Q-learning For Action Selection

Ordelman, S.

Q-LEARNING FOR ACTION SELECTION

by

Sven Ordelman

Student number:		
Course code:	IM9906	
Date:	1st of June 2022	
Thesis committee:	Prof. Dr. Tanja E.J. Vos (chairman),	Open University
	Dr. Pekka Aho (supervisor),	Open University
	Olivia Rodriguez Valdés (supervisor),	Open University

Research Organisation

Thesis information	
Title	Q-learning For Action Selection
Date	1st of June 2022
Course code	IM9906
Degree programme	Open University of the Netherlands, Faculty of Science, Master's Programme in Software Engineering
Student	
Name	Sven Ordelman
student ID number	852203128
Graduation committee	
Supervisor and Chairman	
Name	Prof. Dr. Tanja E.J. Vos
Supervisor	
Name	Dr. Pekka Aho
Supervisor	
Name	Olivia Rodriguez Valdés

Acknowledgment

Whilst doing this study and creating the thesis I have received a great deal of support and assistance.

First I would like to thank my supervisors, Prof. Dr. Tanja E.J. Vos, Dr. Pekka Aho and Olivia Rodriquez from formulating the research questions until finishing the thesis their expertise was invaluable and helped me a lot.

I would also like to thank and acknowledge Moneybird and in special Joost Diepenmaat, Remco de Man en Robbin Voortman for providing the resources and setting up the online environments for testing the Moneybird Application.

Abstract

By testing, bugs can be detected before they are shipped to the end user. Most systems are tested manually or through scripted testing. Testing can also be automated so that tests occur without intervention of a tester. This study is about an automated testing tool called TESTAR. TESTAR tests systems through its GUI, to do this it needs to select actions. Here is where one problem resides with automated testing tools. Which actions does it need to select to reach a high test-effectiveness? In the basis TESTAR selects actions at random, but there has been prior research in using Q-learning for action selection. Q-learning is mostly a model-free approach to machine learning where rewards are given based on the selected action. Based on these rewards the algorithm can make thoughtful decisions. This study builds upon prior research for action selection in TESTAR by looking further into Q-learning algorithms. Previously TESTAR only had a basic Q-learning algorithm.

The main research question of the study is *"How can the test-effectiveness of TESTAR be increased using Q-learning in the action selection step?"*

which is answered using three sub-questions,

- RQ1 - How can different Q-learning algorithms in TESTAR be implemented?
- RQ2 - What reward functions can be used for the implemented Q-learning algorithms in TESTAR based on the available reward metrics?
- RQ3 - How can the performance (test-effectiveness) of a combination of Q-learning algorithm and reward function in web-based applications be measured?

In the first research question two new Q-learning algorithms are implemented, *Double Q-learning* and *QV-learning*. The algorithms were tested on the java application Rachota and showed promising result compared to Basic Q-learning. Which takes us to the second Q-learning where new rewards are introduced. In total there are three existing rewards and this study introduces three new ones. All of those rewards in combination with the three Q-learning algorithms are tested against the Rachota application. In this research question Rachota was also tested using pure random action selection and after comparing the results, promising results were found. The new QV-learning algorithm in combination with a new reward showed a 10% improvement over pure random actions selection. In the third research question these results were confirmed against the web-based application Moneybird which showed that the Q-learning algorithms were not only applicable to java applications but also on web-based applications. Here is also introduced how test-effectiveness can be tested on web-based applications using a coverage API.

This study confirms the main research question and shows improvements in the action selection step using Q-learning. Possible improvement for future work are to give TESTAR

better understanding of forms to prevent a blocking state and to test with other Q-learning algorithms and Q-learning configurations which were out of scope for this study.

Contents

1 Introduction

An application can never be totally bug free. But many bugs can be prevented using testing. Testing can be done by hand, by following a sequence manually, or scripted. A scripted test sequence is written once and can be used every time the system is tested. This is called *scripted testing*. There are also tools that can generate test cases for you without manual testing or scripts. This will lower the need for manual labour since such a tool only needs to be configured once and can be used every times the system needs to be tested. This means, after the initial setup, testing becomes almost effortless. In this thesis we will focus on script-less test automation. We will reference to testing with such a tool as *script-less testing*.

In this thesis, we look especially into script-less testing of a system through its GUI (Graphical User Interface) with TESTAR. TESTAR is a tool that automates testing by going through a system's GUI and generating test cases for the system. If the SUT (System Under Test) crashes or reaches an unwanted state, TESTAR reports this failed test case back to the user.

1.1. AUTOMATING GUI TESTING

As said before, testing a system can be time consuming. This is due to the potentially large execution space of the application [20]. In the software industry, high level tests, such as GUI tests, are often executed manually. This is costly, tedious and error prone[2]. This manual labour can be automated the same way unit tests verify the backend code. High-level GUI tests can be automated by scripting the actions and expectations into test cases, but this still requires manual labour upfront. Examples of tools that allow the user to create *script-based* instructions for automating tests are JFCUnit[1], Selenium Webdriver[2] and Robotium[3] [21] . Another way of creating automated tests is by *capture/replay*, this enables

[1]`http://jfcunit.sourceforge.net/`
[2]`https://www.selenium.dev/documentation/en/webdriver/`
[3]`https://github.com/RobotiumTech/robotium`

the tester to record the manual interaction with the UI. Tools, such as Test Automation FX[4], Selenium IDE[5] and Micro Focus Unified Functional Testing[6] will translate the manual test in a sequence that can later be replayed when the system is under test [21].

But for each newly added functionality, test cases need to be manually created. This is still a tedious and error prone job. To further automate this, tools such as GUITAR[7][12] allow for automated generation of test cases. GUITAR interacts with all UI widgets to try to understand the application. After all interactions are tested, GUITAR can generate test cases for the application. Another tool for automated testing through the GUI and the tool this thesis is about is TESTAR[8] [3].

There are many ways to automate GUI testing, but this sums up a few methods, the goal is to reduce the manual labor cost and still have a thoroughly tested system. Newer tools that do fully automated testing at GUI level also try to improve the total test coverage of the system over manual testing. This means it tests more cases and parts of a system than a human test designer would think about.

1.2. GOAL OF THE RESEARCH

The goal of the research is to improve automated testing at GUI-level using TESTAR. In the basis, TESTAR generates test-cases, it does this by navigating through a system under test. Most of the time the system has more than one action available and TESTAR should choose the most suiting action. Mostly, TESTAR selects actions randomly but this can be improved by introducing action selection algorithms to TESTAR. For example, Q-learning[26] which is a form of reinforcement learning. Using Q-learning, TESTAR learns which actions are the most suitable whilst testing a system. This research will try to improve the test-effectiveness of TESTAR by improving the action selection algorithm with Q-learning. The research will first try to improve the test-effectiveness on Java applications and after that it will use these results on web-based applications to show that the results are dynamically applicable. The eventual goal is to achieve better test-effectiveness compared to random testing.

1.3. STRUCTURE OF THE DOCUMENT

After the Introduction, the document starts of with a chapter about TESTAR. The subsequent chapter is about Q-learning. The fourth chapter of the document is Related Work that gives more information about closely related subjects to automated GUI testing. After the related work, the methodology for the research and research questions will be explained in chapter Research. The sixth, seventh and eight chapter are about RQ1, RQ2 and RQ3 respectively. The ninth chapter is the conclusion which consists of the conclusion and suggestions for future work. The document concludes with the Research Organisation chapter.

[4]http://www.testautomationfx.com/

[5]https://www.selenium.dev/selenium-ide/

[6]https://www.microfocus.com/en-us/products/uft-one/overview

[7]https://sourceforge.net/projects/guitar/

[8]https://testar.org/

2
TESTAR

As previously introduced, TESTAR is a tool for the automated testing of a system through its GUI. In the basis, it does not need much setup and it can start testing an application immediately. A TESTAR sequence is further visualised in Figure 2.1.

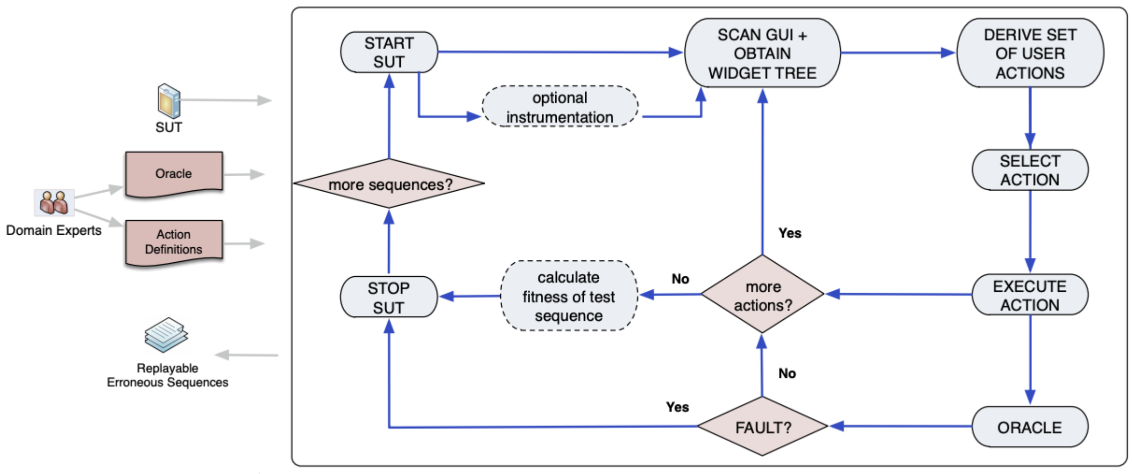

Figure 2.1: TESTAR sequence[31]

TESTAR works using a basic sequence that consists of six steps. These six steps are the following[31]:

1. TESTAR starts the System Under Test (SUT);
2. TESTAR obtains the GUI's state, the state consists of the visible widgets represented in a tree and their properties like position, size, etc.
3. TESTAR derives a set actions;
4. TESTAR selects and executes an action;

5. TESTAR checks if the state resulting from the action is valid. If the state is invalid, the sequence generation is stopped and the sequence leading to the invalid state is saved and can be replayed;

6. TESTAR goes to step 1, unless the given amount of sequences is generated. A sequence in TESTAR is a test run of the SUT with a configurable amount of actions. If the amount of actions is reached or TESTAR finds an invalid state the sequence is ended.

Sequences closely related to the research are further explained in the following subsections.

2.1. TESTAR DERIVES A SET ACTIONS

TESTAR derives a set of actions from the widgets, possible clicks, text inputs, etc. It does this using algorithms and configurable filtering options [10; 32].

One of those algorithms is *sensible actions*, this algorithm determines which actions can probably be used on a widget. For example, a scrollbar can scroll and an input field cannot. Next to that, the system checks which actions are enabled or disabled. With this information, TESTAR can make better choices about which action it is going to select.

Another algorithm looks into prioritization, this algorithm, is called *deriving top level actions* and can be combined with the sensible actions algorithm. According to the researchers: "Widgets that are on top of the layout hierarchy are more likely to lead to actions that trigger state transitions", meaning that the application should favour top actions and work its way down the layout hierarchy. This makes it possible to prioritize top-level widgets by including the detected hierarchy.

Another prioritization algorithm is *deriving new actions*. This algorithm compares the previous actions to the actions currently available. According to the researchers, prioritizing new actions would possibly lead to creating new sequences.

To further improve the testing, it should also be possible to tell TESTAR which actions it should not perform. This must be possible, because doing incorrect actions could lead to unwanted states. For example, minimizing and maximizing a window or clicking on external links would not lead to new interesting test result. That is why TESTAR should concentrate on the SUT only. To do this, TESTAR includes the possibility to *filter actions*. Filtering actions is possible with the use of regular expressions. TESTAR checks if a UI widget confirms to a regular expression and would not derive that action. Another way is the click filter that can used by tester to select widgets in TESTAR that should not be executed. The most flexible way is to use *programmatic filtering* that allows the tester to use the TESTAR Java protocol to filter certain widgets. This is demonstrated in example 2.1.

Example 2.1: Programmatic filtering

```
for (widget : state) {
    if (widget.get(WebTags.Href).contains("Undesired-URL")) {
        continue; // skip this widget
    } else { // derive actions as defined
```

Next to filtering, it is also possible to use *pre-specified actions* which allows TESTAR to use a login form in one go for example. This makes testing of a SUT faster and prevents it from hanging on a certain page without progress because the page is dependent on specific input. This is demonstrated in example 2.2.

Example 2.2: Pre-specified actions

```
for (widget : state) {
    if (widget.get(Tags.Title).contains("Signin")) {
        new CompoundAction.Builder()
            .add(new Type("username"))
            .add(new KeyDown(KBKeys.VK_TAB))
            .add(new Type("password"))
            .add(new Keydown(KBKeys.VK_ENTER)).build();
    }
}
```

2.2. TESTAR SELECTS AND EXECUTES AN ACTION

TESTAR selects and executes an action, by default the action selection mechanism selects an action randomly. It does this by looking at the derived actions set and randomly executes one of those actions.

Much improvement can be done with action selection, the first improvements were implemented in the derived actions step. But, in the action selection step there is also much room for improvement. Recent studies look into using reinforcement learning, especially Q-learning, to select the actions [10; 32]. Q-learning is a machine learning algorithm which most of the time does not need to be pre-trained since most Q-learning algorithms are model free. By implementing reinforcement learning in TESTAR, TESTAR works as shown in algorithm 1.

Algorithm 1 RL algorithm of Anna Esparcia et. al.

```
Require: $R_{max}$                                          ▷ max reward: $R_{max} > 0$
Require: $\gamma$                                           ▷ discount: $0 < \gamma < 1$
Require: $\alpha$                                           ▷ learning rate
 1: $initializeQValues()\ sequence$
 2: $s \leftarrow getStartingState()$
 3: repeat
 4:     $availableActions \leftarrow getAvailableActions(s)$
 5:     $a \leftarrow selectMaxAction(availableActions)$          ▷ Select action
 6:     $performAction(a)$
 7:     $s' \leftarrow getReachedState()$
 8:     $reward \leftarrow \begin{cases} R_{max} & \text{if } x_a = 0 \\ \frac{1}{x_a} & otherwise \end{cases}$     ▷ Reward for action
 9:     $Q(s, a^*) \leftarrow reward + \gamma\ max_{a \in A_{s'}} Q(s', a)$     ▷ Learn
10:     $s \leftarrow s'$
11: until s' is the last state of the sequence
12:
```

The researchers [32] found out Q-learning was superior over testing with random actions if the parameters were selected correctly. But, there is also much room for improvement by exploring other and possibly better metrics for calculating the rewards. This will further improve Q-learning in TESTAR.

The researchers [32] knew the changes were improvements since they designed a way to measure and compare performance to another configuration in TESTAR application. To achieve this, they used three metrics to calculate the performance, the first is *abstract states*, i.e. "the number of different states, or windows in the GUI, that are visited in the course of an execution". The second metric is the *longest path*, which checks if the deeper parts of the application are visited and not only the surface. They do this by calculating the longest path where no loops occur. The third and last metric is the *Minimum and Maximum coverage per state* which keeps track of the minimum and maximum state coverage. This number is the rate of executed actions over the total available actions in a given state. With this performance score TESTAR can be compared to another configuration of TESTAR. In the case of this study, different action selection algorithms can be compared.

2.3. Goal of TESTAR

The goal of TESTAR is to improve test-effectiveness without having to deal with the high labour cost of manually testing the application. The manual creation of test cases can be error prone and is very costly, especially in applications under continuous development [20]. If TESTAR can fully replace manual forms of testing it can speed up the development process and increase test-effectiveness. This results in an application which is more thoroughly tested. Bugs can be detected earlier and automatically.

3 Q-learning

Q-learning is mostly a model-free approach to machine learning using reinforcement learning. Reinforcement learning *learns during the execution* of the application[26], compared to the process consisting of two steps, training and inference, which is needed for supervised and unsupervised machine learning.

Q-learning can be used to train an entity called an agent to accomplish a given task. This agent receives the current state of the application by observing the environment. Which in the case of the study is an application, by observing the application it builds the state accordingly. Upon retrieval of the state the agent chooses the next action. The environment executes the selected action which results in a change of state. The agent receives this new state and assigns a reward to the executed action. Based on this reward, a positive or negative number, the agent trains itself to create a policy for selecting the best actions.

Q-learning is based on the concept of a *Quality Matrix*. This is a matrix with a size of $N \times Z$. N is the number of possible states the environment can be in. Z is the number of possible actions an agent can perform. When selecting a row of the Quality Matrix, which represents a state, the agent computes the maximum value in a row and selects the corresponding actions. The Quality Matrix is initialized with random values or zeros and is updated by using formula (3.1) [26]:

$$Q_{new}(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a)\Big) \tag{3.1}$$

In (3.1) the variables are referring to:

- s_t, s_{t+1} are the current and next state in the environment.
- a_t, a_{t+1} are the current and next selected action by the agent.
- γ is the discount factor, which is $\gamma \in [0,1]$. It states how much of the long-run rewards need to be taken into account, compared to the immediate rewards.
- α is the learning rate, which is $\alpha \in [0,1]$. It states how much the newer knowledge needs to replace the older knowledge.

- r_t is the reward value.

With this formula it is possible to continuously update the Quality Matrix. With the updates in the matrix the agent learns and builds an *action policy*. This does not imply that the action policy is a good policy. A key factor in Q-learning is selecting the right metrics to determine the reward. If the reward is not good, the resulting policy will never be good.

Above we explained the basic implementation of Q-learning, but there are many other variants of Q-learning which are divided in two groups. Single agent and Multi agent algorithms [15]. In the case of action selection we only need one agent, the one that selects the action. That is why we only look into 5 single agent Q-learning algorithms:

1. Incremental multistep
2. Deep Q-learning
3. QV-learning
4. Double Q-learning
5. Hierarchical Q-learning

Which are further explained in the following sections.

3.1. INCREMENTAL MULTISTEP

In comparison to the default single step Q-learning algorithm, basic Q-learning (which was previously introduced as the basic implementation), incremental multistep allows a credit to be carried through multiple steps. In this case a credit is a variable which impacts the rewards of multiple states. It does this by using Temporal-Difference learning [28] for creating estimates of the outcome of actions. It creates these estimates based on the maximum rewards of the previous states. This makes the previous states impact the eventual reward which means that each given reward is a credit which influences multiple states. This estimate is combined with the basic Q-learning[23] algorithm which results in an Incremental Multistep algorithm reward. The algorithm is shown in algorithm 2.

Algorithm 2 Incremental Multistep Q-learning by Peng et al.

```
1:  $Q(x,a) = 0$ and $Tr(x,a) = 0$ for all $x$ and $a$
2:  repeat
3:      $x_t \leftarrow$ the current state
4:      Choose an action $a_t$ according to current exploration policy
5:      Carry out action $a_t$ in the world. Let the short-term reward be $r_t$, and the new state be $x_{t+1}$
6:      $e_t^{'} = r_t + \gamma V_t(x_{t+1}) - Q_t(x_t, a_t)$
7:      $e_t = r_t + \gamma V_t(x_{t+1}) - V_t(x_t)$
8:      for each state-action pair $(x,a)$ do
9:          $Tr(x,a) = \gamma\lambda Tr(x,a)$
10:         $Q_{t+1}(x,a) = Q_t(x,a) + \alpha Tr(x,a) e_t$
11:     end for
12:     $Q_{t+1}(x_t,a_t) = Q_{t+1}(x_t,a_t) + \alpha e_t^{'}$
13:     $Tr(x_t,a_t) = Tr(x_t,a_t) + 1$
14: until end
```

3.2. DEEP Q-LEARNING

Deep Q-learning is a combination of Convolutional Neural Networks with basic Q-learning. Convolutional Neural Networks can be visualized as trees with each branch having its own weight. Based on those weights an input is parsed into an output. The output resembles the estimation of the Neural Network. But to make a good estimation, the weights for each branch must be determined. A Neural Network does this by training from pre-defined samples. Which means that if a neural network is trained using states with as output the correct Q-values, it can make guesses about the Q-values for unseen states.

A trained Neural Network is called a model. In Deep Q-learning two models are used. The first model is the Q-network which is used to estimate the Q-values. It is continuously trained after an iteration using a random mini-batch of previous states, actions and rewards from the replay memory. This means that this model adapts to the system. The second model is used as the target network which is an unbiased estimator to avoid errors in training the Q-network. This target network synchronizes with the Q-network after a given number of iterations to make sure the Q-network still makes sense. Except for calculating the Q-values using the Q-network the algorithm works the same as basic Q-learning [5].

3.3. QV-LEARNING

QV-learning is almost the same as basic Q-learning but instead of learning from actions (rewards) the Q-value learns from the V-value which is updated using a Temporal-Difference learning. This means it does not consider actions which is the case when only using basic Q-learning [33]. The Q-value of QV-learning is calculated using equation 3.2.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha\Big(r_t + \gamma V(s_t + 1) - Q(s_t, a_t)\Big) \tag{3.2}$$

The big difference between basic Q-learning is the usage of the V-value which can be continuously calculated using equation 3.3.

$$V(s_t) := V(s_t) + \beta\Big(r_t + \gamma V(s_t + 1) - V(s_t)\Big) \tag{3.3}$$

The pseudo code for this algorithm is shown in algorithm 3.

Algorithm 3 QV-learning

```
1: $Q(s,a) = 0$ and $V(s) = 0$ for all $s$ and $a$
2: repeat
3:     Choose action $a$ based on $Q(s,\cdot)$, observe $r$, $s'$
4:     $V(s_t) \leftarrow V(s_t) + \beta(r_t + \gamma V(s_t + 1) - V(s_t))$               ▷ Calculate the $V(s_t)$ value
5:     $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma V(s_t + 1) - Q(s_t, a_t))$   ▷ Calculate $Q$ value using the $V$ value
6:     $s \leftarrow s'$
7: until end
```

3.4. DOUBLE Q-LEARNING

Q-learning does not perform well in an environment with a random variable. That is why double Q-learning was invented. In comparison to basic Q-learning, double Q-learning has two Q-functions each updated with the value of another Q-function. Both Q-functions should learn from a different set of experiences and both values are used to select the next action. By training two Q-functions the random variable is less prominent [29].

The Q-values for Double Q-learning are updated using algorithm 4.

Algorithm 4 Double Q-Learning by Hado et al.

```
1: Initialize $Q^A, Q^B, s$
2: repeat
3:     Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s^{'}$
4:     Choose (e.g. random) either UPDATE(A) or UPDATE(B)
5:     if UPDATE(A) then
6:         Define $a^* = arg\ max_a Q^A(s^{'}, a)$
7:         $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s^{'}, a^*) - Q^A(s,a))$
8:     else if UPDATE(B) then
9:         Define $b^* = arg\ max_a Q^B(s^{'}, a)$
10:        $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s^{'}, a^*) - Q^B(s,a))$
11:    end if
12:    $s \leftarrow s'$
13: until end
```

3.5. Hierarchical Q-learning

To solve problems that arise when using Q-learning in a larger state-action space, hierarchical Q-learning was invented. It divides actions in higher and lower level actions. For example, driving a car can be tested with the higher level action, drive from A to B, and the lower level actions up, down, left and right. The main action, called an abstract action, is driving to B but lower level actions need to be used to get the result of the high level action [4]. The algorithm for hierarchical Q-learning is shown in algorithm 5.

Algorithm 5 Hierarchical Q-learning by Chen et al.[6]

```
1: Initialize $Q(hs, ha)$ arbitrarily.
2: repeatfor each episode
3:     Initialize $hs, k \leftarrow 0$
4:     repeatfor each step of episode
5:         Choose $ha$ for $hs$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
6:         Take action $ha$, observe $R$, $hs^{'}$
7:         (i) $Q_k(hs, ha) \leftarrow (1-\alpha_k)Q_{k-1}(hs, ha) + \alpha_k[R + \gamma max_{ha'} Q_{k-1}(hs^{'}, ha^{'})$
8:         (ii)        ▷ update the $Q$ value for corresponding landmark state $l$ and action $a$
9:         if $hs \in QS$ then
10:            $Q(l,a) = (1-\eta)Q(l,a) + \eta Q(ha, ha)$,
11:            $a \in A, \eta \in [0,1)$ ,
12:            $\eta = \eta^m, m > 1$ is a scalar constant.
13:        end if
14:        (iii) $hs \leftarrow hs^{'}$, $k \leftarrow k+1$
15:    until $hs$ is terminal
16: until learning process ends
```

4 Related Work

TESTAR is not the only testing tool that uses reinforcement learning. There are many testing tools which are using reinforcement learning. In this chapter, research which uses reinforcement learning with testing through the GUI is looked into.

4.1. Reinforcement learning with a greedy policy

4.1.1 Veanes et al.

In Veanes et al. [30] researchers are trying to test reactive systems. Since reactive systems can have spontaneous and unexpected outcomes, the researchers test the systems "Online", which is described as on-the-fly testing. The researchers are using a form of *model-based testing*. In model-based testing, the tester uses a model of the system's behaviour to detect any discrepancies between the IUT (Implementation Under Test) and the model. If the observed responses do not conform to the model or the IUT rejects an action which should be possible according to the model, a *conformance failure* occurs. A principal concern of online testing is choosing a *strategy* for selecting the actions. A bad strategy may fail in provoking certain behaviours or can take a very long time to achieve good coverage. Selecting actions at random is a wasteful strategy since this can cause the tester to select actions which were previously tested. That is why the researchers used reinforcement learning. The researchers created an algorithm for online testing a Robot. This algorithm uses Q-learning to determine which action it takes. For selecting actions they used three strategies. The first one (Lct) was a greedy strategy where they always selected the action with the lowest cost, allowing for fast exploration. The second strategy (Rlc) is choosing actions where the cost will likely be inversely proportional to its current cost. Meaning that the least frequent actions are favored. The third strategy (Rnd) was choosing the actions randomly. The outcome of their research was that Lct showed the most promising results. Both Lct and Rlc favored transitions to new abstract states but with Lct the coverage of the system increased faster. Rnd did not perform as well as Lct and Rlc. The implementation tries to minimize the cost of actions, the actions resulting in high rewards have a low cost.

For this thesis, Veanes et al. confirms that a random action selection approach is a bad approach. It also shows that a greedy approach is probably the best approach to choose. Implying that when choosing a policy for the Q-learning algorithms, a greedy policy should be favored. This is why for this study, a greedy approach is used.

This was one research about testing a system using reinforcement learning, but there are many others. Especially if you look into testing Android applications. Degott et al. [7], Koroglu et al. [16] [17] and Gu et al. [11] have all researched into testing an Android application through its GUI.

4.1.2 Degott et al.

In Degott et al. [7] researchers investigated how they could make the test suit understand interactions with the application. They needed to do this since Android applications can have a variety of interactions. Such as swiping, scrolling and clicking. Each app behaves differently and building a model which can detect the interactions for a certain app would probably fail to recognize possible interactions with another. To automatically test the application the researchers needed a technique that adapts the model at run-time. It does this using a pre-conceived notion of likely interactions which gradually become less-likely if the actions do not succeed. The researchers used reinforcement learning to achieve this. The researchers used two strategies, the first one being the $\epsilon - Greedy$ approach, which chooses between selecting a new action or the best-action based on its current knowledge. Since $\epsilon - Greedy$ needs a value for ϵ, which is used to determine the strategy exploration/-exploitation rate, the researchers did a small experiment. The reinforcement-learning algorithm with $\epsilon - Greedy$ was used on 5 randomly selected apps with 4 different values for ϵ, 0.1, 0.2, 0.3 and 0.4. Based on coverage a ϵ of 0.3 scored the highest and is used when gathering the results. The second strategy is the Thompson approach. The Thompson approach is very similar $\epsilon - Greedy$ but it calculates the probability of an action differently. The Thompson approach creates a *B-distribution* based on the wins and trails for each state action. *Wins* and *trails* are variables the test suite keeps track of. *Trails* is the number of how many times the action in a state is used. *Wins* is the sum of all rewards for that action. The reward is 1 if the action results in a GUI change, otherwise it is 0. The probability distributions are sampled for all the elements on screen and the best sample, the one with the highest value, is selected. The researchers found that by using the $\epsilon - Greedy$ and Thompson sampling approaches the baseline reinforcement-learning algorithm was improved by 18% and 24% respectively. The baseline is a reinforcement-algorithm without special action selection strategies.

Next to the two strategies there were also two types of reinforcement learning algorithms. One with, and one without previous knowledge. They created a knowledge base function to load wins and trails based on previous knowledge. This algorithm is shown in algorithm 6.

