Smart Dog for Minecraft

Kim Arnold Thomsen
Rasmus D. C. Dalhoff-Jensen

June 10, 2014

AALBORG UNIVERSITY
STUDENTS STUDYREPORT

Department of Computer Science
Aalborg University
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst
Telephone +45 9940 9940
Telefax +45 9940 9798
http://www.cs.aau.dk

Title: Smart Dog for Minecraft
Subject: Reinforcement Learning
Semester: Spring Semester 2014
Project group: sw103f14

Participants:

Rasmus D.C. Dalhoff-Jensen

Kim A. Thomsen

Supervisor:
Manfred Jaeger

Number of copies: 4

Number of pages: 109

Number of numerated pages: 101

Number of appendices: 9 Pages

Completed: June 10, 2014

Synopsis:

This report argues for the benefit of combining tabular reinforcement learning with feature-based reinforcement learning, to make it possible for agents to have specific behaviour in specific situations in an general environment impractical to express without features. The report describes three different approaches to do so, as well as an implementation of such a system in the game Minecraft. The report describes a set of test showing that two of the three approaches shows benefit in such a scenario.

Contents

Chapter 1

Word Definitions and Abbreviations

This chapter contains a list of word definitions and abbreviations used throughout this report.

AI: Artificial Intelligence.
Current: By "current", we mean the version of something at the time of publishing the report. As such, "current version of the system" refers to the version of the system at the time of publishing the report
Game-state: The state of all possible game attributes at the given point in time.
MDP: Markov Decision Process
Mod: Modification for a piece of software. Usually made by a person not associated with the developers of the original software.
Player: The player refers to the person playing Minecraft.
Player-Character: The player character is the entity the player controls in the videogame Minecraft.
State: The state from an MDP.
State-attribute: In tabular reinforcement learning the state are described via a set of state-attributes.
Tabular: Something relating to the side using state-based learning, as oposed to feature-based learning.
User: A user is a person who interacts with the system, but without being one of the developers.
XML:Extensible Markup Language.

Part I

Introduction

Chapter 2

Introduction to Smart Dog

This chapter provides an introduction to the report, and to the smartdog project

In this report, we aim to document the work that has gone into the smartdog project, and the results of this work. The different parts of this report describe the process of developing an intelligent agent for the video game Minecraft, which uses a combination of two different kinds of reinforcement learning in order to adapt to the environment. The name of the project stems from the agent, which in this case is a dog in the video game.
Throughout this report, we will describe our thesis, the concepts of reinforcement learning, the different components that makes up the system, and a series of tests which aims to show whether the thesis holds or not.

Before moving one to the rest of the report, we would like to give thanks to the following people and organisations for providing software which we incorporate in our project:

- Mojang - The developers of Minecraft.[1]
- The developers of Minecraft Forge, a community made API for Minecraft.[2]
- Andy Khan - For developing the open source Java Excel API[3]

[1] https://mojang.com/

[2] The names of the top contributers can be seen on this page:
`http://www.minecraftforge.net/wiki/Minecraft_Forge`

[3] http://www.andykhan.com/jexcelapi/

Chapter 3

The Minecraft World

This chapter introduces the reader to the video game Minecraft, specifically with its environment, and the different game concepts used to describe it. The chapter is taken in its whole from [4]

According to Mojang (the independent development company who developed Minecraft):

"Minecraft is a game about breaking and placing blocks." [1]

Minecraft is a sandbox game taking place in a world made of layers of perfectly square blocks. With the exception of the lowest layer consisting of bedrock, every other block can be picked up and placed somewhere else.
In addition to the blocks, the game world has creatures in it, including peaceful ones such as cows or sheep, and hostile creatures such as zombies and the creature known in Minecraft as the creeper.
A third category of creatures are considered neutral, in that they will only attack the player if provoked. This category includes wolves and iron golems.
Finally, the player is also present in the environment, as a person trying to survive and build.
Being a sandbox game, Minecraft has no predefined goal or end condition, leading to players coming up with their own ideas about what to do, and what their goals are.

3.1 The technical aspects

Minecraft is a java based game. It is not open source, although users are allowed to make external modifications for the game (known as mods), as long as they do not make any changes to the Minecraft source code, or charge any money for their mod. [6]
In the Minecraft model, the world consists of three different things:

- **Blocks**: The $1 * 1 * 1$ stationary blocks, such as stone, water, and dirt, from which the world is built.
- **Items**: Anything which the player can hold in his hands. These can be used to interact with the environment, in the form of tools, or be placed in the world, which will place the corresponding block there (a stone item will place a stone block, and a flower item will place a flower block.

[1] According to their site, https://minecraft.net/

- **Entities**: Anything which can move. Most entities are affected by gravity. Entities include creatures, dropped items, arrows in mid-air, mine-carts, the player, falling sand, and more.

3.1.1 Minecraft Forge

Minecraft Forge is an unofficial Minecraft API. Our reason for using Forge, is that Mojang have not yet made an official API for Minecraft, and as such Forge is a good choice. Using Forge enables the developer to easily create mod files, without making changes to the Minecraft source, which would go against the license agreement. [6]

Chapter 4

Problem Domain

This chapter explains the problem domain of this project.

With tabular reinforcement-learning[1] one can have an agent learn certain behaviours in an environment, but there are some problems with scaling. If the environment is too complex, it can become practically impossible to create an exact model of the environment. In the game Minecraft the playable map size is $60'000'000 \times 60'000'000 \times 255$ blocks. This is the amount of blocks that can theoretically exist in a Minecraft world that would also be traversable by an agent. If one should only keep track of which block the agent is on, that would result in $918'000'000'000'000'000'000$ different states. This does not take into account such things as an agent's heading, speed, other entities in the environment, or the fact that entities positions are not by block, but by double values indicating where on a specific block the entity is. The Minecraft environment is too complex to be fully described by a tabular reinforcement model. Another factor which intensifies this is that Minecraft runs very much in real time. Not that it doesn't run in its own "clicks" and therefore in turns, but it is not legal to modify the Minecraft files directly. This means that what can be done legally is to access some events happening, making it a virtual real-time system, which is:

"...a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment." [1, p. 2]

With this in mind, using a tabular reinforcement learning approach in any real life scenario can be rather problematic. The environment has to be simplified since we so far haven't been able to fully express the real world. And even if we could, that would properly be to complex a state-space to handle. A favoured example in many game-theory examples would be chess. A reinforcement agent which could observe the chess board and move the pieces would need to know the position of all 32 pieces. If we simplify the example a bit, and imagine that every piece can be on every position of the board that is 64 positions, meaning 64^{32} states resulting in approximately $6,3 \times 10^{57}$ unique states.

Handling this with tabular reinforcement would be impractical, but there exist a feature based reinforcement method which scales better with complex

[1] Meaning state-based reinforcement-learning, as opposed to feature-based reinforcement-learning.

environments. The feature based solution uses features to describe a state instead of keeping track of all possible states. Therefore the amount of information that needs to be saved is limited to the amount of features, rather than an exponential increase in the state-space by state attributes. This means that you can create reinforcement learning agents to handle rather complex environments. The problem lies in that the features are functions and doesn't indicate unique states. Where in a tabular solution a single shift in any state-attribute would mean a completely new state, the feature would only be a function on the same attribute. Such a function would try to inform what action to perform depending on the value of the game-state, and together with the other features the action is determined. To clarify, if we want to describe the x-coordinate of an agent as a state-attribute, the state-attribute would see it as a set of completely independent states and choose appropriate actions for every independent state. If we would want to represent the same as a feature, the feature would indicate the value of the x-coordinate, and a weight would indicate how likely it is that a specific action should be performed depending on the feature value. When using feature-based reinforcement learning, one needs to design the features so that they are meaningful and true. When a feature outputs 2 instead of 1, it should be double as good/bad. The features are therefore general information about the environment instead of unique states. This means that you can create an agent which can see that two different situations are "similar", and will behave the same way in both of them. The benefit of this is that the agent will not end up in a brand new situation where it has to learn a completely new behaviour. The downside to this is that the agent can not have as precise a behaviour as it could have with a tabular model.

With features one could in Minecraft create an agent which flees from dangerous situations and otherwise stays alive. One could simulate the instincts of a animal with features. But such an agent would not be able to understand unique states. That certain behaviour should be different if a feature value has a unique mid-value, rather than a high or low value. There are certainly agents which can benefit from the general characteristic of features. A rock-paper-scissor[2] like strategy-gaming AI which can keep track of the players unit strengths and locations. But there will be optimal plays that will be missed, since such approaches does not track unique states, but rather a general overview. So we have a tabular solution which works as a table of unique states, as illustrated in Figure 4.1, whereas features functions more like a group of advisor's, as the one seen in Figure 4.2.

The tabular solution would, if trained properly, always be better since it handles all possible states, but it is not always practically possible to

[2]With this we mean that certain units are good against some other units and weak towards yet another type of unit.

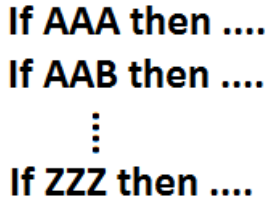


Figure 4.1: Illustrating behaviour in unique states.

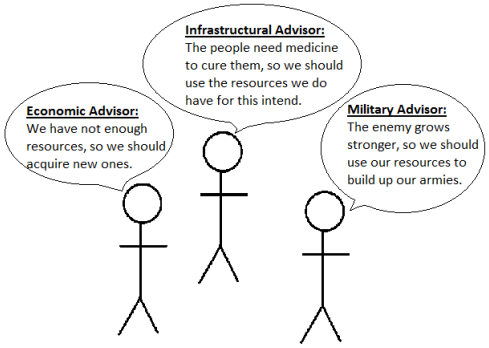


Figure 4.2: Illustrating features as a group of advisors.

create the fully descriptive statespace. The feature based solution would, if designed properly, be able to express the environment, but only in general terms.

4.1 Problem Specification

With the established problem domain, it looks like neither the tabular nor feature based reinforcement-learning is sufficient. It is either impractical and without transferring "known"[3] behaviour or not precise enough but with transferable knowledge. Therefore we can explain the problem with the following problem statement:

How can we develop an intelligent agent which is capable of learning optimal behaviour for different scenarios, while being able to transfer knowledge from one scenario to another as well?

With the problem specification we can split it into these problem specification points:

- Develop an intelligent agent.
- Enable the agent to learn an optimal behaviour for a given situation.
- Enable knowledge transfer between different situations.

4.1.1 Develop an Intelligent Agent

With this we mean to develop an agent which behaves intelligently in a environment. This could be anything from staying alive while gathering

[3]Known from other similar states.

resources, to complex strategic decisions as an AI in a strategy game. In this report we will be focusing on the more simple examples, and keep them as simple as possible while still having some complexity to show.

4.1.2 Enable the Agent to Learn an Optimal Behaviour for a Given Situation

With this we mean that the agent should be able to learn an optimal policy for a specific situation. With a specific situation, we mean one which is in the same general environment, but different to other situations in the environment. A situation is small enough to be expressed with a tabular state-space, whereas the general environment would be impractical, if not practically impossible, to describe with a tabular state-space.

4.1.3 Enable Knowledge Transfer between Different Situations

With this we mean that the agent should be able to transfer general knowledge obtained in one situation to another situation within the same general environment. That is to say that the general environment could be described by factors that are true in all situations which could appear in the environment. For instance that certain entities are bad/good, or that water extinguishes fire.

Chapter 5

Our Thesis

This chapter describes our proposed solution to the Problem Specification in section 4.1.

Our idea is to combine the tabular theory with the feature theory, making a system which benefits from both the precise instructions from the tabular Q-function, and the more general instructions from the feature-based Q-function.

It is our belief that such a combination would outperform either of the two standard models of reinforcement learning when it comes to situations where an agent has to switch between different kinds of environments.

Our plan is to design a decision module so that whenever the agent has to decide on which action to perform, it will consult both the tabular model, and the feature-based model in order to find the action which has the highest combined Q-value.
The Q-value is an estimate of the quality of performing the given action at the given state of time, based on previous observations.
The idea behind using the combined Q-value is to allow the agent to learn a behaviour which is generally smart, and then be able to use another behaviour if it knows one for a specific situation. This should make the agent adaptive, while still being able to specialise.

As such, we intend to provide a solution to our problem specification by developping an intelligent agent which uses tabular Q-learning for learning optimal behaviour for a specific situation, and feature-based Q-learning for learning behaviour that can be transferable trough all situations.

Part II

Theory

Chapter 6

Reinforcement Learning

This chapter will give an introduction to reinforcement learning, and demonstrate different approaches that have been considered for this project's implementation. It is based on [4, Chapter 6], with examples added and several minor changes written.

The general concept is that whenever an agent performs an action, it will receive feedback on whether that action was good, given the context. The agent is then supposed to adapt, so that it will perform better actions, based on the feedback it has been given. A more detailed description of the concept is as follows: At all times, the agent is in a state $s \in S$ (S being the set of all states), which corresponds to the agents knowledge about the environment, whether it represents its location as the coordinates in a grid, or its hand in a game of poker. In each state, the agent must reach a decision on which action $a \in A$ (A being the set of all actions) to perform. After having performed an action, the environment will provide the agent with a new state, as well as a reward r for performing the action a in the previous state. Rewards are usually represented by a decimal value, with positive numbers representing positive feedback, and negative numbers representing negative feedback. The numeric value of the reward indicates the weight of it. This means that 0.8 is a greater reward than 0.6, while -0.7 is a worse reward (greater deterrent) than -0.1.

Figure 6.1 shows this behaviour. The agent are always in a state s. It decides to performs an action, a, that might or might not change the environment. After having performed an action the agent will receive a reward and a new state. The reward might be 0, but is received nonetheless. This is the pattern state-action-reward-state, and is denoted as a 4-tuple (s,a,r,s').

6.1 Markov Decision Process

An MDP is a model of decision making for an agent. It models the states which the agent can be in, and the actions that the agent can perform. The state can describe different things. It might describe the coordinates of the agent's position, it might describe the positions of chess pieces, or it might describe the agent's situation.
The actions that the agent can perform, describes the different ways of interaction that the agent has. It might describe the different ways to move

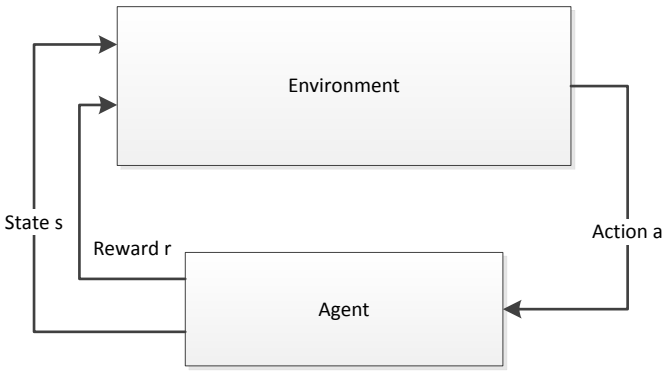


Figure 6.1: This picture shows the agent in its interaction with the environment.

in a grid, different moves available for chess pieces, and actions in a more abstract environment.
The Markov Decision Process also contains a probability matrix which shows what states the agent will end up in when it takes a specific action in a specific state, and how likely it is to get to these states. This means there is a matrix for each action, and if the state-space is large, so will the matrices be. In these matrices. it is then easier to see what will happen to agents when they perform specific actions.
Lastly the Markov Decision Process also contains the reward for taking specific actions from specific states and ending in specific states. This is the way to show the agent its goal. The reward are a numeric value, where the value shows how important the rewards are in comparison to each other, so that some actions becomes preferable to others in specific situations.
Formally, a Markov Decision Process is a 4-tuple $(S, A, P_a(s, s'), R(a, s'))$ where

- S is a finite set of states
- A is a finite set of actions
- $P_a(s, s')$ is the probability matrix depicting the probability of taking the action a in the state s, will lead to s'
- $R(a, s')$is the reward for taking action a in a given state, and ending in the state s'.

We will now show an example. This example will be a depiction of some of the implementation later on, but here we will describe it independently of it.
Lets say we have an agent in a room. It can see what kind of floor it is standing on, and what types of floor is in the rest of the room. There are 4 types of floor, red, yellow, green, and neutral floor. The agent only cares for the coloured parts and it doesn't care how much of the different types there are or where they are, only whether they are there or not. In this specific example Figure 6.2 there is one part of the floor that is red, one part that is yellow and one part that is green. Anything in between is neutral.

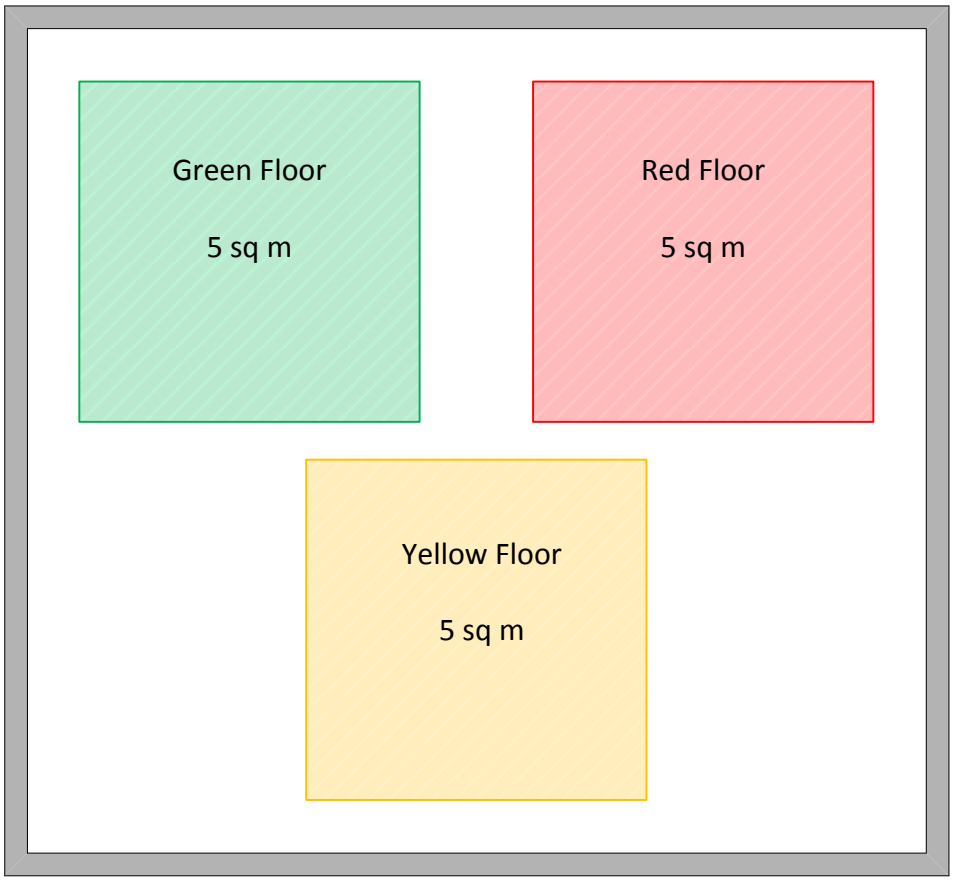


Figure 6.2: This picture shows a concrete example of an agent in a environment.

The agent will get a reward of 1 every time it enters the green part of the floor.
Lets for now say that r = red, y = yellow, g = green, and if a 0 or 1 is appended, it says whether a floor type is seen or not. One specific state could then be $s_1 = \{r, r1, y1, g1\}$ meaning that the agent is standing on red

floor, sees the red floor, and can see both yellow and green floor. The state space used in this example is as follows:

$$S = \{\{r, y, g\} \times \{r0, r1\} \times \{y0, y1\} \times \{g0, g1\}\}$$

having $3 * 2 * 2 * 2 = 24$ states.
Notice that the agent doesn't sense whether there are neutral floors nearby or not, since in this example, we assume that if the floor type is not one of the other types, its neutral. Also notice that the agent have no value for standing on neutral floor. We change the value when the agent enters one of the coloured floor types and not the neutral type, which will be explained in more detail in chapter 11, and why we do this. For now just assume that there is good reasoning behind this design choice.

In this example the set of actions would be
$A = \{move\_to\_red, move\_to\_yellow, move\_to\_green\}$ There are no action to move to neutral due to moving to the neutral part of the floor is uninteresting for the agent. The probability matrix would in this example be three 24×24 matrices. One for each action. Lastly there would be rewards for all action-state pairs. In this example all of them would be 0 except the states where the smartdog is standing on a green block, where the reward $R(*, \{g, *, *, *\})$ would have the value of 1.

6.2 Optimal Policy

The goal of reinforcement learning is to learn an optimal policy for behaviour in a given environment. An optimal policy is one which maximises the discounted reward. By discounted reward, we mean rewards obtained in the future. These rewards are discounted by a discount value γ, indicating that we mean the rewards closer by are preferable over rewards further away. The discount value γ defined as $0 \leq \gamma < 1$. As such, rewards are worth less the further away they are, with the total expected reward being

$$R = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...[5]$$

where t, indicates how far into the future the reward is. For instance $t + 2$ being one step further away than $t + 1$.
This is important since if it was not in place, the agent would find it fine that it could possible get a high reward, and perform actions with no rewards given as long as those could possible lead to the reward in the future. It would not need to perform actions to get closer to getting that reward as long as it would be a possibility in the future, leading the agent to be able to be caught in a loop that gives no rewards and never ends.
If we continue the example from section 6.1, we can see that there would be any number of optimal policies. As long as every second action is move

to green, there are no difference between moving to yellow or to red. An example of an optimal policy could therefore be:

1. move to green
2. move to red
3. repeat

Here it will gather a positive reward every time it takes the action move to green.

6.3 ϵ-greedy

To find an optimal policy the agent needs to explore the environment. While performing actions the agent think is best at the time, there might be actions that leads to greater rewards than formerly discovered. So some way to make the agent somewhat explorative rather than purely exploitative could help find the optimal policy.
With ϵ-greedy, the idea is to always pick the best action, except for ϵ of the time. This will introduce a random element to the decision process, which might make the agent find new and better approaches. This is also useful in case of a dynamic environment, where just because a given state-action pair gave a negative reward at some point, it might not keep doing that.
However, while introducing randomness in the decision making process might be beneficial in the early stages of a static environment problem, it is not beneficial once the agent has discovered an optimal policy. As a result, there are several variations of ϵ-greedy policies, such as introducing decay, meaning that the chance of the agent taking a random action gets lower and lower over time.

Continuing the example from section 6.2. We have a policy which dictates that we go between green and red. Introducing ϵ-greedy with ϵ having a value of 0.1 the agent would take a random action 10% of the time. This would lead the agent to at times go to the yellow part of the floor. In the case described, this would not lead to a better reward. If the environment where to change, however, so that entering yellow would give a reward of 2, this would have a great impact on what would be the optimal policy.

