A Comprehensive Evaluation of Q-Learning Based Automatic Web GUI Testing

Yujia Fan[1], Siyi Wang[1], Sinan Wang[2], Yepang Liu[2,*], Guoyao Wen[3], and Qi Rong[3]
[1]Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
[2]Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China
[3]Huawei Inc., China
12132331,12010339,wangsn@mail.sustech.edu.cn, liuyp1@sustech.edu.cn, wenguoyao,rongqi1@huawei.com
*corresponding author

Abstract—Recently, reinforcement learning (RL) based automatic Web GUI testing techniques are gaining popularity in both academia and industry as they can enable more intelligent exploration of web applications' states. However, the existing RL-based techniques often incorporate special features, such as DFA-guided state recovery or contextual input data generation, making the effectiveness of RL itself unclear. Moreover, while these techniques mostly employ Q-learning (QL), a model-free RL method, they were evaluated with different experimental settings, which could lead to unfair comparisons. Motivated by the two observations, we propose a generic QL-based automatic Web GUI testing framework, and conduct the first systematic evaluation, which considers four QL specific configurations, on two open-source benchmark web applications and one industrial portal website. Based on the experimental results, we discuss several findings regarding the effectiveness of QL-based automatic GUI testing. We believe that our findings can provide useful guidance to industrial practitioners and shed light on future research on leveraging RL to improve automatic Web GUI testing.

Keywords–Automatic GUI Testing; Web Testing; Reinforcement Learning; Empirical Evaluation

1. Introduction

Modern web technology provides users convenient and unified software applications across different platforms (e.g., desktop and mobile) [1]. Due to the continuous prosperity and wide usage of web applications in our daily life, the quality assurance for these applications (or websites) becomes essential. As network-based software products, web applications can be tested by traditional unit testing for the backend's business logic [2] or by user interactions with the frontend [3]. Alternatively, since web applications are made on top of the Hyper Text Transfer Protocol (HTTP), they can also be tested by the data abstraction (i.e., request and response datagrams) defined by the HTTP specification. However, these testing methods couple with the software's underlining development techniques or require writing test scripts, thus bringing extra maintenance efforts to developers.

The aforementioned test methods heavily rely on the knowledge of the website under test (WUT for short) at the implementation level, such as the source code or the API specifications. This major requirement limits their usages for general web testing. Although different websites vary in their functionalities, their user-end interaction patterns are generally similar to each other. On a typical website, a user will click a button for triggering specific functionality, input text contents in editable areas, control the scroll bar to navigate on the web page, etc. This high-level abstraction of user interaction provides opportunities to design and implement generic web testing tools.

Automatic Web GUI testing explores the WUT without manual interference during the testing process. Its goal is to achieve a sufficient coverage criterion on the WUT with the given time budget. Crawljax [4] is a classic early-stage technique for automatic Web GUI testing. However, its content crawling strategy is less intelligent in discovering promising elements that may trigger new web states. This innate limitation is recently tackled by the *reinforcement learning* (RL) based exploration strategy. For example, Zheng et al. proposed WebExplor [5], the first automatic Web GUI testing technique driven by reinforcement learning. Later, Sherin et al. proposed QExplore [6], which is also driven by RL and is capable of generating context-sensitive input data for exploring deeper web pages. Similar to web applications, desktop and mobile apps can also be explored by RL-based techniques [7], [8]. These existing research works indicate the great potential for RL-driven GUI testing for software quality assurance.

However, these existing techniques mostly utilize special features during the exploration in the WUTs, making the contribution of the RL algorithm unclear. For the aforementioned two techniques, WebExplor employs an on-the-fly deterministic finite automaton (DFA) construction process and utilizes the DFA for recovering from the failing states. QExplore also builds a DFA-like graph, but it never uses the graph for assisting its web exploration procedure. Meanwhile, QExplore can generate contextual input data for form fields, which is not implemented by most other techniques. Even without these special features, the RL itself is rarely evaluated in a comprehensive manner. Both WebExplor and QExplore adopt the QL algorithm, a model-free RL approach, for driving automatic GUI testing, but their algorithmic configurations and experimental settings are misaligned. Also consider WebExplor and QExplore as examples, the former takes a constant discount factor $\gamma = 0.95$ while the latter will adaptively choose this parameter during exploration. Actually, for most QL-based automatic GUI testing techniques, including those for desktop and mobile applications, their settings can vary in state abstractions, action representations, reward functions, and so on (Table I summarizes their differences). Furthermore, these

2767-6684/23/$31.00 ©2023 IEEE
DOI 10.1109/DSA59317.2023.00013

works evaluate their implemented tools on different WUTs with different time budgets, making their comparisons unfair. This paper aims to investigate the real effect of QL-based automatic Web GUI testing techniques, thus to provide practitioners empirical supports about how RL techniques can help improve the quality assurance process and maintain dependable web systems. To motivate our work, we surveyed recent research works about RL-based automatic GUI testing and summarized how they differ in several aspects. Based on the result of our literature review, we propose a generic automatic GUI testing framework for web applications, as well as four configurable components in the framework. We set several options inspired by related works for each component and conduct our comprehensive evaluation on three subjects with 217 configurations. After analyzing the experimental results, we offer practical advice for practitioners and researchers on using RL-based Web GUI testing tools effectively.

This paper makes the following three contributions:

- We survey nine recent RL-based automatic GUI testing techniques, and summarize their differences.
- We propose a generic automatic Web GUI testing framework driven by QL, with four configurable components.
- We conduct the first comprehensive experiment to investigate how QL can improve the performance of automatic Web GUI testing, and discuss our findings for practitioners and future researches.

2. BACKGROUND

2.1 Markov Decision Process

The problem of RL is often described as a decision-making process, in which an intelligent agent takes actions in the environment for earning reward. It is generally formalized as a *Markov decision process* (MDP) [16]. An MDP is defined as a 5-tuple $<\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma>$ where $\mathcal{S}$, the *state space*, is a set of states representing how agent observes from the environment at different time moment. At each state, the agent takes an action from the *action space* $\mathcal{A}$. This action typically results in the change of the environment, causing state transition observed by the agent. This is described by the *transition function* $\mathcal{P}$. After state transition, the environment will provide an immediate reward to the agent based on a *reward function* r, and the agent will adapt its strategy in order to guide future decision-making.

The goal of RL is to find a policy π that guide the agent to choose a sequence of actions $a_i \in \mathcal{A}$ $(i = 0, 1, \ldots, T)$ that maximizes the *return* at the specific time t. This return is denoted G_t and represented as the accumulative reward with a *discount factor* $\gamma \in [0, 1]$:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{T} \gamma^k r_{t+k} \tag{1}$$

The action-value function $Q_\pi(s, a)$ represents the expected return if the agent follows the policy π and executes action a at state s. By some simple mathematical derivations [17], the bellman equation $Q_\pi(s, a)$ can be obtained, which describes the relationship between the values of the current state-action pair and the next state-action pair.

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[G_t | \mathcal{S}_t = s, \mathcal{A}_t = a] \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s', a') \end{aligned} \tag{2}$$

$Q_\pi(s, a)$ tells how good can be an action performed in the certain state, which helps decision-making of the agent. As a result, an RL problem can be solved by training this Q-function to approximate the real action-value function implicitly defined in the environment.