Algorithm 6 Knowledge-base function to reuse a priori knowledge alongside our $\epsilon - Greedy$ and Thompson sampling approaches by Degott et al.

```
1: function KNOWLEDGE-BASE(action, class)
2:     $p \leftarrow probability(action, class)$
3:     $wins \leftarrow p\ x\ \psi$
4:     $trails \leftarrow \psi$
5:     return wins, trails
6: end function
```

By comparing the reinforcement learning algorithms the researchers found that the algorithm without previous knowledge outperformed the one with by 8% in terms of statement coverage. To further improve the reinforcement learning algorithms they used a statically gathered model. This model is used as a basis in the reinforcement learning algorithm and is adjusted according to app behaviour. This algorithm is shown in algorithm 7

Algorithm 7 Fitness Proportionate Selection with reinforcement learning. Starting from the widget class probabilities from the crowd-based model and dynamically tuning these values according to the behavior of the app under testing by Degott et al.

```
1: Input: $n > 0$
2: function FITNESS-PROPORTIONATE-SELECTION-WITH-REINFORCEMENT LEARNING
3:     for $(a, c) \exists A \cdot C$ do
4:         $p \leftarrow crowd - based - model(action, class)$
5:         $wins \leftarrow p\ x\ \psi$
6:         $trails \leftarrow \psi$
7:         $P(a|C(w)) \leftarrow \frac{wins_{(a,C(w))}}{trails_{(a,C(w))}}$
8:     end for
9:     while stop criteria not met do
10:        $S = (w, P(a|w)) \forall w$ in the current screen
11:        $e \leftarrow originalStochasticSelectAlgorithm(S, P)$
12:        $r \leftarrow e.w.perform(e.a)$
13:        $wins_{(a,C(w)} \leftarrow wins + r$
14:        $trails_{(a,C(w))} \leftarrow trails + 1$
15:        $P(a|C(w)) \leftarrow \frac{wins_{(a,C(w))}}{trails_{(a,C(w))}}$
16:     end while
17: end function
```

The researchers found that with a statically gathered model they could improve the baseline reinforcement-algorithm by 20%. When using the statically gathered model with the $\epsilon - Greedy$ and Thompson the algorithm performed worse by 4% and 8% respectively. This shows that knowledge obtained during testing is more valuable than knowledge from a previous model.

For this thesis there can be built upon some of the results from Degott et al. For example, when using a greedy policy with a Q-learning algorithm the found exploration rate can be used. Next to the implementation of the greedy policy there is also known that it is better to prefer algorithms without previous knowledge over algorithms with. This is useful when choosing which Q-learning algorithms to implement in TESTAR.

4.2. Q-LEARNING FOR TESTING APPLICATIONS

4.2.1 Koroglu et al.

QLearning-Based Exploration (QBE) [16] is proposed: a fully automated black-box testing methodology that explores GUI actions using Q-Learning. Specifically, QBE learns from a set of existing applications the kinds of actions that are most useful in order to reach a particular objective such as detecting crashes or increasing activity coverage. The rewards applied with Q-Learning are based on activity or instruction coverage.

In another research regarding testing on Android applications from Koroglu et al. [17], the researchers highlight the problem that many automated GUI testing applications can obtain a high amount of activity coverage and are good in detecting errors. But, those applications cannot guarantee that they tested essential parts of the system. The researchers introduce the ability to write UI test scenarios written in a language called Gherkin. These test scenarios will be used by the application to check if the output of the Android application conforms to the scenario. The Android application is tested using reinforcement learning. Next to basic Q-learning the researchers also implemented Double Q-learning.

For this thesis it is important to note that measuring test-effectiveness does not always resemble the workings of the application. Essential parts of the system can remain untested. Another take away from Koroglu et al. is that the researchers state that Double Q-learning is more robust compared to basic Q-learning. It improves learning performance by lowering the maximization bias (overestimation). This occurs when wrong Q-values are used for learning which can be caused by a bad reward. It would be good to compare the findings of this thesis to the findings of Koroglu et al. to see if Double Q-learning also improves learning in TESTAR compared to basic Q-learning.

4.2.2 Adamo et al.

In Adamo et al. [1] researchers describe a Q-learning based technique to automatically explore the GUI of an Android application. They do this to improve code coverage in test suites in comparison to random test generation. The reward function used is a counter based reward function. The reward received is inversely proportional to the times an action has been executed. Meaning an action executed for the first time receives the highest reward. The researchers did not use the basic Q-learning function but created their own. The Q-function the researchers created takes into account the immediate reward and the reward from events in future states. This results in the possibility for an action resulting in a low reward to be taken if the rewards of the events in the resulting state are very high. Since the reward is only based on a counter the rewards for all the actions are easily calculated. The researchers also created a dynamic discount factor. This discount factor is proportional to the number of actions in a state. If there is a low number of possible actions in a state it has a higher discount factor. This is needed since the created Q-function can cause the agent to ignore states with a low number of actions, due to the fact that the actions are taken more often. By using a dynamic discount factor this can be reduced. The dynamic discount factor prioritizes future rewards over immediate rewards when having a low number of actions. The described implementation resulted in a improvement of 4.14% to 18.83%. Depending on the system under test these are promising results.

For this study the research is very relevant since it tries to solve a very similar problem but on another platform. The reward function used in this study is similar to the counter based reward function of Mark Dourlein which will be used in this study. The described

Q-function would also be a good one to test with rewards which can be calculated from a single state like the counter based reward. But since this study uses Q-functions and rewards like an Image Recognition reward which cannot be calculated without actually using the action this is out of scope.

4.2.3 Pan et al.

In Pan et al. [22] researchers propose Q-testing. Q-testing is a method of testing Android applications from the GUI. The researchers do this using Q-learning. The strategy is curiosity-driven, meaning that the algorithm is guided towards states it is curious about. This strategy is accomplished by continuously adjusting the reward of an event according to the importance of a state. The reward is calculated based on the differences in the current state and those in memory. The states in memory are a part of the previously visited states. To improve the test efficiency of Q-testing even further, the researchers also introduced a neural network which divides different states based on functional scenarios, which helps to test certain functions of an application more easily.

The researchers compared the results of Q-testing to Monkey, which is random testing. The researchers found that Q-testing improved Monkey testing by 3.12% on average when comparing the reached instruction coverage. This information is based on the results of 50 applications. The researchers also compared based on discovered faults which Q-testing improved Monkey with 88 faults detected (197 over 109).

For this thesis, the study of Pan et al. shows that it is possible to reach better test-effectiveness using Q-learning. This is useful since it shows that it is possible what this thesis is trying to achieve. The reward used is also very similar to rewards of this thesis since it compares previous states with the achieved state. Due to the scope of creating and training a neural network this thesis does not include algorithms with a neural network, but it would be a good algorithm to try in further studies.

4.2.4 Harries et al.

In Harries et al. [13] researchers propose a reinforcement learning framework for functional software testing. This framework is called DRIFT. DRIFT uses a symbolic representation of the user interface to determine the actions in the GUI. To select these actions DRIFT uses Q-learning as the reinforcement learning algorithm and combines this with a Graph Neural Network for getting the value of a state-action. The researchers used DRIFT to test the operating system Windows 10. For the Q-learning algorithm, DRIFT uses multiple reward functions, one for each specific objective the framework is trying to achieve.

The results of the study show that the algorithm is better than a random agent by two orders of magnitude. For this thesis the research again shows that Q-learning is a good direction to improve test-effectiveness. Most other parts of this study are not really relevant to this thesis since DRIFT uses multiple rewards. Those rewards need to be created for each specific task of an application to test the system. This comes closer to writing test-cases and in this thesis multiple applications are used. Writing specific rewards for each of the applications will require much time. It also means that TESTAR will not be as good with applications it does not have rewards for.

4.2.5 Eskonen et al.

In Eskonen et al. [9] researchers try to solve the same problem as this thesis is solving, improving action selection in GUI testing. They also do it using reinforcement learning with the addition of deep learning. The researchers trained a DNN (deep neural network) using images of applications shrinked to 128 by 128 pixels. By using these images as training data the DNN learned complex patterns in the application. The DNN can then be used to favor certain actions. When an application is under test, the algorithm feeds an image to the DNN. The DNN returns the probability if an pixel should be clicked for each pixel. The algorithm maps these pixels to actions and combines the probabilities. An action is then selected and executed.

The researchers compared the algorithm to random action selection and Q-learning. In the case of Q-learning the researchers used a counter-based reward. With this reward the first click on an action gets the highest reward, whilst all others get gradually lower rewards by dividing 1 by the click count.

The researchers showed that deep learning was superior to the other two algorithms. In a span of 10.000 actions, Deep Reinforcement Learning reached 5-7 times as many unique states. The researchers also did another test by comparing the URL maximization over 100 clicks with real testers. The created algorithm was even superior over inexperienced users and almost as good as an experienced user after 100 clicks, whilst random and Q-learning were the worst. This shows that the proposed solution is performing well in their experiments. The downside is that it needs to be trained for each application to learn the application patterns. The training takes longer if the application is bigger. But after training the algorithm is possibly as good as a human tester. It also is much faster, it does 100 clicks in 10 seconds compared to 5 minutes for the human tester.

For this thesis this is relevant since it shows that next to the Q-learning algorithms used in this study, there are other improvements which can possibly improve the action selection even further. It also shows that Q-learning can indeed be better than random action selection, which is what this study is trying to solve for TESTAR. The reward used in Eskonen et al. is very similar to the counter based reward of Dourlein which means that it is a good reward to test with.

4.3. Another form of reinforcement learning, SARSA

4.3.1 Gu et al.

In Gu et al. [11] another research regarding testing Android applications, researchers are implementing AimDroid which is an Activity Insulated Multi-level strategy to model-based testing for Android apps. This strategy seeks to maximize coverage and fault detection while controlling the length of sequences. AimDroid consists of two steps, first AimDroid systematically discovers every unexplored activity using a BFS (breadth-first search) algorithm. Afterwards it insulates a discovered activity in a so called "cage". In this "cage" the activity will be extensively exploited using a reinforcement learning guided fuzzing algorithm. The reinforcement learning algorithm used here is a different type of reinforcement learning algorithm used for this thesis. In Gu et al. they use SARSA, the difference between SARSA and Q-learning is according to the thesis of Mark Dourlein [8]: "The difference between SARSA and Q-learning is that SARSA is on policy and Q-learning is off policy. The difference is in updating the Q-value. In Q-learning, this is based on the maximum Q-value of all available actions. In SARSA, this is based on the Q-value of the selected action."

For this thesis the research of Gu et al. does not seem very relevant due to the fact that the researchers are using SARSA and this study uses Q-learning. But, it is good to note that after this study further improvements can be made to the testing strategy to improve the test-effectiveness of TESTAR. For example, by adding the AimDroid unexplored activity search and using a "cage" to test the activity. In AimDroid they use SARSA which is not used in this study, SARSA is already implemented in TESTAR as shown in Mark Dourlein's thesis.

4.4. Existing reinforcement learning rewards in TESTAR

4.4.1 Mark Dourlein

Some findings of the thesis of Mark Dourlein [8] are very useful for this study. Especially the design of the reward functions. In total the thesis lists three used reward functions. In this thesis these reward functions are used in the experiments.

1. Counter based, the counter based reward function calculates a reward using the following formula $\frac{1}{N}+1$, where N is the number of times the action has been executed before. This reward function will give higher rewards to the least executed actions. This will possibly lead to prioritization of the least executed actions.

2. Image Recognition, the image recognition based reward function will compare the image from the previous state to the image of the current state. The comparison will return a percentage as a similarity score. This percentage is used for calculating the reward. The formula for the reward is $1-N$ where N is the similarity score represented as a value between 0 and 1. This reward function will possibly lead to fast exploration of all the views of an application.

3. Widget Tree, the widget tree based reward function will compare the widget trees of the current state and the state prior to the current state. The reward is the sum of the values of all nodes. A changed node will return one, otherwise it will return zero. This reward function will possibly lead to the prioritization of actions which inflict a large change in available widgets.

4.5. Testing systems using test-cases

4.5.1 Mariani et al.

Another example is created by Mariani et al. [19]. The researchers created AutoBlack-Test which tests application through the GUI and generates test cases for the recorded behaviour. This is useful since you do not have to write test cases manually. AutoBlack-Test automatically builds a behaviour model and selects possible test cases which are then executed. The AutoBlackTest application is visualized in Figure 4.1.

In comparison to TESTAR that starts testing the system from startup, AutoBlackTest uses pre-recorded test cases. These test-cases are the expected behaviour from and to a certain state. Instead of testing from the startup, AutoBlackTest begins testing the behaviour only after the initial test-case state has been reached, or if it cannot be reached, the last possible state. By doing this, testing can be directed and parts of the system included in test-cases will always be tested. In TESTAR the possibility remains that parts of the system remain untested. By using AutoBlackTest you can ensure that certain parts of the system get tested.

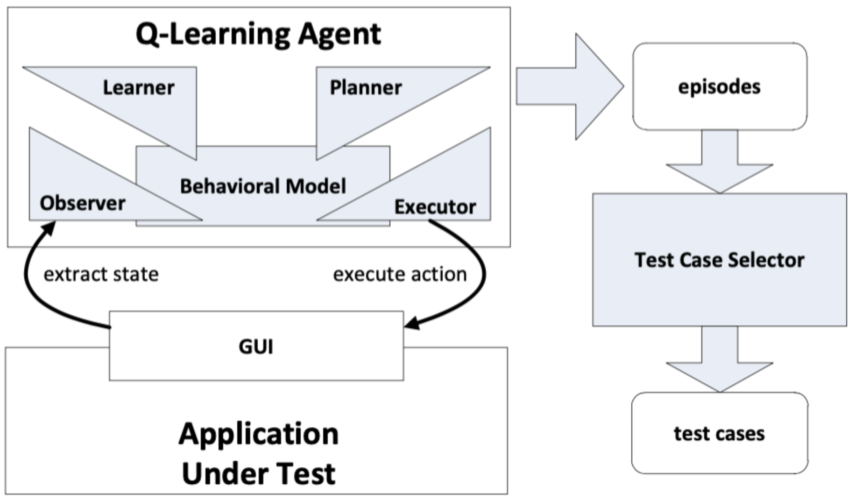

Figure 4.1: AutoBlackTest sequence by Mariani et al.

For this thesis the Test Case Selector used in AutoBlackTest seems irrelevant, it is also not implemented in TESTAR yet. But if in the future it is, the results of this thesis can be compared to TESTAR with AutoBlackTest. This will show if a Test Case Selector is beneficial over a good action selection algorithm.

4.5.2 Li et al.

In Li et al.[18] researchers propose Humanoid. Humanoid is an automated GUI testing generator. This generator is able to learn how humans interact with an application. With this knowledge it can guide test generation. It can prioritize certain actions leading to important states faster. A random generator will test all actions on a page whilst Humanoid will only test the important actions. This generates more meaningful test cases. The researchers trained an implementation of Humanoid using a public dataset of 304,976 human interactions.

The researcher compared the algorithm to random test case generation using many applications. Humanoid was overall better compared to random test case generation but, in most cases the results were close. On some applications random was even better. The researchers also ran the experiments for other algorithms, but overall the algorithms were worse than random.

For this thesis the research is not really relevant since it has nothing to do with reinforcement learning, but it shows that there are different options to improve action selection which could be useful in another study regarding the topic. It also shows that random actions selection can be hard to beat since the researchers only improved it by a small margin whilst many other algorithms failed to outperform random.

4.6. Challenges for automated testing

4.6.1 Su et al.

In Su et al.[27] researchers describe a method of measuring test-effectiveness for GUI testing tools using known bugs. Researchers use a large variety of open source applications which have known bugs. Those applications are tested using a few GUI testing tools. Since the reproducible bugs are known, the found bugs with each testing method can be compared. Researchers found that they can compare the testing tools to each other but comparing with bugs is not the best method since it has many challenges. Some bugs need a certain kind of input to occur or are nested deep in an application. This makes it that finding the bug can be more luck than a recurring occurrence for a testing tool. Some of the insights are also relevant for this thesis:

- Most testing tools randomly generate input, which makes detecting bugs based on specific input hard. Testing tools must improve on form input.
  - For TESTAR this is also the case and applications with relatively many forms immediately have a lower test-effectiveness due to the fact that TESTAR cannot progress. This is shown when testing on Moneybird in chapter 8.
- To reach certain states, applications require a specific sequence of chosen GUI actions. These action patterns are not easily followed by a testing tool. A testing tool can easily have a good run of three actions but do something else on the fourth. Testing tools have a very hard time to detect interaction patterns in applications.
  - For TESTAR this is also the case, applications with a higher complexity reach a lower coverage. This can be seen if Rachota in chapter 7 is compared to Moneybird in chapter 8. Moneybird has many forms and most forms consist of multiple actions or pages which TESTAR has to go through. Since TESTAR does not recognize these patterns, the simpler application, Rachota, gets a much higher coverage. This can possibly be mitigated with better Q-learning functions/rewards but only up to a certain point. TESTAR still need to complete the full form, consisting of multiple actions without knowing if the reward will be good. TESTAR is currently not able to recognize these patterns. This means that it cannot know which actions belong to a certain form or scope which makes TESTAR choose wrong actions from time to time.

In this thesis the test-effectiveness is not measured based on bugs, but based on code coverage. For this thesis the research of Su et al. is relevant due to the insights on testing tools. TESTAR also faces the previously outlined issues. For this thesis this confirms that these problems are not one-off issues in the thesis experiments but that it is a more common problem.

4.7. Imitation learning

4.7.1 Humphreys et al.

In Humphreys et al. [14] researchers try to create an algorithm which acts like a human when completing tasks on MiniWob++, a suite for web-browser tasks. The researchers do this by combining imitation learning, which in this case means that it learns from human behaviour by imitating their actions, and reinforcement learning. The research train both algorithms from scratch using a weighted mixture. The results of the study are promising since the algorithm received human like performance.

The researchers compared the combination of imitation learning and reinforcement learning with the individual algorithms and found that both imitation- and reinforcement learning did not perform as good individually as the combined algorithm. For this study this is relevant because it shows that after this study a created reinforcement learning algorithm can be extended with other algorithms such as imitation learning to achieve even better performance. Imitation learning would be very useful to try out in combination with a reinforcement learning. The only downside is that in the article the researchers do not go into detail about which reinforcement learning algorithm and reward is used which makes it hard to replicate with TESTAR.

4.8. OTHERS

In [25] the use of reinforcement learning is considered with the goal of building an agent that can learn to achieve tasks using mobile apps. Rewards are based on progress made in those tasks, so as to incentivize intermediate task steps. State and action space are derived for the phone's internal representation of screen elements. A parallelism could be observed between this study and the objective of creating agents that learn to use an application with the ultimate task of testing.

Recently, [24] describes ARES, a Deep Reinforcement Learning approach for black-box testing of Android apps, using a Deep Neural Network to learn the best exploration strategy. Moreover, another tool, FATE, was proposed to fine-tune the hyperparameters of Deep Reinforcement Learning algorithms on simulated apps, since it is computationally expensive to carry it out on real apps.

5
Research

5.1. RESEARCH QUESTIONS

Now that the topic of the study is known, we look into what this study will add to this topic. In this study the main question will be "How can the test-effectiveness of TESTAR be increased using Q-learning in the action selection step?". To answer this question first we look into implementing different kinds of Q-learning algorithms and reward functions for those algorithms. After that part of the study the test-effectiveness, also referred to as the performance of TESTAR, will be measured. It can be measured in multiple ways, this study will also look into how it can be measured, especially in web-based applications. In total there are three research questions which are the following.

- RQ1 - How can different Q-learning algorithms in TESTAR be implemented?
- RQ2 - What reward functions can be used for the implemented Q-learning algorithms in TESTAR based on the available reward metrics?
- RQ3 - How can the performance (test-effectiveness) of a combination of Q-learning algorithm and reward function in web-based applications be measured?

5.2. RQ1

As previously introduced in chapter 3, there are many different Q-learning algorithms. All of those algorithms have a different take on calculating the Q-value and training the agent. Due to the scope of the study, two algorithms will be implemented. In this case the algorithms *QV-learning* and *Double Q-learning*. The choice was made through comparing the five single agent algorithms from chapter 3:

1. Incremental multistep.
2. Double Q-learning.

3. Deep Q-learning.
4. Hierarchical Q-learning.
5. QV-learning.

To determine which algorithms will be implemented the different Q-learning algorithms were compared to each other. To determine if they are suitable they were compared using the following metrics, complexity (how difficult are the algorithms to implement, how many weeks will they take?), learning (how do the algorithms learn?), suitability (how suitable are the algorithms for testing with TESTAR). By filling in the metrics for each algorithm we get the results as shown in table 5.1.

Table 5.1: Q-learning algorithms compared

Algorithm	**Properties**
Incremental multistep	**Complexity** Incremental multistep allows a credit to be carried across multiple states, it is similar to QV-learning in a way that the Q-value is not only calculated based on the current state of the application but using multiple states. It does this by using temporal difference learning. The expected time to implement the algorithm is 3 weeks. **Learning** Incremental multistep learns the same as basic Q-learning, it learns from a given reward using the calculated Q-values. The difference is that it also adds temporal-difference learning. By adding temporal-difference learning, the algorithm Q-value is calculated based on the current and previous states. **Suitability** Incremental multistep is suitable for this study since it is similar to the already proven to work basic Q-learning.
Double Q-learning	**Complexity** Double Q-learning trains two policies instead of one, it does this to reduce the random factor of a system. Since it is similar to basic Q-learning it is expected to be implementable in around 2-3 weeks. **Learning** Double Q-learning learns by training two different policies/agents. When updating a Q-value, the existing Q-values from both agents are used. **Suitability** Since basic Q-learning is already proven to work in TESTAR double Q-learning can be implemented without further studies. This makes it very feasible for this study.
	Continued on next page

Table 5.1: Q-learning algorithms compared: **continued**

Algorithm	**Properties**
Deep Q-learning	**Complexity** Since Deep Q-learning uses neural networks which must be trained and fit for TESTAR, implementing Deep Q-learning algorithm will approximately take 6-8 weeks or more. The difficulty of the implementation sits in creating a good model for the use across multiple systems. The model must be verified not to be over-fit and different neural networks should be compared. If looked deeper in to Deep Q-learning it could be a full study on its own. **Learning** Deep Q-learning learns by training a neural network using samples from previous experiences. This neural network will then predict the Q-value. **Suitability** Because the variety of systems TESTAR can be used on, it would be hard to create a neural network which suits multiple systems in the scope of this research. Another reason this algorithm suits the study not very well, as Degott et al. pointed out, is that previous knowledge does not achieve greater results in the case of testing.
Hierarchical Q-learning	**Complexity** Hierarchical Q-learning uses a high-level actions and low-level actions. The implementation is a bit more complex than basic Q-learning and should take 4 weeks. The complexity is found in creating the hierarchy of actions and defining when a high level action is completed. **Learning** Hierarchical Q-learning learns from the Q-values. The Q-value is calculated a bit different compared to basic Q-learning. It takes into account not only the normal, in this case low-level actions but also an abstract action. **Suitability** It is currently not really suitable for TESTAR, since TESTAR must be easy to run on multiple systems, and it is hard to define which actions are high-level and which actions are low-level. To implement this Q-learning algorithm a study must first investigate if it is possible to accurately detect high-level and low-level actions in an application.
	Continued on next page

Table 5.1: Q-learning algorithms compared: **continued**

Algorithm	**Properties**
QV-learning	**Complexity** QV-learning works almost the same as basic Q-learning but with the addition of a V-value for temporal difference learning, this value helps calculating the reward based on multiple states instead of 1. The time to implement this algorithm will be approximately be 2-3 weeks. **Learning** The learning works the same as basic Q-learning, the only difference is that the Q-value is calculated differently. The Q-value in QV-learning is updated using a V-value. This adds temporal-difference learning to TESTAR. Temporal-difference learning is used to calculate the Q-value based on all previous states instead of only the latest. **Suitability** The QV-learning algorithm will be very suitable for this study since it is similar to the proven to work basic Q-learning. The addition is using a suitable policy and implementing the correct Q and V functions.

By looking at the results, two algorithms, Deep Q-learning and Hierarchical Q-learning, are not easily implementable due to the high complexity and need further study on how they could be implemented in TESTAR. The other three algorithms, QV-learning, Incremental multistep and Double Q-learning are very feasible for TESTAR. The choice was made to not implement Incremental multistep because it is similar to QV-learning, both algorithms learn from all previous states using Temporal-Difference learning. The chosen algorithms to implement are *QV-learning* and *Double Q-learning*.

5.3. RQ2

Before we can start with investigating possible reward functions, we first need to know which metrics can be extracted from TESTAR in each state. When looking into the source code there are a few things that could be interesting metrics for creating a reward function. The metrics are interesting since they are subject to change after the selected action. This means they do not stay the same during the execution of the system under test. In general there are a few classes where metrics can be extracted from.

- With the *ActionSelectionUtils.java* it is possible to get the new set of actions. This new set of actions are actions that were not available in the previous state. This metric can be useful since we also know all the current actions and the previous actions. There can now be calculated how much the system has changed since the last state.
- In the *AbstractState.java* it is possible to get metrics based on the current state. It is possible to get the number of visited and unvisited actions in the current state. By comparing this to the number of current actions we also evaluate how much of the current state has been tested.

- Another interesting metric is to compare a screenshot of the current state to a screenshot of the previous state and determine how much the GUI has changed after a selected action.

In table 5.2 we see a overview of the metrics.

Metric	Explanation
Number of current actions	The number of current actions is the number of actions in the current state.
Number of previous actions	The number of previous actions is the number of actions in the previous state.
Number of new actions	The number of new actions is the number of new actions since the previous state.
Number of visited actions	The number of visited actions is the number of actions that have been selected prior to this state.
Number of unvisited actions	The number of unvisited actions is the number of actions that have not been selected prior to this state.
Similarity score	By comparing a screenshot from the previous state to a screenshot of the current state we can calculate a percentage of how similar the application is.
Percentage of state change	By dividing the number of new actions by the number of current actions we know how much the state has been changed to the prior state.
Percentage of tested state	By dividing the number of visited actions by the number of visited actions we know the percentage of how much of the state has been tested.
Percentage of untested state	By dividing the number of unvisited actions by the number of visited actions we know the percentage of how much of the state remains untested.
Percentage of GUI change	By dividing the number of changed pixels by the amount of total pixels we know how much of the GUI has been changed to the prior state.

Table 5.2: State metrics in TESTAR which can be used in rewards

These are globally available metrics, but there are also metrics which can be extracted based on the action itself. In table 5.3 these metrics are displayed.

Metric	Explanation
Has the action been selected before	True or false if a selected action has been selected already before the previous state.
Z-index	On which level does the action sit?

Table 5.3: Metrics based on an action in TESTAR which can be used in rewards

With the reward metrics, different reward functions will be designed and implemented into TESTAR. The designed reward functions can then be used in RQ3 to verify which combination of reward and Q-learning algorithm has the highest test-effectiveness.

The different rewards functions will be designed according to heuristics from the research of Mark Dourlein and Degott et al. In both studies it seems beneficial to use a greedy approach in reinforcement learning. For this thesis three reward functions from the thesis of Mark Dourlein will be used. Which are:

1. An image recognition based reward function.
2. A widget tree based reward function.
3. An execution count based reward function.

Next to these reward functions, more reward functions that will prioritize larger changes in the application will be designed.

5.4. RQ3

Once RQ1 and RQ2 have been answered, there are a variety of combinations of reward functions and Q-learning algorithms. The Q-learning algorithms also have multiple configurable parameters. These parameters need tuning to get the best performance from the algorithms. By using the combinations with different parameters, new combinations are created. This means one combination consists of three parts, a *reward function*, a *Q-learning algorithm* and the *parameters*.

To determine which combination has the highest test-effectiveness, the test-effectiveness must be measured. This could be measured on the smaller systems used in the implementation phase but it would be better to collect the data on a larger system. This will show the algorithms also work on a different system and perform on a larger code base.

The application which will be used for this study is Moneybird[1]. Moneybird is accounting software mostly used by smaller companies and self-employed individuals. The platform has a variety of functionalities which can be tested. The choice for Moneybird was made since Sven Ordelman works for Moneybird. This makes the platform accessible and since there is prior knowledge of the code-base it is easy to start testing with it. It also makes easy to implement test-effectiveness measurements in the application. The application is written in Ruby on Rails. This is a web-based framework which is good since TESTAR is already able to extract actions from web-based applications through the browser. This means that to start testing Moneybird with TESTAR not much setup in TESTAR is required.

The definition of what is going to be tested and on which system is now defined. Still unknown is, how the test-effectiveness is going to be measured. The test-effectiveness can

[1]`https://www.moneybird.com/`

be measured in many ways, but for this study the test-effectiveness will be measured using a few metrics which are shown in table 5.4.

Metric	Explanation
Code coverage	The number of executed lines of code as a percentage of the total lines of code resemble the amount of the application that has been reached. This is easy to extract from a Ruby on Rails application since it is not compiled. The number of uniquely reached lines can be easily extracted for example using the ruby gem (a library), simplecov[2]. This library measures the coverage of each file and line.
Time	How long does it take to reach a certain variable, for example 80% code coverage. This measurement will not work for manually created test-cases since the time it took to create them cannot be taken into account.

Table 5.4: Test-effectiveness metrics

To measure test-effectiveness using TESTAR, a baseline should be set. By running TESTAR with random action selection for a certain number of sequences and actions we know the initial variables. The other versions of TESTAR will then be run with the same amount of sequences and actions. After all the experiments, the test-effectiveness can be compared.

5.5. METHODOLOGY

To answer the research questions a few different research methods are used. For RQ1 and RQ2 the research methods are *creation* and *experimentation*. RQ1 and RQ2 will result in an implementation, these implementations will be experimented with on Java applications. For RQ3 the research method is *experimentation* and *creation*, in RQ3 results are collected by doing experiments with the implementations of RQ1 and RQ2. RQ3 will result in a method for measuring test-effectiveness on web-based applications. RQ3 also functions as the *validation* of the research, it validates the results of RQ1 and RQ2 using the larger system Moneybird. Eventually the main question can be answered using the collected results.