6.4 Q-Values

Having established the (s,a,r,s') connection, it is possible for the agent to start figuring out how to pick the next action, based on its experience of the

rewards given to it for a specific state-action combination. If we let $Q(s, a)$ show the expected discounted reward for performing action a in state s when the agent performs optimally thereafter, we can see that it would be beneficial to choose an action with a high expected discounted reward.
Discounted reward refers to the idea that rewards are less valuable the further into the future they are expected to be received.
Two typically used approaches for obtaining these Q-values are tabular Q-learning and feature based Q-Learning described in chapter 7 and chapter 8 respectively.

Chapter 7

Tabular Q-Learning

This chapter will describe the concept of tabular Q-learning

The idea behind tabular Q-learning is to represent the Q-values of every state-action pair in a table. What this means, is that for each state-action pair, there will be a row in a table telling what the current estimate for the Q-value of performing that action in that state is.
Now, in order to determine the expected reward for a given state-action pair, tabular Q-learning uses the previously noted expected reward for the pair, as well as the reward, and the expected reward of the best possible future state-action pair. The formula below shows the approach in more detail:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma max_{a'}(Q(s',a')))^1$$

In the formula, γ is the discount factor, and α is the learning rate, which is a value showing how "conservative" the agent is, meaning whether it favours new values over experience. Both factors have a value between 0 and 1. This way, every time an action a has been performed in a state s, the corresponding table entry for the Q-value $Q(s,a)$ is updated.

If we continue the example from chapter 6, we could describe how such a table looks like. It would be a table with 72 rows of state action pair since we have a state-space of 24 states and 3 actions. The specific scenario This is because which floor types the agent can see does not change, and there are three types of floor it can have been on. So the relevant rows for this specific scenario can be seen in Table 7.1

[1] As seen in [2]

State s	Action a	$Q(s,a)$
$s_{\{r,r1,y1,g1\}}$	$a_{move\_to\_red}$	-0.1
$s_{\{r,r1,y1,g1\}}$	$a_{move\_to\_yellow}$	-0.09
$s_{\{r,r1,y1,g1\}}$	$a_{move\_to\_green}$	1.0
$s_{\{y,r1,y1,g1\}}$	$a_{move\_to\_red}$	0.9
$s_{\{y,r1,y1,g1\}}$	$a_{move\_to\_yellow}$	-0.1
$s_{\{y,r1,y1,g1\}}$	$a_{move\_to\_green}$	1.0
$s_{\{g,r1,y1,g1\}}$	$a_{move\_to\_red}$	1.0
$s_{\{g,r1,y1,g1\}}$	$a_{move\_to\_yellow}$	-0.09
$s_{\{g,r1,y1,g1\}}$	$a_{move\_to\_green}$	-0.1

Table 7.1: An example of Q-values using enumerated states.

One can from this table see that the optimal policy are as described as the example in section 6.2. If the agent should happen to enter the yellow patch it is to move to green.

7.1 Size Factor

An important factor to keep in mind when working with a tabular implementation is the sheer size of the of the domain. The example state-space above had 24 unique states with 3 possible actions, resulting in 74 unique state-action pairs. While this is a relatively low number, more complex environments could have thousands, millions, if not trillions of unique state-action pairs, making the implementation much more complex, and much more space consuming in terms of memory.
In addition, the larger the set of state-action pairs, the longer it will take for the agent's behaviour to converge towards an optimal policy. This is because that in theory, to find an optimal policy an agent has to discover which action $a \in A$ is the best to pick in every state $s \in S$. An agent will take longer to converge towards an optimal behaviour the bigger its domain is, and as such, a tabular solution is not optimal for very large domains. Besides it is not practical to store such amounts of data.

Chapter 8

Feature-based Q-Learning

This chapter will describe the concept of features in reinforcement learning.

Where as the idea of finite state Q-Learning described in chapter 7 enables an agent to learn a policy with completely individual options for each individual state, the approach is not practical for very large state spaces, and impossible for state attributes with non-finite value ranges.

Reinforcement learning with features does not rely on having a stored set of Q-values for each state-action pair. Instead, it calculates the Q-value of a state-action pair on the spot, using a function of the features (In our situation a linear one, although it is possible to make different functions).

8.1 Features

A feature can be seen as a function of a state, which describes a specific piece of information about that state. As an example, in a racing car scenario a feature could indicate the distance to the finish line, while another feature could indicate the speed of the car.
As such, the state can be seen as a set of feature values, the values of each of the feature functions at the given point. It is not necessary for features to be independent of each other. For instance, the features **percentage of women in the room** and **percentage of men in the room** would have to add up to an even 100%, and if one were to increase the other one would have to decrease as a result.
Another example of connected features could be a feature indicating the **danger level** of a situation which could depend both on the **health of the agent**, and the **amount of fire in the area**, which could be described by two separate features.

8.2 Q-values

As mentioned in the introduction to this chapter, Q-values for a state-action pair in feature learning are not looked up in a table, but are instead calculated when needed. In a state space with n features, the Q-value for a state action pair Q(s,a) is calculated as follows:
$Q(s,a) \sum_{i=1}^{n} w_{i,a} \times F_i(s)$
Where $F_i(s)$ is the value of the ith out of n features in the state s, and $w_{i,a}$

Feature	Weight for action a_1	Weight for action a_2
F_1	0.37	3.14
F_2	1.20	−2.31

Table 8.1: An example of the values of weights for a set of features and actions.

is the weight corresponding to the feature F_i and the action a. The weight itself is a variable with a decimal value. An example calculation is provided in section 8.3.

8.3 Weights

In reinforcement learning with features, the goal is to find the values for each of the weights for each feature, so that the agent performs according to an optimal policy. While the feature values describe a state, they do not directly describe the value of being in a certain state s, or the value of performing action a in state s. Instead, the weights do this, by *weighting* the feature values against the actions. What this means, is that for each action, each feature has a weight which indicates the "weight" this feature should have in deciding whether to perform this action or not. To illustrate this, we use an example with a state space with two features (F_1 and F_2), and an action set of two actions (a_1 and a_2). Assume the agent has learned the weights shown in Table 8.1.
Given a state s with feature values $F_1 = 0.5$ and $F_2 = 0.9$, the Q-values are calculated as follows:
$Q(s, a_1) = w_{1,a_1} \times F_1(s) + w_{2,a_1} \times F_2(s) = 0.37 \times 0.5 + 1.20 \times 0.9 = 1.265$
$Q(s, a_2) = w_{1,a_2} \times F_1(s) + w_{2,a_2} \times F_2(s) = 3.14 \times 0.5 + (-2.31) \times 0.9 = -0.509$
In this case, we can see that in state s, it would be best to chose action a_1, as the state-action pair has the highest Q-value of the two.

8.3.1 Weight Adjustments

As mentioned above, the approach for using feature-based learning relies on learning the correct values of the weights. As with the idea of state-based Q-Learning described in chapter 7, the Q-values are updated after each state-action-state transition. However, as the weights are updated and stored, rather than the Q-values themselves, the approach is a bit different. For each Feature, the weights corresponding to the action that was picked have their values updated using gradient descent, a term which will be described in the next section: $w_{i,a} = w_{i,a} + \eta\delta F_i(s)$
$\delta = r + \gamma max_{a'} Q(s', a') - Q(s, a)$
Where η is the gradient descent step size and γ is the discount factor.

8.3.2 Gradient Descent

Similar in nature to Newton's Method [7, p. 35], the idea behind gradient descent is to reach a certain point on a graph by iteratively descending the gradient of the current best estimate of the point.
The approach used here for updating the values of weights is a bit different from traditional use of Newton's method or gradient descent though. Primarily, we do not use the value of a gradient's slope obtained through derivation, but instead approximate the slope using a delta value δ. A delta value is one indicating the difference in value between two measurements. In our case, the delta value is calculated as
$\delta = r + \gamma max_{a'} Q(s', a') - Q(s, a)$
and as such, it would have been more appropriate to call the approach gradient ascent, since the weight will be adjusted towards the direction of the slope of gradient, rather than in the opposite direction, which would be required to find a low point.
What this means is that with gradient ascend, a *positive* slope means we continue in a *positive* direction on the x-axis towards the local maximum, while a *negative* slope using gradient descend would mean we continue in a *positive* direction on the x-axis towards a local minimum. This can be seen in Figure 8.1, where the current estimate is marked x, and the following estimate is marked $x + 1$ for both gradient ascend and gradient descent.

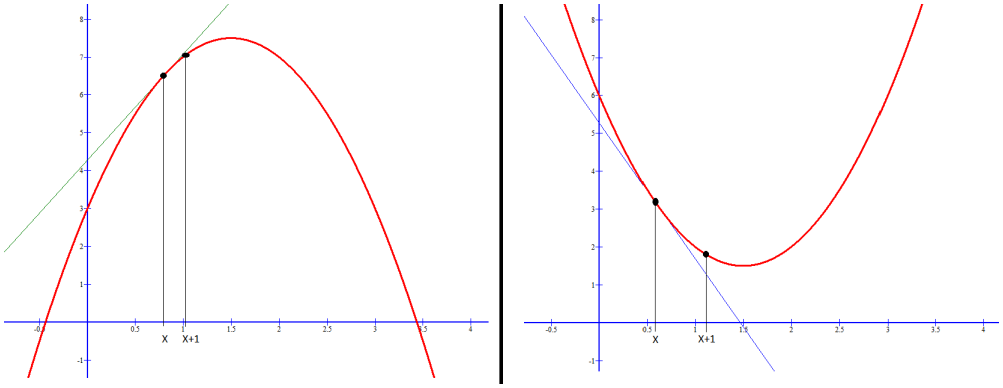


Figure 8.1: Graphs illustrating gradient ascent on the left, and gradient descent on the right.

As we can see, while the approach used here and described in [2] is not identical to the mathematical model, it accomplishes the same goal of making the weight values converge.

Gradient Descent Step Size

On a final note, we must remember the gradient descent step size. The purpose of this value is to control and adjust convergence. Too small a value, and the system will be very slow at reaching the point of convergence, and too large a value and it will continuously overstep it.

Chapter 9

The Combined Approach

This chapter contains a description of the Combined Approach, which is the main point of this project.

By combining Feature-based Q-learning and tabular Q-learning we hope to enable an agent to transfer knowledge from one situation to another, while still enabling it to learn specific behaviour for the given specific situation.
We intend to use tabular Q-learning to allow the agent to learn behaviour specific to a given situation, and use Feature-based Q-learning to learn general behaviour. The idea is then that if the agent has learned general behaviour from some situation(s), the agent will will benefit from it when coming to a new situation, more so than starting with no prior knowledge at all.
As the two different sides represents two different parts of the agent's understanding of its situation, they each have their own rewards. As such, performing an action which earns a reward on the feature side does not mean that the agent necessarily earns the same reward on the tabular side.

9.1 Basics

First, we need to clarify a few things:

- $s_1 \in S_1$ indicates a state from the state-space covered by the tabular representation.
- $s_2 \in S_2$ indicates a state from the state-space represented by features.
- On its own, $s \in S$ represents a state of the full state-space, that is to say the product of the divided states: $s = s_1 \cdot s_2$
- s' indicates the state that the agent enters while performing action a in state s. s'_1 is the part of s' seen by the tabular side, while s'_2 is the part seen by the feature side.
- r_1 indicates a reward assigned to the tabular side.
- r_2 indicates a reward assigned to the feature side.
- $w_{i,a}$ represents the weight corresponding to the feature F_i and the action a.
- γ is the discount factor $0 < \gamma < 1$.

- α is the learning rate $0 < \alpha < 1$.

- η is the gradient descent step size, which is a constant $\eta > 0$.

It is normal to implicitly read Q as being the maximum expected reward. In our case there are a few places where it is not so. We will therefore note when we use maximum and when we do not.

9.2 Decision Making

In order to decide which action to pick, the agent will consult both the tabular and the feature side. As such, for each action the Q-value is:
$Q(s, a) = k_1 Q(s_1, a) + k_2 Q(s_2, a)$
$k_1 + k_2 = 1$
The reason for using the constants k_1 and k_2 is that we do not want the combined value to be much bigger than its parts, as that would cause problems with Unified-Q and Unified-a. In the current version of the implementation, both weights are 0.5 to weight the equally in the decision making.

9.3 Approaches for Q-value Updates

During initial analysis and design, we came up with 3 different approaches for updating the Q-values. The reason behind these different approaches is that there is no standard approach for combining tabular-based Q-learning and feature-based Q-learning, as such a combination is not a standard.
In the sections subsection 9.3.1 through subsection 9.3.3 we will look at the different approaches.

9.3.1 Separate Update

The idea behind the Separate Update is that if the two state-spaces are sufficiently different, we would not expect them to converge to a single policy. As such, the idea of this approach is that the Q-functions for the individual state-spaces are updated based on individual values. That is to say that the best possible action will be picked for the given state-space, and the corresponding Q-value will be used as an estimate for the expected reward.

$Q(s_1, a) = (1 - \alpha)Q(s_1, a) + \alpha(r + \gamma max_{a_1'} Q(s_1', a_1'))$
$w_{i,a} = w_{i,a} + \eta \delta F_i(s_2)$
$\delta = r + \gamma max_{a_2'} Q(s_2', a_2') - Q(s_2, a)$

9.3.2 Unified-Q Update

The idea behind the Unified-Q Update is that the maximum expected future reward is based on the Q values from both state-spaces. $max_{a'}Q(s', a'))$ represents the highest value of the sum of expected future rewards for both state-spaces over the best possible action a'.

$Q(s_1, a) = (1 - \alpha)Q(s_1, a) + \alpha(r_1 + \gamma max_{a'}Q(s', a'))$
$w_{i,a} = w_{i,a} + \eta\delta F_i(s_2)$
$\delta = r_2 + \gamma max_{a'}Q(s', a') - Q(s_2, a)$

9.3.3 Unified-a Update

The idea behind the unified-a update is to use the same action for updating the Q values as the one that would be picked in s. This means the action which gives the highest unified value, but otherwise use the individual expected values for the update. It differs from unified Q in that it uses the individual expected rewards to update the value.
It is also worth noting that this is the rare case where we don't necessarily choose the action with the highest individual value. The value of either the tabular or the feature based representation might be lower than other possibilities as long as the unified value is the highest option.

$Q(s_1, a) = (1 - \alpha)Q(s_1, a) + \alpha(r_1 + \gamma Q(s'_1, a'))$
$w_{i,a} = w_{i,a} + \eta\delta F_i(s_2)$
$\delta = r_2 + \gamma Q(s'_2, a') - Q(s_2, a)$
where
$Q(s', a') > Q(s', a'')$
$a' \neq a''$
$\forall a'' \in A$

9.4 Transferring Knowledge

One of our ideas behind the combination of tabular Q-learning and feature-based Q-learning is to enable the agent to transfer knowledge from one scenario to another. To understand the idea of knowledge transferring, consider a tabular implementation with a large state-space. If we imagine an agent working in a situation described with this large state-space, it would be able to learn an optimal policy for that specific set-up. Now imagine the agent is put in another situation described by the same large state-space. These two situations are not the same, and as such the states the agent observes in the first situation are not the same as the ones it will observe in the new situation.

In this case, the agent does not benefit from having been in the first situation due to the nature of tabular Q-learning, where an agent has to have observed each unique state in order to learn an optimal behaviour. As such, there is no knowledge transfer in tabular Q-learning.
With feature-based Q-learning, knowledge transfer is possible, as the agent does not need to have found the Q-values for each possible feature value combinations, but rather on the weights assigned to each feature-action pair, which is a much smaller set.
However, as the Q-values in a feature-based implementation are based on a linear function of the features, such an implementation lacks the precisions in unique states that a tabular implementation provides.
We believe that by combining a tabular approach with a feature-based approach, we can develop an agent which can transfer knowledge from one scenario to another, while still being able to learn an optimal policy for each scenario. We believe that in learning general behaviour represented on the feature side in one scenario and then transferring that knowledge to another scenario, the agent will have a better starting point in the other scenario, compared to if it did not have this transferred knowledge.

Part III

Components

Chapter 10

Overview of the system

This chapter provides an overview of the system, along with information required for better understanding the following chapters

This chapter serves as an introduction to the component part of the report. That is to say the description of the different components that the implementation of the system consists of.
As this is an introductory chapter, the descriptions of the different components will be brief, and there will be forward referencing to show where these components are described in more detail.

10.1 Why do we use Minecraft?

To start off, we felt that it would be a good idea why we have used the video game Minecraft as a platform for our implementation. The decision to use Minecraft goes back to our previous semester, where we decided that it would be interesting to work with an existing platform, and an existing user community. Minecraft felt like a good choice for this, since Mojang[1] allows for modding, and the game world itself consists of blocks, making it similar to the grid worlds used in the reinforcement learning examples we have seen.
As we had used Minecraft as a platform in the previous semester, we decided to also use it this semester, so that we could expand on the work we had already done, and allow us to do more than if we were to start from scratch.

10.2 Conceptual System Structure

Before we get too technical, we are going to show the general idea of how the system is designed. Standard Minecraft does not use reinforcement learning, and as such we need an interface between our reinforcement learning components, and the Minecraft platform. Figure 10.1 shows how our system uses two components to interact with the Minecraft environment: A sensory module, and an actuator module.
The sensory module is used to obtain the state from the environment, that is to say transform the game-state into the correct reinforcemnt-learning-state, which includes detecting the correct values for each state-attribute and each feature.
At this point, the observed state along with reward obtained for performing

[1] The company behind Minecraft

the action leading to this state is given to a component called the reinforcement module.
This module is responsible for updating the tabular Q-values and feature weights of the previous state-action pair, and to decide which action to perform next based on the newly observed state, as described in chapter 9.
Finally The actuator module is designed to perform the action that the reinforcement module has decided on. It does this by giving the SmartDog entity specific instructions of how to perform this action in the Minecraft environment.
When the action has been performed, the model has gone full circle, and the sensory module will observe a new state, and so on.

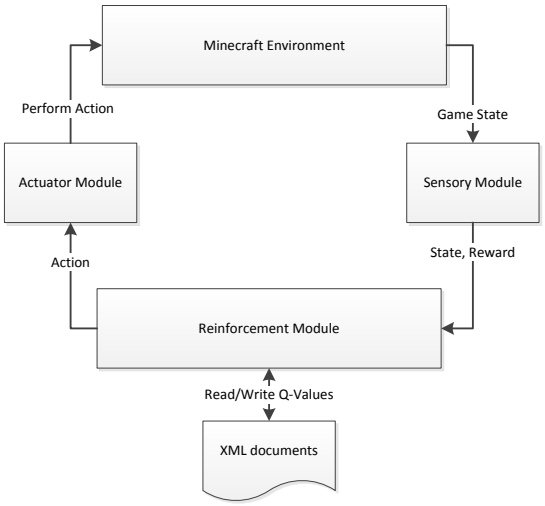


Figure 10.1: Figure showing the concepts of how the main reinforcement learning components interact.

10.3 System Flow Chart

Getting a overview of how the system works technically can be done by looking at the cycle that the smartdog entity constantly iterates through. This flow can be seen in Figure 10.2. We here see the never-ending flow for the smartdog. It starts out with running the sensory module to see if a state-change has occurred. If it has, the system will wait some time to gather rewards and then go through the notions of updating Q-values and weights, make a new decision on what action to perform, and then initiate said action. If there is no state-change however, we need to check whether

the smartdog has stopped moving, a trick we utilise to know whether it has finished the movement part of an action. For now "no" means that the agent has started an action and is still moving, which is why we in such a case go back and look for either a state-change or that the smartdog stops moving. If it has stopped moving, it means that we can check whether the agent has initiated the second part of an action, and then initiate it if not, and check if it is done if we have initiated it, to finally be able to update, decide and begin a new action. Most of these checks that lead back to another iteration are due to Minecraft being a virtual real-time system and we therefore needs to wait for things to happen in the environment. We discuss this further in chapter 12. Something to notice is that if the smartdog is moving, and while it moves the state changes, the agent will stop, update, decide on a new action and begin it. This means that taking an action does not necessarily mean that the smartdog gets to complete the action. On the reinforcement side, we treat this as a completed action. An action which just failed, but with no consequences on the reinforcement side of it. This was an early design choice since the environment is non-deterministic.

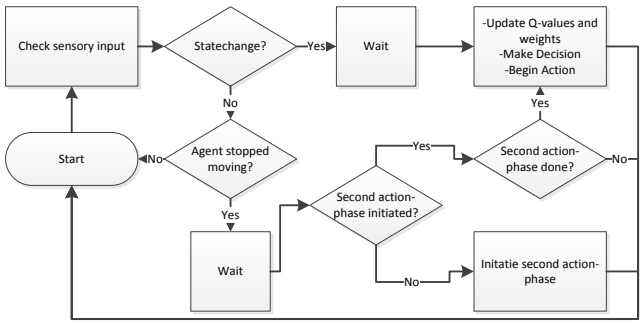


Figure 10.2: Figure showing the flow of the system.

Continuing our example from the theory part, we can show a cycle where the smartdog notices a change in the state, that it is now on red floor. It waits, and updates the Q-values and weights. It then decides that the best thing to do is to move to the green floor. Checks it sensors again, but no change just yet and the agent is still moving, so it reiterates. It keep reiterating until it notices a state-change when it enters the green floor, where it gets a reward, updates, takes the decision to go to the red floor, and starts the action.
Other actions,such as push button, could have completed its movement to the button, and then completed the action of pushing it, but the "move to

colour" actions will never succeed as they per definition will get a state-change before their action is done. This is not a problem, as this also effectually means that those specific actions are done.

10.4 The EntitySmartDog Class

The EntitySmartDog class is the one which handles the smartdog entity as the name suggests. What's more interesting though is that it is here that we both connect to the Minecraft environment as well as organise and call the rest of our system. The class inherits from the Minecraft EntityTameable class which means that the smartdog entity are connected directly into the Minecraft environment and its game ticks. When we then create a sensor, actuator and reinforcement object for the entity, and call these when the smartdog are updated by Minecraft through the method onLivingUpdate, which we will talk about in chapter 12. The flow of this method are also the one we depict in Figure 10.2, and is how our system flows. It is therefore that the EntitySmartDog class is the connecting point of our system. We have build it with modularity in mind, so that if we where to change system we could still use our three modules. If a module needed to be changed we could more or less do so with only minor trouble. We have additionally designed the sensor and actuator module such that new actions and sensors can easily be added and existing actions and sensors can be changed. We have also made it so that the smartdog can respawn on user-defined positions so that it will be easier to test our smartdog.