It is worth mentioning that, the action space can either be state-independent or state-dependent (denoted as $\mathcal{A}_s$). For example, a Block Maze robot in the puzzle only has four directions to explore in all time, no matter where its position is [18]. As for GUI testing, the action set depends on the number of interactable elements in the current user interface [5].

2.2 Q-Learning

In most RL problems, the transition function $\mathcal{P}$ is stochastic and implicit. This means, the agent cannot obtain any information about which state will the environment transfer to before the action is executed. In the web testing scenario, without accessing the source code, an agent cannot predict the change of web page before executing GUI actions. Therefore, the agent can only learn sampled data by interacting with the environment and observing the state transitions.

Q-learning [19] is a widely-adopted model-free RL approach in GUI testing [5], [7], [8]. It does not require the state transition probability distribution in advanced. The core data structure of Q-learning is called a Q-table. The rows of Q-table represent all the states in the environment and the columns represent the actions that can be taken in a given state. In Q-table, each entry $Q(s, a)$ (i.e., the Q-value) is continuously updated to approximate the real action-value function. At each time step t, the agent can choose action a_t based on the Q-values of the current state s_t according to certain policy. After executing the chosen action, the agent receives reward r_t and reaches to next state s_{t+1}. And then it can update Q-value $Q(s_t, a_t)$ using the following formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{3}$$

Apparently, the Q-values can tell how good an action is in the given state. However, the agent cannot always choose the action having the greatest Q-value, since doing so may suffer from a local optimal and lose the change of exploring more unseen states. Such behavior policy should be avoid, especially in the early stage of training. A more frequently-used policy is ε-greedy, which is proposed to balance the exploration and exploitation in a transition-independent decision-making scenario [17]. Following this policy, the agent randomly selects an action from the action set with a probability of ε to explore more unseen states, and chooses the "best" action with $1-\varepsilon$ to exploit prior experience:

TABLE I
SUMMARY OF EXISTING RL-BASED AUTOMATIC GUI TESTING TECHNIQUES

Tool	Year	AUT	RL Algorithm	State Abstraction	Action Definition	Reward	Behavior Policy	Hyperparameters
AutoBlackTest [7], [9]	2014	desktop	Q-learning	elements set	simple operation or complex operations	GUI change	ε-greedy	$\varepsilon = 0.8$ $\alpha = 1$ $\gamma = 0.9$
TSETAR [10], [11]	2016	desktop	Q-learning	elements set	valid operations	curiosity	maximum	$\alpha = 1$ $\gamma \in \{0.2, 0.5, 0.95\}$
QBE [12]	2018	Android	Q-learning	exploration extent	operation categories	GUI change or crash detection	ε-greedy	$\varepsilon = 0.005$ $\gamma = 0.9$
ClassicQ [13]	2018	Android	Q-learning	elements set	valid operations	GUI change and curiosity	ε-greedy	self-adaptive ε $\alpha = 1$ $\gamma = 0.9$
QDorid [14]	2019	Android	DQN	elements set	operation categories	GUI change	ε-greedy	self-adaptive ε
Q-testing [8]	2020	Android	Q-learning	LSTM embedding	valid operations	GUI change	ε-greedy	$\gamma = 0.99$
Image-Based [15]	2020	web	A3C	snapshot	pixels of snapshot	new state found	actor network	$\gamma = 0.99$
WebExplor [5]	2021	web	Q-learning	HTML tags	valid operations	curiosity	Gumbel-softmax	$\alpha = 1$ $\gamma = 0.95$
QExplore [6]	2023	web	Q-learning	elements set	valid operations	curiosity	maximum	$\alpha = 1$ self-adaptive γ

$$a_t = \begin{cases} \underset{a'}{\operatorname{argmax}}\, Q(s, a') & \text{with probability } 1-\varepsilon, \\ \text{random action} & \text{with probability } \varepsilon. \end{cases} \tag{4}$$

In Equation 3, $r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ consists of the immediate reward r_t and the predicted value $\max_{a'} Q(s_{t+1}, a')$. The latter term is sampled by policy $\max Q$, which is different from the behavior policy for choosing action. Such *off-policy algorithm* is more convenient as it can reuse previous training samples and tend to have smaller sample complexity [17]. Due to this reason, Q-learning is widely used in practice, especially in the GUI testing scenario.

3. RELATED WORK

In this section, we survey recent research work about automatic GUI testing driven by RL techniques. To better compare each work, we break down the testing process into several key components, and discuss how these work implement them in their approaches. They are summarized in Table I.

3.1 RL Algorithm

The process of automatic GUI testing can be regarded as an MDP problem. For general GUI testing, the application under test (AUT) can be regarded as the environment where agents, i.e., the testing programs, continuously perform actions and receive feedback and reward scores from the environment. This problem formulation not only applies for websites [5], [6], [15] but also for desktop applications [7], [9], [11] and mobile apps [8], [12]–[14]. This problem can be solved by different kinds of RL algorithms. However, given the fact that the testing process is mostly conducted at the GUI front-end while the source code is not always available, model-free methods are preferred. This is why QL, as one of the typical model-free RL methods, is widely adopted by recent works regarding automatic GUI testing [5]–[9], [11]–[13]. Besides, some works adopt deep reinforcement learning methods for their capability of representing complex state and action spaces. For example, Vuong et al. [14] utilized deep Q-network (DQN) to approximate the WUT's behavioral model thus to guide the agent's action selection. Eskonen et al. [15] proposed an image-based approach driven by asynchronous advantage actor-critic (A3C) RL algorithm, in which multiple deep neural networks are responsible for valuing states and detecting available actions.

3.2 State Abstraction

State is a snapshot representation of the environment (i.e. application under test) observed by the testing agent. Ultimately, a state abstraction should preserve sufficient knowledge about the current environment and discard most irrelevant information (e.g., changes of advertisements). However, due to the dynamic nature of GUI applications, state abstraction can easily result in state space explosion thus limit an RL agent's performance [20]. To address this issue, AutoBlackTest [9] represents the state using simplified visible elements in the current GUI, such as the number of items in a listbox or the length of strings in a textarea, while the detail contents are ignored. Similarly, TESTAR [10] collects values of all stable properties of the widgets in the current screen for state abstraction. Vuong et al. continuously proposed ClassicQ [13] and QDorid [14] to interact with the Android app and generate test cases based on an on-the-fly generated model. They define a state by the activity name (the GUI class of Android apps) and the set of GUI events available on the corresponding screen. QBE [12] divides states into five categories based on the number of enabled actions in the current GUI, which makes the trained model universal for different Android apps. Q-Testing [8] novelly utilizes a neural network to measure the similarity between two states for determining whether they are in the same scenario, thus merging similar states. For web applications, the HTML code can precisely represent web states as it encodes structural characteristics (the DOM tree) of the web pages. For this reason, WebExplor [5] analyzes the page's front-end HTML code, converts it to a sequence of tags and calculates the similarity between two sequences

Algorithm 1: Q-learning based automatic web testing

Input: Entry web page p, hyperparameters $\alpha, \gamma, \varepsilon, \tau$, fail state f, Q-table $Q = [q(f, a_f)]$

$s :=$ getStateAbstraction(p)
$A :=$ detectAction(p)
Initialize $q(s, a)$ for all $a \in A$ and add them into Q
repeat
  $a :=$ chooseAction$(s, Q, \varepsilon, \tau)$
  $failed, p' :=$ executeAction(a, p)
  if *failed* **then**
    $s' := f$
  else
    $s' :=$ getStateAbstraction(p')
    if s' is new state **then**
      $A' :=$ detectAction(p')
      Initialize $q(s', a')$ for all $a' \in A'$ and add them into Q
  $r :=$ getReward(s, a, s')
  $q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}_{s'}} q(s', a') - q(s, a)]$
  if $s' = f$ **or** no state transition for a while **then**
    $p \leftarrow$ reset()
    $s \leftarrow$ getStateAbstraction(p)
  else
    $p \leftarrow p'$
    $s \leftarrow s'$
until time budget exhausts

to identify unique states. QExplore [6] represents state as the collection of attribute values of the interactive DOM elements in the web application.