6
Using Q-learning in TESTAR

This chapter is about RQ1 *How can different Q-learning algorithms in TESTAR be implemented?*. In chapter 5 the decision was made to implement two, Double Q-learning and QV-learning. First this chapter will look into how a current Q-learning algorithm is implemented in TESTAR and next the new algorithms will be implemented. Finally the algorithms are compared to each other by running a few experiments. These experiments will show if the new algorithms have been successfully implemented in TESTAR.

6.1. Q-LEARNING IN TESTAR

Before an implementation of new algorithms can be added to TESTAR, knowledge of how TESTAR incorporates a Q-learning algorithm now must be present. A global overview can be seen in figure 6.1. This figure shows the reinforcement learning package in TESTAR. In this figure the full lines show which classes have instances of other classes, whilst the striped lines show which functions are being called from a function. The main class of the reinforcement learning package is a ModelManager. The ModelManager manages all the factors of the Q-learning system. From getting reward, to getting the next action it manages everything. A Protocol in TESTAR can just instantiate the ModelManager and ask for the next action, the ModelManager handles everything and returns the next action. This is useful since other parts of TESTAR do not need knowledge of the presence of Q-learning.

The ModelManager works as follows, it creates an abstract state model with abstract states. After each action, the ModelManager uses the RewardFunction and QFunction to get the Q-value for the previous selected action and state combination. It saves this Q-value to the abstract state. To select the next action, the abstract state model along with the current state is sent to the action selector. The action selector determines the current set of possible actions based on the algorithm which uses the Q-values and a policy to select the next action.

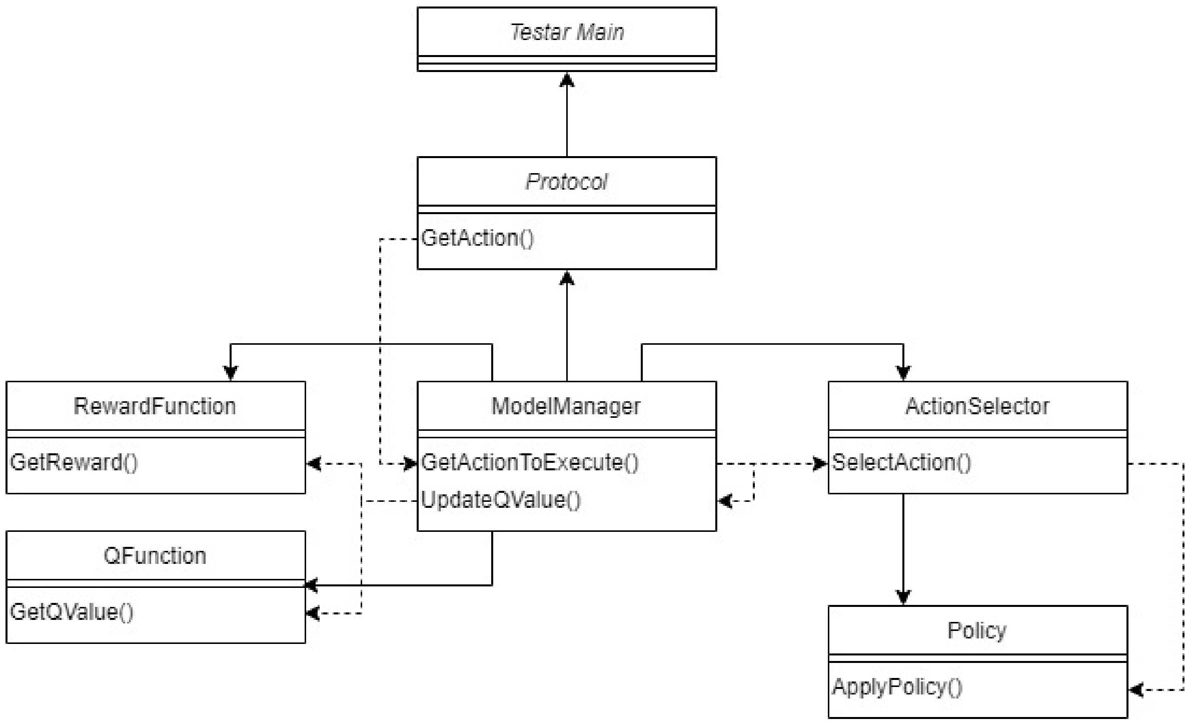

Figure 6.1: TESTAR basic Q-learning class diagram

In the class diagram the classes which make Q-learning in TESTAR work are shown. But besides some function names the diagram does not show details of the exact implementations. That is why below some pseudo code is included which should make the class diagram easier to understand.

First the ModelManager calls the GetReward() function. For this function no pseudo code is given since the pseudo code for the reward functions are in chapter 7. After the ModelManager gets the reward it passes this to the QFunction along with the executed and previously executed action. The psuedo code for this QFunction is shown in algorithm 8 below.

Algorithm 8 Basic Q-learning QFunction's (GetQValue())

```
1: Initialize $oldQValue, newQValue, actionUnderExecution, executedAction, \alpha, \gamma, defaultQValue$
2: if executedAction != null then
3:     $oldQValue := executedAction.getAttributes().getVValue$
4: end if
5: if actionUnderExecution != null then
6:     $newQValue := executedAction.getAttributes().getVValue$
7: end if
8: return $oldQValue + \alpha * (reward + \gamma * newQValue - oldQValue)$;
```

After the ModelManager receives the calculated QValue from the QFunction, it saves this value in the Q-Matrix. After the QValue is updated the system can select a new action. The ModelManager asks the ActionSelector to select an action. The ActionSelector on its turn calls the policy which in this study is the GreedyPolicy with an ϵ of 0.3. The pseudo code for ApplyPolicy() is shown in algorithm 9 below.

Algorithm 9 Basic Q-learning Policy's ApplyPolicy()

```
1: Initialize $Actions, \epsilon, qValuesActionsMultimap$
2: for $action \in actions$ do
3:     $qValuesActionsMultimap \leftarrow (action.qValue, action)$
4: end for
5: $qValues \leftarrow qValuesActionsMultimap.keys$
6: if $qValues.empty$ then
7:     return $null$
8: end if
9: $maxQValue = Collections.max(qValues)$
10: $selectedAction = qValuesActionsMultimap.get(maxQValue)$
11: return $selectedAction$
```

After the ModelManager receives the selectedAction it passes it on to the Protocol which in turn gives TESTAR the action. TESTAR executes the action and the whole sequence starts over.

To implement our own implementation of a Q-learning algorithm, a new configuration can be made with altered versions of the QFunction, Policy or other parts of the system. Since TESTAR communicates through the ModelManager that acts as a interface, classes and functions used by the ModelManager can easily be altered and changed.

6.2. QV-LEARNING

QV-learning works almost the same as basic Q-learning except it uses a Q- and V-function instead of only a Q-Function. To implement this in TESTAR, two classes were created. The first being the QVFunction class that uses a V-value to calculate the Q-value using equation 6.1.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha\Big(r_t + \gamma V(s_t + 1) - Q(s_t, a_t)\Big) \tag{6.1}$$

The V-value in this equation is previously calculated using the Vfunction class which implements equation 6.2.

$$V(s_t) := V(s_t) + \beta\Big(r_t + \gamma V(s_t + 1) - V(s_t)\Big) \tag{6.2}$$

The α, β, γ are configurable parameters in TESTAR. The implemented formulas can be found in Wiering et al. [33]. This paper also states the policy to be used is the Greedy Policy that was already implemented in TESTAR prior to this study. A class diagram for the QVLearning implementation is shown in figure 6.2.

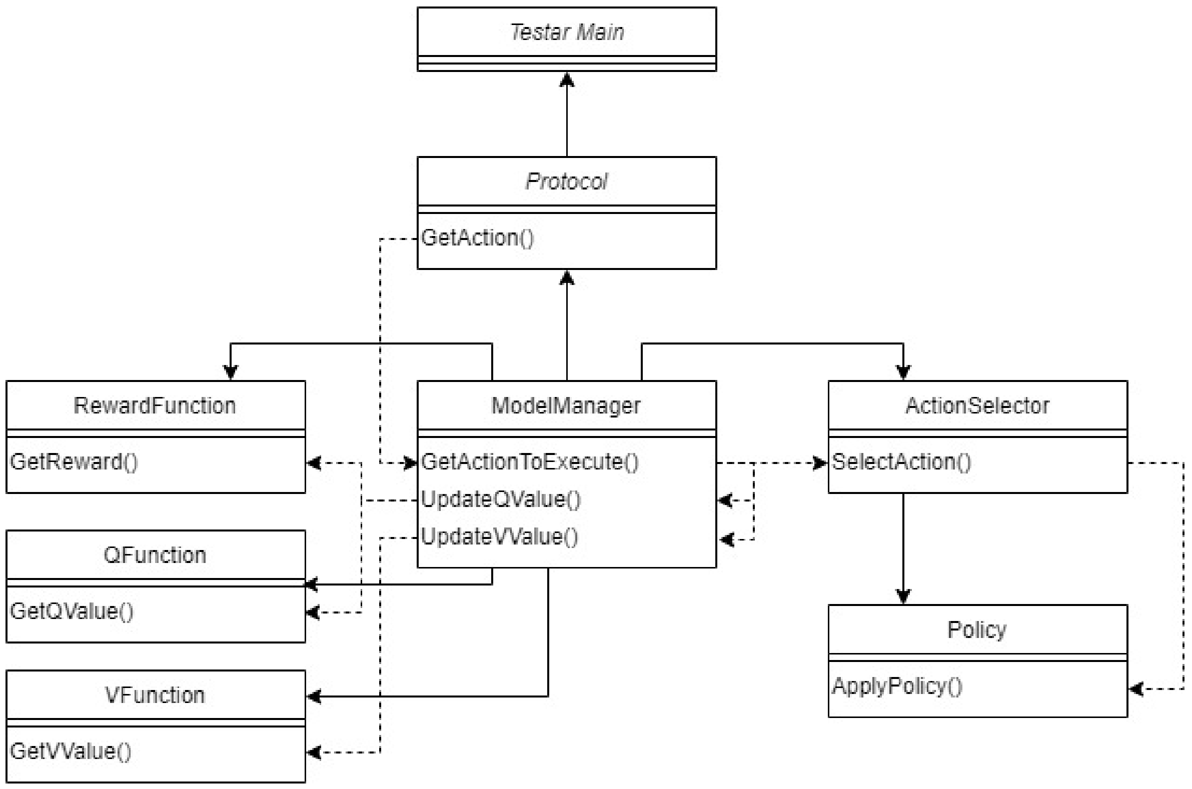

Figure 6.2: TESTAR QV learning class diagram

If the QV-learning implementation is compared to the basic Q-learning implementation by code, the largest change is found in the QFunction and the addition of the VFunction. For these classes the pseudo code is shown below.

Algorithm 10 QV-learning VFunction

```
1: Initialize $oldVValue, actionUnderExecution, executedAction, \beta, \gamma$
2: if executedAction != null then
3:     $oldVValue := executedAction.getAttributes().getVValue$
4: end if
5: if actionUnderExecution != null then
6:     $newVValue := executedAction.getAttributes().getVValue$
7: end if
8: return $oldVValue + \beta * (reward + \gamma * newVValue - oldVValue)$;
```

Algorithm 11 QV-learning QFunction

```
1: Initialize $oldQValue, vValue, executedAction, \alpha, \gamma$    ▷ The V Value is passed to the function
2: if executedAction != null then
3:     $oldQValue := executedAction.getAttributes().getQValue$
4: end if
5: return $oldQValue + \alpha * (reward + \gamma * vValue - oldQValue)$;
```

6.3. Double Q-learning

Another implemented Q-function is Double Q-learning. Double Q-learning, as the name suggests, uses two Q values instead of one. By adding a tag to the abstract state model for storing the second Q-value, each state can now have two Q-values. The calculation of a new Q-value is based on both Q-values. The algorithm from Hado et al. is used here.

Algorithm 12 Double Q-Learning by Hado et al.

```
 1: Initialize $Q^A, Q^B, s$
 2: repeat
 3:     Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s^{'}$
 4:     Choose (e.g. random) either UPDATE(A) or UPDATE(B)
 5:     if UPDATE(A) then
 6:         Define $a^* = arg\ max_a Q^A(s^{'}, a)$
 7:         $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s^{'}, a^*) - Q^A(s,a))$
 8:     else if UPDATE(B) then
 9:         Define $b^* = arg\ max_a Q^B(s^{'}, a)$
10:         $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s^{'}, a^*) - Q^B(s,a))$
11:     end if
12:     $s \leftarrow s'$
13: until end
```

When a Q-value has to be selected, the algorithm chooses randomly between updating Q-value A or Q-value B. The other Q-value corresponding with the selected action is passed as a parameter in the calculation. To implement this, a new Q-function has been added to TESTAR, using two Q-values instead of one. Since there now are two Q-values, the existing policies for action selection do not work anymore. So a new policy must be created. Researchers in Degott et al. showed that the best policy to use is $\epsilon - Greedy$ with a ϵ of 0.3, this is what is implemented into TESTAR. The whole Q-learning class diagram is shown in figure 6.3.

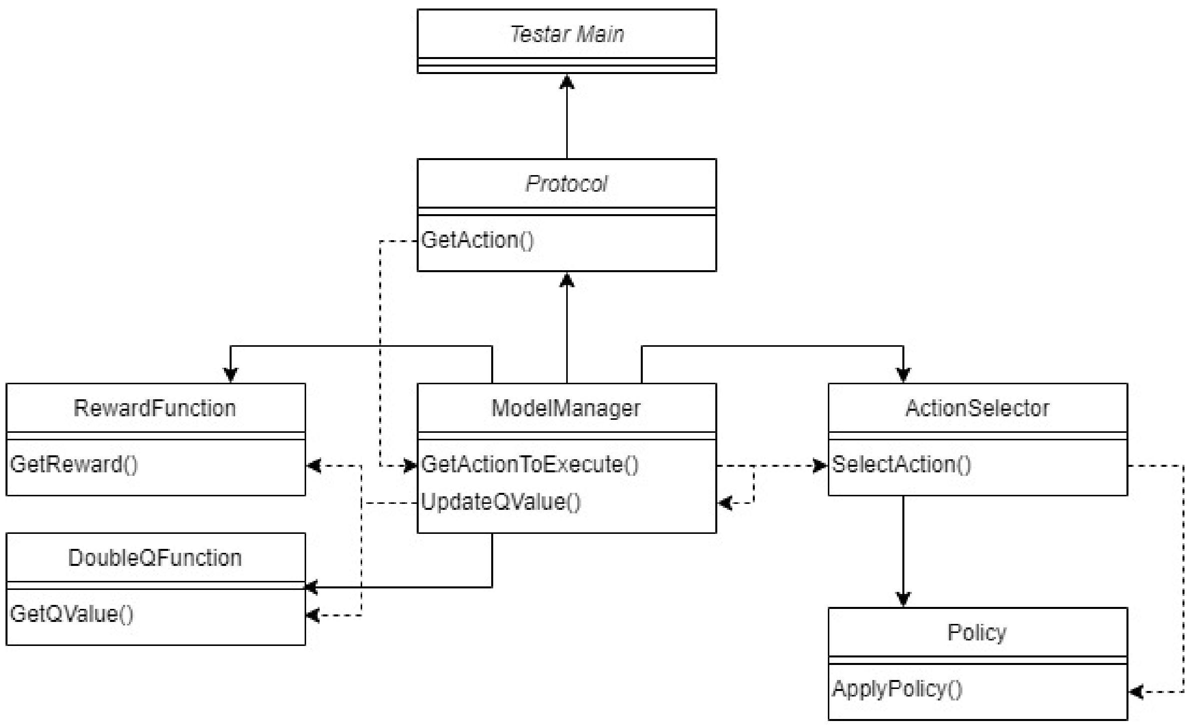

Figure 6.3: TESTAR Double Q-learning class diagram

6.4. Experimenting with the Q-learning implementations

At this point of the study there are two newly added Q-learning algorithms, it is possible to compare these to the already existing Basic Q-learning algorithm. This is done as a form of preliminary testing to know if the Q-learning algorithms work. If the algorithms are only tested with newly added rewards it cannot be concluded why an algorithm fails. This is because it can be due to the reward or due to the Q-learning algorithm. By testing the Q-learning algorithms separately it can be confirmed if the algorithms work.

To compare the algorithms, first data must be collected by doing some experiment runs. Our test setup consists of 10 computers, each executing the tests for each algorithm. The experiment is executed on 10 computers to limit the effect of outliers. The choice was made to use 10 computers because less computers would increase the effect outliers and more computers would increase the cost in time or resources that one experiment take. Since multiple experiments are needed it would require much more computers or much more time. For each algorithm:

- The Widget Tree Reward is used, this reward is explained in figure 13 in chapter 7. This reward is used since it was already present in TESTAR and proven to function which means that if an algorithms does not work it cannot be related to the reward.

- The policy used for the experiments is a $\epsilon - Greedy$ policy with a ϵ of 0.3. This policy and ϵ value is chosen based on the findings of Degott et al. [7] as explained in chapter 4.

- The α and γ values are the same for each Q-learning algorithm. They are the same because this decreases the chance of external factors influencing the results. The values used are α of 1.0 and γ of 0.99

- For QV-learning a default β of 1.0 is used
- An existing protocols for Spaghetti and Rachota are used.

By using the same reward and policy for each algorithm, the possibility of a different policy/reward having influence on the test-effectiveness of a algorithm is limited. All the algorithms have the same application and dataset available which allows for an equal comparison. Each test consists of 10 sequences with 300 actions. This means that each computer executes 3000 actions.

The first experiments are using a really small java application called Spaghetti, this application has been used with TESTAR before and required minimal setup. This application consists of 2,094 instructions and only 68 possible branches. From this experiment there can be concluded if TESTAR is able to run a full test of an algorithm without crashing and if the results are useful. To compare the algorithms better they must be used against a larger application. This is why a second test run on a java application called Rachota was executed. The choice for Rachota was made since it is a large application which TESTAR already has existing configuration for it which makes it easy to start the experiments. Rachota has in total, 53,416 instructions with 2,874 possible branches. From the results of those experiments the algorithms can be compared.

6.4.1 Running with Spaghetti

First the results collected on Spaghetti are looked into. For the basic Q-learning algorithm the results in figure 6.4 show that it reaches around 82% to 90% instruction coverage and 66% branch coverage.

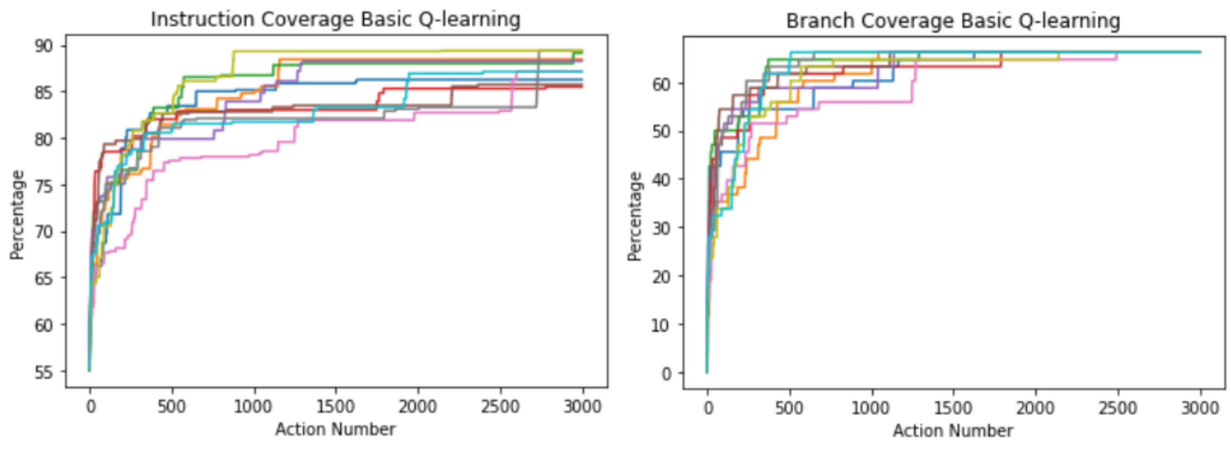

Figure 6.4: Spaghetti Basic Q-learning results

For the new QV-learning algorithm in figure 6.5 it shows that it reaches around 82% to 88% instruction coverage and 66% branch coverage. Since it is a small application conclusions cannot be made accurately, but what can be seen is that QV-learning is a bit more consistent in the early phase of the algorithm. Both reach the same and possibly maximum branch coverage percentage, whilst basic Q performs a bit better on the instruction coverage, but by a minimal margin due to the small number of possible instructions.

For the new Double Q-learning algorithm in figure 6.6 it shows that it reaches around 80% to 87% instruction coverage and 66% branch coverage. The branch coverage is the

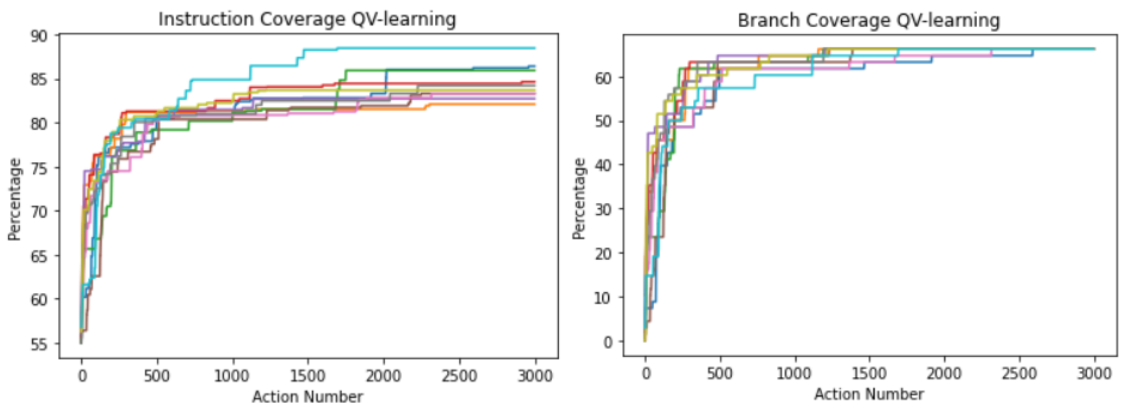

Figure 6.5: Spaghetti QV-learning results

same as the other algorithms, but the instruction coverage is slightly worse. But again due to the size of the application a conclusion should not be made.

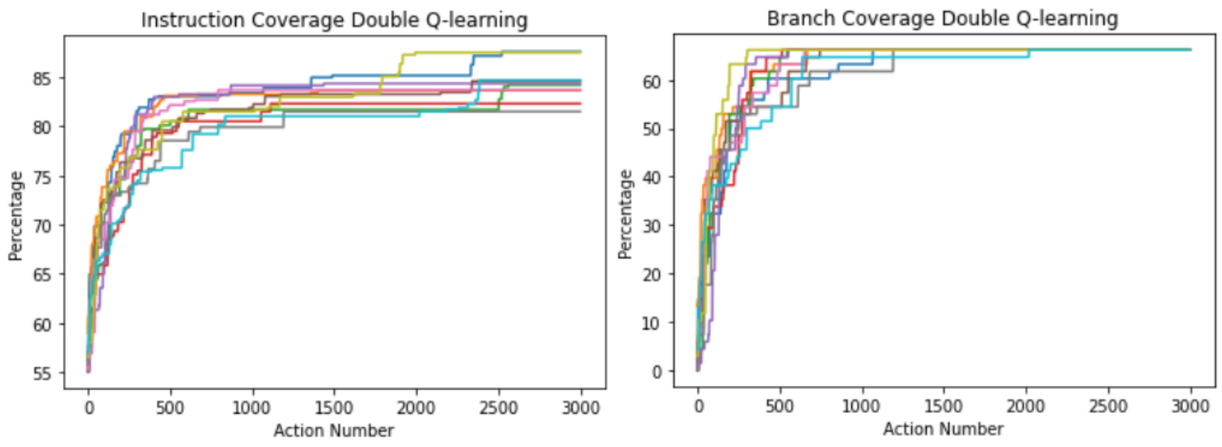

Figure 6.6: Spaghetti Double Q-learning results

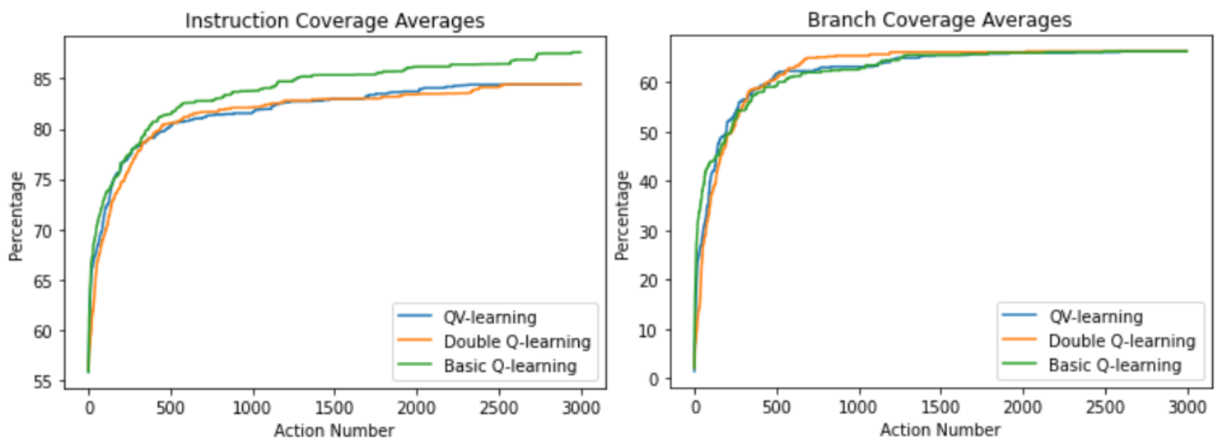

Figure 6.7: Spaghetti Combined results

For each of the results an average can be calculated, these averages are combined for all the algorithms in a single graph. These results are shown in figure 6.7. If looked at the graphs it can be seen that Basic Q-learning performs best for instruction coverage whilst Double Q-learning reaches the 66% branch coverage the fastest. From these graphs it can be noted that all the algorithms are working and can be used for testing on Rachota.

6.4.2 Running with Rachota

The results for Rachota are much more interesting. For the basic Q-learning algorithm, the results in figure 6.8 show that in some occasions it reaches 68% instruction coverage and 40% branch coverage, but it is not very consistent. The algorithm performs well in some tests whilst being very bad in others. It is also good to note that not all test runs consist of 3000 actions due to the fact that if an error is found in a sequence, the remaining actions will not be executed. This is why some tests will have a flat line towards the end.

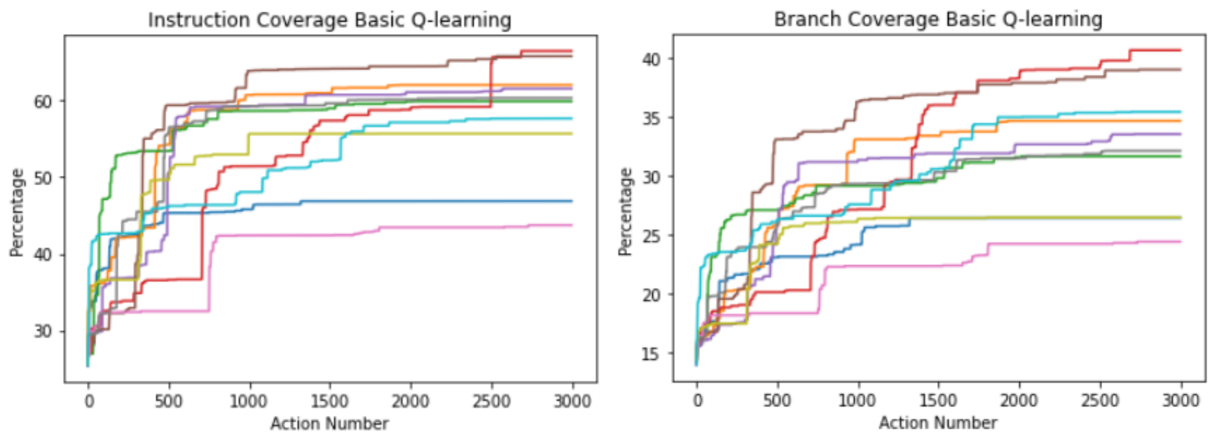

Figure 6.8: Rachota Basic Q-learning results

For the new QV-learning algorithm the results in figure 6.9 show that it reaches around the same instruction and branch coverage as Basic Q-learning in its best run, but it is far more consistent. With QV-learning keeping the instruction coverage within a 8% range, Basic Q-learning does it in 27%.