10.5 The Following Chapters

The following chapters in this part of the report all describe different components of the system. They are written in such an order as to facilitate the understanding of the system.
We start out by describing the state space, and explaining the real-time perspective of the system. We then present the components related to sensing, deciding and performing, and finish off by describing data management and testing.
In the chapters where we expand on previously existing components, we describe the changes briefly at the end of each chapter.

10.5.1 Source Code

In the chapters which use source code. we have used the notation "..." to indicate that code has been omitted. We use this to omit code in order to fit the code snippets to the page, and only omit code which is not necessary

in order to present the functionality of the component.
We use standard Java comments, utilising single- and multi-line comments when needed. An example of this can be seen in Listing 10.1.

```
1 //This is a single line comment.
2
3 /* This comment uses
4  * multiple lines
5  */
6
7 ... //This indicates omitted code.
```

Listing 10.1: An example of comments and omitted code.

Chapter 11

State and State Space

This chapter contains a description of the statespace used in the system, and how its different parts are designed.

As described in chapter 9, we combine the tabular state-space and the feature-based state-space to form a combined state-space.
The following sections will describe how state-attributes and features are handled in the implementation, and how the state handles which actions are possible.

11.1 State-Attributes and Features

In this implementation, we use states from the combined state-space to hold information for both the tabular states, and the feature-based states.
The values for each of the state attributes and features are obtained through the sensory module. See chapter 13 for more information.
The state contains the state-attributes shown in Table 11.1.

What block was I on? is an attribute which indicates which of the *interesting* blocks the agent is either standing on, or has been on previously. The blocks that are considered *interesting* are the red block, yellow block, green block, hostile block, and the hopper. If the agent steps onto any of these, the value of that attribute will change. Otherwise the value will stay the same as the one it had in the previous state.
The reasoning behind this design choice is because we stops the agent if the state changes and makes it decide upon a new action. If we did not have it like this, it would decide upon a new action each time it enters or leaves one of the interesting blocks. If we did not stop the agent mid-way because of state-changes and only noticed what state it had after performed action

State-Attribute	Number of Possible Values
What block was I on?	5
Are there red blocks nearby?	2
Are there yellow blocks nearby?	2
Are there green blocks nearby?	2
Are there hostile blocks nearby?	2
Are there hoppers nearby?	2
Are there buttons nearby?	2

Table 11.1: The state-attributes, and their number of possible values

Feature	Number of Possible Values
Health	20
Are there edible items nearby?	2
Are there non-edible items nearby?	2
Is there something in my inventory?	2

Table 11.2: The features, and their number of possible values.

we could have this attribute as "what block am I on".
The other attributes have boolean values, which are either `true` if there are one or more of the specific block nearby, or `false` otherwise.

In addition, the state contains the features shown in Table 11.2.

It is worth noting that the values v of each of the features are in the interval $0 \leq v \leq 1$.
Theoretically, features can have countably infinite value ranges, but in our current implementation, they have countably finite value ranges. This is not because we needed to limit them, but because we have intended for some of the boolean valued features to be more expressive in the future.

11.2 Possible Actions

As something new this semester, the state is also in charge of showing which actions are possible to perform in that state, and which are not. As opposed to the previous semester, where the agent could attempt to move to red blocks even though there were none nearby, in the system from this semester, the agent can only perform actions which have a chance of succeeding. For this purpose, there is a method called `GetPossibleActions()` which returns a list of the actions that are possible to perform in the state. Note that the method returns a list of byte values. The reason for this is that each action $a \in A$ is represented by a byte value.

```
1  public ArrayList<Byte> GetPossibleActions()
2  {
3      ArrayList<Byte> actions = new ArrayList<Byte>();
4      boolean shouldWaitingBeAllowed = true;
5
6      if (GetFeatureValue(BasicInfo.
           FEATURE_IS_THERE_SOMETHING_IN_MY_INVENTORY) == 1)
7          {
8          actions.add(BasicInfo.ACTION_DROP_ITEM);
9          shouldWaitingBeAllowed = false;
10     }
11     if (GetFeatureValue(BasicInfo.
           FEATURE_ARE_THERE_NON_EDIBLE_ITEMS_NEARBY) == 1  &&
```

```
        GetFeatureValue(BasicInfo.
        FEATURE_IS_THERE_SOMETHING_IN_MY_INVENTORY) == 0)
12      {
13          actions.add(BasicInfo.ACTION_PICK_UP_NON_EDIBLE_ITEM);
14          shouldWaitingBeAllowed = false;
15      }
16      ...
17      if (GetFeatureValue(BasicInfo.
        FEATURE_IS_THERE_SOMETHING_IN_MY_INVENTORY) == 1 )
18      {
19          //FIXME this should be changed so we can determine if
                the inventory is edible
20          actions.add(BasicInfo.ACTION_EAT_ITEM);
21          shouldWaitingBeAllowed = false;
22      }
23      ...
24      if (areThereButtonsInSight == true)
25      {
26          actions.add(BasicInfo.ACTION_ACTIVATE_BUTTON);
27          shouldWaitingBeAllowed = false;
28      }
29      if (areThereRedCustomBlocksInSight == true)
30      {
31          actions.add(BasicInfo.ACTION_MOVE_TO_RED);
32          shouldWaitingBeAllowed = false;
33      }
34      ...
35      if(shouldWaitingBeAllowed == true)
36      {
37          actions.add(BasicInfo.ACTION_WAIT);
38      }
39      return actions;
40  }
```

Listing 11.1: The method `GetPossibleActions`

As seen in Listing 11.1, the method starts with an empty list, and adds actions to it when it determines that they are possible. For istance, the agent can only perform the action **Activate Button** if there are buttons nearby, and it can only perform the action **Eat Item** if it has something in its inventory. Finally, the agent is only allowed to perform the action **Wait** if none of the other actions are possible. This is to prevent errors in the system where the agent would be unable to pick a valid action.

11.3 Changes from previous semester

This component has been modified to contain information for both the tabular-state, and the feature-based state. The state-attributes continue to have a fixed number of possible values, while the feature values are

less limited, allowing any value between 0 and 1. In addition, the method `GetPossibleActions` has been added, which determines which actions can be performed in the current state.

Chapter 12

Virtual Real-Time Perspective

This chapter describes how the system handles the real-time aspects of the Minecraft platform.

One of the things that we have been troubled with is that Minecraft is a virtual real-time system. It is not legal to make changes to Minecraft's own code[1], but completely fine to create mods which access it from the outside. We use the programming interface to Minecraft called Forge[2], which uses Scala[3] to connect with the Minecraft code, making it possible for programmers to use Java to write their mods without modifying the Minecraft source files.
But this also means that what is available is a virtual real-time system, as the developer will have to work with the platform, rather than be able to modify it.
We can read in the code that Minecraft runs in its own game ticks, which are used to simulate time in the game world. We have also seen that all entities have a method called textttonLivingUpdate, which runs "often" in the Minecraft environment - about 20 times a second. It is not guaranteed to have exactly 20 ticks per second, but it is the closest we can get to the game tick.
All creature entities have this method, which is called as long as the entity is alive. The method is called to make the entity "do something", which could be moving to somewhere, making a sound, or other things. We can use this method by creating our own smartdog entity, and run the agent's components from the method.
This is where all our code starts. `onLivingUpdate` is where our system is connected to Minecraft, and where the smartdog's senses are used, decisions are started, and actions performed. We use the dog's senses "often", about 4 times a second, and start the decision process if its senses tell us that there is a state change or that the smartdog is done with its former action. Actions are started when a decision has been made.

[1] According to [6]
[2] A community made API:
`http://www.minecraftforge.net/forum/`
[3] An object-functional programming and scripting language:
`http://en.wikipedia.org/wiki/Scala_%28programming_language%29`

12.1 Using the `onLivingUpdate` Method

The method `onLivingUpdate` is very relevant to how we handle the real-time platform. As mentioned above, `onLivingUpdate` runs 20 times a second, but we do not need to run our part as often. Therefore, we have set in a counter **counterForOnLivingUpdate** which is used in Listing 12.1, line 7 and 8. Besides this, we have set in a delay from when a smartdog has finished its action or the state has changed, until it decides on a new action. This is because deciding on a new action will also update the Q-values and feature weights of the former state-action pair. To do so correctly all rewards given for the action performed must have been obtained, but since they are given in an environment that might have a delay, we have a delay as well, so that we collect the rewards for the right action. The variable **counterForDelayingThoughts** is this delay. We have 3 situations, one where we are ready to make a new decision and only need to wait for the delay, one where we count the delay up, and one where we need to make the decision.
The first situation can be seen in Listing 12.2, and the last situation can be seen in Listing 12.3. The second situation simply consists of incrementing a counter, so there is no point in highlighting it further.

```
1  public void onLivingUpdate()
2  {
3    super.onLivingUpdate();
4   ...
5   if (!this.worldObj.isRemote)
6   {
7     counterForOnLivingUpdate++; //This is to do our code less
          frequent.
8    if (counterForOnLivingUpdate >= 5)
9    {
10     if (counterForDelayingThoughts == -1)
11     { //This happens when a action is completed and rewards has
           been collected.
12       //that is to say that the state has changed or the action
             is done, and we need to figure out which,
13     //and start the counterForDelayingThoughts.
14     ...
15     }
16     else if (counterForDelayingThoughts < BasicInfo.
           THOUGHT_DELAY_MAXIMUM)
17     { //This is the delay so that the dog can retrieve rewards.
18     counterForDelayingThoughts++;
19     }
20     else if (counterForDelayingThoughts >= BasicInfo.
           THOUGHT_DELAY_MAXIMUM)
21     { //This is after the delay.
22     //Here we make a decision and start an action, or do the
```

```
        second part of an action.
23      }
24      counterForOnLivingUpdate = 0;
25    }
26    }
27  }
```

Listing 12.1: The simplified method `onLivingUpdate`

So to begin with we are in a situation where a new decision needs to be made, which can be seen in Listing 12.2. This can be caused by the state changing, or the smartdog having performed the moving part of its action. In either case we notice which with a flag and starts the delay counter.
In the case of state change, we also check whether the smartdog has died (as seen in Listing 12.2, line 9), since we need to save the data before Minecraft notices that the entity is dead and removes its thread. Normally this is not a problem since Minecraft takes a while to do so, but with the delay this might become a problem, which is why we ignore the delay in the case where the dog dies.

```
1  if (counterForDelayingThoughts == -1)
2  { //This happens when a action has completed and rewards has
       been collected.
3   //that is to say, a new decision needs to be made.
4   if (smartdogSensor.CheckSensoryInput(this))
5   {//If the state has changed
6    counterForDelayingThoughts = 0; //starts the delay
7    endedDueToStateChange = true;
8    //If the dog is dying, the data needs to be saved before
        minecraft finds out about the death, and kills the thread
9    if(smartdogSensor.GetCurrentState().GetIsTheDogDead())
10   {
11    smartdogReinforcementModule.MakeDecisionEGreedy(
          smartdogSensor.GetCurrentState(), smartdogSensor.
          GetTabularRewardAndReset(), smartdogSensor.
          GetFeatureRewardAndReset());
12    }
13  }
14  else if (this.getNavigator().noPath())
15  { //This is if the dog has ended its move phase
16   counterForDelayingThoughts = 0;
17   endedDueToStateChange = false;
18  }
19 }
```

Listing 12.2: Part of the `onLivingUpdate` method

Hereafter we have the situation after the delay where we want to make the decision, which is seen in Listing 12.3. We first want to get the newest state to use as our s' by running the sensory module. If the action has been

stopped prematurely due to a statechange, we make a new decision and start the action from this decision.If it stopped due having finished moving, we will use the first iteration to signal the actuator to finish its action. This could for instance be activating a button it has moved to. If in the next iteration it has completed its full action, we make a new decision and signal the actuator to perform the action decided upon. Otherwise, agent will just wait until the next iteration and check again.

```
1  else if (counterForDelayingThoughts >= BasicInfo.
       THOUGHT_DELAY_MAXIMUM)
2  { //This is after the delay.
3   counterForDelayingThoughts = -1;
4   //We make sure that s' is the newest state
5   smartdogSensor.CheckSensoryInput(this);
6   if (endedDueToStateChange)
7   {
8    //This line determines what action to perform as well as
         sending the state and reward to the reinforcement module
9    byte actionToPerform = smartdogReinforcementModule.
         MakeDecisionEGreedy(smartdogSensor.GetCurrentState(),
         smartdogSensor.GetTabularRewardAndReset(), smartdogSensor
         .GetFeatureRewardAndReset());
10   smartdogActuator.PerformAction(actionToPerform, this);
11   standardActionLocked = false; //So that the dog can perform
         actions again, if an action are interrupted by a
         statechange.
12   //Update the number of actions taken
13   ...
14  }
15  else
16  {
17   if (standardActionLocked == false)
18   {
19    standardActionLocked = true;
20    this.smartdogActuator.FinishAction(this);
21   }
22   //If the dog has just started finishing its action, it should
         not also manage to finish it in the same iteration, so
         that it waits before making a new decision
23   else if (this.smartdogActuator.GetAndResetIsActionDone()) //
         This is when the full-round action has been performed
24   {
25    //This line determines what action to perform as well as
         sending the state and reward to the reinforcement module
26    standardActionLocked = false;
27    byte actionToPerform = smartdogReinforcementModule.
         MakeDecisionEGreedy(smartdogSensor.GetCurrentState(),
         smartdogSensor.GetTabularRewardAndReset(),
         smartdogSensor.GetFeatureRewardAndReset());
28    smartdogActuator.PerformAction(actionToPerform, this);
29    //Update the number of actions taken
```

```
30      ...
31    }
32   }
33 }
```

Listing 12.3: Part of the `onLivingUpdate` method

12.2 Synchronisation

This section is inspired by [4, Chapter 11] As mentioned before we are working with a virtual real-time system, and have limited access to Minecraft's code. One of the problems we have encountered is that we could not determine when an action was done. We could not directly access when the smartdog was done and had finished moving to its destination. Therefore, we use a periodic check to see if the agent *has stopped* moving, as opposed to being able to listen for an event triggered when the agent *stops* moving. To make sure that all actions are found, also those that does not include moving, we make the dog make a minuscule jump, not visible to the player but visible to the sensory module. This way we always know when the smartdog is done moving[4, Chapter 11]. We also found that we need more time than $\frac{1}{20}$th of a second to collect the rewards, which we handle with the delay.

12.3 Changes in The Entity Smart Dog Class

This class has been modified primarily to slow the agent down in order to ensure that rewards are given to the correct state-action pair. It has also gotten the addition to finish actions since we have added actions that does other things than simply moving.

Chapter 13

Sensory Module

This chapter contains a description of the sensory module used by the agent.

The sensory module is the eyes and ears of the agent. Without it, the agent would be blind and deaf in its decision making. The purpose of the sensory module is to provide the agent's decision module with a fitting description of the agent's situation. This description is represented by a state.

13.1 Checking for State-Changes

In the typically presented implementations of reinforcement learning, the system is turn based. That is to say the agent will be in some state s, perform action a, and end up in some state s', earning a reward r. The transition (s, a, r, s') is instantaneous, in that nothing happens in between. In the real time model we are using, the transitions are not instantaneous. If for instance the agent would move from one block to another, an amount of time would pass in between being in state s and reaching state s'.
The question is then how often the sensory module should check for state changes. A simple approach would be to check for state changes after having performed action a. However such a solution would remove the possibility to detect state-changes happening before an action has been completed. An example of such a state-change is seen in Figure 13.1, where an agent picking **Move to green**, while standing on a red block would have its action interrupted by moving over the yellow block in the middle, as that would cause a state change.
If we check the system for state-changes too often however, the system could suffer from performance issues, as checking the values of state-attributes and features can be taxing.
As such, the sensor checks for state changes periodically, at about 4 times per second. The method used in checking this can be seen in Listing 13.1. Every time this method is called, it will check the current values of each of the state-attributes and features. At the bottom of the method, it will compare the current values to the previous values, and if they are different it will have found a state-change, and will return `true`. It is worth noting that the check for whether the current state is equal to the previous state is based on the values of the state-attributes, and not the feature values. The reason for this is that if we were to introduce features which were to be based on either distance or time, their values could theoretically change

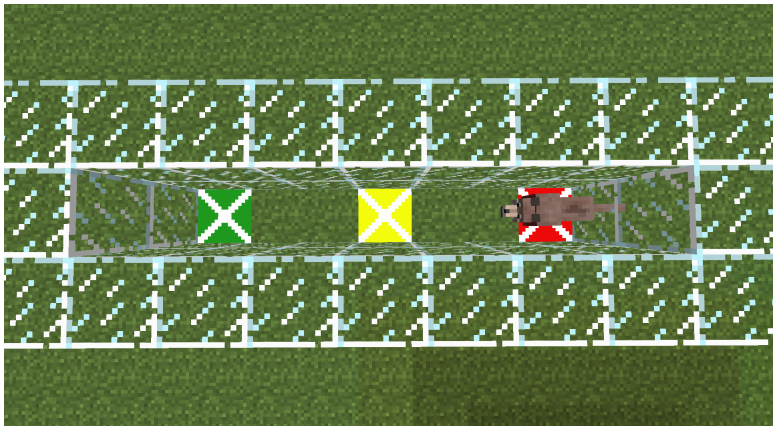


Figure 13.1: Picture showing a situation where the action can be interrupted.

between each check, and the agent would have its actions interrupted far more often.
We have considered implementing a version in the future where a certain change in feature values will merrit a state change, but that is not in the current version.

```
1  public boolean CheckSensoryInput(EntitySmartDog entitySmartdog)
2  {
3      updateFlag = false;
4      oldSmartDogState = smartDogState;
5      smartDogState = new SmartDogState();
6      CheckWhatBlockWasIOn(entitySmartdog.posX, entitySmartdog.
           posY, entitySmartdog.posZ, entitySmartdog.worldObj);
7      CheckForNearbyBlocks(entitySmartdog.posX, entitySmartdog.
           posY, entitySmartdog.posZ, entitySmartdog.worldObj);
8      CheckAmIHealthy(entitySmartdog);
9      CheckAreThereItemsNearby(entitySmartdog);
10     CheckAmIHoldingSomething(entitySmartdog);
11
12     if (!oldSmartDogState.equals(smartDogState))
13     {
14         updateFlag = true;
15     }
16     return updateFlag;
17 }
```

Listing 13.1: The method `CheckSensoryInput`.

13.2 Obtaining a state

In order to obtain the appropriate values for the state-attributes and features, the sensory module uses several different methods to obtain the values

of each of these as seen in Listing 13.1.
Each of these methods relate to a different amount of state-attributes/features. For instance, the method `CheckAmIHoldingSomething` only affects the feature **Is there something in my inventory?**, while the method `CheckForNearbyBlocks` affects the attributes: **Are there red blocks nearby?**, **Are there yellow blocks nearby?**, **Are there green blocks nearby?**, **Are there hostile blocks nearby?**, **Are there hoppers nearby?**, and **Are there buttons nearby?**.

13.2.1 Health

The method `CheckAmIHealthy` is somewhat different from the other sensor methods in that, not only does it find the value relating to the health of the agent, it also assigns a reward relating to the change in health, and signals the system if the smartdog is dying.

```
1  public void CheckAmIHealthy(EntitySmartDog entitySmartDog)
2  {
3      float health = entitySmartDog.getHealth();
4      if (health <= 0)
5      {
6          health = 0; //Just in case the health should go below
               the expected limit
7
8          if (amIDying == false) //The dog should only be
               punished once per death
9          {
10             AddFeatureReward(BasicInfo.REWARD_FOR_DYING,
                   entitySmartDog);
11             //NB We considered adding the reward to the Tabular
                    part as well, but decided not to, as the
                   attributes do not describe the health of the
                   dog, and as such can not tell if it is close to
                    dying.
12             /* It might have worked in the first scenario, but
                   can not be guaranteed to work in all scenarios
13              * The tabular part represents what the dog has
                    learned from the player, while the features
                    corresponds to its survival instinct
14              */
15             amIDying = true;
16             smartDogState.SetIsTheDogDead(true);
17             String ownerName = entitySmartDog.getOwnerName();
18             SmartDog.RespawnDog(ownerName, entitySmartDog.
                   worldObj, entitySmartDog.GetSpawnX(),
                   entitySmartDog.GetSpawnY(), entitySmartDog.
                   GetSpawnZ());
19         }
20     }
21     else if (health > 0)
```

```
22      {
23          amIDying = false;
24      }
25
26      if (health > 20) //Just in case the health should go past
            the expected limit
27      {
28          health = 20;
29      }
30
31      //Reward/punish the dog according to its change in health
32      //This is done on the feature part only
33      double healthReward = health - previousHealth;
34
35      if (healthReward > 0.0)
36      {
37          AddFeatureReward(healthReward * BasicInfo.
                REWARD_FOR_HEALING_HALF_HEART, entitySmartDog);
38      }
39      else if (healthReward < 0.0)
40      {
41          AddFeatureReward((-1*healthReward) * BasicInfo.
                REWARD_FOR_LOSING_HALF_HEART, entitySmartDog);
42      }
43
44      double featureHealthValue = 1 - (health / 20);
45      smartDogState.SetFeatureValue(BasicInfo.FEATURE_HEALTH,
            featureHealthValue);
46      previousHealth = health;
47  }
```

Listing 13.2: The method `CheckAmIHealthy`

Looking at Listing 13.2, we can see that the method starts by reading the health value from the entity representing the agent in the game. For a living agent, this value should be $0 < health \leq 20$.
If the health value is zero or below, the agent is considered to have died. In this case, the method will add a reward of -100 to the feature side as a death toll. In addition, the method will indicate that the agent is dead, which is necessary in order to ensure that the Q-values are updated before Minecraft kills the thread connected to the agent. Finally, the sensor will signal the system and request that the smartdog be respawned at its corresponding spawn point.
Should the smartdog be in a situation where it is not dead, the method will instead calculate its change in health. It is calculated as the difference between the current health, and the health it had the previous time the `CheckAmIHealthy` method was run. If the difference is greater than zero, the agent has been healed since the last check, and will earn a reward on the feature side proportional to the amount of healthpoints restored. If the

difference is less than zero, that agent has been hurt since the last check, and will earn a negative reward on the feature side proportional to the amount of healthpoints lost.
Finally, the feature value **Health** is calculated. It is worth noting that it is inversely proportional to the amount of health the smartdog has left. That is to say the feature will have the value of 0 if the smartdog has 20 health points left, the value of 0.5 at 10 healthpoints, and the value of 1 when the smartdog has no health left.