3.3 Action Definition

Most of the aforementioned works directly define interactable GUI elements together with GUI events as the concept of action in MDP [5], [6], [8], [10], [11], [13]. Unlike them, AutoBlackTest [9] defines two types of action: simple action and complex action. The former refers to atomic GUI actions executed on a single widget, such as button click or text input; while the latter represents a workflow of simple actions, such as form fields filling and submitting. QBE [12] and QDorid [14] both adopt a categorized representation of actions. The former defines system events and GUI events and divides them into several subcategories, such as *click*, *swipe*, or *menu*; the latter classifies actionable components on the screen into four groups, like *input* or *navigation*, and sends UI events according to their belonging groups. Eskonen et al. propose an image-based approach [15], which uses GUI snapshot as input. Their work defines actions as the pixels in the snapshot while actions will be chosen based on the aggregation of per-pixel probabilities.

3.4 Reward

In order to help the testing agent effectively exploring the GUI application, testers need to design reasonable reward function that tells how "good" is an action on the given state. A part of techniques leverage on the extent of GUI change (i.e. proportion of new widgets to all of the widgets) to estimate the quality of an applied action [7]–[9], [12]–[14]. Besides GUI change, some techniques set the inverse number of times that an action has been executed as reward to encourage exploration. Their intuition is that rarely executed actions have more potential to trigger unvisited states, thus inspire the agent's curiosity [5], [6], [10], [11], [13]. In order to define platform-specific testing objective, QBE [12] also introduces a new reward function for Android crash detection in its testing framework. Additionally, these basic rewards can be combined for multiple testing objectives. For example, ClassicQ [13] sums up a GUI change term and an execution frequency term (i.e., curiosity) as its receiving reward values.

3.5 Behavior Policy and Hyperparameters

Considering how to balance the choices of exploring new states and exploiting past experience, majority of the QL-based methods employ ε-greedy strategy to select action based on Q-values in current state. The parameter ε can either be a fixed value or be dynamically changed during testing. ClassicQ and QDorid start with $\varepsilon = 1$ to maximize exploration and then gradually decrease it until a final minimum value of 0.5 to follow the experience. WebExplor measures the weights of valid actions in relevant state using Gumbel-Softmax method [21], and converts the weight value to probability to increase the randomness of action selection. As shown in the updating formula of Q-values at Equation 3, there are two more hyperparameters: discount factor γ and learning rate α. In most works, both of them are fixed but different numerical values, as shown in Table I. On the contrary, QExplore calculates γ based on the available actions in DOM states. Specifically, it utilizes an exponentially decreasing discount, such that state with fewer actions will receive greater γ, thus encourages the agent favors future reward rather than immediate reward at the state. Unlike traditional RL problems in which agents are trained in offline mode, most RL-based automatic GUI testing techniques make the training and running intertwined. This requires the agent can learn from the environment as soon as possible, leading to a preference of greater learning rate α.

4. Q-Learning Based Testing Framework

4.1 Overview

As previously discussed, RL-based methods widely applied to drive automatic GUI testing, especially for web applications. However, these RL-based methods mostly add special modules such as restart strategy or input generation algorithms to achieve better functionality exploration capability. On the contrast, the RL framework itself is rarely experimented in isolation, despite the fact that it has many configurable components and most work adopt different algorithmic configurations. Taking the most popular RL algorithm, QL, as an example,

the QL-based automatic GUI testing (QL-testing for short) methods vary in terms of state abstraction, action definition, reward function, hyperparameters, and behavior policy, as shown in Table I. However, the real effects of these variants are rarely evaluated and discussed.

Based on existing work, we summarize a generic workflow of QL-testing, as shown in Algorithm 1. It accepts multiple inputs and will repeat web GUI exploration in a given time budget. The algorithm only illustrates the general workflow of GUI exploration. Though it is flexible to add arbitrary testing objectives into the workflow, for example, recording coverage [22], or detecting presentation failures [23].

To start GUI exploration, an entry web page p, typically the index URL of the WUT, should be passed into the workflow. It also requires four hyperparameters for configuring algorithmic preference of QL, which will be introduced in §4.5. To deal with failed interactions (e.g., 404 status code, web crashing, or entering out-of-domain pages), a fail state f is introduced for error recovery routine. Accordingly, we set a_f as the sole action in f, i.e., $\mathcal{A}_f = \{a_f\}$. Since valid action is generally not available when the testing procedure fails, we implement a_f as a restart operation. In GUI testing, interactable elements are often distinct among different GUI pages. In other words, the available actions are state-dependent in our context. Therefore, we construct the Q-table as a key-value mapping, where key is a pair of state and action (s, a), and value is the Q-value representing the expected return of applying action a in state s. For fail state, the Q-value $q(f, a_f)$ is set to a large negative constant so that the QL-agent will avoid choose actions that are tend to fail once the fail state is encountered.

QL-agent first abstracts the entry page into state s (Line 1) and detects a set of valid actions A for s (Line 2). As it is the first state during exploration, the agent needs to initialize Q-values for all actions in this state in the Q-table (Line 3).

After initialization, the agent repeats the following exploring procedure until the time budget exhausts. Given the current state s and the Q-table Q, the agent will choose an action a from all available actions $\mathcal{A}_s$ stored in the Q-table, and the action will be chosen according to the behavior policy defined by the Q-table and hyperparameters ε or τ (Line 5). Then, it executes action a in the current web page p (Line 6), which is accomplished by sending corresponding GUI events to the web browser. The action execution can result in changes in the environment, typically the transition of web pages. Occasionally, some failures may occur and the agent should handle such fail state for continuing exploration. The function `getStateAbstraction` returns a flag *failed* indicating whether action a was successfully applied, and the resulting web page p'. If the execution is failed, we set the next state s' as the fail state f (Line 8); otherwise, the agent will observe s' by abstracting the new web page p' (Line 10). If the state s' is an unseen state, we should detect all available actions in it and initialize their Q-values (Lines 11-13).

The algorithm then calculates a numeric reward for guiding agent's exploration. This reward is based on the state s, the applied action a and the resulting state s' (Line 14). With this reward, the agent will update the corresponding Q-value $q(s, a)$ according to Equation 3 (Line 15).

