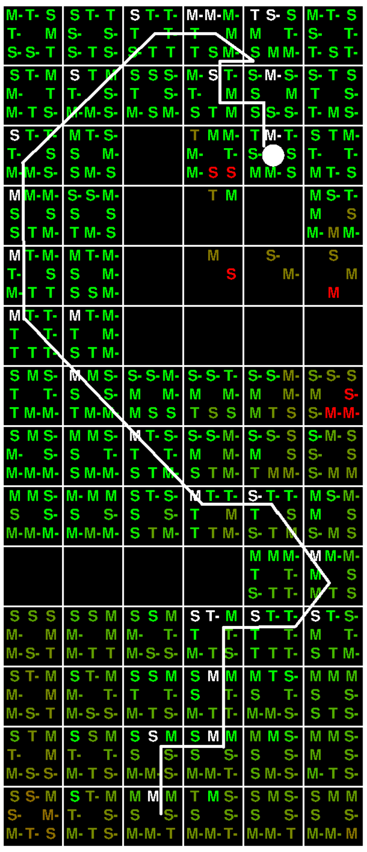


Figure 3.9: The solution path (in white) found by agent 2 over the course of 286 runs.

Succes Ratio (%)	60
Average Actions Taken	33.2
Average Duration (s)	27.7

Table 3.2: The average performance of agent 2's solution path over 10 evaluation runs.

3.3 Discussion

3.3.1 Comparison of the Methods

Learning Process

In our experiment the second agent was both slower and less performant than the first agent. To further illustrate the difference in learning speed we combined the graphs of the learning processes of both agents into one graph (fig. 3.10).

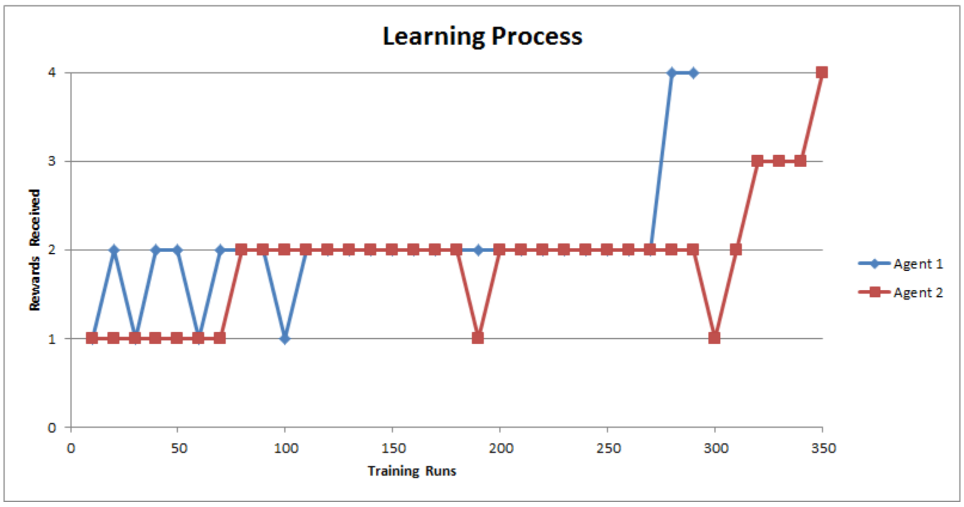


Figure 3.10: The progress of both agents towards the goal over time, in terms of the amount of positive rewards they received in every tenth run.

This graph shows that the agent using the first discretization method was at all points except for run 100 ahead or equal to agent 2, which uses the second discretization method. Agent 1 also finished its learning 57 runs ahead of agent 2, meaning it took agent 2 almost 20% longer to find a solution path. A reason for this may be found in the additional movement directions that were allowed for agent 2, because allowing movement in the 4 diagonal directions on top of the 4 straight movement directions doubles the state-action space. This means that the agent generally has to learn the optimal action for more states before it can find its deterministic solution path. There are also simply more ways for the agent to die, because more state-action pairs result in stepping into lava, meaning more runs end without receiving rewards for subgoals further into the mission and the goal itself.

Solution Path

An explanation for the 40% failure rate for agent 2 may be found in our use of the second discretization method, where diagonal movement is allowed. One shortcoming of this method is that in the case of diagonal movement from a state, the agent can exit the state directly through the corner, to the left of the corner or to the right of the corner, where each of these three exit points brings the agent to a different state. So a diagonal movement action from one state, can potentially bring the agent to three different other states. The result then depends on where exactly the agent entered the state, which may differ slightly between runs because of different delays in computation or in the client-server interactions of Project Malmo.