13.3 Collecting Rewards

A final responsibility held by the sensory module is to collect rewards. What this means is that all rewards earned through performing action a are stored in the sensory module until requested by the decision module in order to update the Q-values. The sensory module stores a seperate value for tabular and feature rewards.
The method shown in Listing 13.3, shows the method used for retrieving feature rewards. Note how it resets the value of the reward when the method is called and the reward is returned. The reason for doing this is that the reward should only be given once.
If the collected reward is zero, it will instead return a value called **Living cost**, which has a value of -0.1 at the time of writing. The reason for having this value is that we wish to encourage exploration by assigning a negative reward to all state-action pairs which do not produce a non-zero result.

```
1  public double GetFeatureRewardAndReset()
2  {
3      double temporary = featureRewardCollector;
4      featureRewardCollector = 0;
5
6      if (temporary == 0)
7      {
8          return BasicInfo.LIVING_COST; //This is so that actions
               without rewards are discouraged. Should help
              correct wrong behavior
9      }
10     return temporary;
11 }
```

Listing 13.3: The method `GetFeatureRewardAndReset`

13.4 Changes from Previous Semester

The sensory module has been modified in order to incorporate features. Amongst other things, this module can now detect the health of the agent,

and assign a reward to the agent based on the changes to its health. That is, reward for increasing the health value, punishment for losing health.
In addition, the module allows the agent to detect buttons and hoppers, as well as items lying on the floor, both edible and non-edible.

Chapter 14

Decision Making Module

This chapter contains a description of the decision making module used in this project.

The decision making module is the brain of the agent. Without it, the agent would do nothing, or at the very least make no decisions.
The purpose of this component is to make decisions on what actions to perform, based on the current state and previous knowledge, and to update that knowledge based on the rewards earned.
At the heart of this component is the method `MakeDecisionEGreedy` seen in Listing 14.1.

```
1  public byte MakeDecisionEGreedy(SmartDogState smartDogState,
       double tabularReward, double featureReward)
2  {
3      formerState = currentState;
4      currentState = smartDogState;
5
6      //The if is to secure that the state is not the impossible
           dummy state.
7      if (formerState.GetWhatBlockAmIOn() != (byte) - 1)
8      {
9          //Calculating the new Q-values and updating the old
10         //UpdateQValuesUnifiedQ(tabularReward, featureReward);
11         //UpdateQValuesUnifiedA(tabularReward, featureReward);
12         UpdateQValuesSeperate(tabularReward, featureReward);
13     }
14
15     //If the dog is dead, we want to save the data, but only
           ONCE!
16     if (smartDogState.GetIsTheDogDead() == true &&
           hasTheDogBeenConfirmedDead == false)
17     {
18         SaveQData();
19 hasTheDogBeenConfirmedDead = true;
20 //we use -1 to indicate that this action could not be
       performed, as the dog is dead.
21 SmartDog.AddRewardToStatisticsData(tabularReward,
       featureReward, (byte)-1, smartDogState.GetFeatureValue(
       BasicInfo.FEATURE_HEALTH));
22     }
23
24     if (BasicInfo.E_GREED > Math.random())
25     {
26 ArrayList<Byte> possibleActions = smartDogState.
       GetPossibleActions();
27 performedAction = possibleActions.get(random.nextInt(
```

```
        possibleActions.size()));
28      }
29      else
30      {
31  //The action that are going to be performed, and that we
        therefore can use to figure out what was performed last
        time in updating the Q-value
32  performedAction = GetMaxUnifiedA();
33      }
34
35      //write the reward to the statistics, and tell which action
            is picked
36      if(hasTheDogBeenConfirmedDead == false)
37      {
38  SmartDog.AddRewardToStatisticsData(tabularReward,
        featureReward, performedAction, smartDogState.
        GetFeatureValue(BasicInfo.FEATURE_HEALTH));
39      }
40      return performedAction;
41  }
```

Listing 14.1: The method `MakeDecisionEGreedy`.

The method starts by confirming that the former state is indeed a previously obeserved state, and not an empty dummy state. The situation where this would occur, is when the smartdog has just spawned into the world, and has not been in a previous state. In such a case, there would be no state to update the Q-values for.
If the former state is indeed a real state, a method will be called to update the Q-values. In the version shown here, the method used is `UpdateQValuesSeperate`, but the others are used in other implementations.
After that, the method will check if the smartdog is dead. If that is the case, the Q-values currently stored in volatile memory will be saved to non-volatile memory to prevent data loss when the agent's runtime is suspended by Minecraft. A flag will be updated to make sure this is only done once, and the obtained rewards will be added to the statistics data.
Finally, we come to the decicion making part of the method. Since we are using an ϵ-greedy approach, we start by determining if we should take a random action, or make an informed decision.
If are in the ϵ% of the time, we pick a random action, which is to say pick an action randomly from the list of possible actions. Otherwise, we make an informed decision by using the method `GetMaxUnifiedA`, which looks through the list of possible actions, and picks the one which has the highest combined Q-value. Finally, the method adds data to the statistics, and informs the system of which action to perform.

The next two sections will go into more detail about how decisions are

made, and updates are performed.

14.1 Making Decisions

When making an informed decision, the method `GetMaxUnifiedA` seen in Listing 14.2 is used.

```
1  private byte GetMaxUnifiedA()
2  {
3      ArrayList<Byte> possibleActions = currentState.
           GetPossibleActions();
4      byte maxAction = possibleActions.get(0);
5
6      double QTabular = qLearningDocumentCreator.
           GetQFromStateAndAction(currentState, maxAction);
7      double QFeature = GetQFromFeaturesAndAction(maxAction,
           currentState);
8
9      double maxUnifiedQValue = (BasicInfo.TABULAR\
           _IMPLEMENTATION\_WEIGHED\_AGAINST\_FEATURES * QTabular)
            + ((1 - BasicInfo.TABULAR\_IMPLEMENTATION\_WEIGHED\
           _AGAINST\_FEATURES) *  QFeature);
10     double contenderValue;
11     for (byte i = 1; i < possibleActions.size(); i++)
12     {
13         QTabular = qLearningDocumentCreator.
               GetQFromStateAndAction(currentState,
               possibleActions.get(i));
14         QFeature = GetQFromFeaturesAndAction(possibleActions.
               get(i), currentState);
15  contenderValue = (BasicInfo.TABULAR\_IMPLEMENTATION\_WEIGHED\
       _AGAINST\_FEATURES * QTabular) + ((1 - BasicInfo.TABULAR\
       _IMPLEMENTATION\_WEIGHED\_AGAINST\_FEATURES) *  QFeature);
16
17  if (contenderValue > maxUnifiedQValue)
18  {
19      maxUnifiedQValue = contenderValue;
20      maxAction = possibleActions.get(i);
21  }
22     }
23     return maxAction;
24 }
```

Listing 14.2: The method `GetMaxUnifiedA`

The purpose of this method is to find the action which has the highest combined Q-value.
To find this action, the method looks at the list of possible actions, and starting with the first one, it iterates through the list. For each action a, the Q-value from the feature side and the tabular side are added together

to form a **contenderValue**. If this value is higher than the previous best, a will be considered the **maxAction**, that is to say the action with the (currently) highest Q-value.
At the end, the method will return **maxAction**, telling the system which of the possible actions has the highest Q-value, meaning that it is the one which will be performed.

14.2 Updating Q-values

As mentioned in section 9.3, we have three different approaches for updating the Q-values of the combined statespace. In the next three sections, we will go through the implementation of each of them.

14.2.1 Seperate Update

The method seen in Listing 14.3 shows the method used for updating the Q-values using the *Seperate Update* approach. The method starts by using the standard approach from chapter 7 for calculating the new Q-value on the tabular side, and then updates the Q-value for the state-action pair $(formerState, performedAction)$.
After that, the method begins updating the value of the feature weights. First of, it calculates the δ (delta) value according to the standard approach from subsection 8.3.1. Then, it goes through each feature, updating the value of the weight, based on the former value, and $\eta \times \delta$ multiplied by the value of the feature.

```
1  private void UpdateQValuesSeperate(double tabularReward, double
       featureReward)
2  {
3      //update the tabular representation
4      //This is the Q update algorithm. Q[s,a] <- (1-alfa) Q[s,a]
           + alfa(r+gamma*max_a'(Q[s',a'])).
5      double newQ = (1 - BasicInfo.ALPHA) *
           qLearningDocumentCreator.GetQFromStateAndAction(
           formerState, performedAction) + BasicInfo.ALPHA * (
           tabularReward + BasicInfo.GAMMA * GetMaxQFromTabular(
           currentState));
6      qLearningDocumentCreator.UpdateQ(formerState,
           performedAction, newQ);
7      ...
8      //update the feature representation
9      double newWeight;
10     double delta = featureReward + BasicInfo.GAMMA *
           GetMaxQFromFeatures() - GetQFromFeaturesAndAction(
           performedAction, formerState);
11     double valueOfFeature;
12
```

```
13      for (byte featureToUpdate = 0; featureToUpdate < BasicInfo.
            NUMBER_OF_FEATURES; featureToUpdate++)
14      {
15          valueOfFeature = formerState.GetFeatureValue(
                featureToUpdate);
16          newWeight = featuresDocumentHandler.
                GetWeightFromFeatureAndAction(featureToUpdate,
                performedAction);
17          newWeight = newWeight + (BasicInfo.
                STEP_SIZE_FOR_GRADIENT_DESCENT * delta *
                valueOfFeature);
18          featuresDocumentHandler.UpdateWeight(featureToUpdate,
                performedAction, newWeight);
19      }
20      ...
21  }
```

Listing 14.3: The method `UpdateQValuesSeperate`.

14.2.2 Unified-Q Update

The `UpdateQValuesUnifiedQ` method is different from `UpdateQValuesSeperate` in that it uses the same estimate for Q_{MAX} on both the tabular and the feature side. To start of, the method calls another method which calculates the maximum unified Q-value.
Then, the new tabular Q-value for the state-action pair is calculated the same way as before, except that it uses the joint maximum Q-value rather than just the maximum Q-value on the tabular side.
For the feature side, the difference lies in the δ value. Again, instead of using the maximum Q-value on the feature side, the joint maximum Q-value is used.
Everything else is similar to the previous approach, which is why most of the code has been omitted.

```
1   private void UpdateQValuesUnifiedQ(double tabularReward, double
        featureReward)
2   {
3       double maxUnifiedQ = GetMaxUnifiedQ();
4       //update the tabular representation
5       double newQ = (1 - BasicInfo.ALPHA) *
            qLearningDocumentCreator.GetQFromStateAndAction(
            formerState, performedAction) + BasicInfo.ALPHA * (
            tabularReward + (BasicInfo.GAMMA * maxUnifiedQ));
6       ...
7       double delta = featureReward + BasicInfo.GAMMA *
            maxUnifiedQ - GetQFromFeaturesAndAction(performedAction
            , formerState);
8       ...
9   }
```

Listing 14.4: The difference in the method `UpdateQValuesUnifiedQ` compared to `UpdateQValuesSeperate`.

14.2.3 Unified-a Update

The `UpdateQValuesUnifiedA` method is different from `UpdateQValuesSeperate` in that it uses the same estimate for the best action a' on both the tabular and the feature side.
To start of, the method calls `GetMaxUnifiedA` which we saw in Listing 14.2. Then the method calculates the new Q-value on the tabular side, using **unifiedAction** as s'. Then, the δ value is calculated again, using **unifiedAction** as s' as well. As with *Unified-Q*, the rest of the method is similar in performance to the *Seperate* approach

```
1 private void UpdateQValuesUnifiedA(double tabularReward, double
      featureReward)
2 {
3     byte unifiedAction = GetMaxUnifiedA();
4     //update the tabular represenation
5     double newQ = (1 - BasicInfo.ALPHA) *
          qLearningDocumentCreator.GetQFromStateAndAction(
          formerState, performedAction) + BasicInfo.ALPHA * (
          tabularReward + (BasicInfo.GAMMA *
          qLearningDocumentCreator.GetQFromStateAndAction(
          currentState, unifiedAction)));
6     ...
7     double delta = featureReward + BasicInfo.GAMMA *
          GetQFromFeaturesAndAction(unifiedAction, currentState)
          - GetQFromFeaturesAndAction(performedAction,
          formerState);
8     ...
9 }
```

Listing 14.5: The difference in the method `UpdateQValuesUnifiedA` compared to `UpdateQValuesSeperate`.

14.3 Changes since Previous Semester

This module has been changed to use reinforcement learning using the combined model, rather than using the old tabular model. As described in chapter 9, we have three different approaches for updating the Q-values, and each of these has an implementation in the reinforcement learning module.
In addition, the agent is no longer capable of attempting to chose an illegal action. That is to say move to red, when there are no red blocks nearby.

Chapter 15

Actuator Module

This chapter contains a description of the actuator module used by the agent

The actuator module is the muscles of the agent. Without this component, the agent would be unable to interact with the minecraft environment. The sections in this chapter describes the actions that the agent can perform, along with the approaches used in performing them.

15.1 Actions and Possible Actions

When signaled to do so, the actuator can perform one of the following actions:

- Wait
- Activate button
- Move to hopper
- Pick up non-edible item
- Pick up edible item
- Drop item
- Move to red
- Move to green
- Move to yellow
- Eat item

Each of these actions consists of two parts: A movement part, and a finishing part. While all of the actions have these two parts, not all of them use them both. **Activate button**, for instance, uses both. First it moves to the position of a nearby button, and then it pushes it.
Move to red only uses the movement part, moving to a red block, and doing nothing for the second part. On the other hand, **Drop item** does not move out of place for its movement part, but drops its carried item(s) for the second part.
The actions will be described in more detail in the sections 15.1.1 through 15.1.6.
Note that while not mentioned in the sections, all actions initiate their

movement by performing a small vertical ascent, before doing the rest of their movement part. This is nescesary for the system to always be able to detect when the agent has *stopped* moving, as opposed to not having moved at all.

15.1.1 Wait

While initially designed to enable the agent to wait for a beter time to act, the wait action is currently only used to allow the agent to be able to perform an action when none of the others are available.
The wait action is very simple, doing nothing on the movement part, and simply signaling that the action is done when performing the finishing part.

15.1.2 Activate Button

This action enables the agent to activate buttons in Minecraft. The agent will first attempt to move as close to the button as it can. If it is close enough, it will push the button when it comes to its finishing part. Whether it succeeded or not, it will signal that it is done.
This action is considered impossible to perform if there are no buttons nearby.

15.1.3 Move to block

This section describes the whole category of actions which has the purpose of moving the agent to a block; red, yellow, green, or a hopper-block. In either case, for its movement part, the agent will attempt to move so as to position itself on top of the block, or as close as possible to it if it is part of a wall, or otherwise out of reach.
For its finishing part, the actuator will simply signal that the action has been performed.
These actions are considered impossible to perform if there are none of the corresponding blocks nearby.

15.1.4 Pick Up Edible/Non-Edible Item

As these two actions are fairly similar, they are described together.
These actions enable the agent to pick up edible and non-edible items. For the movement part, the agent attempts to move to the position of the nearest stack of items of the preferred type. If the agent is close enough, it will pick up the corresponding item as its finishing action.
Whether the agent picked up an item or not, the actuator module will signal that the action has been performed.
These actions are considered impossible to perform, if there are no items

of the corresponding type nearby, or the agent already has items in its inventory.

15.1.5 Drop Item

This action enables the agent to drop the item(s) it is currently holding. For its movement part, the agent does nothing, waiting to perform its finishing part. For its finishing part, the agent drops the item(s) currently in its inventory on the floor at its current position. Whether or not the agent actually had anything in its inventory, the actuator module will now signal that the action has been performed.
This action is considered impossible to perform if the agent has an empty inventory.

15.1.6 Eat item

This action enables the agent to eat the item currently in its inventory. For the movement part, the agent stays in place. For the finishing part, it is easier to get an understanding of the action by having a look at the code of how it is performed, which can be seen in Listing 15.1.
First off, the method starts by determining if the agent has anything in its inventory or not. If it does, the method will determine if the item(s) in the stack are edible or not. If they are edible, the method will heal the agent, and reduce the number of items in the agent's inventory.
Finally, whether there was something in the inventory or not, the actuator module will signal that the action has been performed.
This action is considered impossible if the agent has nothing in its inventory. In a future version, this might be changed so that the action will only be possible if there is something *edible* in the agent's inventory.

```
1  private void ActionEatItem(EntitySmartDog smartDog)
2  {
3      if (smartDog.GetInventory() != null)
4      {
5          ItemStack stack = smartDog.GetInventory();
6
7          if (SmartDogUtility.IsItemOfType(stack, SmartDogUtility
               .TYPE_EDIBLE))
8          {
9              smartDog.heal(SmartDogUtility.FoodHealingValue(
                   stack));
10             stack.stackSize--;
11             ...
12             if (stack.stackSize > 0)
13             {
14                 smartDog.SetInventory(stack);
15             }
```

```
16              else
17              {
18                  smartDog.SetInventory(null);
19              }
20          }
21      }
22      isActionDone = true;
23  }
```

Listing 15.1: The method `ActionEatItem`.

15.2 Performing an Action

Performing an action using the actuator module is a two step process, as mentioned in the previous section. First, the agent moves, and then it performs the remainder of the action.
The reason for this split is to allow the sensory module to run while the action is being performed. To get a better understanding of the process, we will look through the code following an example. Assume that for this example, the agent has decided to activate a button.
Looking at Listing 15.2, we see that the first thing the method does is to set a boolean value called **isActionDone** to `false`. This shows that the action has not yet been completed. After that, the **currentAction** is updated to indicate that the action the actuator is currently performing is activating a button.
Going through the if-statements, we find the one corresponding to the action, and see that we need to use the method `ActionMoveToButton`.

```
1  public void PerformAction(byte smartdogAction, EntitySmartDog
       smartDog)
2  {
3      isActionDone = false;
4      currentAction = smartdogAction;
5
6      if (smartdogAction == BasicInfo.ACTION_WAIT)
7      {
8      }
9      else if (smartdogAction == BasicInfo.ACTION_ACTIVATE_BUTTON
           )
10     {
11         ActionMoveToButton(smartDog);
12     }
13     else if (smartdogAction == BasicInfo.ACTION_MOVE_TO_HOPPER)
14     {
15      ActionMoveToHopper(smartDog);
16     }
17     else if (smartdogAction == BasicInfo.
           ACTION_PICK_UP_NON_EDIBLE_ITEM)
```

```
18      {
19          ActionMoveToNonEdibleItem(smartDog);
20      }
21      ...
22  }
```

Listing 15.2: The method `PerformAction`.

Now the actuator knows which movement method to use. The method can be seen in Listing 15.3. To start off, the method finds the x, y, and z coordinates of the button. Then, the actuator makes the agent jump, which makes it possible to assume that if the smartdog is not moving, its movement has stopped, rather than not yet started.
Finally, the method uses the built-in navigation system in Minecraft to make the agent move towards the button.

```
1  private void ActionMoveToButton(EntitySmartDog smartDog)
2  {
3      int x = smartDog.getSensoryModule().getButtonX();
4      int y = smartDog.getSensoryModule().getButtonY();
5      int z = smartDog.getSensoryModule().getButtonZ();
6      //This is to make a minor change in the movement of the dog
           so that we can check when the action is done even when
           the dog can't move.
7      smartDog.motionY = 0.01;
8      smartDog.getNavigator().tryMoveToXYZ(x, y, z, 1);
9  }
```

Listing 15.3: The method `ActionMoveToButton`.

When the actuator has moved the agent to the button, it will not do anything else, until the system has discovered that the agent has stopped moving. At this point, it will signal the actuator to finish the action, which will call the method seen in Listing 15.4. What the method does, is to check the value of **currentAction** to see which method it should use to finish up performing the action.
In this case, the method to be used is `ActionActivateButton`. If the action had been wait, or move to hopper, we can see that there is nothing to do on the finishing part, and it will signal that the actuator has finished the action right away.

```
1  public void FinishAction(EntitySmartDog smartDog)
2  {
3      if (currentAction == BasicInfo.ACTION_WAIT)
4      {
5          isActionDone = true;
6      }
7      else if (currentAction == BasicInfo.ACTION_ACTIVATE_BUTTON)
8      {
9          ActionActivateButton(smartDog);
```

```
10      }
11      else if(currentAction == BasicInfo.ACTION_MOVE_TO_HOPPER)
12      {
13       isActionDone = true;
14      }
15      else if (currentAction == BasicInfo.
            ACTION_PICK_UP_NON_EDIBLE_ITEM)
16      {
17          ActionPickUpItem(smartDog);
18      }
19      ...
20  }
```

Listing 15.4: The method `FinishAction`.

Now comes the part where the actuator finishes the activate button action. The method can be seen in Listing 15.5.
First,the method gets the coordinates of the button in question, and calculates the distance to the agent. If the distance is below an acceptable threshold, the actuator will figure out what kind of button it is, and then activate the button. Finally, whether the procedure was succesful in activating the button or not, the actuator will signal that the action has finished being performed.

```
1   private void ActionActivateButton(EntitySmartDog smartDog)
2   {
3       int x = smartDog.getSensoryModule().getButtonX();
4       int y = smartDog.getSensoryModule().getButtonY();
5       int z = smartDog.getSensoryModule().getButtonZ();
6
7       if (smartDog.getDistance(x, y, z) <= BasicInfo.
            ACTIVATION_RANGE)
8       {
9           if (smartDog.worldObj.getBlockId(x, y, z) == Block.
                stoneButton.blockID)
10          {
11              Block.stoneButton.onBlockActivated(smartDog.
                    worldObj, x, y, z, null, 0, 0, 0, 0);
12          }
13          else if (smartDog.worldObj.getBlockId(x, y, z) == Block
                .woodenButton.blockID)
14          {
15              Block.woodenButton.onBlockActivated(smartDog.
                    worldObj, x, y, z, null, 0, 0, 0, 0);
16          }
17      }
18      isActionDone = true;
19  }
```

Listing 15.5: The method `ActionActivateButton`.

15.3 Changes Since Previous Semester

The module has been modified to allow the agent to perform several new actions, including activating buttons, picking up edible/non-edible items, moving to a nearby hopper, and eating edible items. As a result of these changes, the actuator has been modified to use the split model of performing actions which it uses now.

Chapter 16

Knowledge Base

This chapter describes the components used in storing and accessing the tabular Q-values and feature weights.

We need to store the weights for our feature based reinforcement learning and the q-values for our tabular reinforcement learning. We do this externally so that they can be saved from session to session, and we do so using xml files: One for the tabular data and one for the feature data.