To continue exploration, the agent should determine whether the previous action results in a fail state, or the exploration procedure traps in a same state, i.e., no state transition, for a while (Line 16). In such case, we will reset web browser and re-compute state abstraction, while the Q-table itself will remain for preserving knowledge about the WUT to the agent (Line 17-18). The `reset` function can be implemented by restarting from the entry page, the least-visited page, etc. No matter which cases, their entries are already presented in the Q-table, so the initialization process will not be executed at this step. If no failures were detected, the next web page p' and its state s' will be continuously explored in the following iteration (Line 20-21).

In this testing framework, the method of state abstraction `getStateAbstraction`, the behavior policy of choosing next action `chooseAction`, the formula of reward function `getReward` and the four hyperparameters are configurable (we underscore them in the algorithm for highlighting). We will introduce the details in the later subsections.

4.2 State Abstraction

In web testing, the environment can not be directly perceived by the agent due to its highly complex and dynamic content space. Instead, it requires a *state abstraction* module which converts a web page in WUT into a simplified representation that RL agent can memorize and make decisions. When designing such a state abstraction method, one needs to take two aspects into consideration. On the one hand, an abstract representation should contain sufficient information of a certain web page such that the agent can divide different web states in a reasonable way. On the other hand, the granularity of state abstraction can also affect the effectiveness of web testing. For a coarse abstraction method (e.g., URL-based abstraction), two loosely-related web pages may be categorized as the same state, which can result in missing critical interactions or misjudging that whether a state is visited before during the exploration process. On the contrast, a fine-grained abstraction method may preserve too much insignificant information about the web page, such that the number of states to maintain will increase dramatically, causing state explosion problem [24].

We implemented two methods for state abstraction in our framework, adopted by previous works about QL-testing [5], [6]. The next two subsections will introduce them in detail.

4.2.1 Tag Sequence of HTML

The first method is to extract tag sequence from HTML document as a simplified representation. Each web page is associated with an HTML document, where the visible elements to users are those tags enclosed by the `<body>` and `</body>` tags. This abstraction method does not care about detailed differences between two web pages, such as figures or texts, but simply preserve the tag hierarchy that define the web page structure. For instance, the abstract representation of following HTML document,

```
<!DOCTYPE HTML>
<html>
    <head><title>Title</title></head>
    <body>
        <div style="font-size:16px">
            <p>Paragraph</p>
        </div>
        <a onclick="func1">button1</a>
        <a onclick="func2">button2</a>
    </body>
</html>
```

can be reduced to its serialized tag sequence:

```
<div><p></p></div><a></a><a></a>
```

The state abstraction method can determine whether two web states are the same by simply comparing their tag sequences.

4.2.2 Set of Elements

Another method is to represent a state by the set of unique interactable elements detected on the GUI. Such kind of "Elements Set" abstraction can also be employed in testing Android apps [14]. In this method, a web state is represented as: $s = \{e_1, e_2, ..., e_n\}$, where e_i are all interactable elements detected from state s's GUI. In a web page, GUI elements, especially interactable elements (i.e. buttons, input areas), often reflect functionalities on the given state. From the QL algorithm's perspective, as Q-table should maintain state-action pairs, this kind of abstraction can guarantee that each action in the state is always valid.

Usually, an element in an HTML document can be uniquely identified with XPath locator [25], so an element can be stored as its XPath string at implementation level. As such, the abstract representation of the example above will be a set of two XPaths of the two <a> elements.

4.3 Reward Function

Reward is a numerical value that tells the agent how "good" or "bad" an action is in a given state. A good reward function helps calculate immediate reward after a web action is applied to achieve more effective web testing. In this subsection, we will discuss two existing reward functions and their variants.

4.3.1 State Change

From viewpoint of GUI testing, a state changes more mean to meet more new functionalities of WUT, thus should be more preferable to the agent. Such kind of reward prefers actions that can lead to significant change between the current state and the next state. Therefore, we can compare two consecutive states to calculate the extent of change as reward [7]. Obviously, for different state abstraction methods, the definitions of state changes are also different.

When states are abstracted as sequences of tags, Gestalt pattern matching [26], a string-matching algorithm, is used to calculate their similarity $sim(s, s')$. In web testing, the action that leads to more perceivable changes is preferable. Thus we use $1 - sim(s, s')$ as a reward to encourage agent to choose actions triggering much-dissimilar states.

If states are abstracted as their element sets, considering new actions came out in the next state, more new actions mean the previous state changes much more. Therefore, the reward function in this case is given by the formula [13]:

$$r = \frac{|s' \backslash s|}{|s'|} \tag{5}$$

4.3.2 Curiosity

The notion of curiosity has been proposed to address the problem of coarse reward in RL, which encourages the agent to explore diverse states [27]. From perspective of action, exercising diverse action is more likely to visit varying states, so we can guide the agent to select the action with lower executing frequency [10]. At the implementation level, the agent will maintain a table to record the number of times each action has been executed, denoted by $N(a)$, and use its reciprocal value as a reward:

$$r = \frac{1}{N(a)} \tag{6}$$

4.3.3 Hybrid

In addition to single aspect, ClassicQ has its reward taking both state change and curiosity into consideration [13]. Merely using state change will be facing a dilemma of jumping back and forth between two greatly vary states, while applying curiosity may waste time on uninterested actions. Therefore, they can be combined to overcome their own limitations. A simple combination is to take their summation:

$$r = \frac{|s' \backslash s|}{|s'|} + \frac{1}{N(a)} \tag{7}$$

Furthermore, we can define other variants base on these simple reward functions. For example, Equation 8 shows a piecewise form of reward function. According to the definition, if the agent discovers a new state, which is not visited before during web testing, we can give it a large reward value. Otherwise, we simply use the curiosity reward function for encouraging fewer executed actions.

$$r = \begin{cases} r_{new\_state} & s' \text{ is a new state,} \\ \frac{1}{N(a)} & \text{otherwise.} \end{cases} \tag{8}$$

4.4 Behavior Policy

Behavior policy concerns how to choose an action in a given state. It often faces a problem of balancing "exploiting" past experience or "exploring" unvisited states. The former means agent will choose action under the guidance of Q-table, while the latter means it randomly chooses from valid actions in the current state. As Q-values can tell an agent which actions to be preferable, totally rely on such guidance can result in sub-optimal behaviors and fail to accumulate maximal reward [17]. Therefore, exploration becomes essential. While exploitation directly utilizes information from Q-table, exploration is usually realized by introducing some degree of randomness to extend the agent's search space.

Our testing framework implements two types of behavior policies adopted by previous works. Both of them take advantages of Q-table and randomness to balance the choices between exploitation and exploration, as illustrated below.

4.4.1 ε-greedy

An effective and popular behavior policy in RL, especially for QL, is ε-greedy, where ε is the probability of randomly choosing actions, as shown in Equation 4. Also note that the extreme values of ε enable some special behaviors to the agent.

For example, if $\varepsilon = 1$, the agent behaves similarly to a random agent (or a Monkey tester [28]). If $\varepsilon = 0$, the agent will always choose the action with highest Q-value, which can lead to falling into a local optimum.