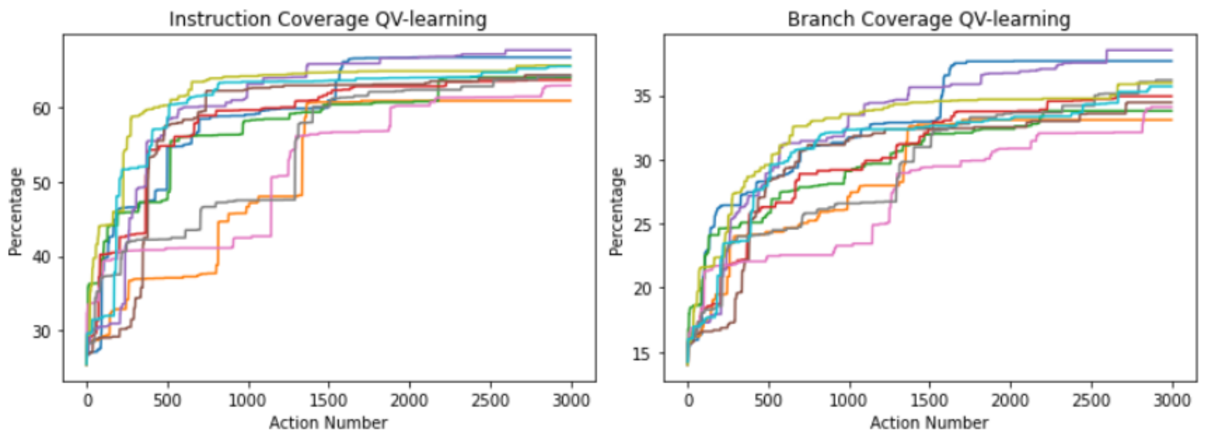

Figure 6.9: Rachota QV-learning results

For the new Double Q-learning algorithm the results in figure 6.10 show that it performs very similar to Basic Q-learning. This is possibly due to the fact that Double Q-learning is just a combination of 2 Basic Q-learning algorithms with the added goal to lessen the possibility of highly rewarding random behaviour of the application. If Basic Q-learning is compared to Double Q-learning it confirms this. It can be seen that the results are a bit more consistent but by far not as consistent as QV-learning.

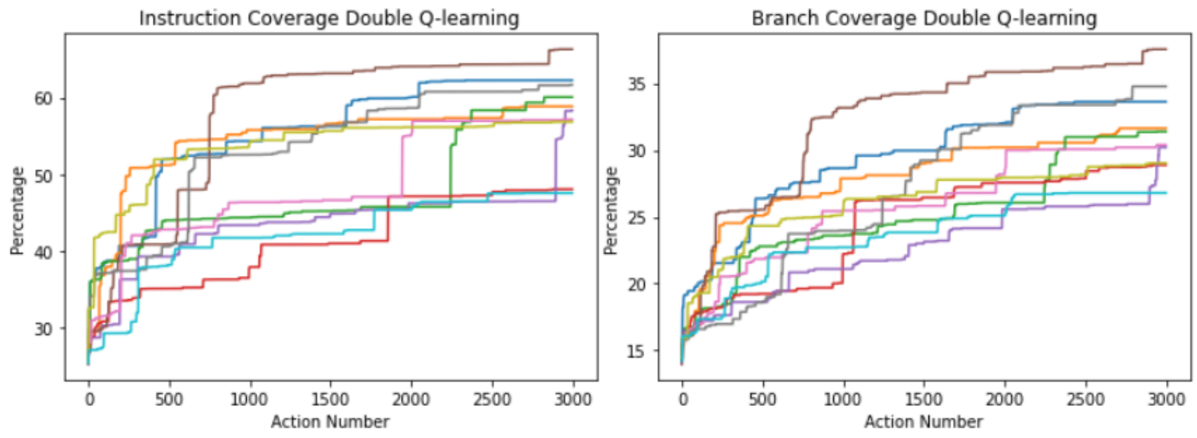

Figure 6.10: Rachota Double Q-learning results

For each of the results the averages can be calculated and combined into a graph. These results are shown in figure 6.11. If looked at the graphs it can be concluded that QV-learning performs the best. This is due to the fact that it performs much more consistent compared to the other algorithms. Basic Q-learning and Double Q-learning both have a good run but on average they perform worse than QV-learning. That is also the reason why Double Q-learning eventually catches and will possibly perform better compared to Basic Q-learning. It starts slower since it needs twice as much data to make good decisions but this allows for more consistent runs.

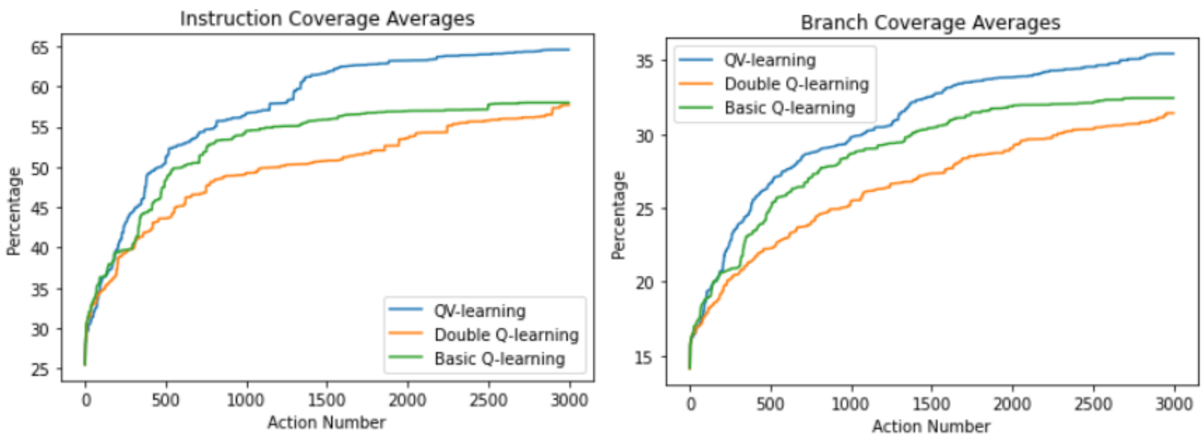

Figure 6.11: Rachota Combined results

It can be concluded that QV-learning is the clear winner and shows the most promising results. Double Q-learning will eventually be better than Basic Q-learning but needs more time to start.

7 Rewarding Q-learning algorithms

The created Q-learning algorithms are not the only aspect to better testing with Q-learning in TESTAR. The Q-learning algorithms also need rewards, without rewards the Q-learning algorithm does not know what is good or bad.

This chapter is about RQ2 *What reward functions can be used for the implemented Q-learning algorithms in TESTAR based on the available reward metrics?*. This chapter will introduce newly designed rewards based on the metrics described in chapter 5. The existing and new rewards will be compared to each other after running some experiments. The rewards will also be compared to TESTAR with random action selection, which shows if using Q-learning is beneficial.

The choice was made to use a $\epsilon - Greedy$ policy with an ϵ of 0.3. The rewards should comply with this policy by giving higher rewards for actions which should be favored.

7.1. Existing rewards implemented in TESTAR

In TESTAR the rewards are separate classes which are called from the ModelManager. Each reward implements the same functions which are declared in the *RewardFunction* interface, in the case of the study the function which is very important is the $GetReward(...)$ function. This function is called by the ModelManager after each action and a reward must be returned. Most of the additions to TESTAR in this chapter are about the implementation inside the $GetReward(...)$. This means that most of the implementation is about the *RewardFunction* in figure 7.1.

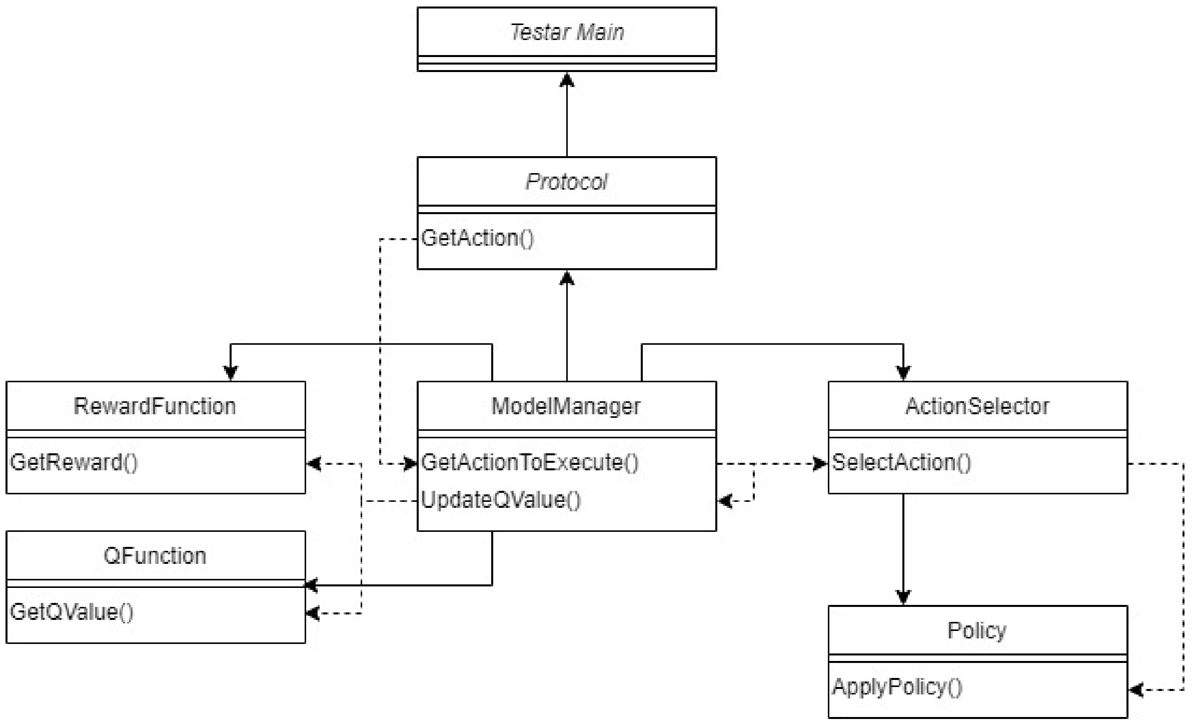

Figure 7.1: TESTAR Q-learning class diagram

In the current version of TESTAR there are two already implemented reward functions which can be useful for this study. The first one is an *image recognition* reward function which calculates a similarity score between two screenshots. The similarity score is a score between 0 and 1. To calculate the reward this similarity score is subtracted from 1. This means that when inflicting more change, the reward gets higher. The formula for this reward is shown in equation 7.1.

$$r_t := 1 - SIMILARITY\_SCORE \tag{7.1}$$

Another reward function is a *widget tree comparison* reward function. This reward function compares the number of widgets in the current state to the number of widgets in the previous state and bases the reward on those two values. It does this by first getting the number of widgets from the current state and then calculating how many widgets are persistent in the current and previous state. To calculate the reward, the number of persistent widgets are divided by the total number of widgets in the current state. The algorithm for calculating the reward is algorithm 13.

Algorithm 13 Widget Tree Comparison Reward Algorithm

```
1: Initialize $currentState, previousState$

2: $currentWidgets := currentState.GetWidgets()$
3: $previousWidgets := previousState.GetWidgets()$
4: $numEqualWidgets := (currentWidgets \cap previousWidgets).count()$
5: $reward := \frac{numEqualWidgets}{currentWidgets.count()}$
```

The last reward function is the *counter based* reward function. This is a simple reward function which just counts how many times an action has been visited, and gives a lower reward if an already visited action is executed another time. The formula for this reward is shown in equation 7.2.

$$r_t := 1/executionCount \tag{7.2}$$

All rewards give a higher reward when inflicting more change in the application. This is good since the chosen policy $\epsilon - Greedy$ favors higher Q-values. This means that if an action is good it should get a high reward so that the policy favors this reward in the future. This is why both rewards are very useful for this study.

7.2. NEW REWARDS AND ALTERATIONS

When comparing the used metrics in the rewards from the previous section to the metrics in table 5.2 it can be concluded that many metrics are already used. Metrics which are not used yet are the number of visited actions and the number of unvisited actions. With those metrics new reward functions can be created. For this study three new reward functions were created that are described in the subsections below.

7.2.1 Widget Tree Image Recognition Comparison Reward

The first reward is a *combination* of the image based reward function and the widget tree comparison reward function. This reward is created because both rewards are proven to work with TESTAR and both give higher rewards when more change occur. One benefit of combining the rewards is that they already work with TESTAR which means that the rewards do not have to be tested before combining them. By combining the rewards, a reward can be calculated based on two different factors, the GUI and the Widget Tree. This creates a reward which has more knowledge of the system and can hypothetically make more educated guesses.

The new reward function is called *WidgetTreeImageRecognitionRewardFunction* and uses both reward functions and combines the values to get a new reward. A parameter can be supplied to this reward function which defines the distribution of the widget tree reward to the image recognition reward. For example, if this parameter has a value of 0.7, the widget tree reward counts for 70% whilst the image recognition reward only counts for 30%. The used algorithm is algorithm 14.

Algorithm 14 Widget Tree Image Recognition Comparison Reward Algorithm

```
1: Initialize $currentState, previousState, widgetTreeComparisonShare$

2: $widgetTreeReward := WidgetTreeRewardFunction.GetReward()$
3: $imageRecognitionReward := ImageRecognitionRewardFunction.GetReward()$
4: $reward := \frac{widgetTreeReward \times widgetTreeComparisonShare}{imageRecognitionReward \times (1 - widgetTreeComparisonShare)}$
```

This reward function will be tested with 9 different distribution parameters (0.1 - 0.9) which allows for finding the best possible configuration of this reward.

7.2.2 Unvisited and Visited Actions Comparison Reward

A second reward function created for this study is the *Unvisited and Visited Actions Comparison*, this algorithm gives a reward based on the percentage of unvisited actions compared to the total number of actions. Actions which progress to a state with a high number of unvisited actions get a higher reward in comparison to a state with a low number of unvisited actions. States where all actions have been visited return a negative reward, which penalizes the action, this possibly allows for faster exploration of the system due to the fact that a large number of unvisited actions in a state results in exploring new actions. The algorithm for this reward is algorithm 15.

Algorithm 15 Unvisited and Visited Actions Comparison Reward Algorithm

```
1:  Initialize $currentState, executedAction$

2:  $executedAction.executionCount += 1;$
3:  $reward := -0.5$
4:  $unvisitedActions := 0$

5:  **for each** $action$ **in** $currentState.getActions()$ **do**
6:      **if** $action.executionCount == 0$ **then**
7:          $unvisitedActions += 1$
8:      **end if**
9:  **end for**

10: **if** $unvisitedActions > 0$ **then**
11:     $reward := \frac{unvisitedActions}{currentState.getActions().size()}$
12: **end if**
```

7.2.3 Penalized Unvisited and Visited Actions Comparison Reward

The third and last reward function created for this study is the *Penalized Unvisited and Visited Actions Comparison*, this algorithm does almost the same as the previous algorithm, it gives a reward based on the percentage of unvisited actions compared to the total number of actions. The only difference is, it penalizes the executed action if it has been executed previously and there are still unexecuted actions in the previous state. This prevents the algorithm from possibly preferring one action and repeatedly executing it only because it leads to a state with many unvisited actions. The algorithm for this reward is algorithm 16. This reward function will lead to exploration of the system and will try to execute as many unexecuted actions as possible.

Algorithm 16 Penalized Unvisited and Visited Actions Comparison Reward Algorithm

```
1: Initialize $currentState, executedAction$

2: $executedAction.executionCount += 1;$
3: $reward := -0.5f$
4: $unvisitedActions := 0$

5: for each $action$ in $currentState.getActions()$ do
6:     if $action.executionCount == 0$ then
7:         $unvisitedActions += 1$
8:     end if
9: end for
10: $previousStateUnvisitedActions := 0$

11: for each $action$ in $previousState.getActions()$ do
12:     if $action.executionCount == 0$ then
13:         $previousStateUnvisitedActions += 1$
14:     end if
15: end for

16: if $previousStateUnvisitedActions > 0$ and $executedAction.executionCount > 1$ then
17:     $reward := -1f$
18: else if $unvisitedActions > 0$ then
19:     $reward := \frac{unvisitedActions}{currentState.getActions().size()}$
20: end if
```

7.3. Defining the experiments

At this point of the study there are three Q-learning algorithms and a total of 6 rewards, 3 new and 3 old. To know which reward is a good reward for a certain algorithm some experiments can be run. The experiments will show which combination of reward and Q-learning algorithm reaches the highest test-effectiveness. The experiments will also show if the newly added rewards improve Q-learning over the old ones. This can be seen by comparing the test-effectiveness of the different algorithms. The test setup is exactly the same as in chapter 6. The only difference is that different rewards are going to be used.

7.3.1 Research question: what do we want to find out

For the second research question, the goal is to find the configuration of Q-learning algorithm and reward which is the most effective in exploring a system under test. This is measured through the test-effectiveness. The test-effectiveness of an algorithm is in this case the instruction and branch coverage percentage which the algorithm reaches after 10 sequences of 300 actions. By comparing the results of all experiments it can be concluded which combination of Q-learning algorithm and reward performs the best.

7.3.2 Dependent variables

The dependent variables are the instruction coverage and branch coverage. As explained in the previous section, these variables can be used to compare the different Q-learning configurations. To create good conclusion about the algorithms, the dependent variables must be available after each action. This is a must since algorithms that reach similar coverage can be compared by looking at the whole test run. To get the coverage, it must be extracted from the SUT. For the experiments in this chapter the SUT is Rachota. Since Rachota is a Java application, the coverage can be measured using JaCoCo (Java Code Coverage). This is a Java library for measuring code coverage. By using JaCoCo, code coverage can be extracted after each action. This information is then put into a file which later can be used to see the reached instruction and branch coverage of the algorithm at any action.

7.3.3 The system under test

The system under test is Rachota, Rachota is a Time Tracking application which is written in Java. It is a simple application with not many forms. This is beneficial for TESTAR since it can easily use most widgets without needing specific input to continue. Most of the Rachota application is accessible and is not hidden behind the need for certain input. This makes it easy for TESTAR to access most of the application.

7.3.4 Studying Q-learning algorithms and rewards: the factors

The factors of our study are the Q-learning algorithms and the rewards functions. We want to compare the different Q-learning algorithms from Chapter 6 combined with the rewards from this chapter to random action selection. All the other independent variables are kept constant. We will study the following rewards as factors:

1. Unvisited and Visited Actions Comparison Reward
2. Penalized Unvisited and Visited Actions Comparison Reward
3. Widget Tree Reward
4. Image Recognition Reward
5. Execution count
6. Widget Tree Image Recognition Reward *with a share of 0.1*
7. Widget Tree Image Recognition Reward *with a share of 0.2*
8. Widget Tree Image Recognition Reward *with a share of 0.3*
9. Widget Tree Image Recognition Reward *with a share of 0.4*
10. Widget Tree Image Recognition Reward *with a share of 0.5*
11. Widget Tree Image Recognition Reward *with a share of 0.6*
12. Widget Tree Image Recognition Reward *with a share of 0.7*
13. Widget Tree Image Recognition Reward *with a share of 0.8*
14. Widget Tree Image Recognition Reward *with a share of 0.9*

The algorithms will be referred differently to make it an easier read. Basic Q-learning will be referred to as BSQ, QV-learning will be referred to as QV and Double Q-learning will be referred to as DBQ. In total the experiment consists of (3×14) 39 test runs.

7.3.5 The constant independent variables

We used the same SUT-specific configuration for Rachota for all reward/Q-learning-combinations, hence these are independent variables:

- **State abstraction**: Since information can change continuously in an application, TESTAR must be able to identify similar states and group them together. If TESTAR cannot do this, Q-learning is not possible. This is because if one letter changes on a page and this is seen as a new state, almost all states have the default value as Q-value which means that there is no learning. To group these states TESTAR uses state abstraction.

  In TESTAR, an abstract state can be based on multiple concrete states. Every time a new concrete state is reached, the ModelManager checks if there is an abstract state which is based on the concrete state. If there is no such abstract state it will create a new abstract state. When creating an abstract state, it converts the actions in the state to abstract actions. These actions only have attributes which are required to identify the action. This is useful since if there is a change in the current state, the abstract version of the actions should stay the same. Which causes multiple concrete state to relate to the same abstract state.

- **Action derivation**: When using Rachota a specific TESTAR protocol is needed to derive certain actions. By default TESTAR builds a Widget Tree and passes this on to the protocol. The Rachota protocol iterates through the Widget Tree to add possible actions to the Widgets. Such as scrolling, clicking and text input.

- **Filters**: The Rachota protocol sometimes helps TESTAR. When TESTAR tries to create an invoice in Rachota, the protocol automatically forces a correct price and hour input so TESTAR can continue with the form. This enables TESTAR to test the application more thoroughly without failing the form continuously.

Other independent variables for the experiment, which are constant values and are defined in the test.settings file:

- Oracles: none
- Minimum waiting time between executed actions: 0.1
- Action duration: 0.1
- Sequences: 10
- Number of actions per sequence: 300
- Q learning parameters:
  - α: 1.0
  - γ: 0.99
  - ϵ: 0.3
  - β: 1.0

7.3.6 Design of the experiment

To be able to draw valid conclusions about the results and deal with the randomness of TESTAR all test runs of the experiment are repeated 10 times. 10 times should be enough to detect any outliers and make assumptions about the accuracy. Concurrent Virtual Machines (VMs) were used for running the tests. Each of the virtual Windows machines are configured with a 2.8GHz CPU and 8GB RAM.

7.4. Results of the experiment

7.4.1 Unvisited and Visited Actions Comparison Reward (Rachota)

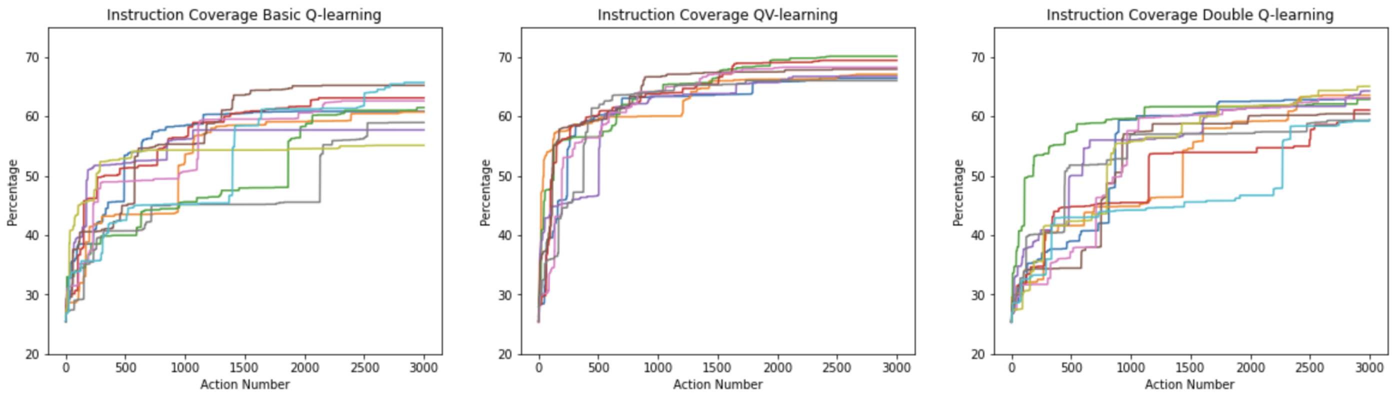

Figure 7.2: Unvisited and Visited Actions Comparison Reward - Instruction Coverage on Rachota

By looking at the instruction coverage QV performs best. Whilst BSQ performs worst. If QV is compared to DBQ the final consistency is around the same. The instruction coverage of QV is around 5% higher which means that in the current dataset QV performs better. Due to the fact that DBQ needs twice as much data since it is essentially two Q-learning algorithms and in the end DBQ is still increasing faster compared to QV, they could perform the same in a longer run. But since time and resources are not infinite, QV is the better one.

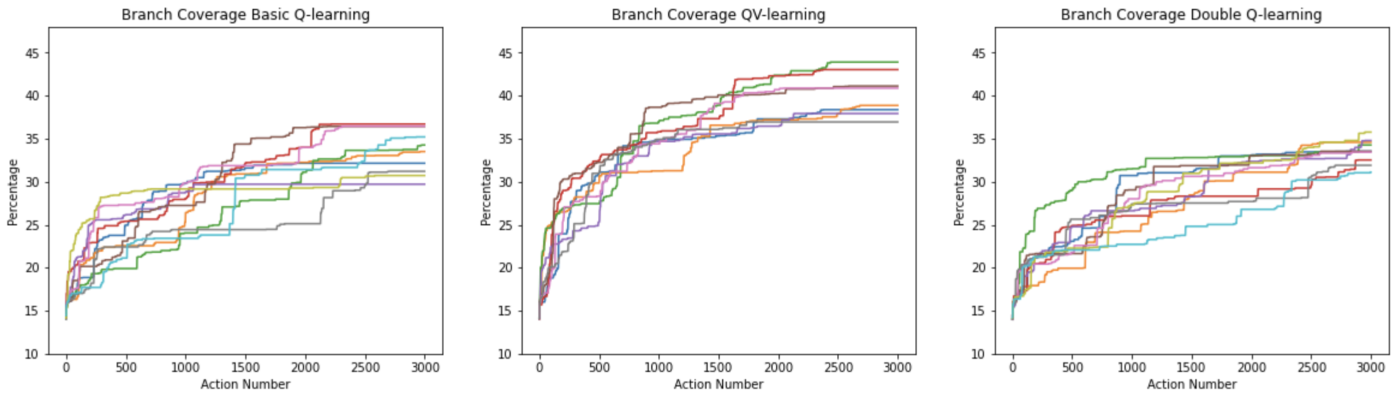

Figure 7.3: Unvisited and Visited Actions Comparison Reward - Branch Coverage on Rachota

In branch coverage we see around the same behaviour as with instruction coverage, BSQ performs worst. QV and DBQ perform well but QV is faster which causes a 5% difference in the end.

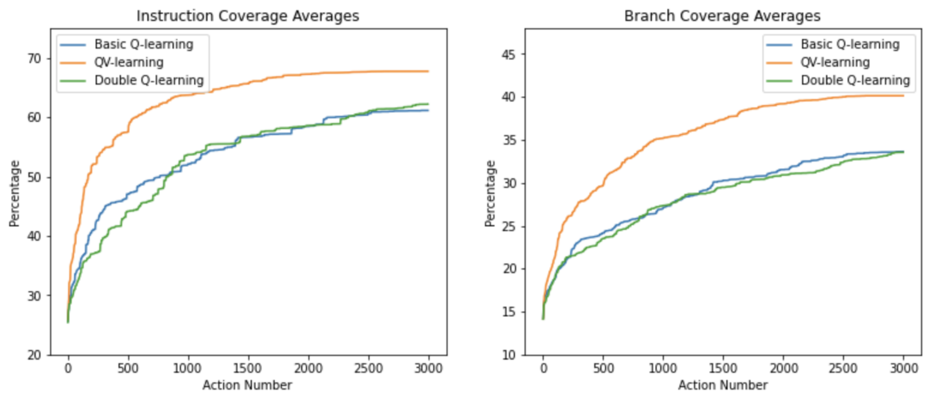

Figure 7.4: Unvisited and Visited Actions Comparison Reward - Average Coverage on Rachota

Previous findings can also be concluded from the average graphs in figure 7.4. QV is much faster but in the end DBQ starts climbing towards the coverage of QV. Based on these test results QV is the best algorithm in combination with this reward function, but DBQ is not far behind.

7.4.2 Penalized Unvisited and Visited Actions Comparison Reward (Rachota)

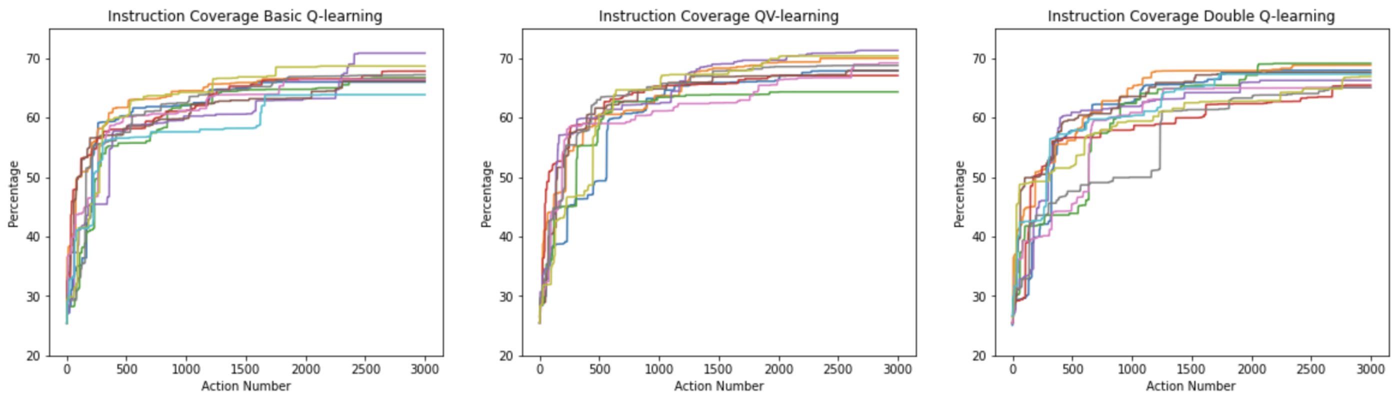

Figure 7.5: Penalized Unvisited and Visited Actions Comparison Reward - Instruction Coverage on Rachota

If a penalty is added to the *Unvisited and Visited Actions Comparison Reward*, the results are very interesting. Both DBQ and BSQ show less variety. It can be concluded from figure 7.5 that the range which the algorithms perform in is much lower compared to the range in figure 7.2. This means for DBQ and BSQ, the penalized version is a much better reward, whilst the algorithm performs around the same for QV.