This can affect the success ratio of the solution path, because there are still many states left where the path back to the solution path is unknown and where no other path to the goal is known. When the agent enters one such state unexpectedly because it entered the previous state slightly to the left or slightly to the right of the expected entry point, the goal will not be found. More learning would be necessary to get solution paths from those states as well.

This would potentially require a lot of repetitions, because the only opportunity for learning is on the rare occassion the agent accidentally gets into one of those states. In those rare occasions the agent would then have to randomly select the right action. On top of that in order to ensure the correct action is taken in the future, the learned reward for this action, which depends on the new reward and the old already known reward, would have to be bigger than the already known rewards. Depending on the learning rate, this itself could take several repetitions.

In the end this would not change the fact that there would likely be multiple different outcomes with different amounts of actions taken and with different durations, but the success ratio could become 100%.

3.3.2 Platform Related Issues

The Project Malmo Platform itself also seems to contribute to the poor stability of the performance of agent 2. The platform only offers two ways to offer subgoal rewards that are relevant to navigation problems, a reward on reaching a position and a reward on touching a certain blocktype. This first type of reward is not an option in combination with our discretization methods, because an agent can reach the block that the position is in, while actually passing by the precise X- and Z-coordinates that define that position. This would make rewards in the states that move towards that block unreliable, because the agent would not reliably get the reward for moving towards that block. So the second type of reward was used in our experiment.

The issue that comes along with this reward on touching a certain block type, is that in some cases the platform somehow registers the agent touching the block while the agent's position coordinates are actually outside the block that the reward is in. Therefore, the agent will not reach the subgoal state in its internal representation of the state-action space, meaning the received reward will be assigned to the wrong state-action pair.

The platform does not offer a way to alter the reward on touching a certain block type functionality in order to account for these problems. It also does not offer more precise rewards which require you to actually fully stand on the subgoal block. These things would be necessary to make the learning process with our discretization methods more reliable.

Another problem is caused by the communication between the agent code on the client and the Minecraft environment on the server. This communication consists of client-server interactions, where the server sends its current world state when the client requests it and the client sends the commands for the agent to the server. This communication causes some delay, which sometimes causes some inconsistency between what the client believes to be the current state of the Minecraft world and the actual state of the Minecraft world, in particular related to the current X- and Z-coordinates of the agent. This makes it possible for the agent to just slightly pass through a subgoal state's outer corner and then pass into the next state in the time it takes for the client to finish its own operations and to receive the new world state. The reward is then assigned to the wrong state.

Chapter 4

Conclusions

The main goal of our research was to compare two discretization methods, based on their learning process and their final solution path. In the first method we discretized the state-action space so that the agent could only move in the north, south, east and west directions, while in the second method we also allowed movement in the north-west, south-west, south-east and north-east directions. We expected the first method to be faster and the second method to perform better in a tradeoff between learning speed and performance. The results show that this tradeoff did not occur in our experiment and that the first method learned faster and performed better.

Therefore trying to make the discretization of the continuous state-action space more precise by allowing movement in more directions did not improve the performance of the agent within a reasonable timeframe.

We also showed some of the functionality of the Project Malmo platform. When it came to implementing our agents with the Q-learning algorithm and the two discretization methods and our mission as an environment with an underlying MDP, Malmo offered all the functionality we needed and more. We did however encounter some very specific problems with the interaction between Project Malmo and our discretization methods (section 3.3.2), which could have been solved with some slight alterations of existing functionality. The platform makes it very easy to represent Minecraft environments as MDPs through its mission system and also offers a lot of template code and missions up front. Importantly, the platform also works as documented, meaning we spent very little time debugging.

The platform does not provide a lot of functionality for visualizing and analysing algorithms and methods. It does however provide the opportunity to quickly and easily generate a lot of data about the performance of an algorithm by letting an agent actually apply the algorithm to problems of varying complexity within Minecraft. It also provides some examples on how to visualize this data.

So overall the Project Malmo platform is very useful for implementing, visualizing and analyzing learning algorithms.

Bibliography

[1] Johnson M., Hofmann K., Hutton T., and Bignell D. The Malmo Platform for Artificial Intelligence Experimentation. In Kambhampati S., editor, *Proc. 25th International Joint Conference on Artificial Intelligence*, page 4246. AAAI Press, Palo Alto, California USA., (2016). https://github.com/Microsoft/malmo.

[2] J. C. Santamaría, R. S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive behaviour*, 6(2):163–217, 1997.

[3] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge: MIT press, 1998.

[4] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.