4.4.2 Softmax

One major limitation of ε-greedy is that, when it takes exploration decision, it chooses equally among available actions. This means the probability of choosing the worst action is the same as choosing the next-to-best action. On the other hand, it will also exploit the same action if its Q-value dominates the other alternatives. A solution to these problems is to assign weighted probabilities to each actions, with respect to their Q-values in the given state. This strategy takes the place of both uniform sampling and greedy choice, which implicitly addresses the exploration-exploitation dilemma. Usually, their weights are calculated by the following Softmax function [17]:

$$p(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{9}$$

It presents how to calculate the probability of an action a in the state s, where τ is a positive value named *temperature*. With lower temperature, the difference in selection probabilities for actions having different Q-values will be greater. In the extreme situation as $\tau \to 0$, Softmax action selection becomes the same as greedy strategy. On the other hand, with higher temperature, the selection probabilities for actions with different Q-values will be more similar. As $\tau \to \infty$, the selection probabilities for all actions will be nearly equal, resulting in a strategy that closely resembles random exploration.

4.5 Hyperparameters

Besides ε and τ employed in behavior policies, there are two more adjustable hyperparameters, α and γ, in QL.

The *learning rate* α controls how fast the agent modifies its estimates [29]. The Q-values in Q-table represent expected returns performing corresponding actions in a given state. Hence they can guide agent to make decisions, and they are updated based on the environment's feedback once an action is performed. After action execution, the agent receives an immediate reward from the environment, and calculates an expected returns for updating Q-value following Equation 3. The extent of update can be controlled by α. When $\alpha = 0$, the Q-value never changes, which means the agent can not learn any knowledge from experience. In web testing, since the training and testing phases are intertwined, most works tend to set α close or even equal to 1, as shown in Table I. Our framework leaves learning rate configurable and testers can control the agent's behavior by turning its value.

When calculating an expected return, the maximum Q-value of next state is multiplied by a *discount factor* γ ranging in $[0, 1]$ (Equation 3). The discount factor determines the importance of future rewards. When it is close to 0, the agent prefers immediate rewards than future rewards, and vice versa. By adjusting this hyperparameter, we can influence the behavior of the QL-agent, and control how much the agent values future rewards over immediate rewards. It is worth noting that, QExplore [6] adopts a self-adaptive $gamma$ based on the numbers of actions in a state.

5. Experimental Setup

5.1 Implementation

To support industrial testing requirements, we design a set of composable APIs such that human testers can specify algorithmic configuration in a low-code fashion. We implement our testing framework on top of Selenium-Java [30] in roughly 17.5k lines of Java code. Currently, our framework supports multiple UI event types [31], such as click or text input on supported DOM elements. The framework will run the main testing loop in a pre-defined time duration, except two cases: the agent enters an out-of-domain URL, or no actions can be found on the current page. In either cases, the agent will generate a fail case (i.e., line 16 in Algorithm 1) and reset the test process. The agent will also be restarted if it traps in the same state for 50 iterations.

We run all the experiments in multiple identical desktop computers, each of them equips a Core i7-8700 processor and 32GB RAM. All the devices are connected to the same network switch for the sake of a consisting network environment.

5.2 Research Questions

To motivate our research and boost the experiments, we proposed the following research questions:

- **RQ1 (Performance of Q-Learning)**: How is the performance of QL-testing, compared to a random baseline?
- **RQ2 (Sensitivity of Settings)**: How do different experiment settings affect the performance of QL-testing?
- **RQ3 (Subjects Deviation)**: How does QL-testing perform on different subject websites?
- **RQ4 (Testing Time)**: How is the QL-testing agent's performance over time?

5.3 Subject Websites

We conduct a comprehensive evaluation of QL-testing using three subjects: Splittypie [32], PetClinic [33], and a large commercial portal website. Splittypie and PetClinic are popular open-source web applications used to evaluate testing tools [5], [34]. The portal website is chosen as a complex environment that is representative of many real-world business websites.

5.4 Evaluation Metrics

To evaluate the effectiveness of the QL-agent with different configurations, we utilized three metrics:

- For the two open-source websites, we use Istanbul to instrument the backend of the websites and retrieve the coverage information during the testing process
- For each subject, we record the number of interacted and detected elements during testing process. Since obtaining a full set of elements of each website is not feasible, we unify detected elements across all testing processes as the denominator for element coverage rate, divided by the number of interacted elements for each test process.
- As the primary goal of web testing is to identify and report errors, we counted the number of unique errors from the console logs in each test process.

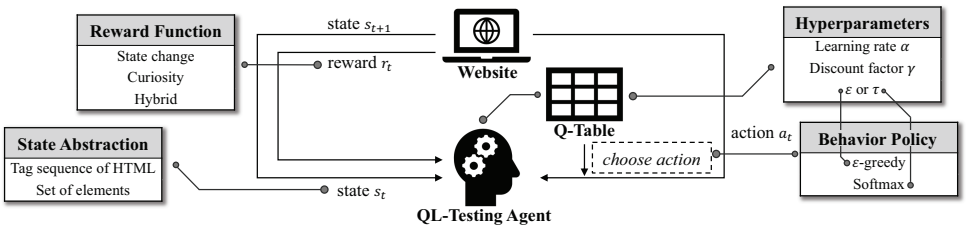


Figure 1. Overview of our experimental configurations

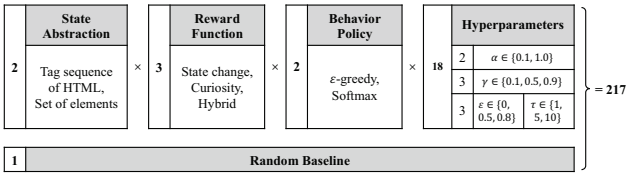


Figure 2. The 217 configurations of components' instances

5.5 Configurations

Figure 1 shows our framework with four components and Figure 2 shows the 216 configurations in our experiment, plus an additional random baseline method.

To compare the effect of different state abstraction methods, we set two options: one is tag sequence extracted from HTML document [5], another is set of unique interactable elements detected at a given DOM tree [6]. We call them Tag Sequence of HTML (TS) and Set of Element (SE).

Two earlier works adopted GUI change scope [7] and action execution times [10] to construct the immediate reward. The former prefers the actions that bring significant change between consecutive states. The idea of the later is based on notion of curiosity, which is also employed by followup tools [5], [6], [11]. Additionally, ClassicQ [13] takes both aspects into consideration. Therefore, we set three reward functions, namely State Change (SC, Equation 5), Curiosity (CR, Equation 6) and Hybrid equation (HB, Equation 7).

Considering how to leverage Q-values, most related works [8], [9], [12]–[14] leverage ε-greedy as their behavior policy, which balances exploration and exploitation by adjusting value of ε. For example, AutoBlcakTest [9] sets $\varepsilon = 0.8$. Unlike other tools, TSETAR [11] and QExplore [6] directly select the action with the maximum Q-value, which is equivalent to setting $\varepsilon = 0$. We also include an option $\varepsilon = 0.5$, such that exploration and exploitation strategy can be taken with equal probability. This results in three ε-greedy behavior policies, corresponding to ε values of 0 (G0), 0.5 (G5), and 0.8 (G8).

Besides, we also use traditional Softmax function to further study the effect of temperature τ. Similar to WebExplor [5], an optional value is $\tau = 1$. As we set 10 as the initial Q-value, we also allow $\tau = 10$. The third option of τ is 5 as a compromise value between 1 and 10. This leads to three Softmax-based behavior policies, which correspond to temperature 1 (S1), 5 (S5), and 10 (S10).