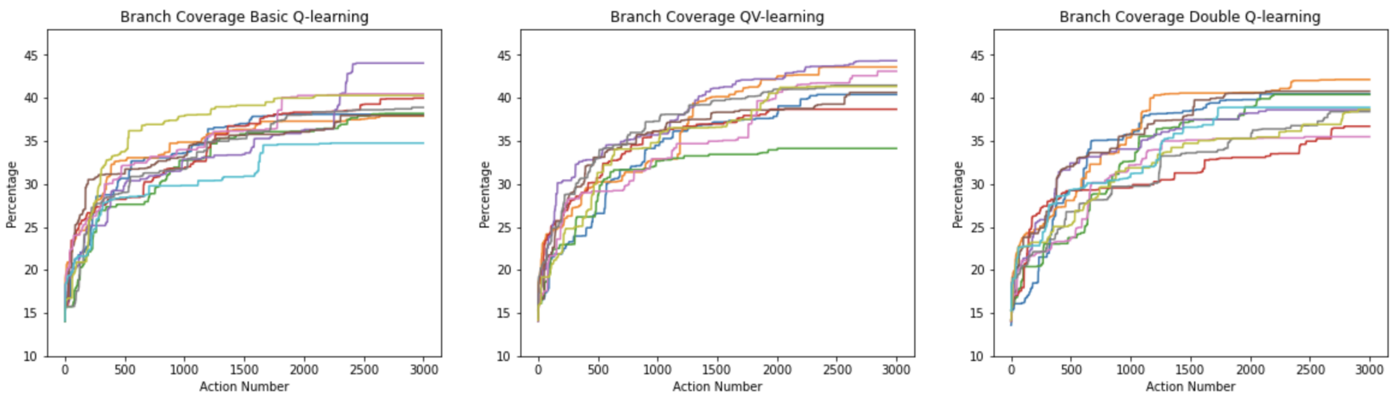

Figure 7.6: Penalized Unvisited and Visited Actions Comparison Reward - Branch Coverage on Rachota

The same can be concluded from the branch coverage graphs in figure 7.6. Both the BSQ and DBQ are getting a higher coverage percentage whilst also showing less variety. From the average graphs in figure 7.7 can be concluded that QV is still performing better but the margin is not very large. A possible reason why QV is performing the same in both algorithms is that the addition of the V value adds more knowledge of the previous states to the algorithm. By introducing a penalty for using an action multiple times whilst there are still unvisited actions, previous knowledge is somewhat introduced to Basic- and DBQ too. Which possibly causes the algorithms to be closer to each other.

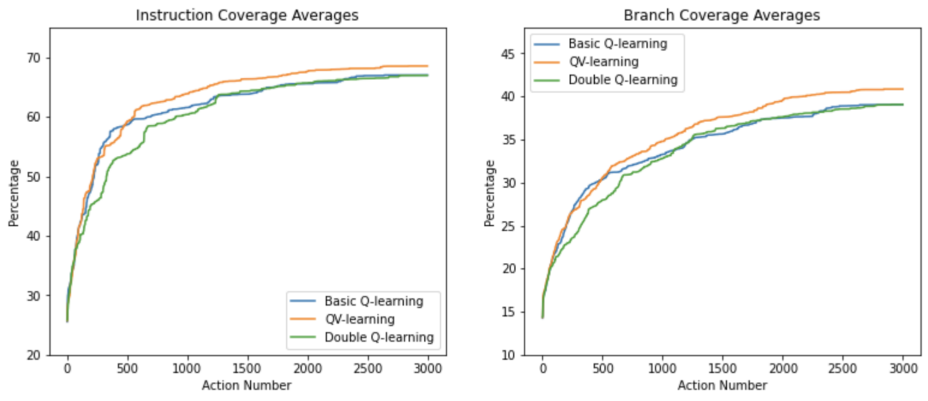

Figure 7.7: Penalized Unvisited and Visited Actions Comparison Reward - Average Coverage on Rachota

7.4.3 Image Recognition Reward (Rachota)

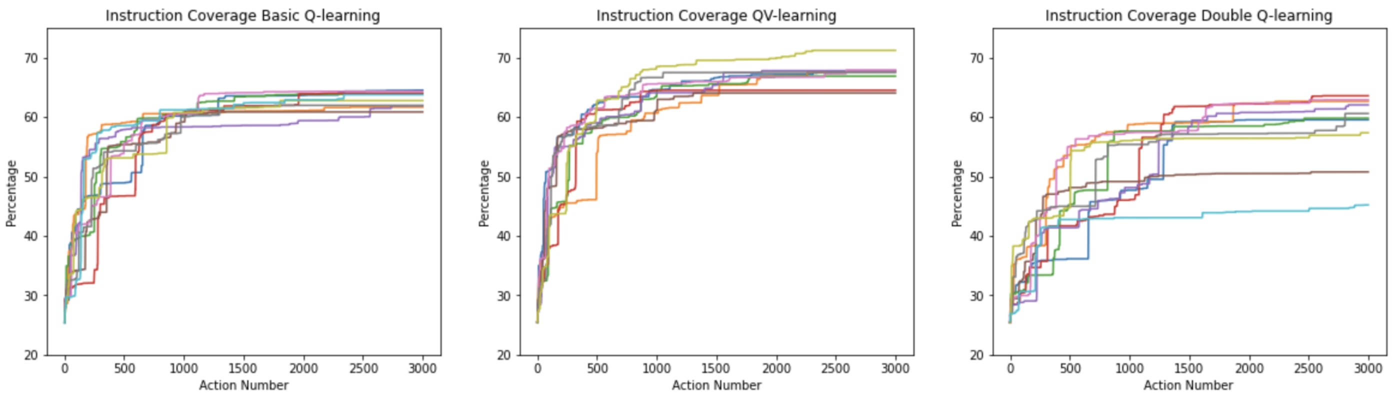

Figure 7.8: Image Recognition Reward - Instruction Coverage on Rachota

One reward that was already implemented in TESTAR is the *Image Recognition Reward*. The gathered results show that QV performs best with DBQ performing the worst. BSQ has not much variety across it runs but reaches a lower coverage compared to QV.

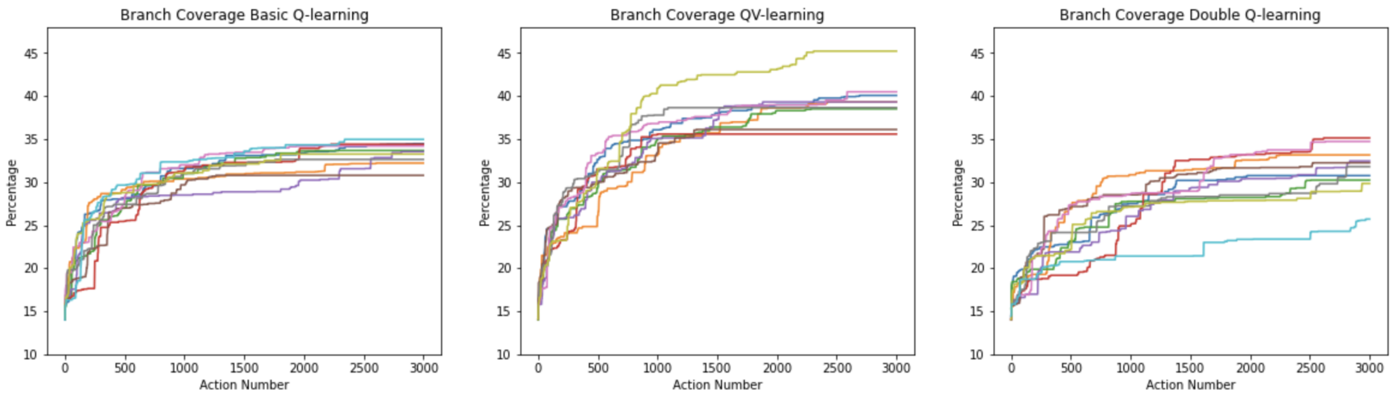

Figure 7.9: Image Recognition Reward - Branch Coverage on Rachota

Just like in the previous results, branch coverage shows the same results as with instruction coverage. From the average graphs in figure 7.10 can be concluded that QV performs the best by far whilst BSQ comes in second. A thing to note is that in the end, all the algorithms are climbing is somewhat the same rate, from this can be concluded that DBQ will probably not catch up with BSQ. This means that DBQ is probably not a good algorithm in combination with the Image Recognition reward.

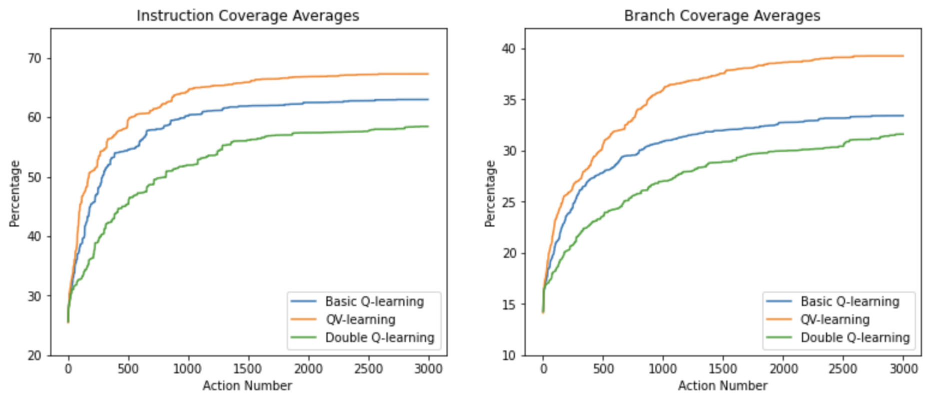

Figure 7.10: Image Recognition Reward - Average Coverage on Rachota

7.4.4 Widget Tree Reward (Rachota)

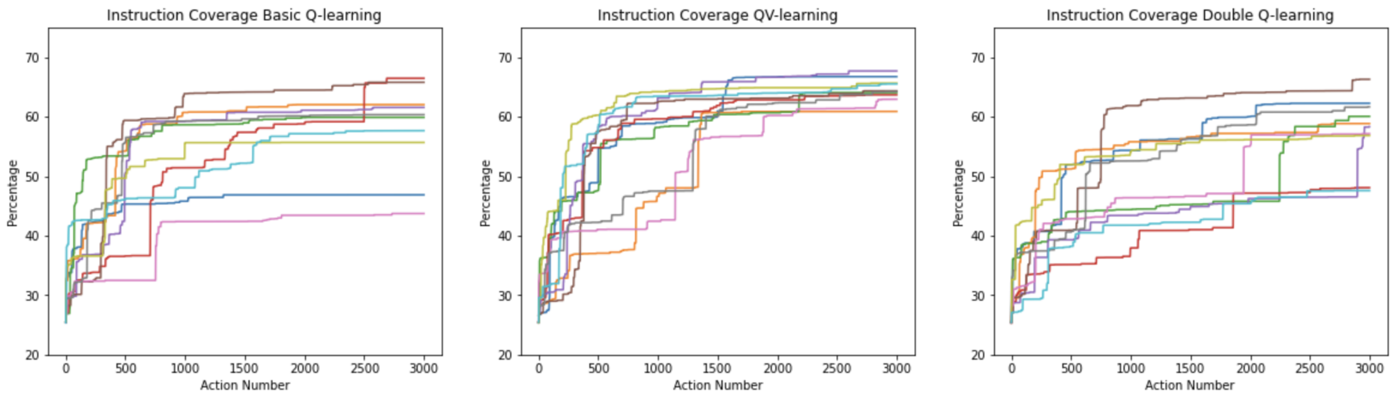

Figure 7.11: Widget Tree Reward - Instruction Coverage on Rachota

The widget tree rewards are the results from chapter 6 displayed in another figure. The graphs indicate that QV performs best and DBQ performs worst. A thing to note is, DBQ's test-effectiveness shows less variety compared to BSQ, possibly due to the more complex algorithm. The reason that the test-effectiveness is worse is probably due to the fact that DBQ needs twice as much data to make good decisions. Which means it needs a longer run to select better actions.

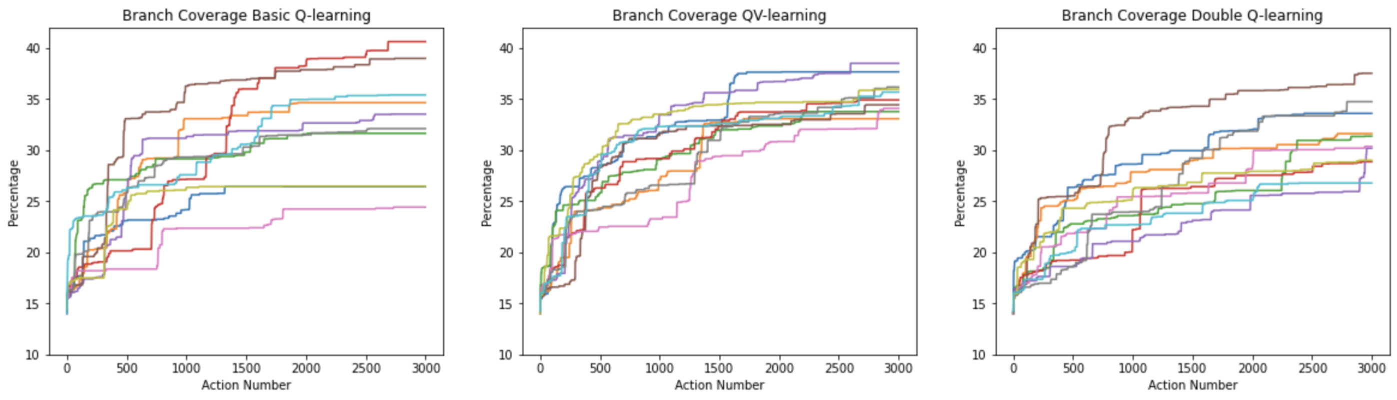

Figure 7.12: Widget Tree Reward - Branch Coverage on Rachota

The same conclusions can be made from the branch coverage and average graphs. When comparing the average graphs to the Image Recognition Reward average graphs in figure 7.10, there can be noted that the Widget Tree Reward is performing at least 5 percentage points worse. These average graphs can be a good reference point for the Widget Tree Image Recognition Reward. For that reward, the averages in figure 7.13 can function as a bottom line and the averages in figure 7.10 as a top line. This makes it useful to see if the new reward works and possibly improves the current rewards. It is also possible to see if the algorithms shows less variety by combining the Widget Tree and Image Recognition Reward.

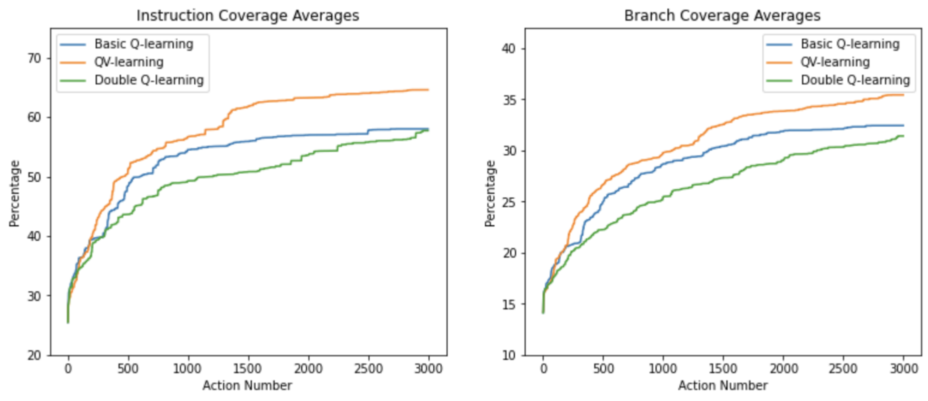

Figure 7.13: Widget Tree Reward - Average Coverage on Rachota

7.4.5 Execution Count (Rachota)

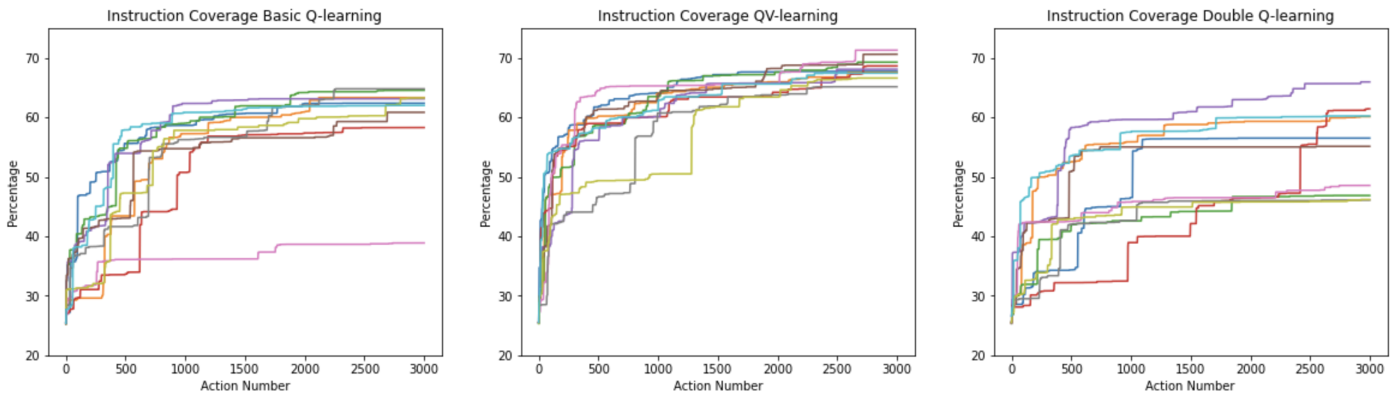

Figure 7.14: Execution Count - Instruction Coverage on Rachota

Since Execution Count and the Unvisited action rewards both calculate the reward based on the execution count of actions, they can be somewhat similar. This is can also be seen when comparing the graphs to the Unvisited and Visited actions reward. Figure 7.2 shows for all the graphs somewhat similar results. From these results it can be concluded that Execution Count works for QV-learning but not as well for the other two rewards.

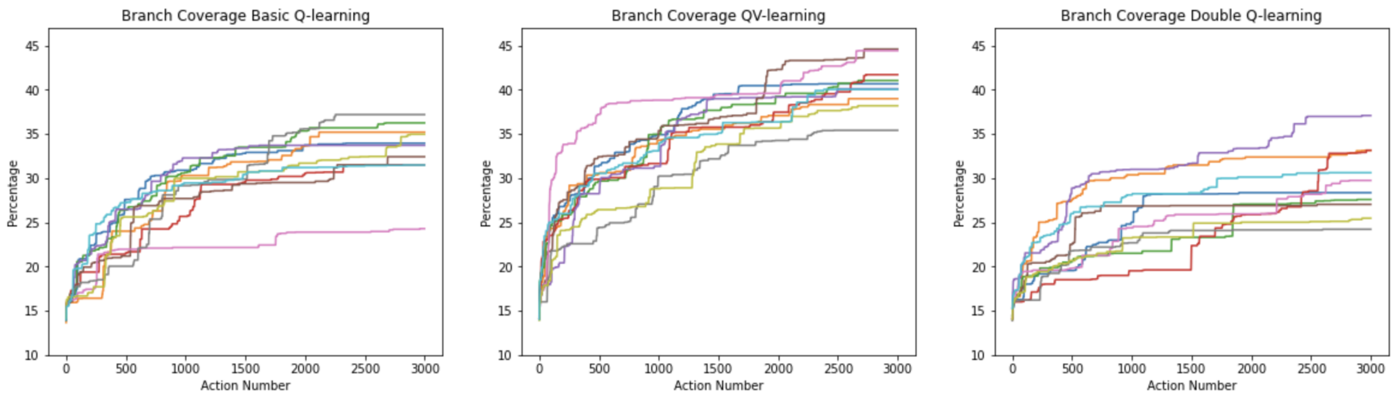

Figure 7.15: Execution Count - Branch Coverage on Rachota

This is also reflected in the branch coverage and average graphs. Which both also show somewhat similar results to the Unvisited and Visited actions reward.

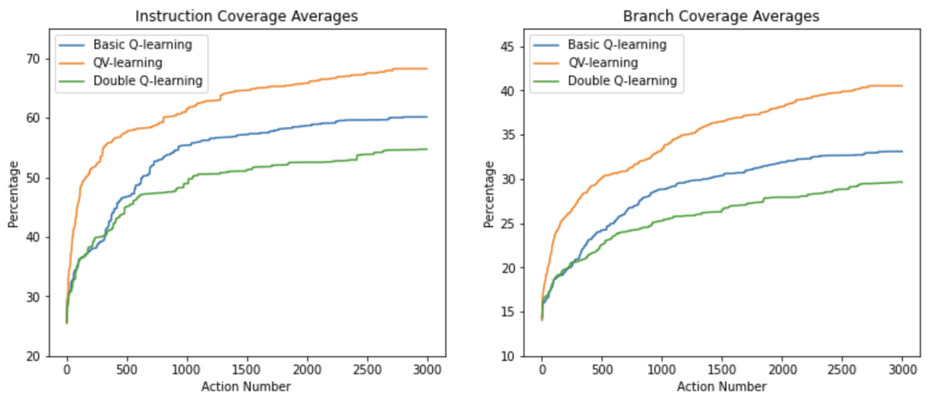

Figure 7.16: Execution Count - Average Coverage on Rachota

7.4.6 Widget Tree Image Recognition Reward *with a share of 0.1* (Rachota)

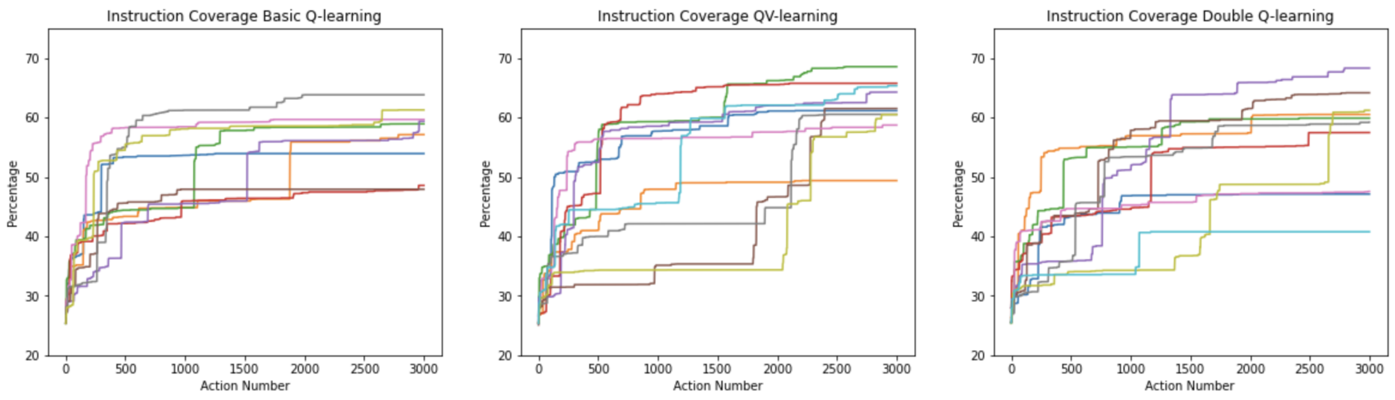

Figure 7.17: Widget Tree Image Recognition Reward *(0.1)* - Instruction Coverage on Rachota

Looking at the results from this reward, the instruction coverage shows that the coverage is slightly worse than the Image Recognition reward in figure 7.8. This could be an early indication that a combination of the Widget Tree and Image Recognition rewards is not a good reward. If the rewards are not enhancing each other, the possibility is that the test effectiveness of the *Widget Tree Image Recognition Reward* will be the average of both individual rewards.

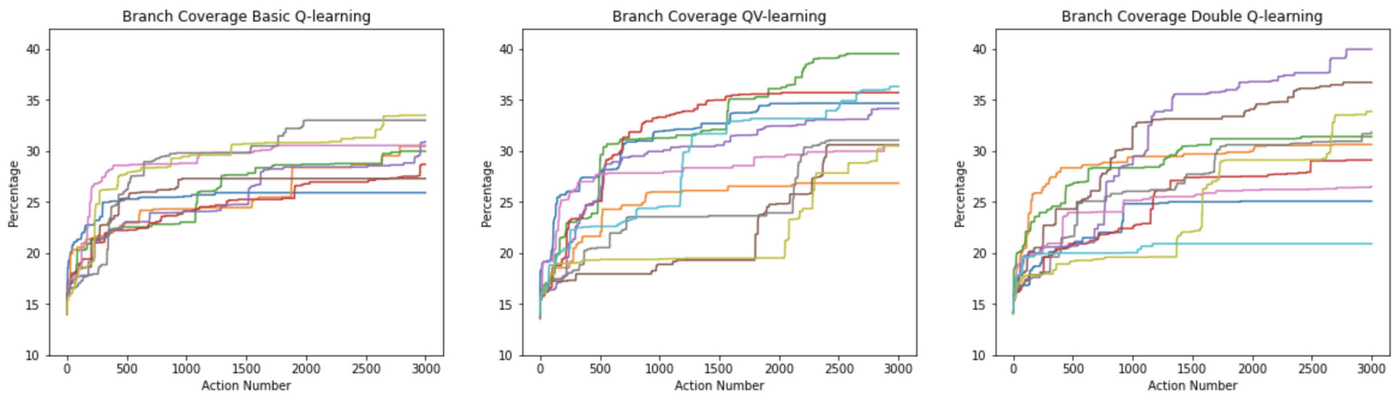

Figure 7.18: Widget Tree Image Recognition Reward *(0.1)* - Branch Coverage on Rachota

The same conclusion can be made from the branch coverage results. The algorithm performs around the same as the Image Recognition but is globally less stable and performs a bit worse.

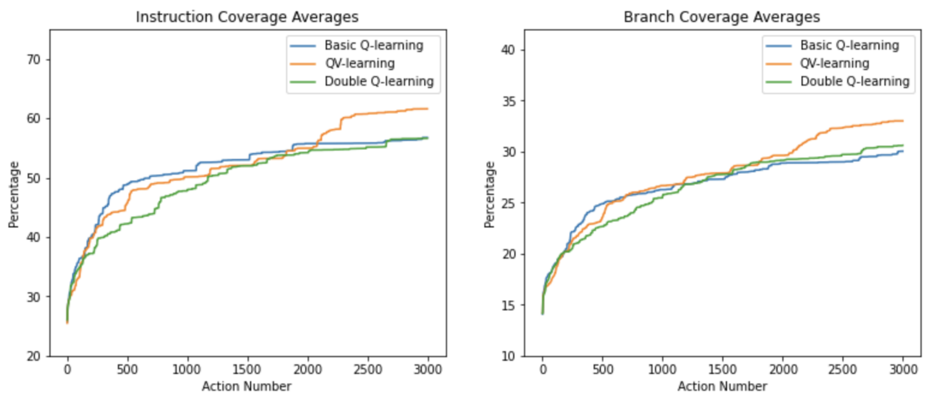

Figure 7.19: Widget Tree Image Recognition Reward *(0.1)* - Average Coverage on Rachota

Previous conclusions are reflected in the average graphs in figure 7.19, when looking at the average graphs from the Image Recognition rewards in figure 7.10 there is a clear difference, QV performs around 5 percentage points less and the other algorithms perform around the same. One thing to note is that at the end of the graphs the algorithms are stabilized with the Image Recognition Reward, but with the Widget Tree Image Recogniton Reward, the coverages are still increasing. This indicates that the algorithm performs worse, but it can possibly still reach the same coverage, it does not perform badly.

7.4.7 Widget Tree Image Recognition Reward *with a share of 0.2* (Rachota)

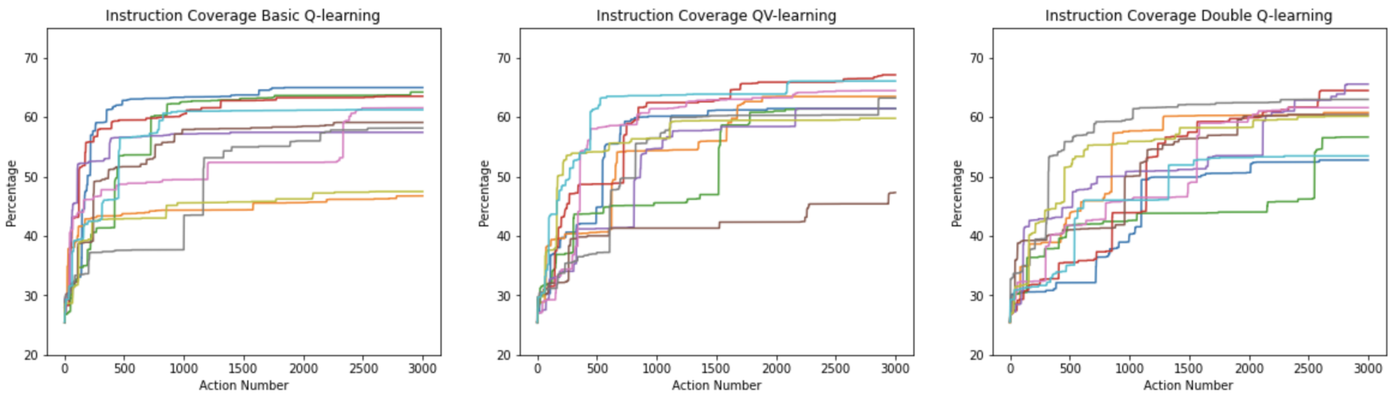

Figure 7.20: Widget Tree Image Recognition Reward *(0.2)* - Instruction Coverage on Rachota

With a share of 0.2, the Widget Tree reward gets a 20% influence on the Image Recognition Reward. From the graphs in figure 7.20 can be concluded that the QV- and DBQ algorithms show less variety than BSQ compared to the algorithm with a share of 0.1. The coverage is around the same, but there are less outliers.