16.1 Q-Learning Documet Handler and Feature Document Handler

The tabular part of our reinforcement model needs to store its q-values for each of its state-action pairs which is 320 states times 10 actions. The feature based part of our reinforcement model needs only to store its weights for all its feature-action pairs which is 10 times 4. We will describe how we do so with the tabular side. The feature side is done in the same way, but with fever entries.
We save the Q-values in an xml document. We first locate the standard folder on the computer for documents, and check whether there is a SmartDogData folder or not, creating one if there is none.
In this folder we create the xml document. We then begin to fill in the xml tree structure. It has a root-node where the first state-attribute is its child-nodes. There is a child node for each value the attribute can have.
These children all have their own child nodes which are based on the next state-attribute with all the values it can have.
This way we create a tree structure where every level of the tree represents a state attribute. This way it is also easy to traverse the tree to find a specific state. Just follow the attributes with the right values.
The last level of the tree is the actions, where each of them have a double value attached to itself.
We create another xml document with line feeds and indentation so that it is humanly readable.
The feature based version is done in very much the same way, except there are only two levels to the tree besides the root node. The first level being the actions and the next being the features. The design is different since we need to find what value each action has in a state corresponding to its features and their weights. Reading and writing to the xml document is also handled, and is just traversel of the trees with a state as input. To read from

the tree, the method will output the value attached to the node. To write to the tree, the method also takes a value as input, and simply overwrites the node's attached value with the new one.

16.2 Changes Since Previous Semester

The changes made to the tabular side of the knowledgeBase component are rather simple, but have resulted in a greatly reduced tabular state-space. First of all, the component's class has been renamed from QLearningDocumentCreator to QLearningDocumentHandler to better emphasize the purpose of the component (it does not just create the file, but also reads from it, and modifies the content).
In addition, we have removed all attributes deemed irrelevant or intended to be replaced with feature values. This includes:

- HostileOrPassiveInSight - Intended to be implemented on the feature side.
- AreThereItemsInSight - Implemented on the feature side
- AmIHealthy - Implemented on the feature side
- FormerAction - Removed.

An attribute indicating whether there are hoppers nearby has been added, as well as a new possible value of WhatBlockAmIOn indicating that the agent is standing on a hopper.

The feature side of the component has been introduced to handle the individual feature weights, and storing them on non-volatile memory (the hard-drive). The FeatureDocumentHandler class is very similar to the QLearningDocumentHandler, although the files it accesses are much smaller in size, since it only stores a weight value for each $(feature, action)$ pair rather than for each individual $(state, action)$ pair. $(s, a) \in S \times A$

Chapter 17

Testing Component

This chapter describes the component used for testing the performance of the agent.

This component has been introduced to enable performance testing. In order to determine the performance of the agent, we need a way to measure the quality of its performance. The goal is for the agent to reach an optimal policy, which is defined as:

> *"one that maximizes total discounted expected reward"*[8, p. 280]

and while that would have been a good unit of measurement, it is hard to obtain through observation.
As such, we measure the total reward the agent obtains during a set number of actions which is 10 in the current version. While this is not the same as measuring the discounted reward, it should still be a valid approach. For instance, if for several instances, taking 10 actions in the start of the experiment yields a lower total reward than 10 actions later in the experiment, it would indicate that the agent has learned a better policy. If the rewards were to become similar during the experiment, it would indicate that the agent has converged to an optimal policy.

The component tracks the agents progress by logging all of the actions that the agent performs in a list, along with the total tabular rewards and feature rewards obtained through those actions. When the list reaches a certain length[1], a new entry will be made in an associated spreadsheet, which contains the cumulative rewards that the smartdog have obtained, along with a list of actions it has performed, and the health of the smartdog when it choose the action.
The list used for tracking actions and health will then be reset, along with the rewards obtained. This enables this component to collect data to prepare for the next entry in the spreadsheet.
If a minimum time has passed[2], the current spreadsheet will be saved on the hard-drive, and a new one will be made, from which the system will continue. The reason for using this approach is that if the system should crash while accessing the spread sheet, we will only lose 15 minutes of data, rather than several hours worth.
In order to create spreadsheets, access and modify them, we have been using

[1] 10 entries in the current version
[2] 15 minutes in the current version

the free, open-source Java Excel API (JXL) [3], which provides methods to create and modify spread sheets.

17.1 Analysing the Test Results

As mentioned above, this component enables us to produce test results, consisting of several data-points of 10 actions, combined with the total rewards earned during those actions.
With these results, it is possible to produce a graph, which shows the performance of the agent over time. Hopefully, the results will show improved performance over time, converging in the end.

Part IV

Testing

Chapter 18

The First Potion Test

This chapter will describe the test where the smartdog operates to maximise its rewards in a test environment with the possibility of dying.

In this test we will find out whether the smartdog is capable of learning to stay alive in a scenario where it will get hurt frequently, and still be able to maximise its tabular reward.

18.1 Test Environment

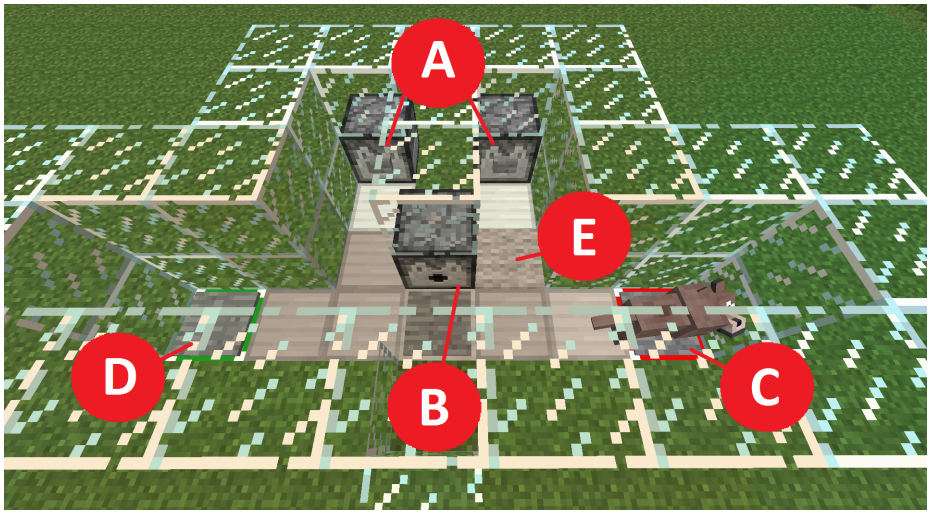


Figure 18.1: Picture showing the setup of the "Potion Survival Test" scenario.

The test environment can be seen in Figure 20.1, which is how it looks in Minecraft.

- At the points "A", there is a button which will dispense a potion which will restore 4 point of the smartdog's health, earning it a reward of 0.4 on the feature side.
- At the point "B", there is a dispenser which fires an arrow at the smartdog whenever it walks in-front of it. This will reduce the smartdog's health with about 4 points, and give it a negative reward on the feature side of -0.1 per lost health point.
- At the point "C", there is one of our red custom blocks. When the smartdog moves to the red block, it will get a reward of 1 on the

tabular side if, and only if, it has been on the green block since last it activated the red block.

- At the point "D", there is one of our green custom blocks. When the smartdog moves to the green block, it will get a reward of 1 on the tabular side if, and only if, it has been on the red block since last it activated the green block.

- At the point "E", there is a block where the smartdog can respawn if it dies during the test.

- In all other situations, the smartdog will receive a negative reward of -0.1 on both the tabular and the feature side.

18.2 Expectations

From the test environment we can see that the smartdog can get rewards by walking between the green and red block. However, doing so will damage the smartdog since it will walk in-front of the dispenser shooting arrows at it.
We therefore expect that after having died some amount of times, it will have learned that when its health is sufficiently low, it will need to go in and activate a button, but otherwise walk between the red and green block to get as many rewards as possible.
We expect the feature reward to average around 0, except for the few actions that leads to death. This is since it will get a negative reward for getting hurt, which has the same numeric value as the reward it will get for healing.

18.3 Test Factors

Each entry in the data set consists of the rewards earned through taking 50 actions, so points on the result graph are rewards gathered through 50 actions
We have run 4 tests: One for each of the 3 update approaches, and one where we have hard-coded a reasonable behaviour by setting the q-values and weights and not letting them update.
The separate update ran for 3 hours and 0 minutes and took 1750 actions.
The unified-a update ran for 2 hours and 51 minutes and took 1700 actions.
The unified-Q update ran for 2 hours and 40 minutes and took 1600 actions.
The hard coded test ran for 6 hours and 36 minutes and took 4190 actions.
In the results we only show the first 1750 actions to fit the other tests.

18.4 Results

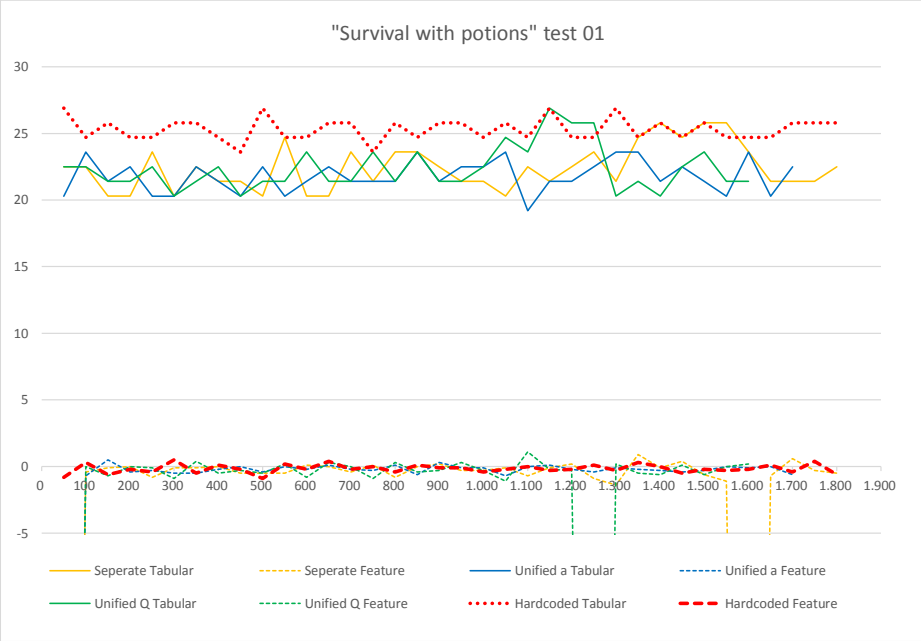


Figure 18.2: Graph showing the first potion survival test results showing the rewards obtained as a function of the amount of actions taken.

In all three update tests the smartdog dies within the first 50 actions. Thereafter the smartdog dies once in the unified-Q and separate tests, but not in the unified-a. The feature rewards seem to have converged after the first 50 actions. The Tabular show some spikes around where the smartdog dies. The unified-a test seems closer to converging than the other two.

18.5 Discussion

As described above, we believe that the agent's policy has converged after the first 50 actions in each case, based on the proximity to the rewards obtained by the hard-coded approach. As such, it is hard to see signs of improvement if all of the improvement is done within the first dataset.
We believe that this scenario is too simple to be able to *show* that the agent learns over time, and as such we have designed a more complex scenario, which can be seen in chapter 19. For that scenario, we intend to have only 10 actions per dataset, as opposed to the 50 actions used for this test. The reason for this change is to be able to better document the changes in obtained rewards over time.
We do see changes in the three update test on the tabular side. The unified-

Q and seperate seem to shift in amount of rewards gathered around where the smartdog dies. The shift could be due to the huge negative value that it gets from the feature side, which then begins to hinder the actions that would otherwise get rewards on the tabular side. It does seem to get back to a more regular reward amount relatively quickly afterwards.
Since we have a system that picks a a certain amount of the actions at random, it does mean that there is the possibility of the observed deaths being due to randomness. If so, we would believe that running the tests for a longer amount of time would let the reward amounts would be a bit more steady, and show better convergence. More test would be needed to show this though.

Chapter 19

Second Potion Test

This chapter describes a test where we use a more advanced version of the first scenario from chapter 18.

In this test we wish to find out if the agent is capable of learning an optimal policy in a scenario with a slightly different reward-function on the tabular side. The reward function is made so that in order to earn a large reward of 2 on the tabular side, the agent first has to take an action which gives it a negative reward of -1 on the tabular side.

19.1 Test Environment

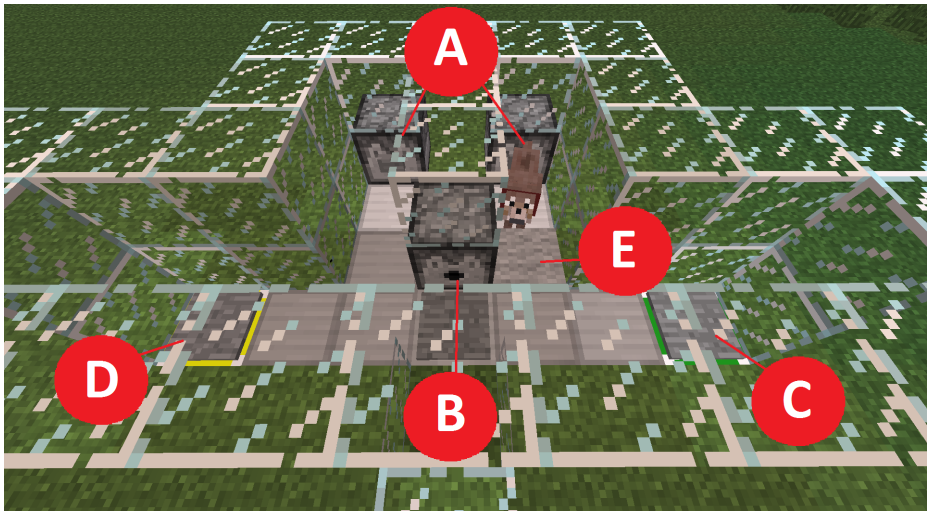


Figure 19.1: Picture showing the setup of the "Second Potion Test" scenario.

The test environment can be seen in Figure 19.1, which shows how it looks in Minecraft.

- At the points "A", there is a button which will dispense a potion which will restore 4 point of the smartdog's health, earning it a reward of 0.4 on the feature side.
- At the point "B", there is a dispenser which fires an arrow at the smartdog whenever it walks in-front of it. This will reduce the smartdog's health with about 4 points, and give it a negative reward on the feature side of -0.1 per lost health point.

- At the point “C”, there is one of our green custom blocks. When the smartdog moves to the green, it will get a reward of 2 on the tabular side if, and only if, it has been on the yellow block since last it activated the green block.
- At the point “D”, there is one of our yellow custom blocks. When the smartdog moves to the yellow block, it will get a reward of -1 on the tabular side if, and only if, it has been on the green block since last it activated the yellow block.
- At the point “E”, there is a block where the smartdog can respawn if it dies during the test.
- In all other situations, the smartdog will receive a negative reward of -0.1 on both the tabular and the feature side.

19.2 Expectations

Besides a few changes on the reward function on the tabular side, this test bears close resemblance to the first potion test in chapter 18. It has other coloured floor, to confuse a later transferring test, and it gets 2 tabular rewards on one side while it get a negative -1 on the other side, but it needs to go between those two to activate the rewards. So We expect the same behaviour as the first potion test: go between yellow and green until health is sufficiently low, then go in and push the button to get healed and return to wander between yellow and green. We do however expect the smartdog to take a bit longer to learn this behaviour since there is a bit more complexity on the tabular side in that it needs to do a negative action to be able to take an action that gives a combined higher reward.

19.3 Test Factors

Each entry in the data set consists of the rewards earned through taking 10 actions, so points on the result graph are rewards gathered through 10 actions
We have turned it down from 50 actions since we could not observe where the smartdog learned the optimal behaviour in the first potion test in chapter 18.
We have run 4 tests: One for each of the 3 update approaches, and one where we have hard-coded a reasonable behaviour by setting the q-values and weights and not letting them update.
The separate update ran for 3 hours and 22 minutes and took 1920 actions.
The unified-a update ran for 7 hours and 58 minutes and took 4730 actions.
In the results we only show the first 1920 actions to fit the other tests.

The unified-Q update ran for 3 hours and 3 minutes and took 1750 actions. The hard coded test ran for 2 hours and 16 minutes and took 1350 actions. We gather the results from the test in three graphs. One for each update approach. In each graph we compare those results to the hard-coded values. The thicker lines in the result graphs are a smoothed out graph that takes the last 10 data point and use the average. This is to better show the flow of rewards gathered. In the graph these are noted by "10 per. bev. gnsn." which is short in Danish for "per moving average". This is due to the software used being in Danish.

19.4 Results

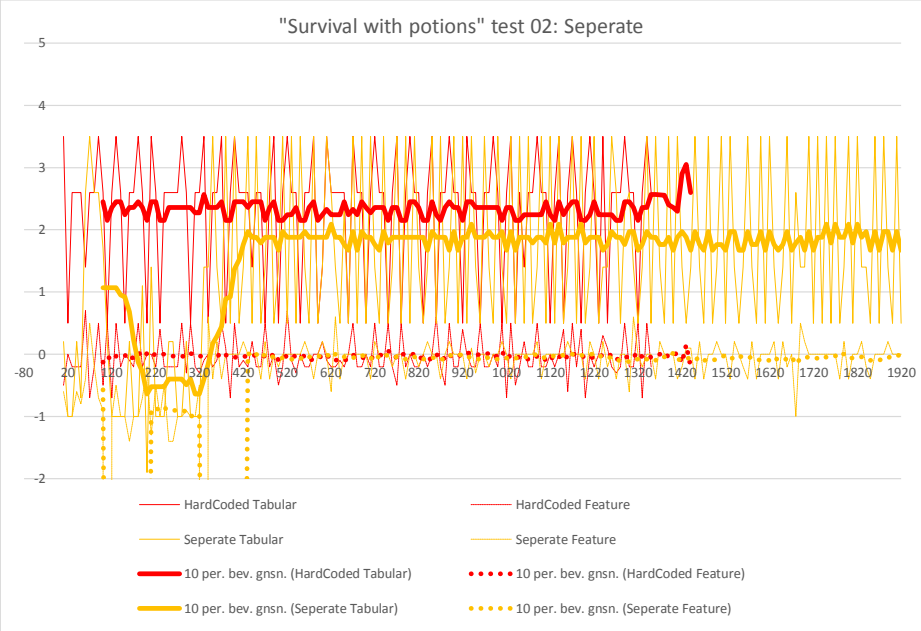


Figure 19.2: Graph showing the amount of rewards obtained using seperate update, as a function of the amount of actions taken.

In Figure 19.2 we see the test results from the potion survival test 2 separate. The Red lines indicate the hard-coded test, and the yellow indicate the separate test. On this it is clear to see that the separate update method seem to converge at about 420 action taken, just after it died for the second time. The converged tabular reward comes close to the hard-coded reward values, but don't get there completely. The Feature rewards seem to converge after the two deaths, and converge at the same value as the hard-coded values.

In Figure 19.3 we see the test results from the potion survival test 2

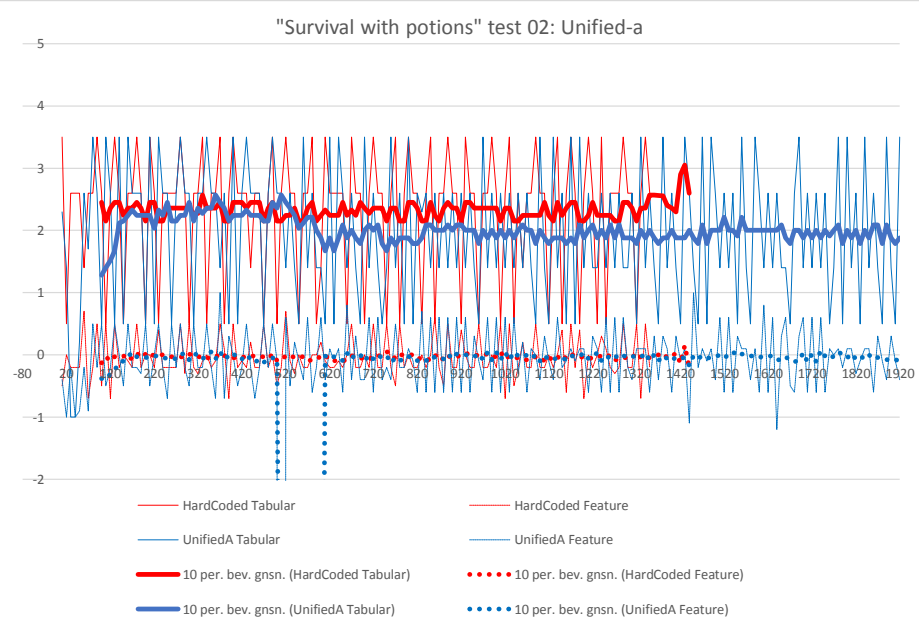


Figure 19.3: Graph showing the amount of rewards obtained using unified-a update, as a function of the amount of actions taken.

Unified-a. The Red lines indicate the hard-coded test, and the blue indicate the Unified-a test. We can see that the Unified-a show signs of convergence to the same reward value as the hard-coded one at about 130 actions taken. It seems to start converging to a lower value after it dies at about 510 actions taken, reaching convergence at 620 actions, after which the reward values seem steady.

In Figure 19.4 we see the test results from the potion survival test 2 Unified-Q. The Red lines indicate the hard-coded test, and the green indicate the Unified-Q test. We can see that the Unified-Q show signs of convergence at about 600 action taken and it seem that it converge to a lower value, after it dies, at 1020 actions taken. The reward amount are, after the graph has converged, close to the hard-coded values but constantly below it.

19.5 Discussion

One thing to notice before we go into any other discussion: The graphs show smoothed out functions which use the average of the last 10 data points from the observed data. This also means that the last data point on those function is an average of 9 null values and the last actual data value.

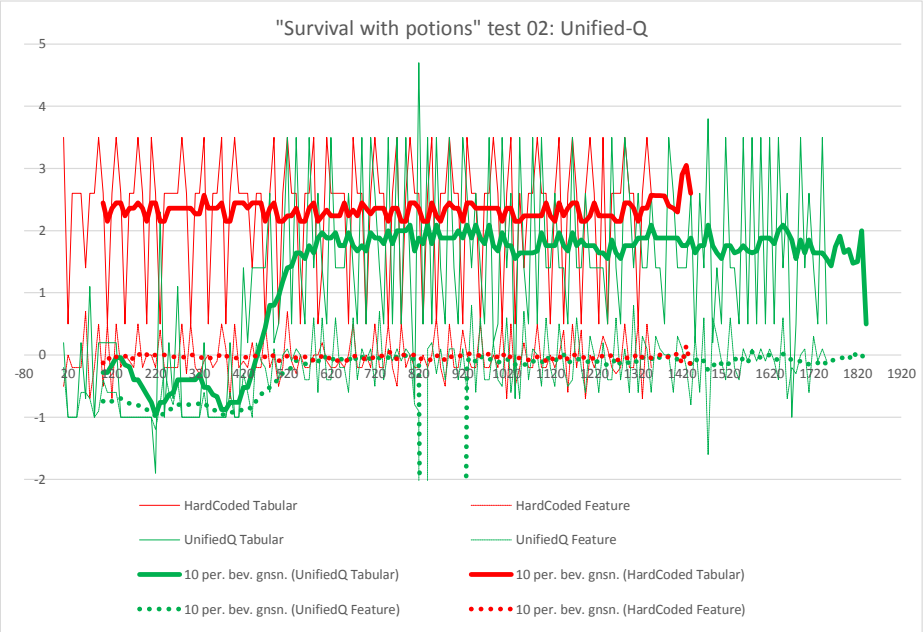


Figure 19.4: Graph showing the amount of rewards obtained using unified-Q update, as a function of the amount of actions taken.