Most works [5], [6], [9], [11], [13] let learning rate $\alpha = 1$ so that QL-agent can learn from environment more quickly. In curious about what will happen if α is smaller, we set two options for α: 0.1 (lr0) and 1.0 (lr1). Similarly, based on the options adopted by previous works [9], [11]–[13], we set three values of discount factor γ: 0.1 (df1), 0.5 (df5), 0.9 (df9). Meanwhile, we set a random baseline, in which the agent randomly chooses actions on the current web page.

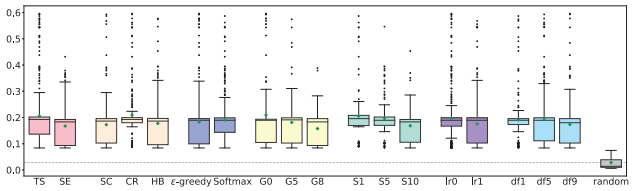
6. Results and Discussion

Each of the 216 configurations were tested on the three subject websites, and each testing process is repeated three times to ensure the results' robustness, which leads to 1,944 data points. For the random baseline, we conducted ten independent test runs on each website, resulting in 30 data points. The duration for each test run was set to 30 minutes. Additionally, for the portal website, due to its complex page structure and contents, we selected two configurations and extended the time budget to 10 hours. Meanwhile, we also ran the random baseline for 10 hours, resulting in three sets of 10-hour data. In total, our experiments required $(216 \times 3 + 10) \times 3 \times 0.5 + 3 \times 10 = 1,017$ hours of workload. In all the test runs, experiment data was recorded at a regular interval of 30 seconds. Upon analyzing the results, we bold our findings that aim to provide actionable advice to developers.

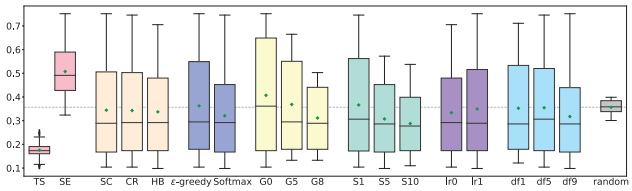
6.1 RQ1: Performance of Q-Learning

We draw the distributions of element coverage in our experiments, as shown in Figure 3. For example, the box plot for **TS** in Figure 3(a) illustrates the distribution of element coverage rate for all the 108 configurations (324 data points) used to test Splittypie with Tag Sequence of HTML state abstraction. Likewise, the box plot for **random** in Figure 3(a) is that for ten tests with random baseline on Splittypie.

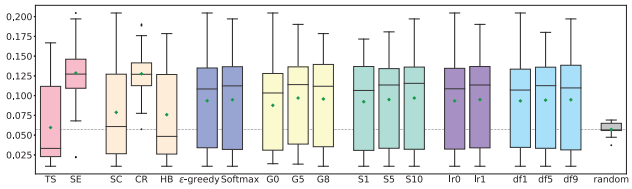
Figure 3 indicates that **the performance of the QL-testing has a higher upper limit but a wider variance than the random baseline**. While QL consistently outperforms the random baseline on Splittypie, this observation does not necessarily hold on PetClinic and portal website. However, the best performance of QL-agent on the latter two websites is much better than random baseline. For example, on PetClinic, QL-based method can achieve a maximal element coverage of 75.14% with configuration (SE, SC, G0, lr1, df9), while random selection can only obtain a mean element coverage of 35.12% within a narrow range. Our results suggest that



(a) Splittypie



(b) PetClinic



(c) portal website

Figure 3. Distributions of element coverage for each component. On Splittypie, the maximal element coverage is 59.62% with configuration (TS, CR, G0, lr0, df9) and maximal coverage by random is 7.45%. On PetClinic, the maximal element coverage is 75.14% with configuration (SE, SC, G0, lr1, df9) and maximal coverage by random is 39.88%. On portal website, the maximal element coverage is 20.45% with configuration (SE, SC, G0, lr0, df1) and maximal coverage by random is 6.91%.

the effectiveness of QL-based testing is contingent on the employed configuration and how well it fits the specific WUT. We also observe similar results on the other metric.

> ***Answer 1***: QL-testing performance fluctuates greatly across different configurations, but can be significantly improved over a random baseline with proper configurations.

6.2 RQ2: Sensitivity of Settings

For pairwise options of the same component, we determined which one yields better performance, by comparing the distributions of their testing metrics (e.g., element coverage shown in Figure 3). To this end, we conducted a set of one-sided Mann-Whitney U-test [35]. Specifically, if p-value < 0.05, we reject the null hypothesis and consider one distribution to be stochastically greater than another. Table II presents the results of the statistical tests, the highlighted options indicate them significantly outperform the others in pairwise comparison. For example, TS can achieve better branch coverage than SE state abstraction on Splittypie, and so on.

Table II reveals that all subjects highlighted options in the **state abstraction** component for all metrics, indicating that it **has a great impact on the performance of QL-testing**. Specifically, TS is superior than SE for all the three metrics

TABLE II
RESULTS OF ONE-SIDED MANN-WHITNEY U-TEST

Branch Coverage				Element Coverage						Reported Errors					
Splittypie		PetClinic		Splittypie		PetClinic		portal		Splittypie		PetClinic		portal	
TS	SE	TS	SE	TS	SE	TS	SE	TS	SE	TS	SE	TS	SE	TS	SE
SC	CR	SC	CR	SC	CR	SC	CR	SC	CR	SC	CR	SC	CR	SC	CR
SC	HB	SC	HB	SC	HB	SC	HB	SC	HB	SC	HB	SC	HB	SC	HB
CR	HB	CR	HB	CR	HB	CR	HB	CR	HB	CR	HB	CR	HB	CR	HB
G0	G5	G0	G5	G0	G5	G0	G5	G0	G5	G0	G5	G0	G5	G0	G5
G0	G8	G0	G8	G0	G8	G0	G8	G0	G8	G0	G8	G0	G8	G0	G8
G0	S1	G0	S1	G0	S1	G0	S1	G0	S1	G0	S1	G0	S1	G0	S1
G0	S5	G0	S5	G0	S5	G0	S5	G0	S5	G0	S5	G0	S5	G0	S5
G0	S10	G0	S10	G0	S10	G0	S10	G0	S10	G0	S10	G0	S10	G0	S10
G5	G8	G5	G8	G5	G8	G5	G8	G5	G8	G5	G8	G5	G8	G5	G8
G5	S1	G5	S1	G5	S1	G5	S1	G5	S1	G5	S1	G5	S1	G5	S1
G5	S5	G5	S5	G5	S5	G5	S5	G5	S5	G5	S5	G5	S5	G5	S5
G5	S10	G5	S10	G5	S10	G5	S10	G5	S10	G5	S10	G5	S10	G5	S10
G8	S1	G8	S1	G8	S1	G8	S1	G8	S1	G8	S1	G8	S1	G8	S1
G8	S5	G8	S5	G8	S5	G8	S5	G8	S5	G8	S5	G8	S5	G8	S5
G8	S10	G8	S10	G8	S10	G8	S10	G8	S10	G8	S10	G8	S10	G8	S10
S1	S5	S1	S5	S1	S5	S1	S5	S1	S5	S1	S5	S1	S5	S1	S5
S1	S10	S1	S10	S1	S10	S1	S10	S1	S10	S1	S10	S1	S10	S1	S10
S5	S10	S5	S10	S5	S10	S5	S10	S5	S10	S5	S10	S5	S10	S5	S10
lr0	lr1	lr0	lr1	lr0	lr1	lr0	lr1	lr0	lr1	lr0	lr1	lr0	lr1	lr0	lr1
df1	df5	df1	df5	df1	df5	df1	df5	df1	df5	df1	df5	df1	df5	df1	df5
df1	df9	df1	df9	df1	df9	df1	df9	df1	df9	df1	df9	df1	df9	df1	df9
df5	df9	df5	df9	df5	df9	df5	df9	df5	df9	df5	df9	df5	df9	df5	df9