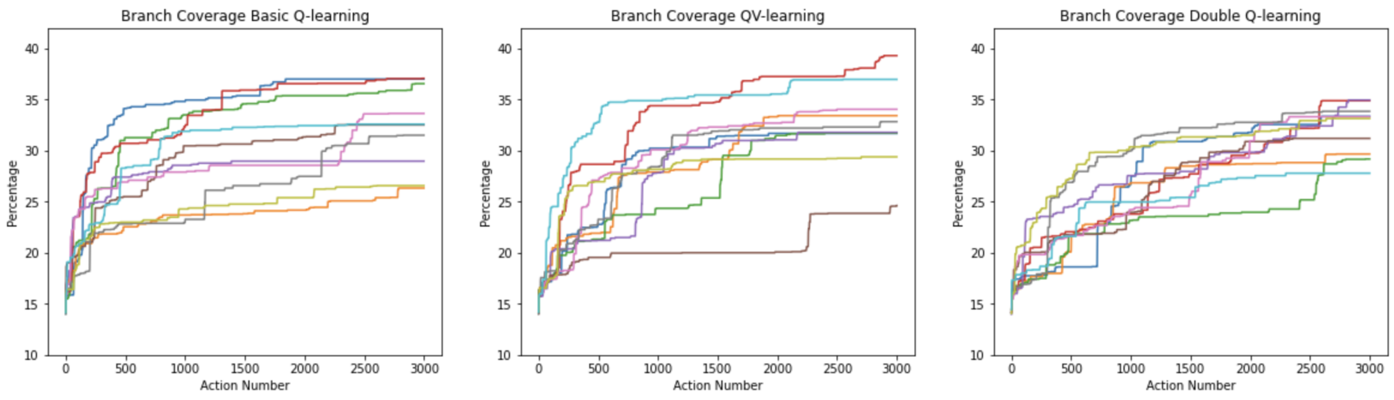

Figure 7.21: Widget Tree Image Recognition Reward *(0.2)* - Branch Coverage on Rachota

The same goes for the branch coverage, it shows more stable test-effectiveness in QV- and DBQ. If looked at the average graphs in figure 7.22, it can be concluded that it performs the same as with a share of 0.1, but is more stable, which is beneficial in the long run. This makes the algorithm more reliable. The algorithms are still performing worse than combined with the Image Recognition reward, but do not show many signs of decreasing in effectiveness, when compared to a share of 0.1.

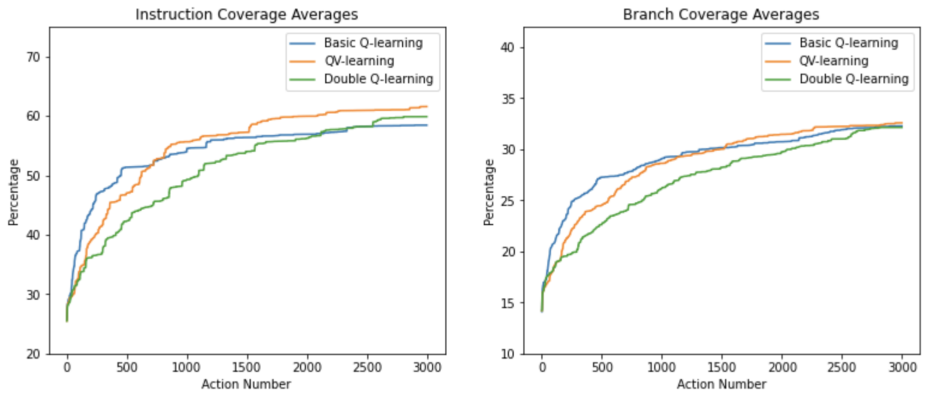

Figure 7.22: Widget Tree Image Recognition Reward *(0.2)* - Average Coverage on Rachota

7.4.8 Widget Tree Image Recognition Reward *with a share of 0.3* (Rachota)

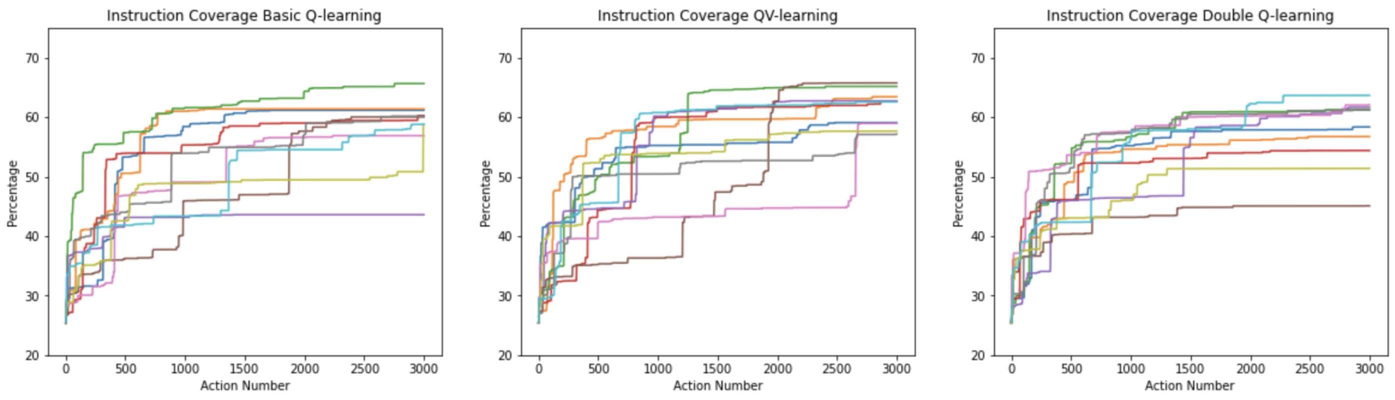

Figure 7.23: Widget Tree Image Recognition Reward *(0.3)* - Instruction Coverage on Rachota

When looking at the results of the reward with a share of 0.3 we see that the test-effectiveness of BSQ and DBQ stay around the same, QV performs a bit worse but by a minimal margin.

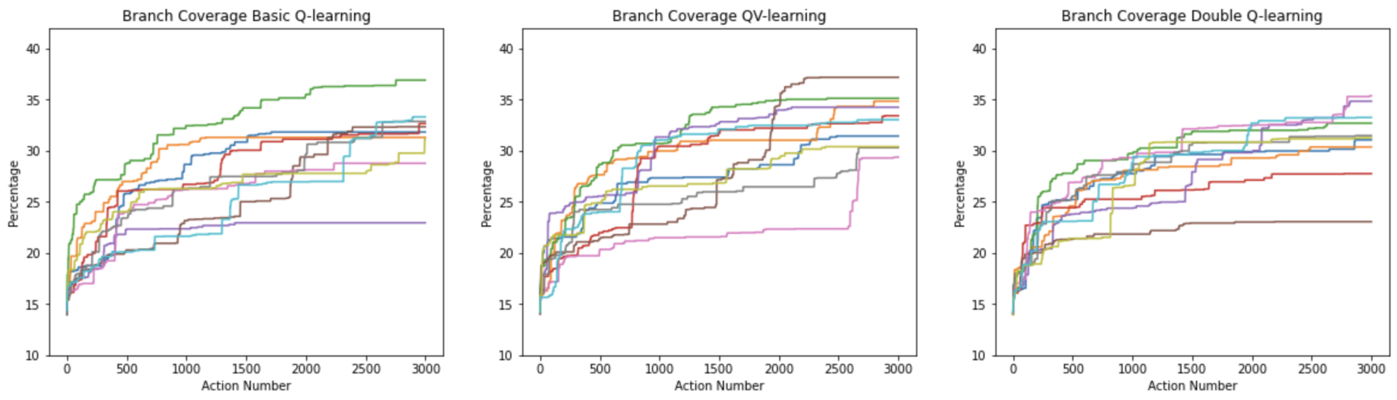

Figure 7.24: Widget Tree Image Recognition Reward *(0.3)* - Branch Coverage on Rachota

This is also reflected in the branch coverage, QV has similar results to runs with a lower share, but not consistently. In the average graphs in figure 7.25 QV performs a bit worse in comparison to the 0.2 average graph in figure 7.22. A thing to note is that QV and BSQ are both performing worse than with the *Widget Tree Reward* averages.

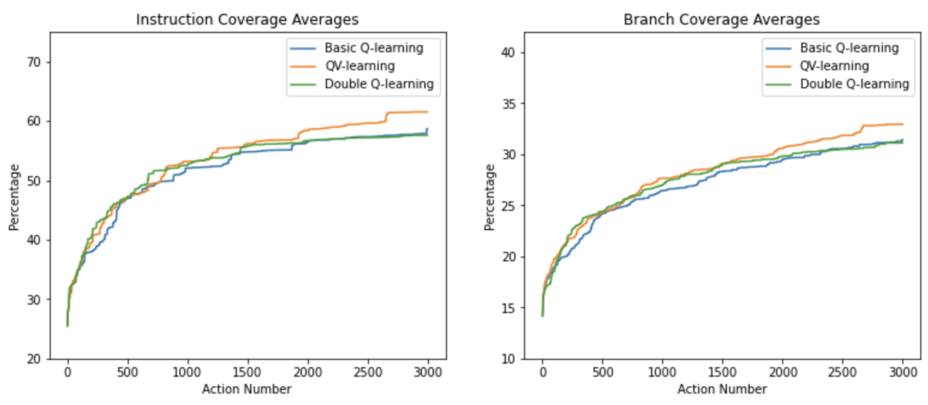

Figure 7.25: Widget Tree Image Recognition Reward *(0.3)* - Average Coverage on Rachota

7.4.9 Widget Tree Image Recognition Reward *with a share of 0.4* (Rachota)

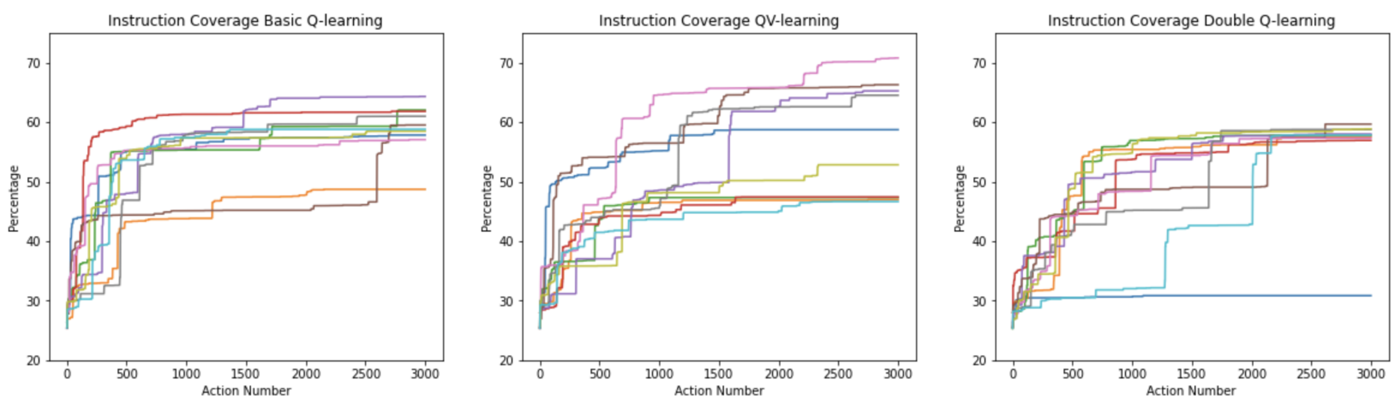

Figure 7.26: Widget Tree Image Recognition Reward *(0.4)* - Instruction Coverage on Rachota

With a share of 0.4 the Basic- and DBQ algorithms perform the same, once again QV is performing worse. For QV there are a few good runs, but overall it performs not really good. Basic- and DBQ are more consistent and show not much variety across all runs.

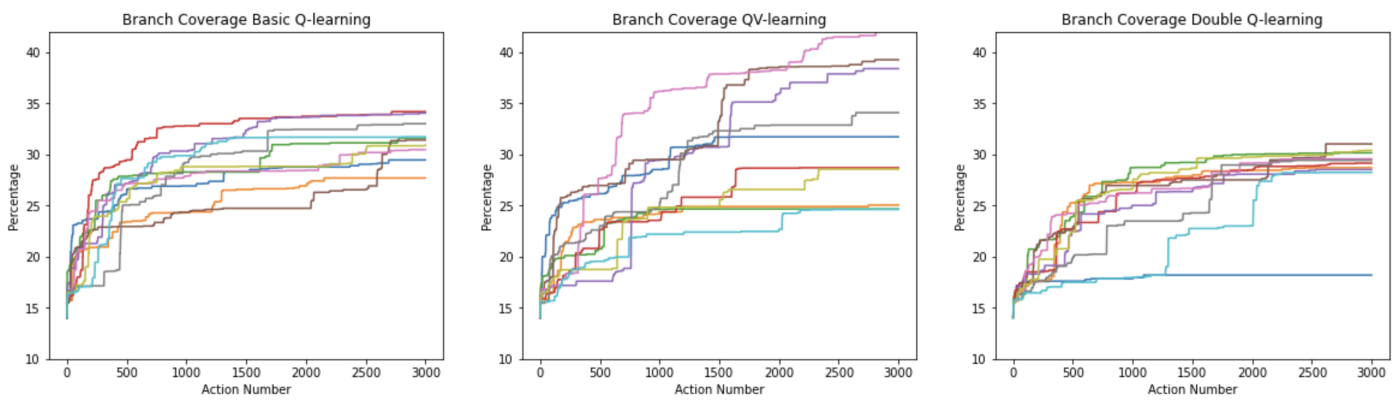

Figure 7.27: Widget Tree Image Recognition Reward *(0.4)* - Branch Coverage on Rachota

This can also be concluded from the branch coverage, QV is performing a bit random with some runs having a really good score, but others perform really bad. This can indicate that the current algorithm is not suited for QV. Overall the reward is not performing as well as the individual counter-parts, Image Recognition and the Widget Tree rewards. This can indicate that this reward is just not a good. The average graphs in figure 7.28 do show that QV is performing worse on the instruction coverage. Due to three good runs, the branch coverage is not lower compared to the average of 0.3 in figure 7.25.

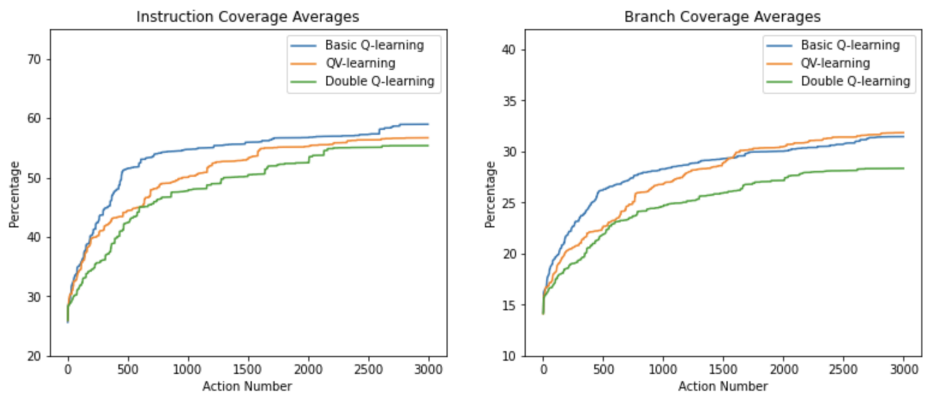

Figure 7.28: Widget Tree Image Recognition Reward *(0.4)* - Average Coverage on Rachota

7.4.10 Widget Tree Image Recognition Reward *with a share of 0.5* (Rachota)

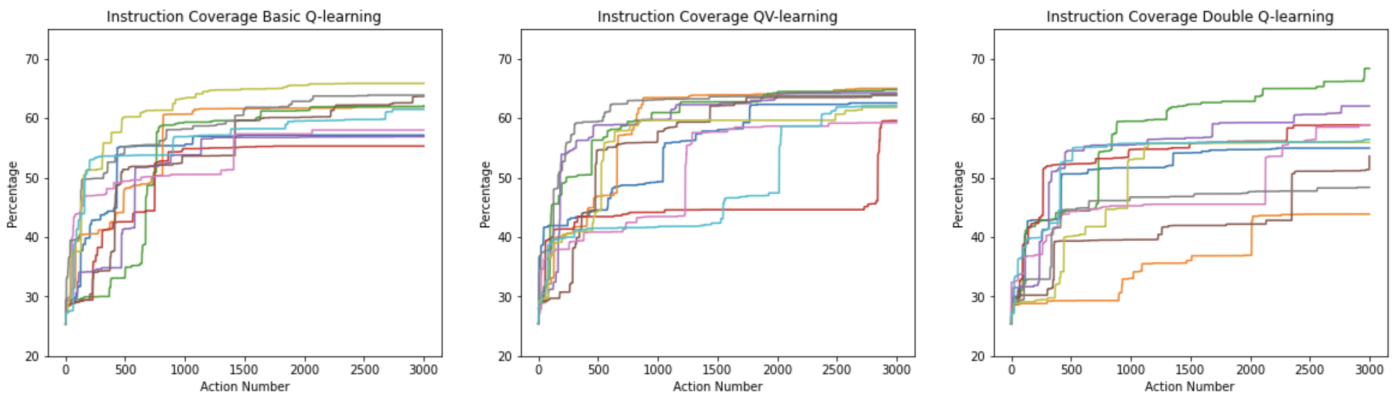

Figure 7.29: Widget Tree Image Recognition Reward *(0.5)* - Instruction Coverage on Rachota

When looking at the results of the reward with a share of 0.5, the results look more promising for BSQ and QV. DBQ does perform really bad. For DBQ some runs are good, but there are runs that perform really bad. It possibly shows that the reward does not return expected values which can cause random behaviour. If the two Q-matrices have totally different rewards for the same actions the chosen actions could be really random.

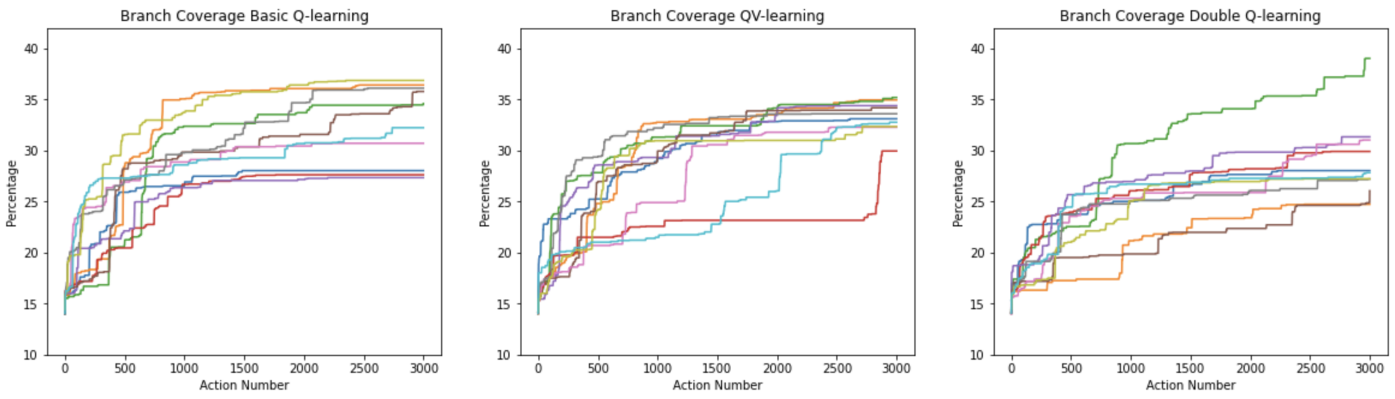

Figure 7.30: Widget Tree Image Recognition Reward *(0.5)* - Branch Coverage on Rachota

The same can be seen in the branch coverage graphs and in the average graphs. DBQ is performing really bad.

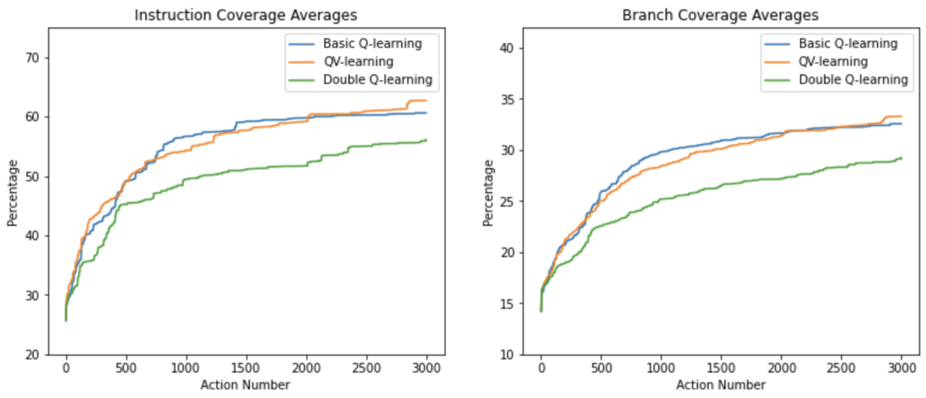

Figure 7.31: Widget Tree Image Recognition Reward *(0.5)* - Average Coverage on Rachota

7.4.11 Widget Tree Image Recognition Reward *with a share of 0.6* (Rachota)

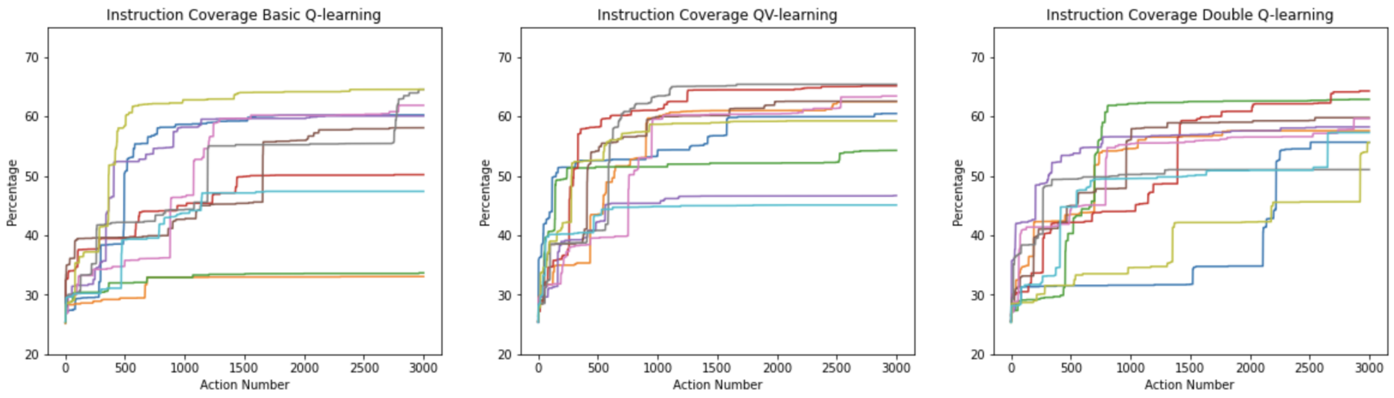

Figure 7.32: Widget Tree Image Recognition Reward *(0.6)* - Instruction Coverage on Rachota

With a share of 0.6, the Widget Tree Reward has a 60% influence over the Image Recognition reward. BSQ has a few bad runs and is less consistent than with a share of 0.5, when looking at the best runs, the algorithms perform somewhat the same.

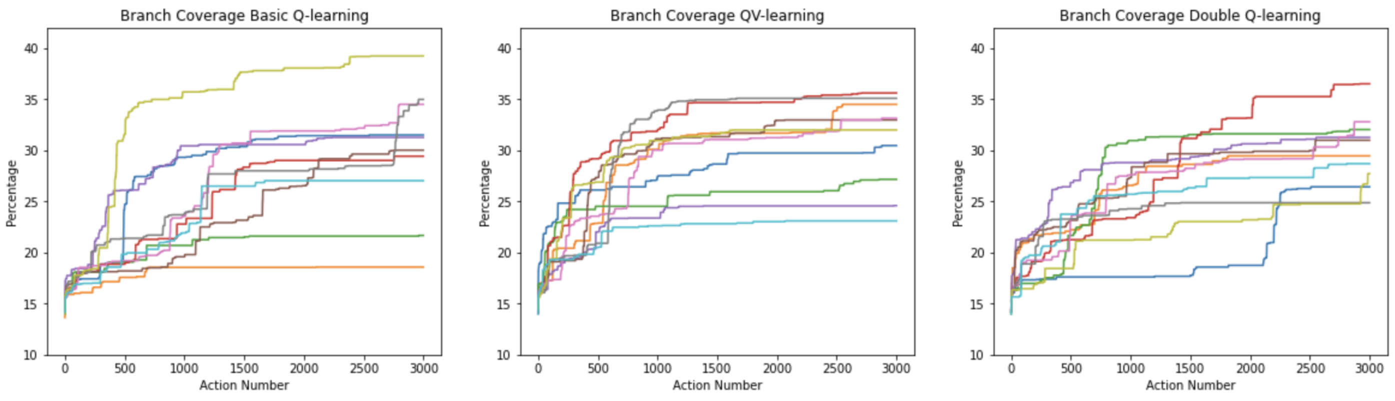

Figure 7.33: Widget Tree Image Recognition Reward *(0.6)* - Branch Coverage on Rachota

The same can be concluded from the branch coverage and average graphs, due to the bad runs the BSQ algorithm is performing worse but overall the test-effectiveness is not declining compared to the previous algorithm with a share of 0.5.

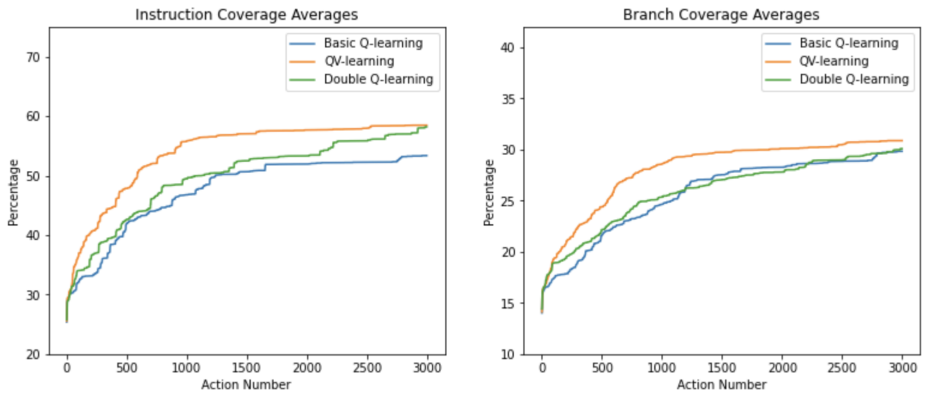

Figure 7.34: Widget Tree Image Recognition Reward *(0.6)* - Average Coverage on Rachota

7.4.12 Widget Tree Image Recognition Reward *with a share of 0.7* (Rachota)

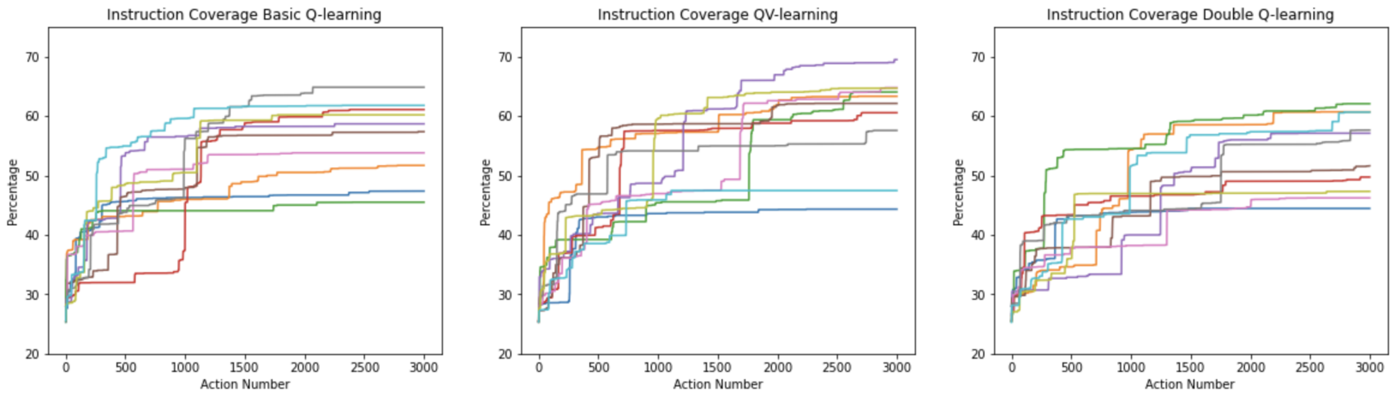

Figure 7.35: Widget Tree Image Recognition Reward *(0.7)* - Instruction Coverage on Rachota

With a share of 0.7 the final spread of the coverage is for all the algorithms very large, which means that the algorithm is not performing very well and that they show forms of random behaviour.