Therefore there seem to be a bit odd behaviour in the last 10 data-points in the smoothed out functions. This is especially visible in Figure 19.4 for the tabular Unified-Q smooth and tabular hard-coded smooth.

In the test it seem that the Unified-a performs best. It seem to show convergence the fastest and when converged have the highest reward value of the three. Also it seems to have a policy getting as much rewards as the hard-coded for a while before it dies and becomes more cautious. The Separate update seem to learn after having died twice, while unified-a and unified-Q seem to learn before death and becomes too greedy, and after death converge on a policy that keeps them alive and with a relatively high reward.

Chapter 20

Testing with Food

This chapter describes a test where the smartdog has to eat food in order to stay alive, and otherwise operate to maximise its rewards.

In this scenario, we will test if the smartdog is capable of surviving in an environment where it will get hurt frequently, and has to learn to eat food in order to survive. The test environment is the same as the one used in chapter 18, except that the dispensers dispenses dog treats, rather than healing potions.

20.1 Test Environment

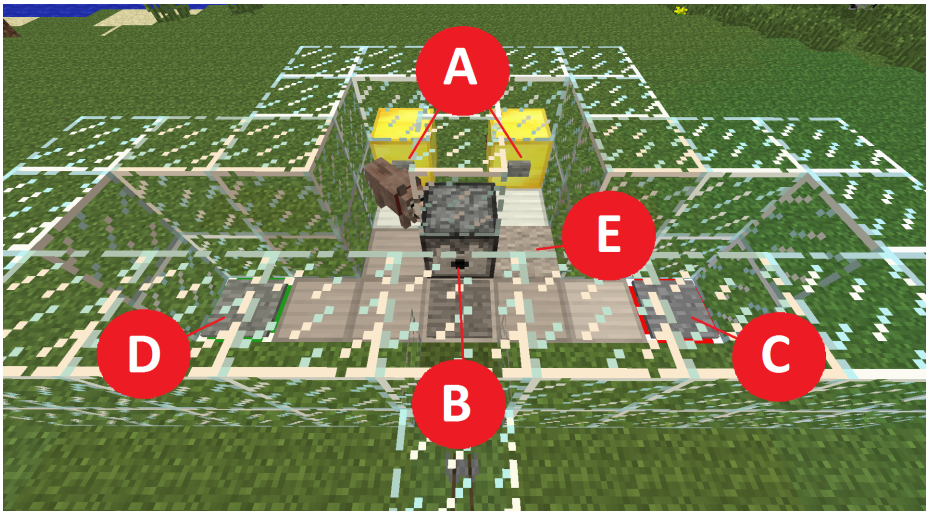


Figure 20.1: Picture showing the setup of the "Testing with Food" scenario.

The test environment can be seen in Figure 20.1, which shows how it looks in Minecraft.

- At the points "A", there is a button which will dispense a dog treat on the floor for the smartdog to pick up and eat. Doing so will restore 1 point of the smartdog's health for each treat it eats. earning it a reward of 0.1 on the feature side.
- At the point "B", there is a dispenser which fires an arrow at the smartdog whenever it walks in-front of it. This will reduce the smartdog's health with about 4 points, and give it a negative reward on the feature side of -0.1 per lost health point.

- At the point "C", there is one of our red custom blocks. When the smartdog moves to the red block, it will get a reward of 1 on the tabular side if, and only if, it has been on the green block since last it activated the red block.
- At the point "D", there is one of our green custom blocks. When the smartdog moves to the green block, it will get a reward of 1 on the tabular side if, and only if, it has been on the red block since last it activated the green block.
- At the point "E", there is a block where the smartdog can respawn if it dies during the test.
- In all other situations, the smartdog will receive a negative reward of -0.1 on both the tabular and the feature side.

20.2 Expectations

In this scenario, we expect the smartdog to be able to learn a policy where it moves between the red and the green blocks to earn tabular rewards, until its health is sufficiently low. At this point, we expect to push the closest of the two buttons in order to get a treat, pick up the treat, and eat it. It should then repeat this process until its health is restored sufficiently for it to go back to moving between the red and the green block. It might die more often than in the previous tests, as it will have to go trough a three step process to restore its health, rather than simply pushing a button as with the previous tests.

20.3 Test Factors

Every row in the data set are 10 actions taken. We run 4 tests. One for each of the 3 update approaches and one where we have hard-coded a reasonable behaviour by setting the q-values and weights and not letting them update. The seperate update test ran for 2 hours and 28 minutes and took 1500 actions.
The unified-a test ran for 3 hours and 1 minute and took 1650 actions.
The unified-Q test ran for 2 hours and 26 minutes and took 1350 actions.
The hard-coded test ran for 1 hour and 31 minutes and took 800 actions.

20.4 Results

Figure 20.2, which shows the tabular side, does not indicate convergence, and even the hard-coded implementation does not show a steady reward

over time.
On the feature side, seen in Figure 20.3, the agent dies much more often than in the previous tests, again not indicating convergence.
It is worth noting that the agent using Unified-Q does not die past its 750th action, and the same applies to the agent using Unified-a, although it happens past the 1200th action.

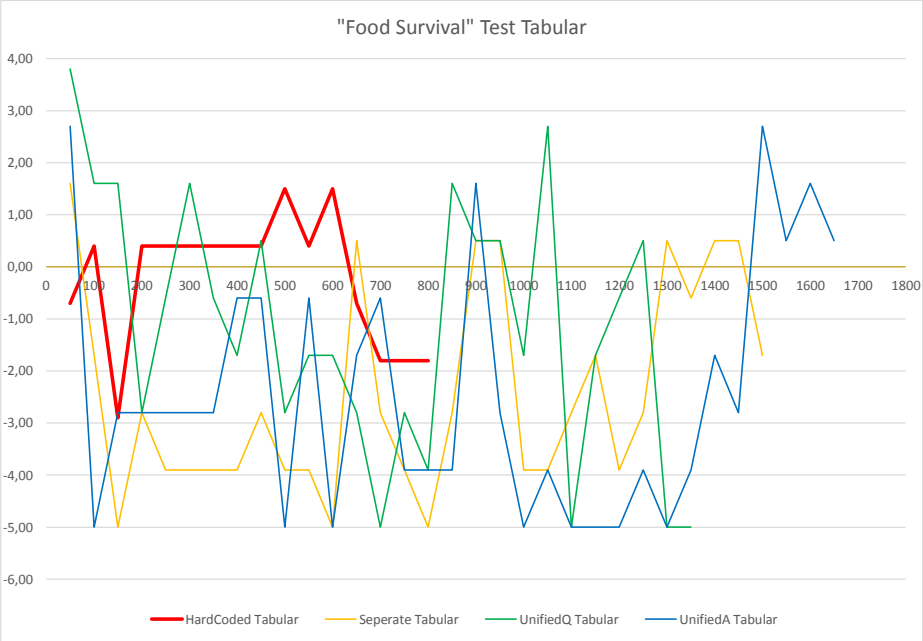


Figure 20.2: Graph showing the amount of tabular rewards obtained as a function of the amount of actions taken.

20.5 Discussion

Looking at the graphs in Figure 20.2 and Figure 20.3, neither of the three approaches shows any clear signs of convergence. We take this to indicate that the agent is not capable of learning an optimal policy in this environment.
The reason for this could be that the reward for restoring the agents health is so low that it is hard to see that there is a benefit to doing it. However, since the hard-coded approach could not produce a more "flat" graph either, a more likely explanation is that our features are not designed well enough. They might need to be more advanced in order to handle a scenario which requires the agent to figure out the three step process that is required in order for it to restore its health in this scenario. One such improvement could be changing the binary-valued features "AreThereEdibleItemsNearby" and

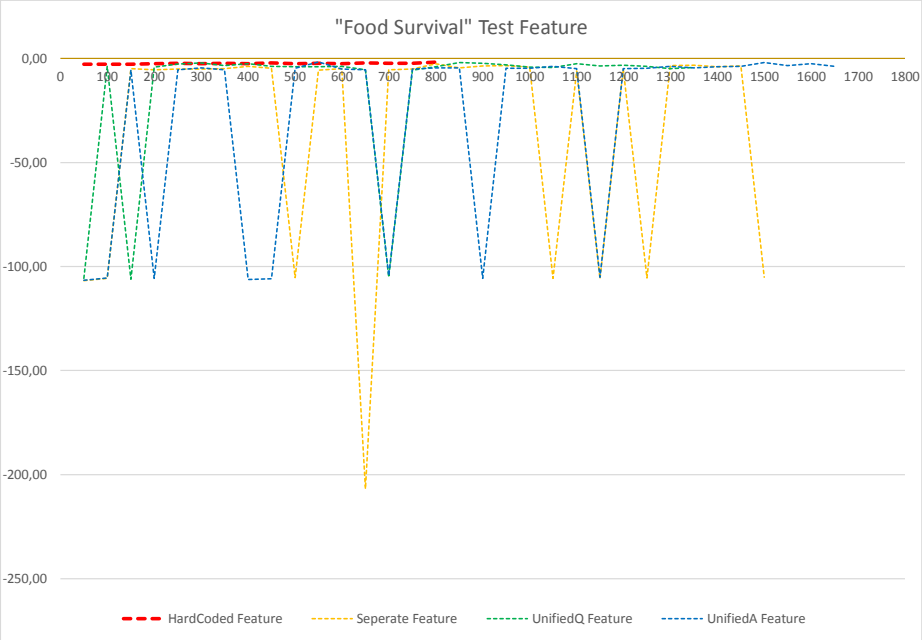


Figure 20.3: Graph showing the amount of feature rewards obtained as a function of the amount of actions taken.

"AmIHoldingSomehting" into numeric-valued features instead.

Chapter 21

Testing Knowledge Transfer

This chapter describes a test where we try to determine if the smartdog is capable of transferring knowledge from one scenario to another.

In this scenario, we will see if the smartdog is capable of using knowledge it has obtained from a previous scenario, in order to perform better than it would have if it did not start out with this knowledge. The test environment is exactly the same as the one used in chapter 19, and we aim to see if it can use the knowledge it has learned from chapter 18.

21.1 Test Environment

In this situation, the test environment is completely identical to the one described in section 19.1

21.2 Expectations

As the environment is identical to the one from chapter 19, we expect the agent to be able to learn the same behaviour as the one from that test. However, as we will be transferring the obtained feature knowledge from chapter 18, we expect it to be able to learn an optimal policy for this scenario faster than without this knowledge, as even though the feature knowledge that we transfer have not yet gotten a fitting value for the "Move to yellow" action, it knows that it can be dangerous to move to a green block, and it has learned that it should push a button when it is low on health.

21.3 Test Factors

Each entry in the data set consists of the rewards earned through taking 10 actions, so points on the result graph are rewards gathered through 10 actions
We have run 3 tests: One for each of the 3 update approaches.
For each test, we have kept the feature-weight values that were obtained for each approach tested in chapter 18. That is to say that unified-a uses the feature-weight values it had learned in the previous scenario, and so on.
The seperate update test ran for 2 hours and 17 minutes, and took 1340 actions.
The unified-a test ran for 2 hours and 16 minutes, and took 1350 actions.
The unified-Q test ran for 2 hours and 36 minutes, and took 1310 actions.

As the scenario is similar to the one from chapter 19, we have not run a new instance of the hard-coded test, but instead try to see how these new results compare to those of the previous test.
We gather the results from the test in three graphs: One for each update approach. In each graph, we compare those results to the results from chapter 19. The thicker lines in the result graphs are a smoothed out graph that displays the moving average of the last 10 data points, just like in the previous test.

21.4 Results

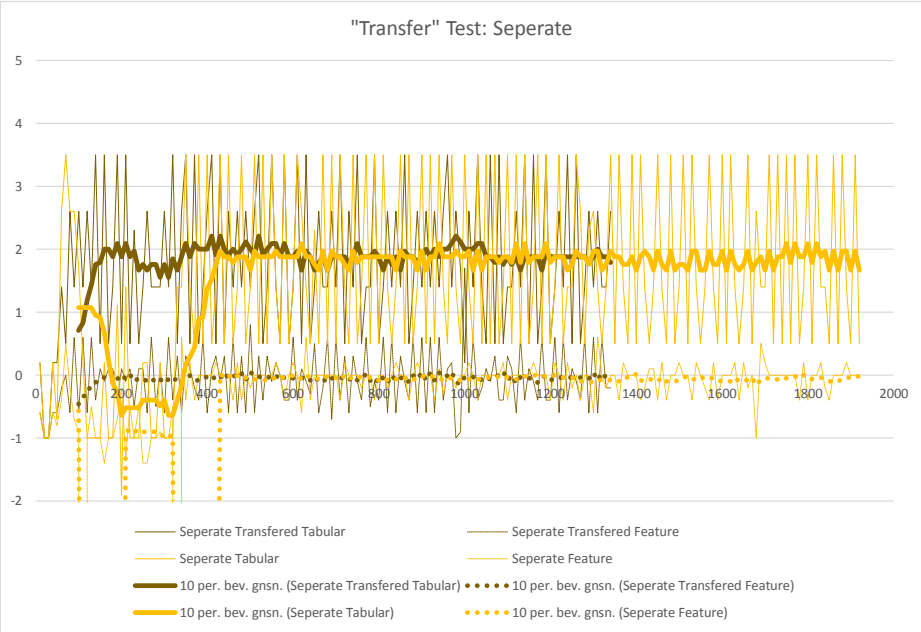


Figure 21.1: Graph showing the amount of rewards obtained using seperate update, as a function of the amount of actions taken.

Looking at Figure 21.1, we can see how using seperate update benefits from transferring knowledge.
First off, it starts to indicate convergence at about 350 actions taken, rather than 420. Additionally, while the agent died twice in the setup where it did not use transferred knowledge, it didn't die throughout the dataset where it had transferred its feature weights from chapter 18.

Looking at Figure 21.2, we see that the number of rewards obtained using unified-a starts to converge at about 300 actions, rather than at about 620 like the test without transferred knowledge. The agent does die, just like in the previous test, but the death happens earlier on.

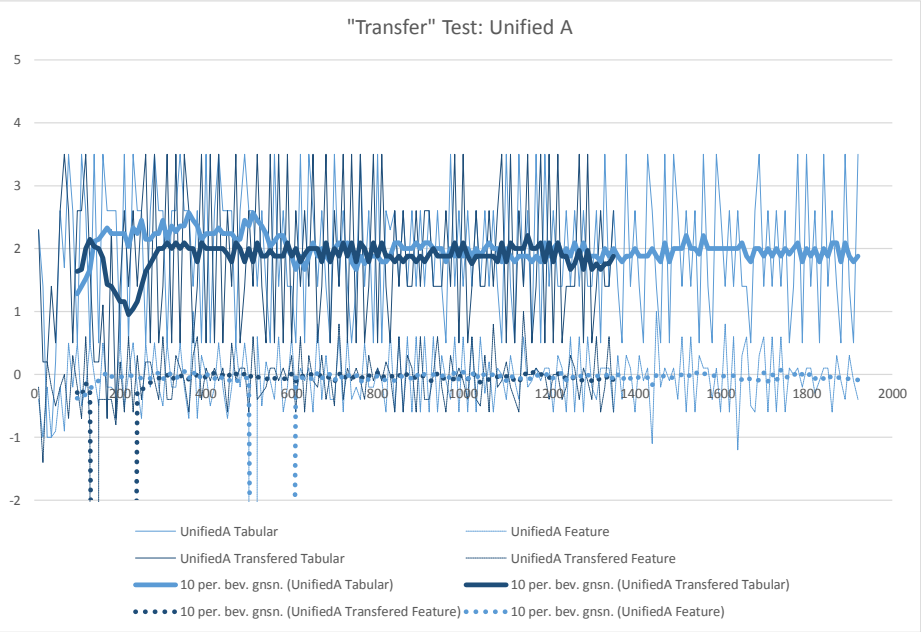


Figure 21.2: Graph showing the amount of rewards obtained using unified-a update, as a function of the amount of actions taken.

Looking at Figure 21.3, we can see that the test running unified-Q has not benefited from having transferred feature weights. Past 400 actions, the results from the previous test have obtained a higher reward than the ones obtained with the test using transferred knowledge. The agent does not die in this new test however.

21.5 Discussion

Looking over the test results, we can see that the tests that were running seperate update, and unified-a both benefited from using the feature-weights from the scenario described in chapter 18. What is interesting to note however, is that unified-Q did not show the same benefit. Looking at Figure 21.3, we *could* argue that the agent's behaviour has started converging at about 100 actions, and thus much earlier than the convergence seen at about 600 actions in the previous test. However, converging at a reward value of about -0.5 per 10 actions is much worse than converging at about 1.8 per 10.
From these results, it looks like implementations running either separate update or unified-a update is capable of transferring knowledge from one situation to another, while an implementation running unified-Q update is not.

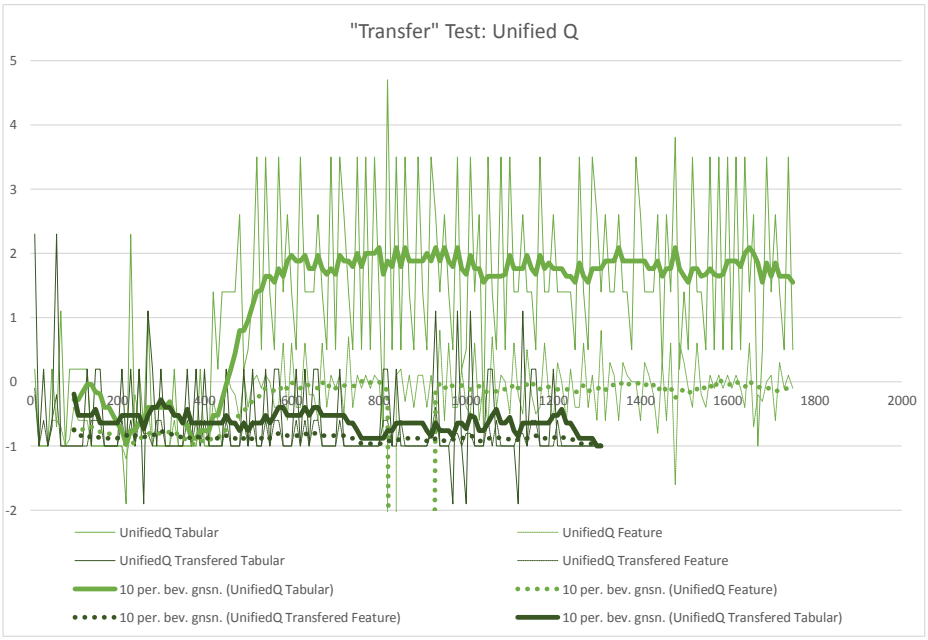


Figure 21.3: Graph showing the amount of rewards obtained using unified-Q update, as a function of the amount of actions taken.

Chapter 22

Test Evaluation

This chapter provides an evaluation of our test results, comparing them with out expectations.

From chapter 18 we cant really see the learning part of the test although we seem to have converged to values that lies close to the hard-coded ones. This indicates that we would have to decrease the amount of actions per datapoint and increase the complexity of the test. This leads us to the second potion test.

The second test gave quite a better result. It showed that the agent learned a behaviour which converged in all three instances. It additionally shows that the three models do have different behaviour, although slight in this scenario. The hard-coded test outperforms them all, but not by much. And although the hard-coded outperforms them in this situation, it would not be able to transfer its knowledge to another scenario, as it can not adjust. Even if the hard-coded test were replaced with a completely tabular policy, learned via changing the tabular state-space to fully express the given scenario, it wouldn't be able to transfer that knowledge. This leads us to the transfer test, where we show that knowledge from the first potion test can benefit the agent in learning the second potion test.

In the transfer test we can see that separate and unified-a profits greatly from having knowledge from a former scenario. They both seem to learn quicker, and converge faster. The separate even avoids to die completely. This is due to the separate having complete separate models for the feature based learning and tabular. Therefore the actions to heal when not having a high health value are the same. In the unified versions, there are also this transfer, but also a bit extra. The unified-a and unified-Q updates their Q-values and feature-weights in a more "unified" fashion. The unified-a updates its weights in accordance to the feature-side's own Q-values. Therefore a Q-value on the feature side will never become the highest one without also receiving a reward on the feature side. Actions will only update the weights according to the maximum feature-side Q-value, as opposed to the unified-Q approach where the maximum Q-value across the two sides are used, and can therefore skew the highest Q-values on the feature side when transferred. We can also see this, as the rewards obtained during the unified-Q test are a lot smaller than the the test where we do not transfer knowledge. If the unified-Q had run for longer it would properly have converged at the same

policy it had found in the previous scenario, but it would take a lot of time to correct the weights to get there.

We also wanted to run another set of test where we made the testing environment a bit more complex by adding a few more steps to getting healed. The smartdog would have to get dog treats, pick them up, and eat them to get healed. These does not seem to learn or converge at all. This might be due to the fact that healing is a three step process of healing one point of health, which would give it a collected reward of -0.1, with -0.1 for two of the actions and 0.1 for the eat action. There are still the large negative reward for dying, but this would indicate that it would take a very long time to learn an optimal behaviour. A more likely reason would be that the features design is flawed. We designed the features connected to eating to be binary, but it might have been better to have them being numeric instead. Being binary means that it can see whether it is holding something or not, and whether there are items nearby or not, but not how many. A redesign of said features would then be necessary. This test was run at the same time as the potion test, but since we got these results and decided that the most likely explanation was bad feature design, we decided to spend our time on the potion test instead. We, unfortunately, did not have the time to redesign the features and then test them afterwards.