on Splittypie, while the latter performs better on the other two WUTs. The reward function plays a significant role in QL-testing performance **on Splittypie and portal website**, where **CR outperforms the other reward functions**. Meanwhile, the behavior policy affects QL-testing **on Splittypie and PetClinic** more, with **lower randomness (either lower ε or τ) parameters leading positive impact**.

Additionally, a learning rate 0.1 can be more effective than that of 1.0. Having lower discount factor also tend to achieve better performance metrics than higher ones.

> ***Answer 2***: State abstraction has the greatest impact on QL-testing performance, followed by behavior policy and reward functions. Priority should be given to state abstraction when designing QL-based Web GUI testing techniques.

6.3 RQ3: Subjects Deviation

In RQ2, we have observed that the performance of the same settings of QL-testing on different WUTs varied. To answer RQ3, we further analyze the relationship between configuration combinations and website characteristics. Specifically, for a performance metric of a website, we extract configurations that rank in the top 5% and bottom 5%, determine the proportion of each option, and present the element coverage results in Table III. For example, among the configurations corresponding to the top 5% element coverage, 48% used CR as reward function, while 27% used SC and 24% used HB.

As shown in row State Abstraction of Table III, TS state abstraction performs well on Splittypie while is disastrous for PetClinic and portal websites, which leads us to conduct a deeper analysis how the mode of state abstraction impact on QL-testing. By scrutinizing the process of the QL agent with TS testing on PetClinic, we observe a phenomenon whereby the TS remains unchanged when triggering the pop-up menu. As a result, the QL agent misses new interactive elements that are crucial to exploration and fails to expand the Q-table.

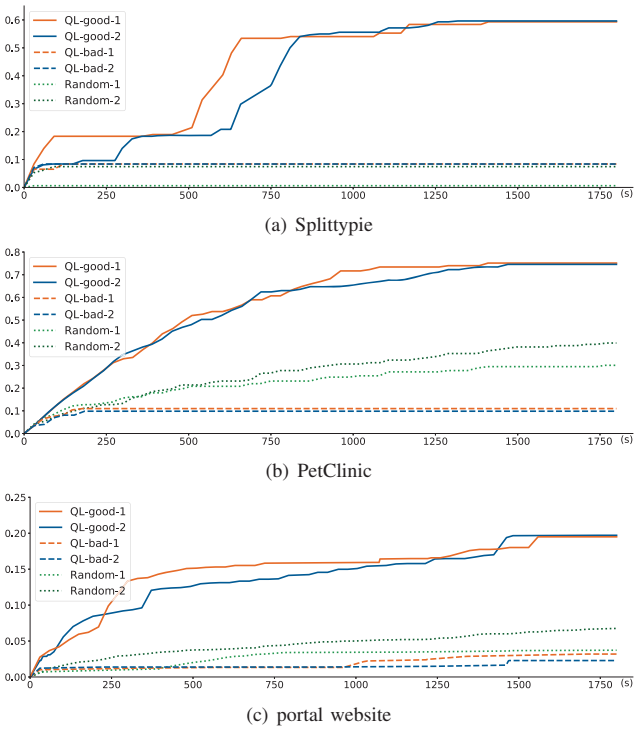
TABLE III
DISTRIBUTIONS OF OPTIONS IN TOP 5% OR BOTTOM 5%

Component	Option	Branch Coverage				Element Coverage						Reported Errors					
		Top 5%		Bottom 5%		Top 5%			Bottom 5%			Top 5%			Bottom 5%		
		Splittypie	PetClinic	Splittypie	PetClinic	Splittypie	PetClinic	portal	Splittypie	PetClinic	portal	Splittypie	PetClinic	portal	Splittypie	PetClinic	portal
State Abstraction	TS	**86%**	0%	0%	100%	**94%**	0%	3%	11%	100%	100%	15%	0%	0%	26%	99%	100%
	SE	14%	**100%**	100%	0%	6%	**100%**	**97%**	89%	0%	0%	**85%**	**100%**	**100%**	74%	1%	0%
Reward Function	SC	26%	34%	41%	47%	27%	**44%**	**41%**	38%	41%	45%	18%	**40%**	**47%**	47%	48%	55%
	CR	**43%**	**41%**	9%	28%	**48%**	31%	31%	9%	26%	0%	**70%**	38%	25%	3%	28%	0%
	HB	31%	25%	50%	25%	24%	25%	28%	53%	32%	55%	12%	23%	28%	49%	25%	45%
Behavior Policy	G0	**37%**	**66%**	15%	41%	**33%**	**75%**	9%	26%	35%	9%	15%	**38%**	16%	16%	41%	21%
	G5	3%	6%	26%	6%	15%	3%	13%	30%	3%	6%	12%	19%	19%	25%	7%	14%
	G8	6%	0%	27%	0%	6%	0%	16%	30%	3%	15%	**21%**	4%	**22%**	29%	1%	16%
	S1	20%	19%	12%	16%	27%	22%	13%	0%	12%	27%	**21%**	27%	16%	10%	22%	12%
	S5	26%	9%	6%	13%	15%	0%	19%	6%	24%	15%	18%	8%	16%	3%	13%	14%
	S10	9%	0%	14%	25%	3%	0%	**31%**	9%	24%	27%	12%	4%	**13%**	16%	16%	24%
Learning Rate	lr0	**74%**	16%	40%	47%	**70%**	22%	**56%**	38%	53%	61%	**52%**	23%	41%	30%	45%	48%
	lr1	26%	**84%**	60%	53%	30%	**78%**	44%	62%	47%	39%	48%	**77%**	**59%**	70%	55%	52%
Discount Factor	df1	29%	22%	29%	16%	**36%**	28%	25%	19%	21%	33%	30%	19%	19%	21%	29%	36%
	df5	**43%**	**44%**	27%	41%	**36%**	**50%**	34%	28%	32%	42%	**39%**	**44%**	25%	32%	28%	33%
	df9	29%	34%	44%	44%	27%	22%	**41%**	53%	47%	24%	30%	38%	**56%**	47%	43%	31%

Different from PetClinic, TS is less effective on the portal website on account of the length of tag sequence, which is too large to store and makes a great influence on computing the change rate between two states. As a result, the immediate reward calculation is inefficient when using the SC or HB reward function, which is related to the mode of state abstraction. In conclusion, **improper state abstraction limits the effectiveness of the QL agent with an incomplete Q-table and also reduces efficiency by consuming considerable computing and storage resources**.