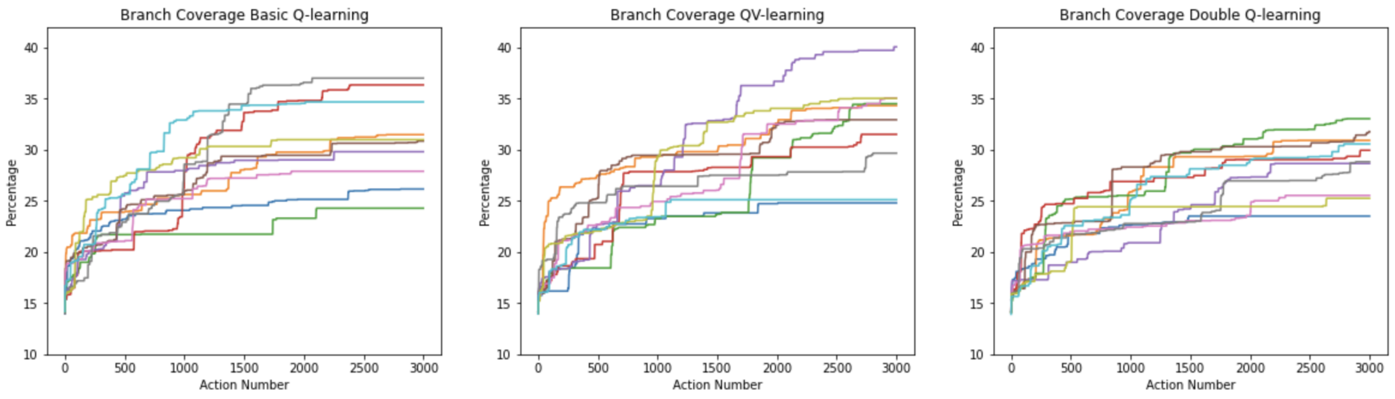

Figure 7.36: Widget Tree Image Recognition Reward *(0.7)* - Branch Coverage on Rachota

This can also be seen in the branch coverage graphs in figure 7.36. Due to the inconsistency of all the runs all algorithms perform around the same which is reflected in the average graphs in figure 7.37.

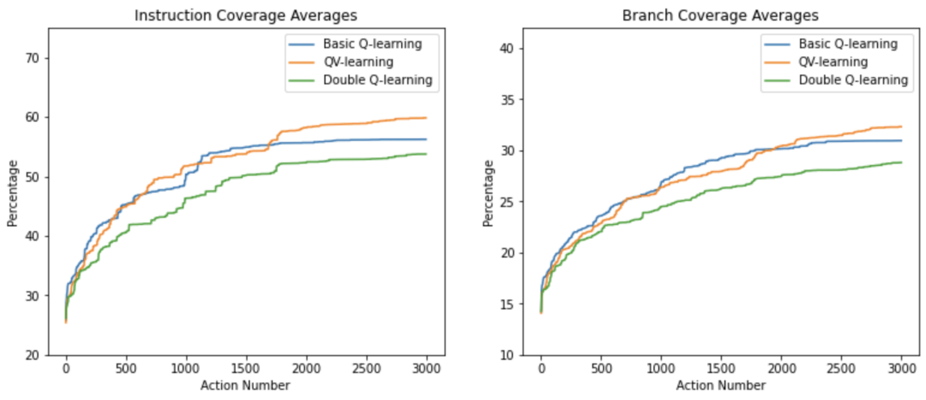

Figure 7.37: Widget Tree Image Recognition Reward *(0.7)* - Average Coverage on Rachota

7.4.13 Widget Tree Image Recognition Reward *with a share of 0.8* (Rachota)

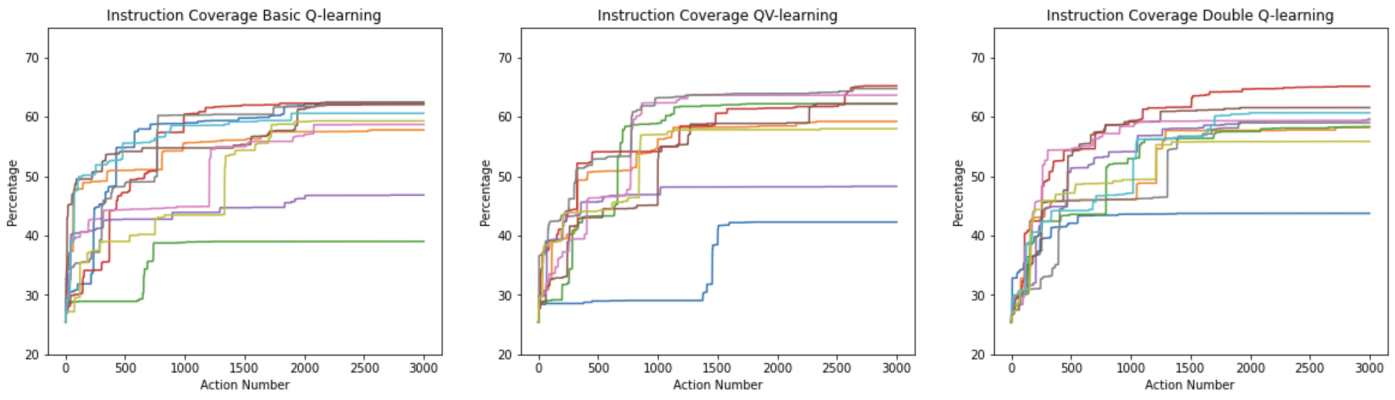

Figure 7.38: Widget Tree Image Recognition Reward *(0.8)* - Instruction Coverage on Rachota

With a share of 0.8 the Widget Tree Reward get a 80% influence over the Image Recognition Reward, this means that the algorithms should perform more like the Widget Tree Reward. This is also reflected in the instruction coverage in figure 7.38. If the instruction coverage is compared to the Widget Tree Reward instruction coverage in figure 7.11. Somewhat the same patterns can be detected, the final coverage does also not differ much.

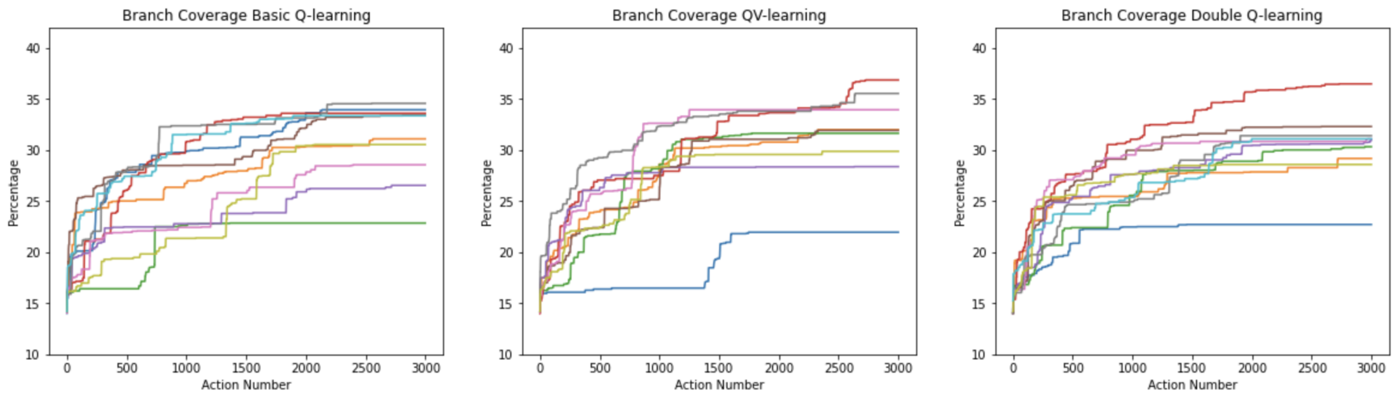

Figure 7.39: Widget Tree Image Recognition Reward *(0.8)* - Branch Coverage on Rachota

The same results can be concluded from the branch coverage in figure 7.39. When looking at the average graphs only QV is performing worse compared to the average graphs in 7.13. The other algorithms perform the same.

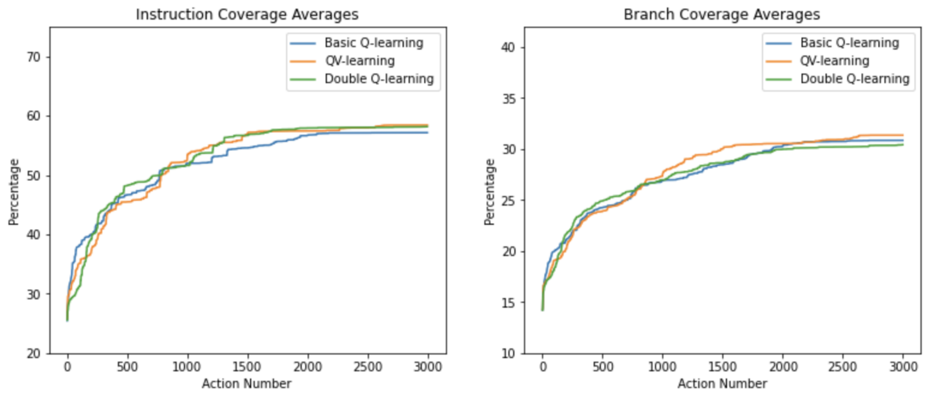

Figure 7.40: Widget Tree Image Recognition Reward *(0.8)* - Average Coverage on Rachota

7.4.14 Widget Tree Image Recognition Reward *with a share of 0.9* (Rachota)

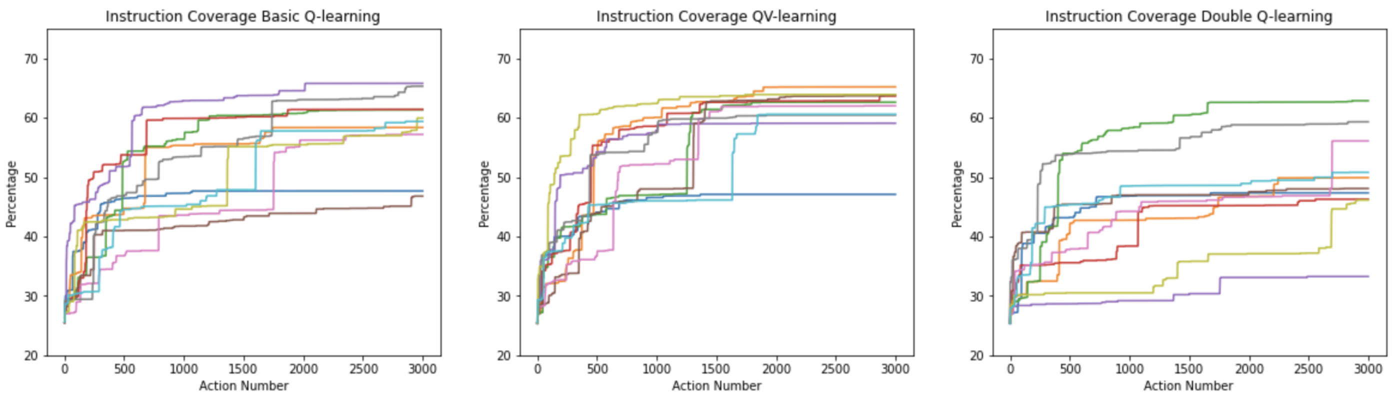

Figure 7.41: Widget Tree Image Recognition Reward *(0.9)* - Instruction Coverage on Rachota

When the Widget Tree reward has a 90% influence over the Image Recognition reward, the algorithms should perform mostly the same as the *Widget Tree reward*, but they do not. They perform worse which means that the reward is not performing well. Even a 10% influence of the Image Recognition has a bad influence on the reward.

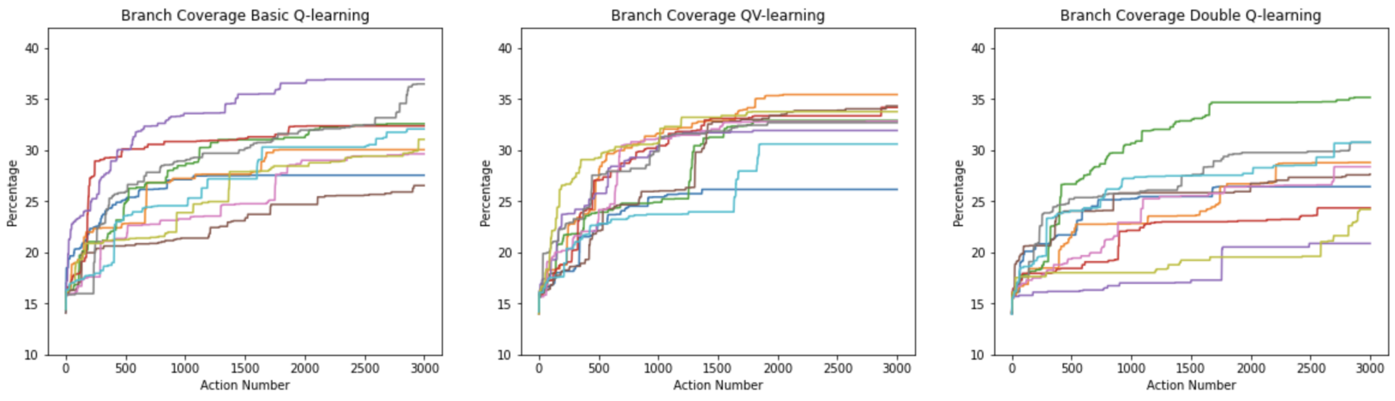

Figure 7.42: Widget Tree Image Recognition Reward *(0.9)* - Branch Coverage on Rachota

As with all the previous results this is also reflected in the Branch Coverage graphs in figure 7.42. In the average graphs there can be seen that all the algorithms are performing worse compared to the average graphs of the *Widget Tree Reward* in figure 7.13.

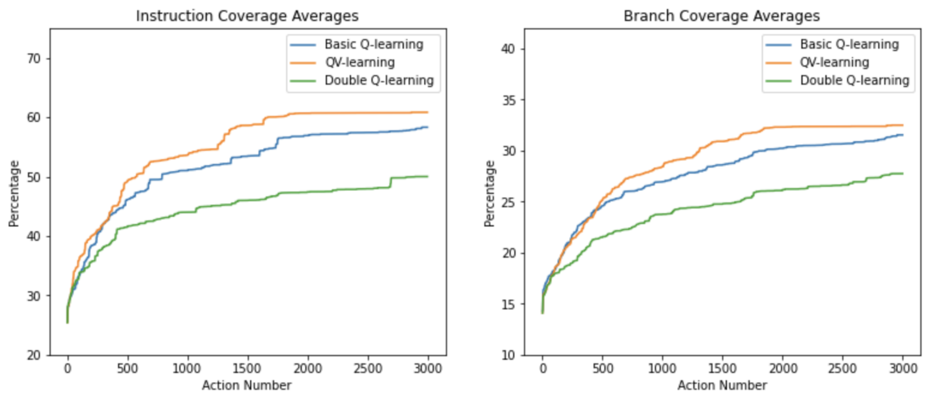

Figure 7.43: Widget Tree Image Recognition Reward *(0.9)* - Average Coverage on Rachota

7.5. Conclusions

From the results the average coverages can be extracted. The average coverages have been put into tables to make it easier to compare the results. In the tables the best coverages are highlighted.

Reward	Instruction	Branch
Unvisited and Visited Actions	61.14 %	33.63%
Penalized Unvisited and Visited Actions	**67.03**%	**39.06**%
Image Recognition	62.97%	33.42%
Widget Tree Reward	58.01%	32.44%
Execution Count	60.15%	33.11%
Widget Tree Image Recognition Reward *(0.1)*	56.74%	30.04%
Widget Tree Image Recognition Reward *(0.2)*	58.44%	32.29%
Widget Tree Image Recognition Reward *(0.3)*	58.67%	31.42%
Widget Tree Image Recognition Reward *(0.4)*	58.97%	31.46%
Widget Tree Image Recognition Reward *(0.5)*	60.62%	32.58%
Widget Tree Image Recognition Reward *(0.6)*	53.37%	29.83%
Widget Tree Image Recognition Reward *(0.7)*	56.24%	30.96%
Widget Tree Image Recognition Reward *(0.8)*	57.14%	30.86%
Widget Tree Image Recognition Reward *(0.9)*	58.32%	31.54%

Table 7.1: Average Basic Q-learning - Reward coverages on Rachota

Reward	Instruction	Branch
Unvisited and Visited Actions	67.74 %	40.14%
Penalized Unvisited and Visited Actions	**68.54**%	**40.86**%
Image Recognition	67.26%	39.26%
Widget Tree Reward	64.58%	35.44%
Execution Count	68.25%	40.53%
Widget Tree Image Recognition Reward *(0.1)*	61.58%	33.01%
Widget Tree Image Recognition Reward *(0.2)*	61.56%	32.59%
Widget Tree Image Recognition Reward *(0.3)*	61.52%	32.95%
Widget Tree Image Recognition Reward *(0.4)*	56.69%	31.84%
Widget Tree Image Recognition Reward *(0.5)*	62.71%	33.29%
Widget Tree Image Recognition Reward *(0.6)*	58.48%	30.88%
Widget Tree Image Recognition Reward *(0.7)*	59.84%	32.31%
Widget Tree Image Recognition Reward *(0.8)*	58.42%	31.36%
Widget Tree Image Recognition Reward *(0.9)*	60.84%	32.50%

Table 7.2: Average QV-learning - Reward coverages on Rachota

Reward	Instruction	Branch
Unvisited and Visited Actions	62.21 %	33.55%
Penalized Unvisited and Visited Actions	**66.96**%	**39.06**%
Image Recognition	58.44%	31.61%
Widget Tree Reward	57.72%	31.42%
Execution Count	54.71%	29.65%
Widget Tree Image Recognition Reward *(0.1)*	56.63%	30.62%
Widget Tree Image Recognition Reward *(0.2)*	59.54%	31.78%
Widget Tree Image Recognition Reward *(0.3)*	57.58%	31.13%
Widget Tree Image Recognition Reward *(0.4)*	55.37%	28.35%
Widget Tree Image Recognition Reward *(0.5)*	56.13%	29.25%
Widget Tree Image Recognition Reward *(0.6)*	58.20%	30.09%
Widget Tree Image Recognition Reward *(0.7)*	53.75%	28.81%
Widget Tree Image Recognition Reward *(0.8)*	58.16%	30.43%
Widget Tree Image Recognition Reward *(0.9)*	50.01%	27.75%

Table 7.3: Average Double Q-learing - Reward coverages on Rachota

Before a comparison can be made, the baseline must be set. The baseline is a test run with a pure random action selection algorithm. This shows if Q-learning improves TESTAR and if the algorithms outperform random action selection.

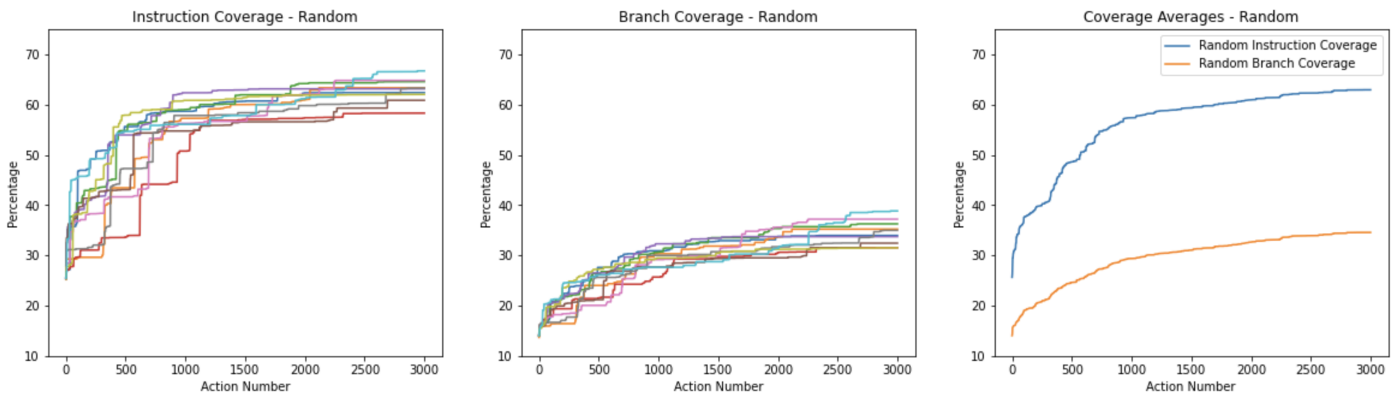

Figure 7.44: Random Action Selection Coverages on Rachota

Algorithm	Instruction	Branch
Random Action Selection	62.93%	34.56%

Table 7.4: Average Random Action Selection - coverages on Rachota

When comparing the results it can be concluded that for all algorithms the best reward is *Penalized Unvisited and Visited Actions Comparison reward*, for QV-learning the *Unvisited and Visited Actions Comparison reward* performs around the same. But for Basic and DBQ the unpenalized version performs much worse. This can be due to the fact that with a penalty more information about previous states is leaked to the algorithm. This information was already present in QV-learning due to the V value, this can explain why adding a penalty does not improve QV-learning by much.

A thing to note is that the combination of both the Image Recognition Reward and the Widget Tree Reward performs very random for almost all the shares and all the rewards are performing worse than the Image Recognition Reward. There are no shown improvements by combining the Widget Tree and Image Recognition Rewards. It would be better to choose one of the individual rewards since those rewards showed a higher test-effectiveness. It could also be that the current reward is not working and the rewards should be combined in a different way. For example always choosing the highest reward of the two.

The top 5 algorithms and reward combinations in order are:

- QV-learning with *Penalized Unvisited and Visited Actions Comparison Reward*
- QV-learning with *Execution Count*
- QV-learning with *Unvisited and Visited Actions Comparison Reward*
- Double Q-learning with *Penalized Unvisited and Visited Actions Comparison Reward*
- Basic Q-learning with *Penalized Unvisited and Visited Actions Comparison Reward*

There is one factor which could possibly alter the ranking of the Q-learning algorithms which is time. If an algorithm reaches a high coverage but performs 2+ times as slow as other algorithms, it could be preferable to choose an algorithm which does not perform as well but is faster. With the lesser performing algorithm you can now run more test runs in the same time frame, possibly increasing the test-effectiveness. By looking at the results almost all of the algorithms took around 6 hours to reach 10 sequences of 3000 actions. Which means the ranking can stay the same.

To fully conclude this research question, the gained coverages must be compared to the Random baseline. If this baseline is compared to the algorithm which existed priorly to this research, Basic Q-learning. It can be concluded that only *Penalized Unvisited and Visited Actions* is performing significantly better, with Image Recognition performing around the same. The same goes for Double Q-learning, here only *Penalized Unvisited and Visited Actions* performs significantly better and *Unvisited and Visited Actions* performs around the same.

The big improvements come when comparing QV-learning to the Random baseline. Except for the *Widget Tree Image Recognition Rewards (0.1-0.9)* all rewards perform significantly better. With some rewards reaching around a 6 percentage points higher instruction- and branch coverage. This shows that Q-learning can indeed be a suiting improvement to action selection in TESTAR.

8 Measuring test-effectiveness on web-based applications

This chapter is about RQ3 *How can the performance (test-effectiveness) of a combination of Q-learning algorithm and reward function in web-based applications be measured?* In this chapter there is explained how the test-effectiveness is measured on the web-based application Moneybird and how the test results are gathered. Eventually the results will show if the algorithms from the previous chapter perform the same on a larger application.

This is needed since in previous chapters, results were gathered on relatively small java applications. For the study the results should be validated on a larger application. This allows for more distinction between the different algorithms since the possibility for a test which is randomly very good becomes significantly lower. There are also much more paths throughout the applications which better algorithms will explore best. Worse algorithms can possibly loop on a few paths. It will also show that an algorithm is not only fit for Rachota but also functions on other applications.

Currently TESTAR is able to test web-based applications but there is no protocol in place for measuring its test-effectiveness. TESTAR is currently only able to measure test-effectiveness of java applications. That is why for this study a test-effectiveness measuring protocol for the web-framework Ruby-On-Rails is written. This can be used on the Moneybird application.

Since the results will also function as a validation the results should be comparable to the results of the previous chapter. That is why the test-effectiveness should be measured in a similar way, using code coverage. The benefit here is that Ruby has an built in coverage API which is useful for extracting data. With the simplecov[1] gem it becomes possible to easily extract the code coverage data. With code coverage as a test-effectiveness measurement we can compare the results to the results of the previously tested java applications. Since both measurements are code coverage we should see similar graphs. The graphs will have different values since in 3000 actions the larger web-based application will not reach

[1]`https://github.com/simplecov-ruby/simplecov`

the same amount of code coverage as Rachota, but the differences between the algorithms can be the same.

8.1. The system under test

As introduced, the system under test is Moneybird, Moneybird is a web-based accounting platform. The web-application is written in Ruby-On-Rails and has many forms and pages. This gives TESTAR access to a wide variety of actions.

8.2. Gathering results

Since a web-based application has some more components than a java application. The web-based application is not run locally on the testing machines. For each testing machine there also is a server instantiated. TESTAR can then access the web-based application by going to the specific URL of the server in the chrome browser. Each server measures the code coverage for which we set-up some API calls. Through $< URL > /app/code\_coverage$ the current code coverage can be requested. The response will include:

- The current percentage of code coverage
- The total number of lines
- The total number of visited lines
- The total number of unvisited lines

TESTAR records this data after every step which allows us to build the coverage graphs which can be compared to the test-effectiveness on Rachota. Another call exposed to TESTAR is $< URL > /app/clear\_coverage$, this call can be used to clear all recorded coverage on the server. This call will be used before the first action so that coverage of different tests will not infer with the current test.

Each server also has access to their own seeded database. This database is reseeded after every test run. This prevents that other test runs could influence another test run by for example deleting or inserting data.

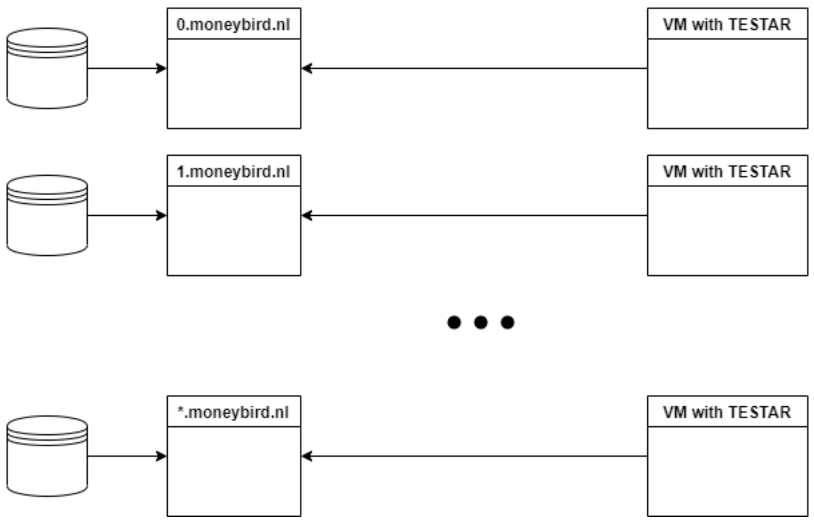

Figure 8.1: Moneybird TESTAR testing environment

Each test will again be run on 10 test machines which each execute 10 sequences of 300 actions. The same algorithms as in chapter 7 are used with the same configuration. The only difference is the SUT and TESTAR protocol.

8.3. RESULTS

In this section the results of the experiments are discussed. But before those can be discussed, first an issue with the Widget Tree related rewards must be clarified. When testing with the *Widget Tree Reward* and *Widget Tree Image Recognition Reward* the experiment showed that it did not perform well on the Moneybird application. Test runs for the other rewards ran quickly in around 3 hours. The Widget Tree rewards were not even on 30% after 24 hours. Meaning that a full test-run would take up 33.33 days of computer time comparing to 1.25 days for the other rewards. The reward was tested on another web application with less widgets, whilst being a bit quicker here it still performed 2-3 times slower in comparison to a Java application. This is why the choice was made to *not* include the Widget Tree rewards in the experiment. They already performed worse on Rachota and by also getting this slow, it is not a feasible reward to test web applications with.