A last thing to note is that results obtained with the hard-coded test might not have been possible to obtain through learning. This is due to us knowing exactly how the environment works and being able to tweak the feature weights to values that create a distinct behaviour. Normally there will be a lot of "next best" actions that we have been able to filter out. It is also worth noting that the hard-coded might not be the optimal behaviour since it is created by us, and we have not spend tremendous amounts of time to calculate the optimal behaviour, but went for one which would act logically and not die.

Part V

Evaluation

Chapter 23

Discussion and Conclusion

In this chapter we will discuss whether our project reach the goals which we had described in the intro part and whether our thesis is correct.

Now that we have run the tests, which are seen in the different chapters in the test part of the report, it is time to see if our solution lives up to the project specification, based on these tests:

"Develop an intelligent agent."

The agent is capable of using reinforcement learning to learn an intelligent behaviour in the situations we have tested, with the exception of the food test. While we would also have liked to add more actions, sensors and features to it so that it would be able to do more complex tasks, this is what time has allowed us to do. We had originally a few more features, but they are likely faulty and the reason we believe that the food test failed. The smartdog does, however, behave intelligently in the simple set-up we have created otherwise, and it do so with a system based on a union of tabular and feature-based reinforcement-learning.

"Enable the agent to learn an optimal behaviour for a given situation."

So far it does not seem like the smartdog agent has learned an optimal policy when compared to the hard-coded version we provided for the tests. However it does seem to have learned a near-optimal behaviour, and with better designed state-space representation it might be able to learn an optimal policy. Designing the right state-space might be problematic since it requires both to take into account the tabular as well as the feature side which we believe will greatly increase the complexity of the design.

"Enable knowledge transfer between different situations."

We use feature-based reinforcement learning to keep information about the general aspects of the environment. We have shown in the transfer test that the separate and unified-a learns and converges faster when having prior general knowledge of the environment. Although that the two test seem very alike their reward function on the tabular side differ greatly, while the feature sides reward function are almost identical. And although the two test environments can be described inside the same tabular state-space, this does not influence the outcome as the observed states are still unique to each of

our tests, and could therefore be viewed as completely separate state-spaces. To clarify, one where yellow and green are observed at all times, and one where red and green are observed at all times.

23.1 Conclusion

The problem statement is as follows:

"How can we develop an intelligent agent which is capable of learning optimal behaviour for different scenarios, while being able to transfer knowledge from one scenario to another as well?"

We have shown that by combining tabular-based and feature-based reinforcement-learning techniques to create an agent, it will be able to learn a near-optimal behaviour for different scenarios while transferring this knowledge from one scenario to another as long as the knowledge transferred is general for the whole environment. We have shown this for two of the three suggested implementation models, specifically Separate and Unified-a. The agent will behave intelligently as long as the state-attributes and features are designed well enough.

Chapter 24

Reflection

Some of our design choices throughout the semester could be questioned. One of which is our chosen platform Minecraft, since it has brought us a lot of trouble and could have been replaced by a simple platform we created ourselves. Why we choose to go with Minecraft anyway is that we wanted to create something that could be used by other people, and to challenge ourselves to make a solution for something already in existence. Would we have been able to create a better agent with our own platform? Probably. Would it have been better to show or disprove our thesis? Likely, but these are the reasons that we still went with Minecraft. We are still in doubt whether it was a good choice or not, but some of the motivation we have gotten for the project has also been from the idea that what we have created might, with some polish, be used by others in the future. Unfortunately the problems arising from working with Minecraft has halted us enough that the functionality of the smartdog is very limited, and in its current state wont be of much use to the standard Minecraft player. But further development might very well be put in in our spare time to bring a smartdog which can follow the player and help in a Minecraft characters daily routine. Another thing to notice that are connected to Minecraft is that our implementation is rather close to being able to run as multi-agent, meaning more than one smartdog. The agents wont be able to communicate, but a few changes to the system so that each individual smartdog gets their own knowledge files and uses those to perform actions and update values and it "should" work out of the box. This is also true for multilayer, as long as we find a way to bind a specific dog to a specific player and their knowledge files.

One of the things that is clear at this point in the project is that more research is needed to fully prove or disprove our thesis. We need to test the different approaches in more complex, individual scenarios with the same general environment. And to do so efficiently could be to develop our own platform to represent the environment, since Minecraft has shown to bee too problematic. Mostly because we do not have access to the code directly, and therefore have to use clever methods, or hacks, to add a reinforcement system as an AI to one of its entities. With our own simple platform, we could keep the environment turn based, and without a graphical animation of what is happening, testing could take seconds instead of hours so that longer tests could simply be run in minutes. And such a platform would not take long to make. Designing the platform to represent a tabular state-space and feature based state-space would also be relatively simple in comparison

to working with Minecraft. But as mentioned above, we have our reasons for initially choosing Minecraft, and at some point we where committed because of how our system looked and how long it would take to change it. Would we have used our own platform in the start if we had known the problems we where faced with via Minecraft? Likely. Would we have had as much fun? Not as much, but less frustration too. Would it have been better for the thesis? Likely. So yes, we would have used our own platform if we had realised this at the beginning.

If one were to assume, which one seldom should, that there were more research on our thesis and that it is actually proven to be able to not just find optimal polices for specific scenarios in an environment as well as transferring general knowledge, but would also be good at it. What if it showed to be a better performing solution than an unspecific feature based solution and a impractical tabular solution? Then what could it be used for?
We have had some ideas throughout the year, and some of them we might even use at a later point. In the domain of games, we have talked about of few applications. The one that came quickest to our minds was a animal simulation. A form for interactive digital pet, which would be far more advanced than that the former digital pets. The system could also be used for creating advanced enemy AIs in strategy games, so that the agent learns about the players specifics and general map related information.
Another rather intriguing idea would be an MMO where the players would not be facing each other in factions, but rather try to survive in small factions/groups against an AI constantly trying to destroy them/the world, and the servers could be won when the AI is destroyed, or lost when the world is. Outside gaming there are other types of applications we have thought of. An advanced AI which would function as a teacher for real people. The agents job would be to learn how best to teach the user by trying different teaching approaches and see which works best. A way to do so could be to put specific types of subjects on the tabular side and types of learning methods on the feature side. These ideas are only that though, ideas. The thesis needs more work, and then more research is needed for any of the ideas to work.

Bibliography

[1] Andy Wellings Alan Burns. *Real-Time Systems and Progamming Languages - Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley, 2009.

[2] David Poole, Alan Mackworth. Artificial Intelligence: Foundations of computational agents., Printed 2010. URL `http://artint.info/html/ArtInt_262.html`. [Online book; Chapter 11.3; Last accessed 05-06-2014].

[3] Andy Khan. Java Excel API - A Java API to read, write and modify Excel spreadsheets, March 2013. URL `http://www.andykhan.com/jexcelapi/`. [Online article; Last accessed 21-5-2014].

[4] Kim Arnold Thomsen, Rasmus D. C. Dalhoff-Jensen. *Helper Dog for Minecraft*. Aalborg University - Student report, 2014.

[5] Mark Humphry. Reinforcement Learning, January 2014. URL `http://www.compapp.dcu.ie/~humphrys/PhD/ch2.html`. [Online article; Chapter 2; Last accessed 8-1-2014].

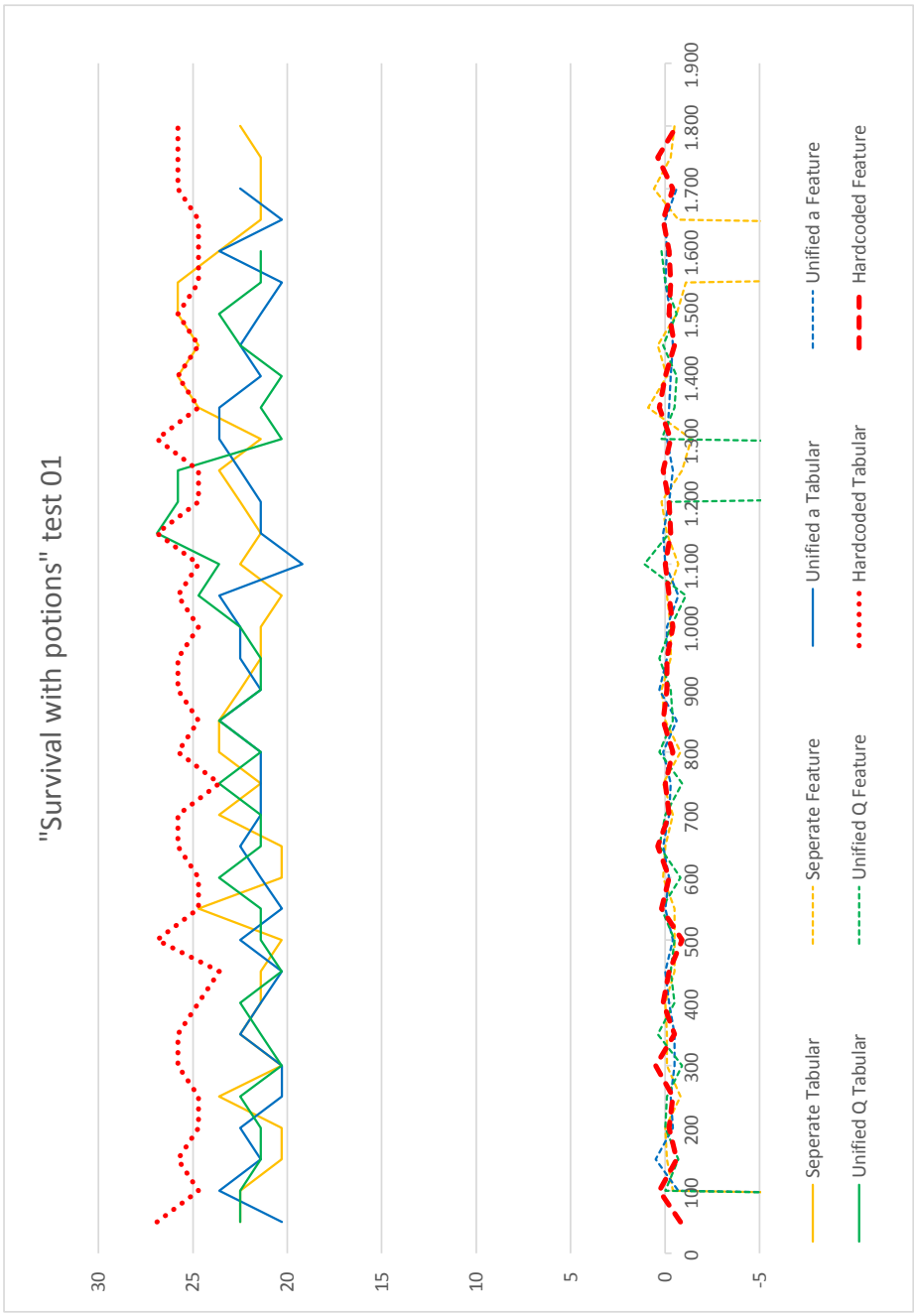
[6] Mojang. Terms of Use, December 2013. URL `https://minecraft.net/terms`. [Online; Last accessed 05-06-2014].

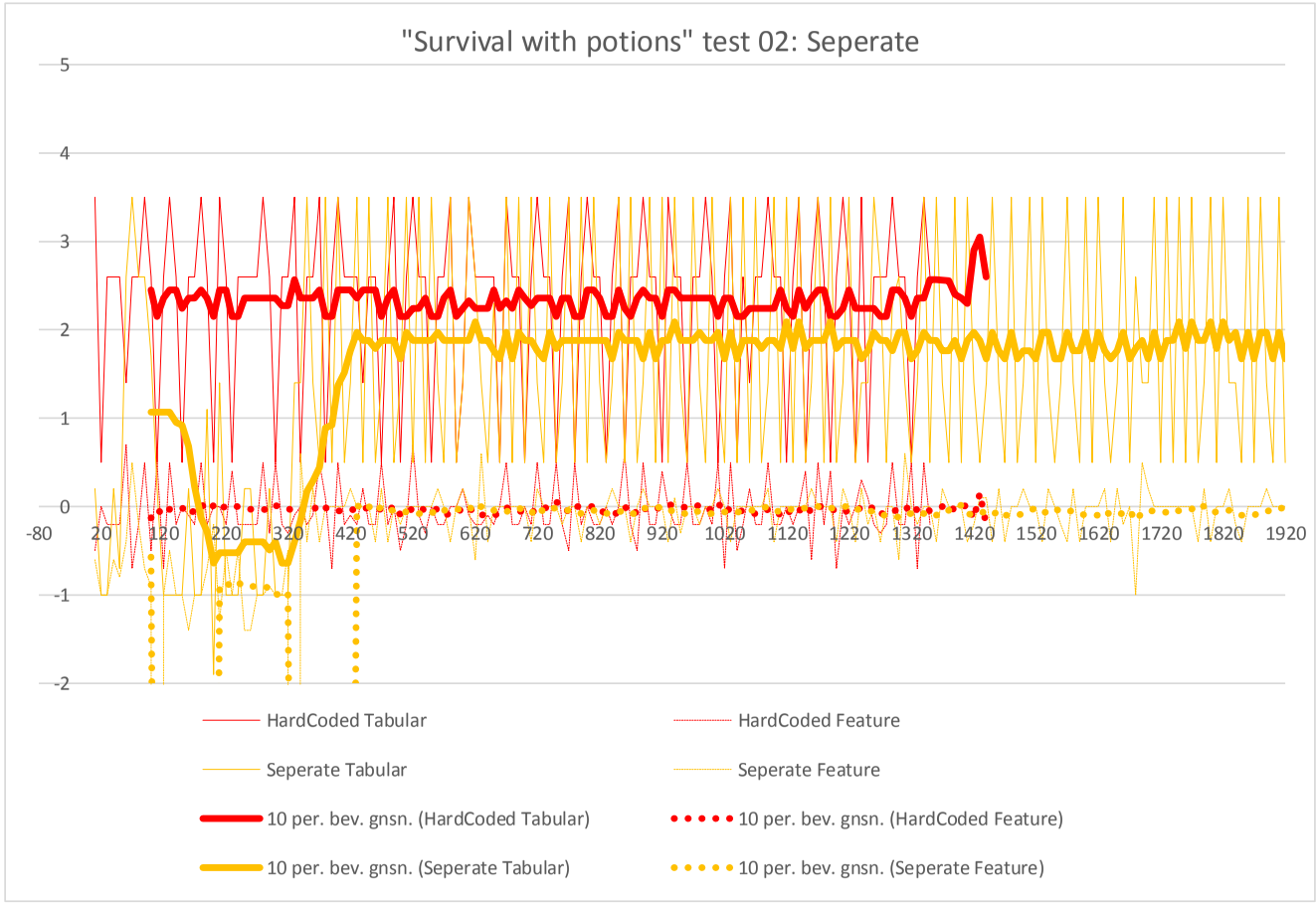
[7] Peter R. Turner. *Guide*^{2} *Scientific Computing*. Macmillan Mathematical Guides, 2000.

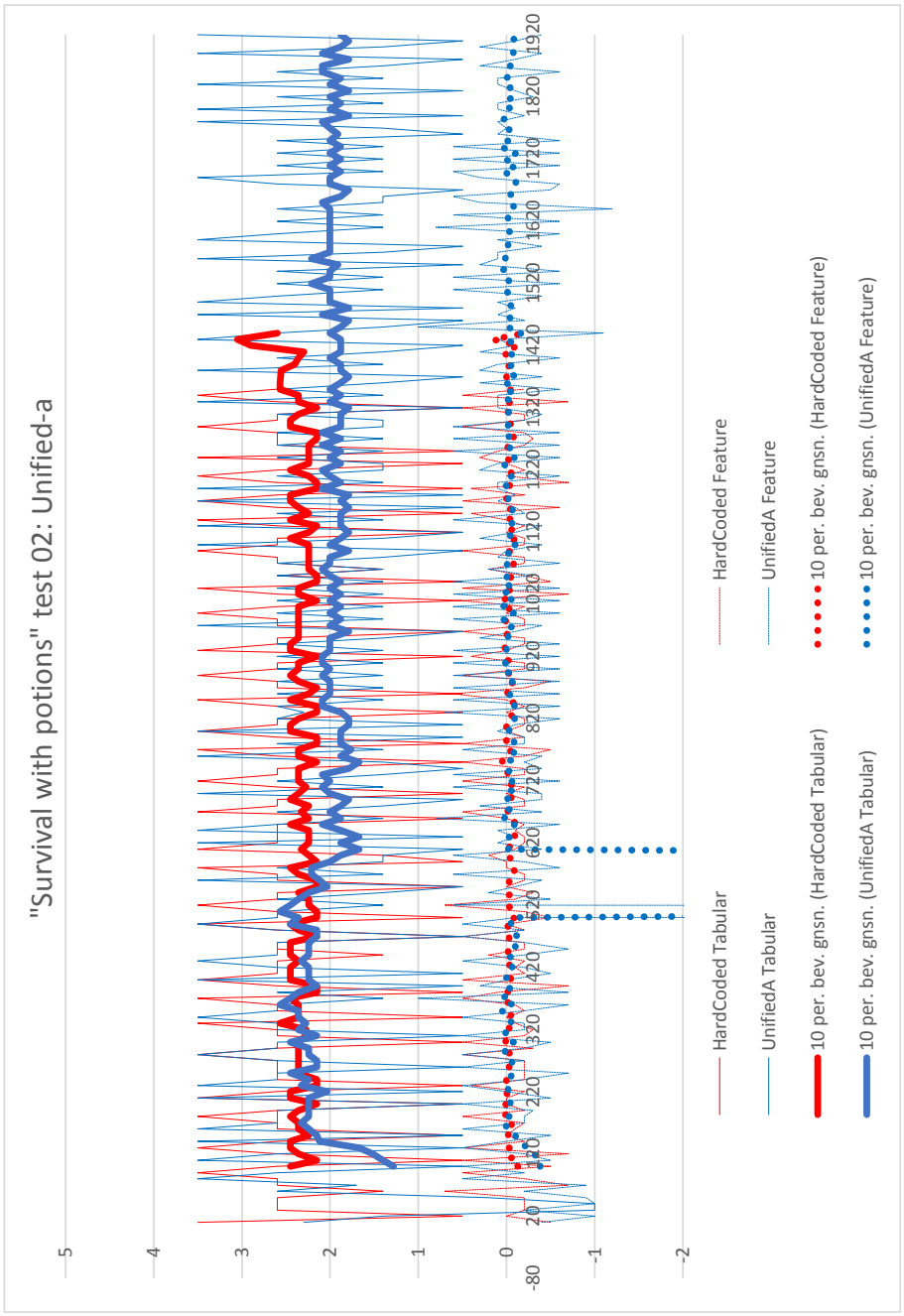
[8] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Machine Learning*, 8:279–292, 1992.