Reward	Basic	QV	Double
Unvisited and Visited Actions	4.60 %	**5.64**%	5.17%
Penalized Unvisited and Visited Actions	5.07%	5.67%	**5.68**%
Image Recognition	4.17%	**5.44**%	4.70%
Execution Count	4.42%	**5.77**%	4.81%

Table 8.1: Average Coverage of Q-learning algorithms on Moneybird

After collecting the results one thing immediately stood out, which is the coverages. The coverages are very low which can mean three things.

1. The application is way too large to discover in 3000 actions
2. TESTAR reaches states which it cannot easily continue from
3. Large parts of the application are inaccessible to TESTAR

The first one can easily be debunked by trying out the application and checking which coverage can be reached. When testing out the application manually for 100 actions the coverage reached was around 13-14%. Which means that TESTAR should be able to reach at least that amount of coverage. Why did it not progress further than those states?

This can be point 2 on the list. Was TESTAR not able to continue is some states? By comparing the manual application try out versus some of the recorded TESTAR runs there can be seen that TESTAR cannot easily fill in required forms or sequences of actions to reach an unseen state. For example creating a new document which requires TESTAR to do 3 specific actions in a row. This must be done without clicking on anything else. The three steps TESTAR must take are hitting the new document button, selecting a contact and hitting save. Since TESTAR does not know how the forms work it does not know how to search for a contact or even that it should save after finding one. This causes that TESTAR only progresses past this point when being really lucky. Most of the times it just continues by clicking on the navigation or doing something irrelevant to the form. This means that point two is indeed a reason that TESTAR does not reach a higher coverage.

Another point on the list is if large parts of the system are inaccessible to TESTAR. This is true, TESTAR does not have access to most settings and the admin panel since from there it could potentially alter the application so much that testing will crash. Other parts of the system work only in a production environment. Such as external connections with third parties and certain background jobs which run periodically on the production environment of Moneybird. Thus being inaccessible lines of code whilst testing. After calculating the lines of code this amounts to around 45% of the application, meaning that at anytime TESTAR cannot progress past 55% code coverage.

Despite having low coverages, the algorithms can still be compared to each other since they are all used in the same environment. This means that the results can be compared to each other. The results can be compared to the results of chapter 7. By looking at the differences between the percentages there can be noted that again QV-learning is the best performing algorithm. In all cases except the Penalized Unvisited and Visited Actions reward (PUR), QV-learning performs best. In the case of PUR, QV-learning almost equals Double Q-learning (0,01 percentage points difference). Which is good since on Rachota, PUR also has the least difference between the three algorithms. The only difference is that on Rachota, QV-learning reached the highest coverage. Another difference is that on the Moneybird application Execution Count performs slightly better than PUR which was the other way around on Rachota. But the difference is marginal.

For the Moneybird application testing time also does not influence the ranking of the algorithms since the tested algorithms and pure random all achieve around the same duration. The really slow algorithms, the algorithms with the Widget Tree Rewards, were already left out from this experiment.

A fun thing to note is that TESTAR also detected two previously unnoticed bugs in Moneybird whilst using the Q-learning protocol. This is not directly an important result for the research but it shows a real world example that TESTAR can be used to detect bugs and errors in applications.

Since the algorithms almost performed the same as on Rachota, the choice was made not to include the graph images in the main section of the thesis. The graphs are included in Appendix A.

To know if there are any improvements using Q-learning a baseline with random action selection must be set. The results of running TESTAR on Moneybird using a pure random action selection algorithm are shown in table 8.2. The experiment consisted of 10 sequences of 300 actions.

Algorithm	Coverage
Pure Random action selection	5.98%

Table 8.2: Average Coverage of pure random action selection on Moneybird

Again, since TESTAR does not know how to progress past forms which causes the coverage to be low. When the Pure Random action selection is compared to the Q-learning results there can be seen that Pure Random action selection performs a bit better. This does not reflect the results from Rachota like the Q-learning algorithms did. This can be related to the fact that TESTAR does not know how to progress past forms. The Q-learning algorithms all favor larger change in the application which most of the times mean new screens with big changes on widgets. This means that most of the Q-learning algorithms will go to forms and click buttons that will cause a larger change in the application (opening forms). Pure Random on the other hand clicks randomly throughout the application which could cause that it clicks more often on smaller actions which does not cause a larger change but still causes the coverage to rise. This also means that random action selection keeps trying forms, which allows it to sometimes succeed. Q-learning will eventually stop using the forms since no progression is made. The graphs for Pure Random action selection are included in Appendix A.

8.4. CONCLUSION

The initial research question of this chapter was *How can the performance (test-effectiveness) of a combination of Q-learning algorithm and reward function in web-based applications be measured?*. In this chapter this was down scoped for Ruby-on-Rails applications where an instruction coverage measuring API was included in the application. TESTAR can use this API to reset and get coverage so it could measure reached coverage after each action. This made it possible to compare the coverages to previous results since they were on the same scale, a coverage percentage per action.

Using the created algorithms from the previous chapters and the new API, the algorithms were tested on the Moneybird application. QV-learning with the rewards PUR and Execution count showed the most promising results, just like in chapter 7. The only problem is that due to the many forms in Moneybird, TESTAR does not reach a high coverage which causes less differences in the algorithms. Pure Random action selection performed better on Moneybird by a margin of 0.21 percentage points which is minimal. To get better results TESTAR must first understand forms better.

9 Discussions

In this chapter the results of the research questions are discussed. There is also explained how these results relate to the related work.

9.1. RQ1: How can different Q-learning algorithms in TESTAR be implemented?

From the results of the first research question, in chapter 6, the study knows that there is potential in the introduced algorithms. This allowed for the algorithms to be used in research question 2. When looking further into the results, the results on Spaghetti differ not much. This is due to the size of the application. Looking back, testing on Spaghetti was not as useful due to this fact since there are no clear differences between the algorithms in the results. That is why in the first research question the algorithms were also tested on Rachota, a larger application. These results showed that after a slow start Double Q-learning perfors similar to Basic Q-learning and QV-learning performs better.

The fact that Double Q-learning performed worse in the beginning can be due to the fact that it needs to fill two Q-matrices. It does this to lower the chance of overestimation as explained by Koroglu et al. [17]. The only downside is that it requires more time to achieve good results.

9.2. RQ2: What reward functions can be used for the implemented Q-learning algorithms in TESTAR based on the available reward metrics?

In the second research question, in chapter 7, the study uses the algorithms from the first research question in combination with multiple rewards. This was done because the algorithms are not the only factor to Q-learning. The results of this research question were compared to random action selection. The results of this research question show that Q-learning can indeed be a better solution to action selection when the right reward and algorithm is chosen. The best combination (QV-learning and the Penalized Unvisited and Visited Actions reward) outperformed random action selection with a 5+ percentage points increase in reached coverage.

In this case it is good to note that from the 42 tested configurations only 9 outperformed random action selection. 5 of those 9 are configurations with QV-learning. For each of the three Q-learning algorithms the top performing configuration uses the same reward. This shows that good configuration is an important factor for Q-learning. Due to the scope of this study the configurations were not tested with different policies. This is due to the fact that with each newly added policy the needed experiments grows exponentially. This is why the related work from Veanes et al. [30] and Degott et al. [7] was used to determine the policy and policy configuration. The chosen policy was a $\epsilon - Greedy$ policy with an ϵ of 0.3. This policy worked for the algorithms since they outperformed random action selection. On the other hand, since the importance of a good configuration is shown, there could be much more potential by using the tested algorithms with different policies.

The results of the research question confirmed that Q-learning can be a better solution to action selection for automated GUI testing. This is consistent with research of Veanes et al. [30], Adamo et al. [1], Su et al. [27], Eskonen et al. [9], Li et al. [18], Humphreys et al. [14] and Harries et al. [13]. All these related work used some sort of reinforcement learning to improve testing over random action selection.

9.3. RQ3: How can the performance (test-effectiveness) of a combination of Q-learning algorithm and reward function in web-based applications be measured?

From the third research question in chapter 8 the results are not as good as initially hoped for. This was due to the fact that the used application has many forms. TESTAR does not know how it can handle or fill these forms which causes that only menu of the application gets tested. Su et al. [27] also experienced this same issue which confirms that it is not only a problem with the application or algorithms used in this thesis. To prevent this, TESTAR must be learned how it can progress past these states. Some related work already have such measurements in place by using Deep Learning to train the test suite for a specific application. This can be found in the studies of Eskonen et al. [9], Li et al. [18] and Humphreys et al. [14].

On the other hand, the research question could still be answered on how to measure test-effectiveness on web-based applications. Due to the fact that this is a very broad topic it was scoped down to Ruby on Rails applications. The choice was made since there was access to the Moneybird application and Ruby on Rails is a web-framework with a large user base. This means that introduced measurements can apply to a large number of web applications. The current measurements are only measured server-side. This means that not all of the actions TESTAR select result in an higher code coverage percentage. To get a more variety after each selected action, coverage measurements in Javascript code can be introduced in. This should give a more detailed graph if an application has many client-side actions. The code-base of the Moneybird application is mainly server-side, this is why there are no Javascript code coverage measurements in this study.

Another part of research question 3 that must be discussed are the rewards related to the WidgetTreeReward. It was found that due to how this reward works, it has a difficult time comparing states with many widgets. This got so slow that 30 runs with any other reward could be completed in the time of one run with the WidgetTreeReward. This is why the choice was made not to use the WidgetTreeReward because it is only usable on applications with a small number of widgets per state. This made the results of research question 3 not fully complete compared to research question 2. This is not a big issue since the reward was already the worst performing in research question 2 and due to the fact that it took so long it will not be a good reward to use.

10 Conclusion: Improvements to test-effectiveness in TESTAR with Q-learning

Before the research can be concluded the problem should be restated. The problem which TESTAR experienced was that whilst it can test systems thoroughly it takes a very long time to achieve this. This is due to the fact that prior to this research one of the best performing TESTAR action selection algorithms was random. Random immediately causes the problem that eventually every action can be selected but there is no way of steering the algorithm in the right direction. Which is why testing takes a very long time. If an algorithm can be steered in the right direction and actively explores new and unique actions the test-effectiveness and performance can be increased. This is why the study looked into Q-learning for action selection. This study further investigated Q-learning by expanding beyond the previously implemented basic Q-learning algorithm.

Looking back at the results one can see that prior to this study, random action selection outperformed basic Q-learning on the tested Java applications. After implementing new algorithms and creating new rewards, more promising results for Q-learning are shown. The two best performing algorithms: QV-learning with the *Penalized Unvisited and Visited Actions* reward and QV-learning with the *Execution Count* reward show up to a 10% increase in test-effectiveness over random action selection on Rachota.

What these results prove is that Q-learning is a suitable solution for action selection in automated GUI testing. Since improvements were made it also answers the main research question: *How can the test-effectiveness of TESTAR be increased using Q-learning in the action selection step?*. This study showed that by using different rewards and Q-learning algorithms, action selection can indeed be improved. In this study there were only a few Q-learning algorithms and rewards investigated. This means that there is possibly even more room for improvement. Which is clarified in the future work section.

10.1. Future Work

As noted previously there are other reinforcement algorithms which remain unexplored in this study. Since there are other reinforcement learning algorithms, there can be recommended to look further into different reinforcement learning algorithms. This possibly improves test-effectiveness in TESTAR even further. Related work in chapter 4 also show that extending a reinforcement learning algorithm with deep learning can improve action selection. For example, this is shown in the study from Humphreys et al. [14].

Next to testing other reinforcement learning algorithms, it is also possible to tweak the current Q-learning algorithms by tweaking their parameters and introducing new policies. This is recommended since chapter 7 shows how important a good configuration is with Q-learning. During this study a single policy and default parameters were used, but by tweaking, for example the γ value in QV-learning, the algorithm could possibly perform even better. This was not done during this study due to the fact that it would require much more experiments. Which would have taken up much more time.

Another possible improvement is making TESTAR understand forms and application behaviour better. Since TESTAR can currently hardly progress past forms which need specific input or action sequences TESTAR does not reach its maximum potential. During the execution of this study researchers are working on improvements with forms in TESTAR. In the future TESTAR will hopefully automatically detect forms and allow users to specify needed input in these forms. This makes it easier for TESTAR to reach more application states. This can further increase test-effectiveness.

List of Algorithms

List of Tables

List of Figures

List of Examples

Bibliography

[1] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 2–8, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6053-1. doi: 10.1145/3278186.3278187. URL `http://doi.acm.org/10.1145/3278186.3278187`. 22, 80

[2] Emil Alegroth, Robert Feldt, and Lisa Ryrholm. Visual gui testing in practice: challenges, problems and limitations. *Springer Science*, 2014. 9

[3] Francisco Almenar, Anna I. Esparcia-Alcázar, Mirella Martínez, and Urko Rueda. Automated testing of web applications with testar. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering*, pages 218–223, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47106-8. 10

[4] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. In *Discrete Event Dynamic Systems: Theory And Applications*, volume 13, pages 41–77, 2003. 18

[5] A. Bayen, A. Jadbabaie, G. J. Pappas, P. Parrilo, B. Recht, C. Tomlin, and M.Zeilinger. A theoretical analysis of deep q-learning. In *Proceedings of Machine Learning Research*, volume 120, pages 1–4, 2020. 17

[6] Chen ChunLin, Dong DaoYi, Li Han-Xiong, and Tarn Tzyh-Jong. Hybrid mdp based integrated hierarchicalq-learning. *RESEARCH PAPERS*, 54(11):2279–2294, 2011. doi: 10.1007/s11432-011-4332-6. 18, 84

[7] C Degott, N P Borges Jr., and A Zeller. Learning user interface element interactions (issta '19, july 15–19, beijing, china). Beijing, China, 2019. 20, 41, 80

[8] Mark Dourlein. Testar and reinforcement learning, 2021. 24, 25

[9] Juha Eskonen, Julen Kahles, and Joel Reijonen. Automating gui testing with image-based deep reinforcement learning. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 160–167. IEEE, 2020. 24, 80

[10] Anna I. Esparcia-Alcazar, Francisco Almenar, Mirella Martınez, Urko Rueda, and Tanja E.J. Vos. Q-learning strategies for action selection in the testar automated testing tool. volume 6, 2016. 12, 13

[11] Tianxiao Gu, Chun Cao andTianchi Liu, Chengnian Sun, Jing Deng, and Xiaoxing Ma andJian Lü. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution*, 2017. 20, 24

[12] Daniel R. Hackner and Atif M. Memon. Test case generator for guitar. *ICSE'08*, 2008. 10

[13] Luke Harries, Rebekah Storan Clarke, Timothy Chapman, Swamy V. P. L. N. Nallamalli, Levent Özgür, Shuktika Jain, Alex Leung, Steve Lim, Aaron Dietrich, José Miguel Hernández-Lobato, Tom Ellis, Cheng Zhang, and Kamil Ciosek. DRIFT: deep reinforcement learning for functional software testing. *CoRR*, abs/2007.08220, 2020. URL `https://arxiv.org/abs/2007.08220`. 23, 80

[14] Peter C Humphreys, David Raposo, Toby Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Alex Goldin, Adam Santoro, et al. A data-driven approach for learning to control computers. *arXiv preprint arXiv:2202.08137*, 2022. 27, 80, 83

[15] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE access*, 2019. 16

[16] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. Qbe: Qlearning-based exploration of android applications. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, 2018. 20, 22

[17] Yavuz Koroglu, , and Alper Sen. Functional test generation from ui test scenarios using reinforcement learning for android applications. *SOFTWARE TESTING, VERIFICATION AND RELIABILITY*, 2020. 20, 22, 79

[18] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073. IEEE, 2019. 26, 80

[19] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Autoblack-test: Automatic black-box testing of interactive applications. 2012. 25

[20] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Automatic testing of gui-based applications. *Wiley Online Library*, 2014. 9, 14

[21] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: an innovative tool for automated testingof gui-driven software. *Springer Science*, 2013. 9, 10

[22] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 153–164, New York, NY, USA, 2020. Association for Computing Machinery.

ISBN 9781450380089. doi: 10.1145/3395363.3397354. URL `https://doi.org/10.1145/3395363.3397354`. 23

[23] Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. In Leslie Pack Kaelbling, editor, *Machine Learning*, volume 22, pages 283–290, 1996. 16

[24] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Deep reinforcement learning for black-box testing of android apps. *arXiv preprint arXiv:2101.02636*, 2021. 28

[25] Maayan Shvo, Zhiming Hu, Rodrigo Toro Icarte, Iqbal Mohomed, Allan Jepson, and Sheila A McIlraith. Appbuddy: Learning to accomplish tasks in mobile apps via reinforcement learning. *arXiv preprint arXiv:2106.00133*, 2021. 28

[26] Sergio Spanò, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Matta, Alberto Nannarelli, and Marco Re. An efficient hardware implementation of reinforcement learning: The q-learning algorithm. *IEEE Access*, 7: 186340–186351, 2019. doi: 10.1109/ACCESS.2019.2961174. 10, 15

[27] Ting Su, Jue Wang, and Zhendong Su. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 119–130, 2021. 26, 80

[28] Richard S. Sutton. Learning to predict by the methodsof temporal differences. 1988. 16

[29] Hado van Hasselt. Double q-learning. 2010. 18

[30] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. 2006. 19, 80

[31] T Vos, P Kruse, N Condori-Fernández, S Bauersfeld, and J Wegener. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015. doi: 10.4018/IJISMD.2015070103. 11, 86

[32] Tanja E. J. Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. scriptless testing through graphical user interface. *Software Testing, Verification Reliability*, pages 26–32, 2021. 12, 13, 14

[33] Marco A. Wiering and Hado van Hasselt. Two novel on-policy reinforcement learning algorithms based on td()-methods. 17, 38

Appendices

List of Figures in Appendices

A

Coverages on the Moneybird Application

A.0.1 Visited and Unvisited Actions Reward

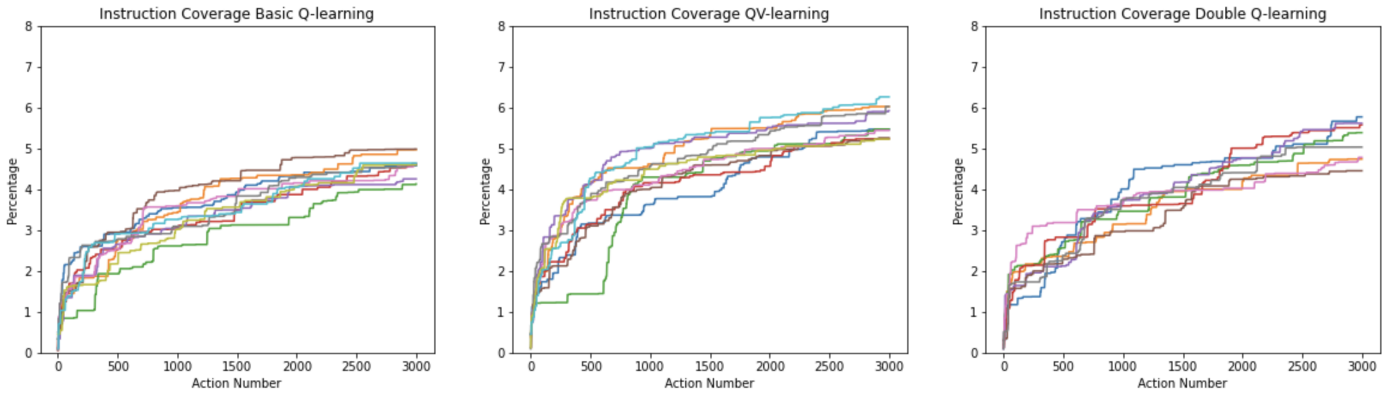

Figure A.1: Visited and Unvisited Actions Reward on Moneybird - All Coverages

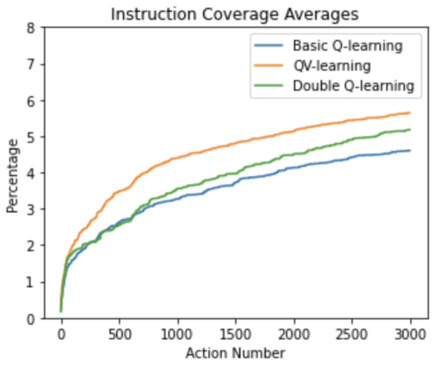

Figure A.2: Visited and Unvisited Actions Reward on Moneybird - Average Coverages

A.0.2 Penalized Visited and Unvisited Actions Reward

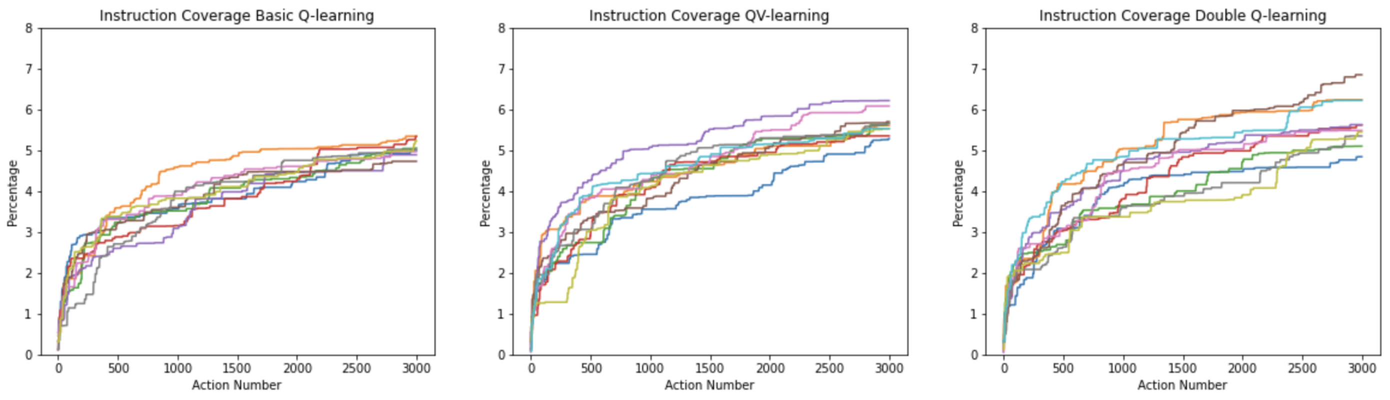

Figure A.3: Penalized Visited and Unvisited Actions Reward on Moneybird - All Coverages

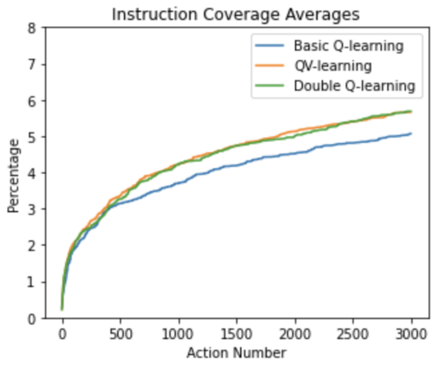

Figure A.4: Penalized Visited and Unvisited Actions Reward on Moneybird - Average Coverages

A.0.3 Image Recognition Reward

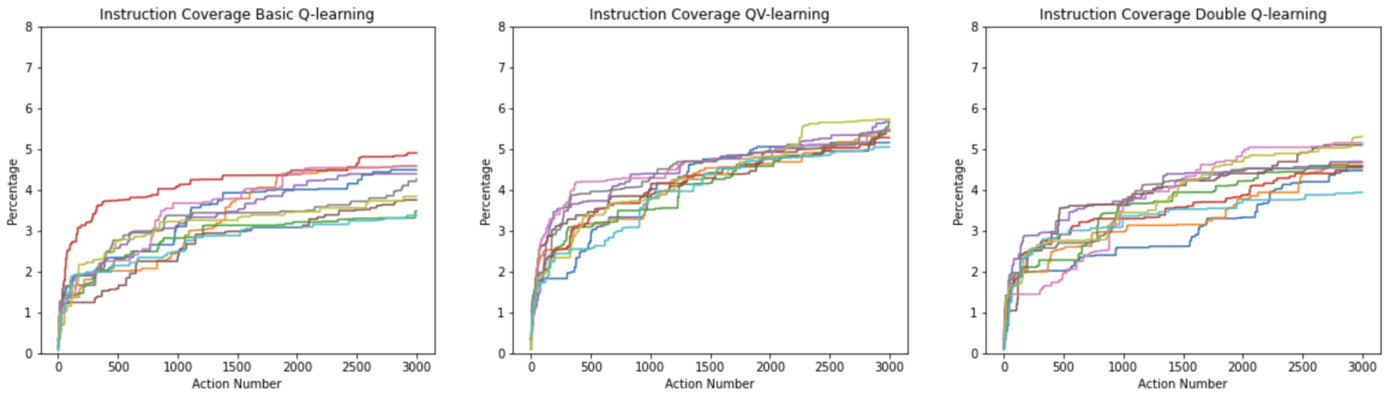

Figure A.5: Image Recognition Reward on Moneybird - All Coverages

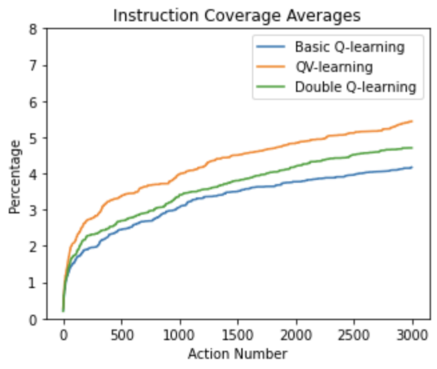

Figure A.6: Image Recognition Reward on Moneybird - Average Coverages

A.0.4 Execution Count Reward

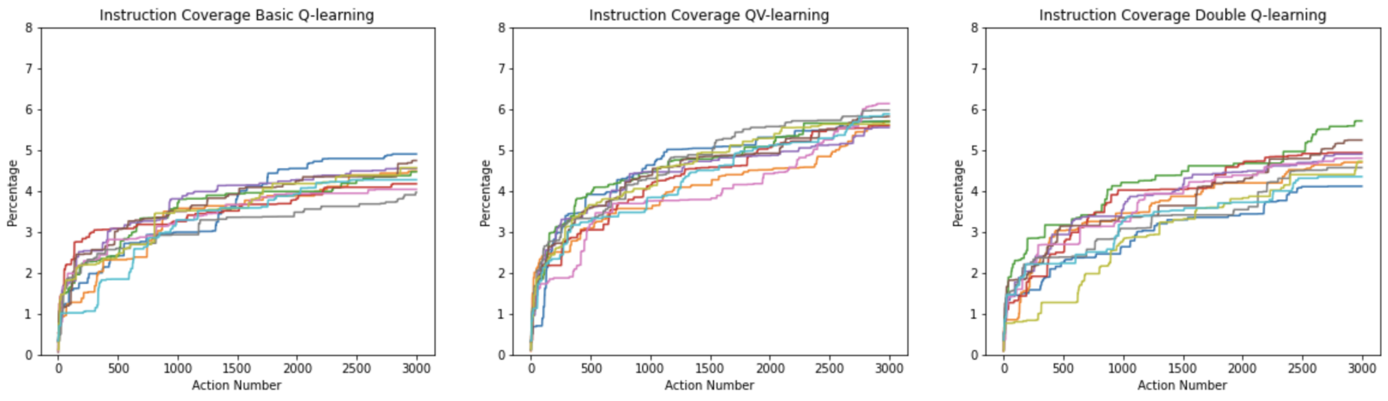

Figure A.7: Execution Count Reward on Moneybird - All Coverages

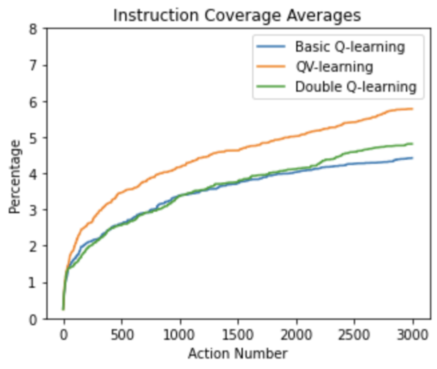

Figure A.8: Execution Count Reward on Moneybird - Average Coverages

A.0.5 Random

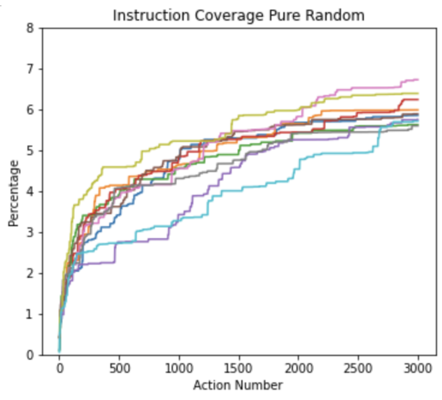

Figure A.9: Pure Random action selection on Moneybird - All Coverages

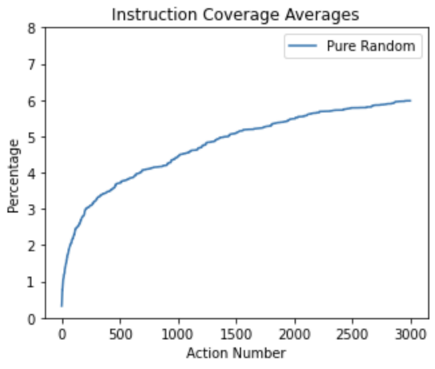

Figure A.10: Pure Random action selection on Moneybird - Average Coverages