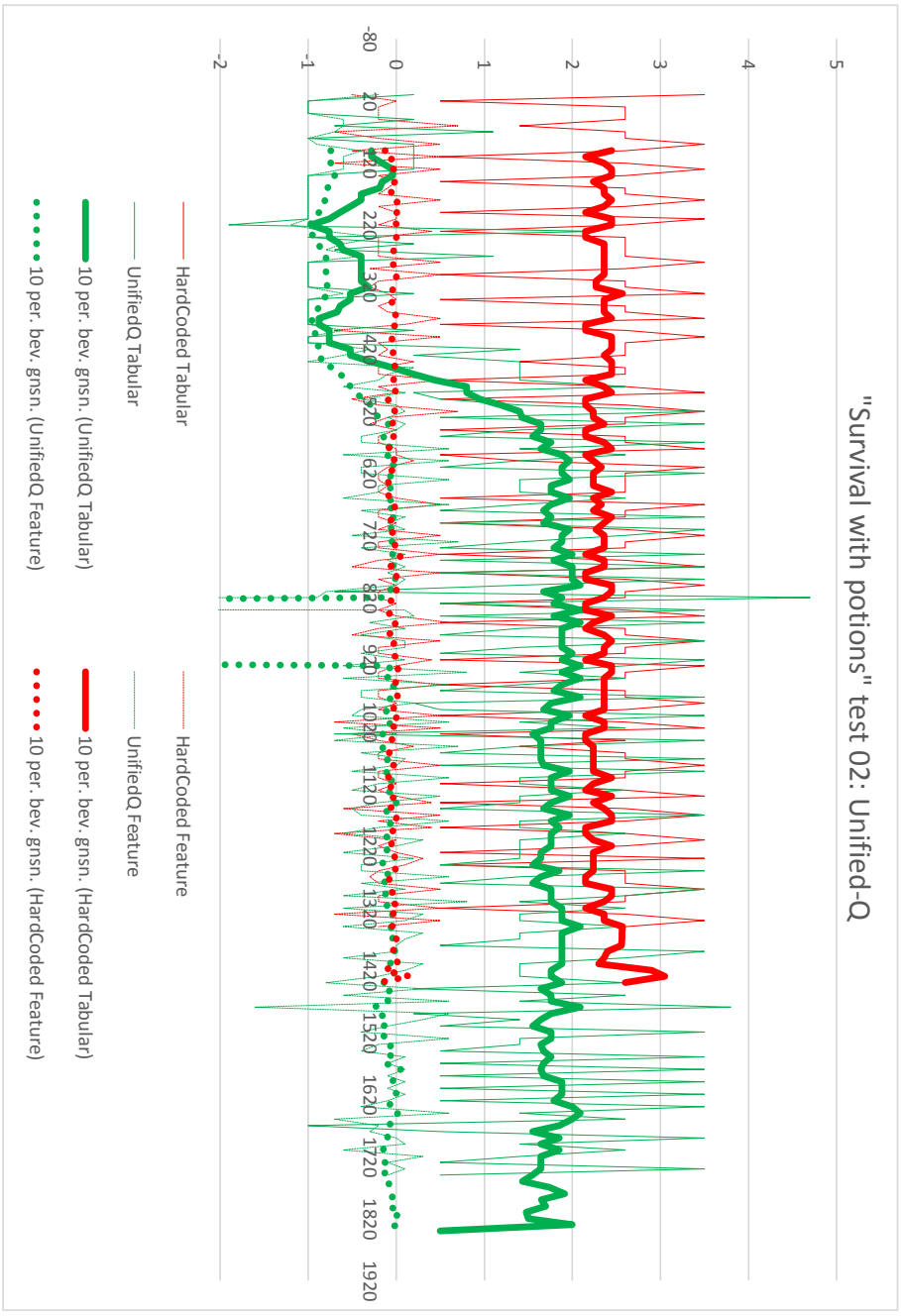
Part VI

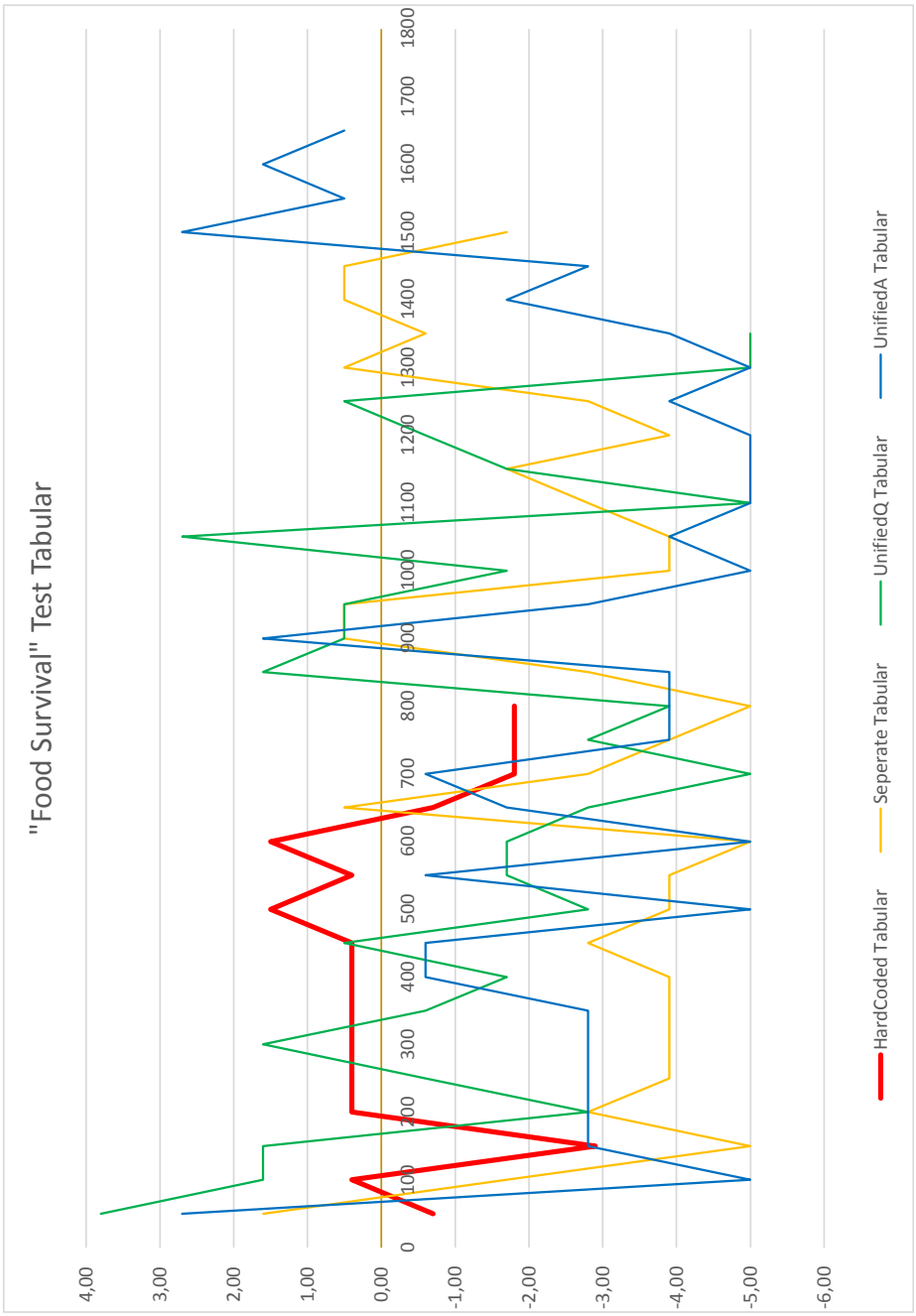
Appendex

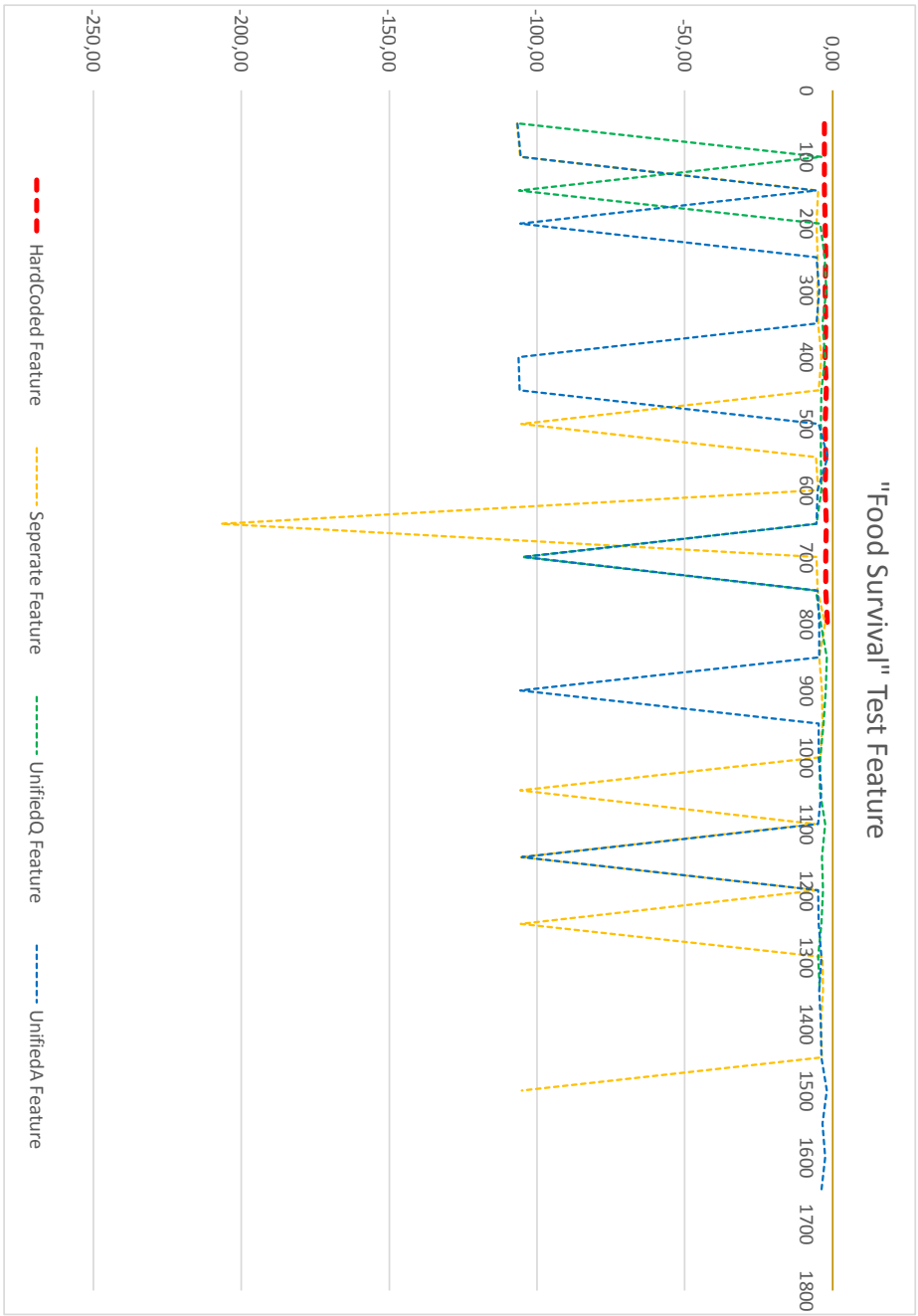

"Survival with potions" test 01
30
25
20
15
10
5
0
-5
0
100
200
300
400
500
600
700
800
900
1.000
1.100
1.200
1.300
1.400
1.500
1.600
1.700
1.800
1.900
Seperate Tabular
Seperate Feature
Unified a Tabular
Unified a Feature
Unified Q Tabular
Unified Q Feature
Hardcoded Tabular
Hardcoded Feature

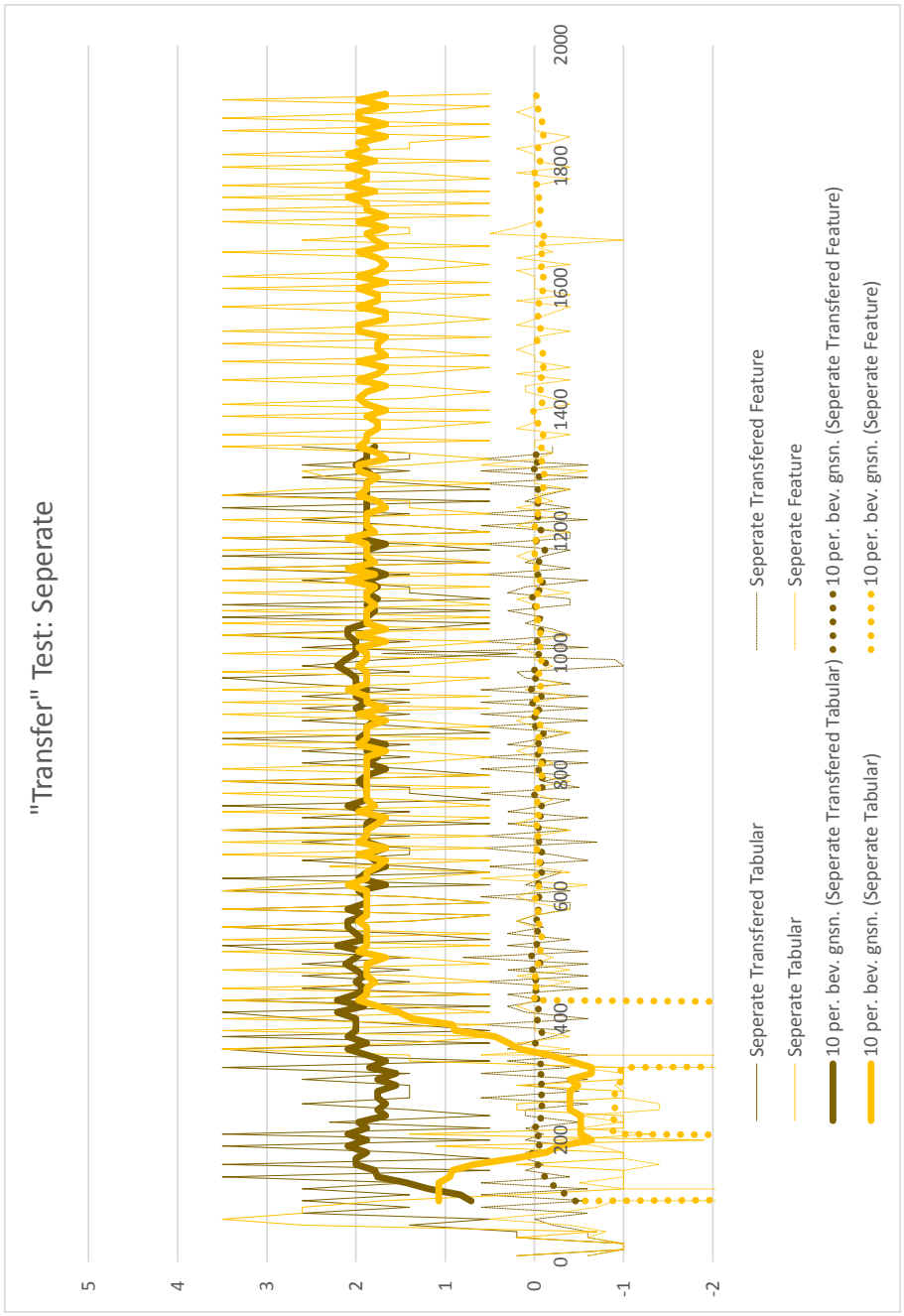

"Survival with potions" test 02: Seperate
5
4
3
2
1
0
-1
-2
-80
20
120
220
320
420
520
620
720
820
920
1020
1120
1220
1320
1420
1520
1620
1720
1820
1920
HardCoded Tabular
HardCoded Feature
Seperate Tabular
Seperate Feature
10 per. bev. gnsn. (HardCoded Tabular)
10 per. bev. gnsn. (HardCoded Feature)
10 per. bev. gnsn. (Seperate Tabular)
10 per. bev. gnsn. (Seperate Feature)

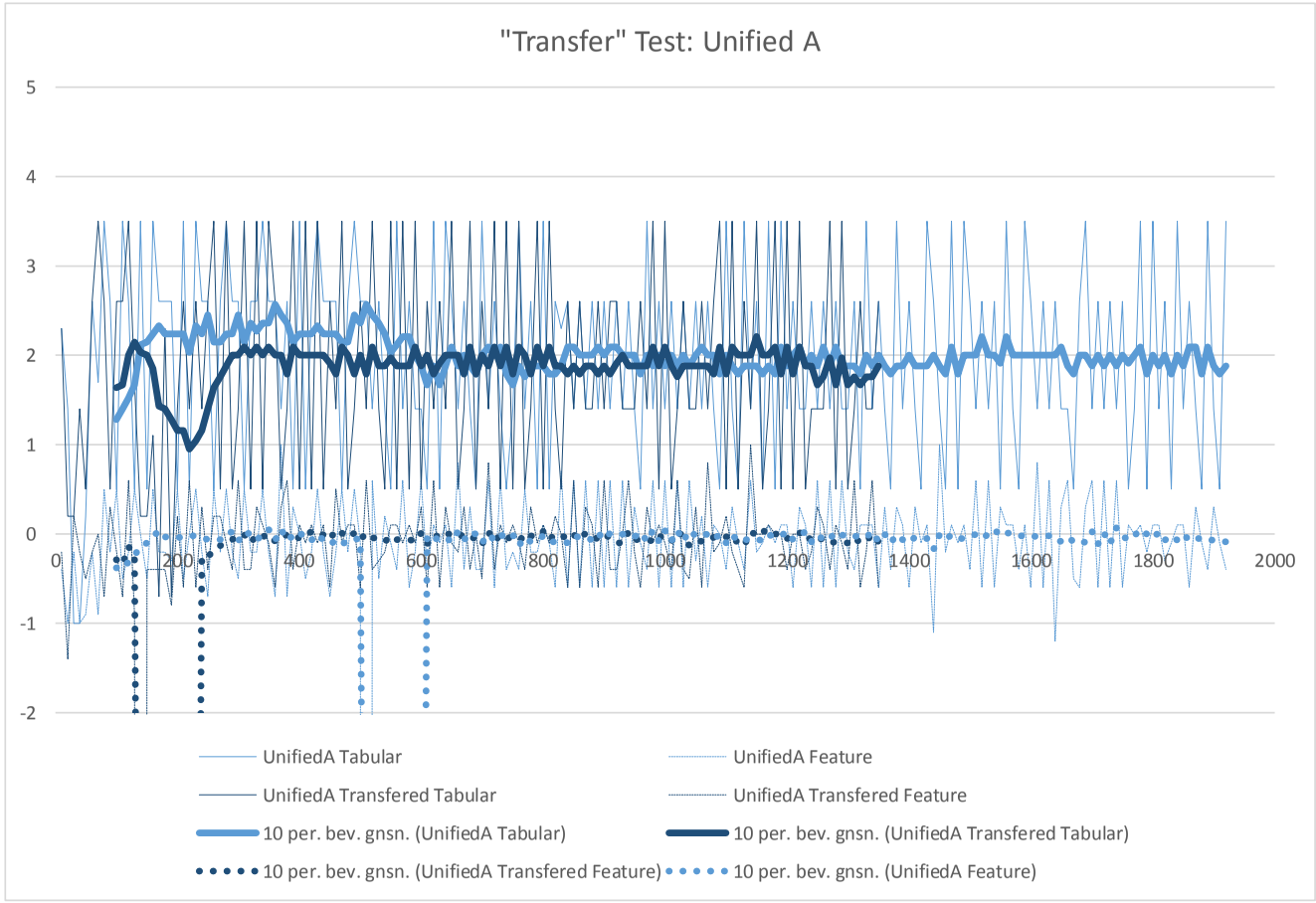

"Survival with potions" test 02: Unified-a
5
4
3
2
1
0
-1
-2
-80
20
120
220
320
420
520
620
720
820
920
1020
1120
1220
1320
1420
1520
1620
1720
1820
1920
HardCoded Tabular
UnifiedA Tabular
10 per. bev. gnsn. (HardCoded Tabular)
10 per. bev. gnsn. (UnifiedA Tabular)
HardCoded Feature
UnifiedA Feature
10 per. bev. gnsn. (HardCoded Feature)
10 per. bev. gnsn. (UnifiedA Feature)

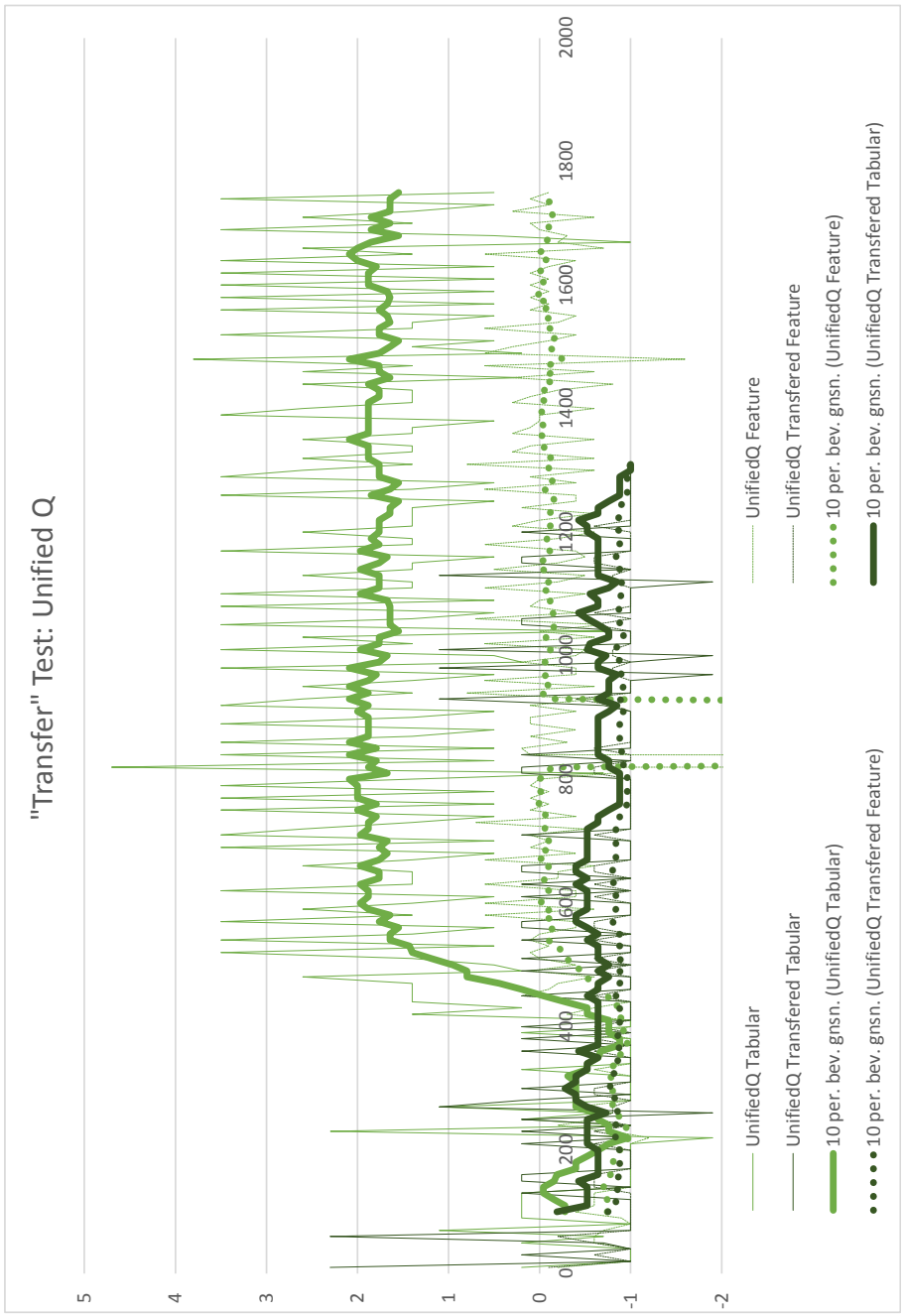

"Survival with potions" test 02: Unified-Q
5
4
3
2
1
0
-1
-2
-80
20
120
220
320
420
520
620
720
820
920
1020
1120
1220
1320
1420
1520
1620
1720
1820
1920
HardCoded Tabular
UnifiedQ Tabular
10 per. bev. gnsn. (UnifiedQ Tabular)
10 per. bev. gnsn. (UnifiedQ Feature)
HardCoded Feature
UnifiedQ Feature
10 per. bev. gnsn. (HardCoded Tabular)
10 per. bev. gnsn. (HardCoded Feature)


"Food Survival" Test Tabular
4,00
3,00
2,00
1,00
0,00
-1,00
-2,00
-3,00
-4,00
-5,00
-6,00
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
HardCoded Tabular
Seperate Tabular
UnifiedQ Tabular
UnifiedA Tabular


"Food Survival" Test Feature
0,00
-50,00
-100,00
-150,00
-200,00
-250,00
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
HardCoded Feature
Seperate Feature
UnifiedQ Feature
UnifiedA Feature


"Transfer" Test: Seperate
5
4
3
2
1
0
-1
-2
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Seperate Transfered Tabular
Seperate Tabular
10 per. bev. gnsn. (Seperate Transfered Tabular)
10 per. bev. gnsn. (Seperate Tabular)
Seperate Transfered Feature
Seperate Feature
10 per. bev. gnsn. (Seperate Transfered Feature)
10 per. bev. gnsn. (Seperate Feature)


"Transfer" Test: Unified A
5
4
3
2
1
0
-1
-2
0
200
400
600
800
1000
1200
1400
1600
1800
2000
UnifiedA Tabular
UnifiedA Feature
UnifiedA Transfered Tabular
UnifiedA Transfered Feature
10 per. bev. gnsn. (UnifiedA Tabular)
10 per. bev. gnsn. (UnifiedA Transfered Tabular)
10 per. bev. gnsn. (UnifiedA Transfered Feature)
10 per. bev. gnsn. (UnifiedA Feature)


"Transfer" Test: Unified Q
5
4
3
2
1
0
-1
-2
0
200
400
600
800
1000
1200
1400
1600
1800
2000
UnifiedQ Tabular
UnifiedQ Transfered Tabular
10 per. bev. gnsn. (UnifiedQ Tabular)
10 per. bev. gnsn. (UnifiedQ Transfered Feature)
UnifiedQ Feature
UnifiedQ Transfered Feature
10 per. bev. gnsn. (UnifiedQ Feature)
10 per. bev. gnsn. (UnifiedQ Transfered Tabular)