Regarding the behavior policy, it presents a trend that **increased exploitation can lead to excellent performance on two small-scale open-source websites**. Synthesize all metrics, the top two behavior policies are G0 and S1 on Splittypie and PetClinic, which turns out that lower randomness (either lower ε or τ) helps QL-agent explore WUTs in a guided manner. We select test processes with great performance on each subject and check the clicks of actions in Q-table. On Splittypie and PetClinic, more than 95% and 89% of actions in the Q-table have been executed, with over 50% and 47% executed two or more times. In contrast, the click rate on the portal is only 10%, with only 0.5% of actions clicked two or more times. The experimental results confirm that on the two small-scale websites, Q-table is able to gather information and guide the agent effectively, suggesting that behavior policy can be inclined to follow the Q-table's guidance. However, for the large-scale websites, Q-table has not acquired sufficient knowledge, indicating that increased randomness is more conducive to exploration.

Additionally, **combining with certain reward functions, the behavior policy G0 carries a potential risk of unwanted behaviors**. During the testing process, it was observed that an agent configured with a combination of G0 and HB(or SC) became stuck in a loop of navigation between two webpages. On the one hand, HB and SC place high value on state changes, resulting in a high reward for actions that navigate between two distinct webpages. On the other hand, the QL-agent with G0 only selects the action with the highest Q-value, which further increases the reward accumulation.



(a) Splittypie

(b) PetClinic

(c) portal website

Figure 4. Element coverage of QL-testing and random baseline over time

Answer 3: There is no one-size-fits-all option for effective QL-testing on all websites. Therefore, configurations that fit the features of the WUT deserve to be carefully chosen in practice.

6.4 RQ4: Testing Time

Figure 4 shows the growing curves of element coverage achieved by different testing agents. For example, Figure 4(a) displays the change of element coverage on Splittypie. Here we select two QL configurations that exhibit outstanding performance (QL-good), two that show poor performance (QL-bad) and two different results run by the random baseline. **For two open-source websites, the exploration reaches**

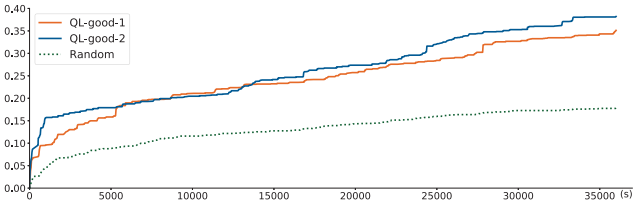


Figure 5. Element coverage on portal website in 10 hours

saturation within 30 minutes, as indicates by the plateauing of the curves. Compared to the QL-bad and random baseline, the coverage rate of QL-goods increase more rapidly and reach a higher value eventually. However, unlike the previous works [5], [6], all QL-testing agents present inefficiency in further exploring the WUTs. This finding suggests that **the exploration capability of QL-testing is still limited**, and auxiliary modules (e.g., DFA in WebExplor, or contextual input generation in QExplore) are necessary to further improve the effectiveness.

For the portal website, the element coverage continues to increase within 30 minutes, indicating that it has not yet reached saturation. To further test it, we used the two QL configurations that perform well and run them in a time budget of 10 hours. We also run the random for the same duration. The resulting element coverage curve is shown in Figure 5. Even with a longer time budget, the element coverage of QL-testing agent is still increasing. It indicates that extending the testing time can indeed improve the testing sufficiency on large-scale websites. However, it also shows the limitation of such single-threaded testing agents, no matter what their underlining driven algorithms are.

Answer 4: QL-testing can quickly reach saturation when exploring small-sized WUTs. Adding time budget can indeed improve test adequacy on large websites.

7. Threats to Validity

There are three potential threats to the validity of our study. Firstly, although we attempt to ensure consistent network conditions during web testing, network issues and the inherent randomness of Q-learning may still have influenced the results. While we conduct three repeated experiments to reduce uncertainties, we cannot totally eliminate them.

Secondly, there is a time lag between the moment of state abstraction and that of action selection during testing. It may lead to the selected actions being unavailable due to incomplete web rendering. Although we attempted to detect whether the web page is fully loaded and avoid interactions during dynamic DOM changes, these tasks are very challenging. These limitations may affect the validity of our results.

Additionally, we only used three WUTs. Even though the two open-source websites have being widely used in related research works, and the portal website is representative as a complex real-world web application. However, it may still limit the generalizability of our findings.

8. Conclusion and Future Work

In this paper, we surveyed nine recent RL-based automatic GUI testing techniques and systematically evaluated QL-based automatic Web GUI testing on top of a generic testing framework. According to the experiment results on two open-source benchmark websites and one industrial portal web application, we discussed several findings regarding the effectiveness of QL-based web testing. In the future, we plan to extend our testing framework to support more powerful RL algorithms, such as deep-RL [14] or multi-agent RL [36]. We also plan to evaluate other variants of RL configuration, such as state abstraction by near-duplicate detection [37] or self-adaptive hyper-parameters [6].

Acknowledgment

The authors thank Chuan Jiang and Yirui He for their contributions of developing the system prototype used in this study. The authors also thank the DSA 2023 reviewers for their valuable comments. The work is partially supported by Huawei and the Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562).

References

[1] S. Yu, C. Fang, Y. Yun, and Y. Feng, "Layout and image recognition driving cross-platform automated mobile testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1561–1571.

[2] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 3–12.

[3] M. Leotta, M. Biagiola, F. Ricca, M. Ceccato, and P. Tonella, "A family of experiments to assess the impact of page object pattern in web test suite development," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 263–273.

[4] A. Mesbah, E. Bozdag, and A. Van Deursen, "Crawling ajax by inferring user interface state changes," in *2008 eighth international conference on web engineering*. IEEE, 2008, pp. 122–134.

[5] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 423–435.

[6] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, "Qexplore: An exploration strategy for dynamic web applications using guided search," *Journal of Systems and Software*, vol. 195, p. 111512, 2023.

[7] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Autoblacktest: a tool for automatic black-box testing," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 1013–1015.

[8] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[9] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic testing of gui-based applications," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 341–366, 2014.

[10] S. Bauersfeld and T. Vos, "A reinforcement learning approach to automated gui robustness testing," in *Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012)*, 2012, pp. 7–12.

[11] A. I. Esparcia-Alcázar, F. Almenar, M. Martínez, U. Rueda, and T. Vos, "Q-learning strategies for action selection in the testar automated testing tool," *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, pp. 130–137, 2016.

[12] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 105–115.

[13] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.

[14] ——, "Semantic analysis for deep q-network in android gui testing." in *SEKE*, 2019, pp. 123–170.

[15] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 160–167.

[16] M. Morales, *Grokking deep reinforcement learning*. Manning Publications, 2020.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.

[19] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[20] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[21] E. Jang, S. Gu, and B. Poole, "Categorical reparametrization with gumble-softmax," in *International Conference on Learning Representations (ICLR 2017)*. OpenReview. net, 2017.

[22] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 571–580.

[23] S. Mahajan, B. Li, P. Behnamghader, and W. G. Halfond, "Using visual symptoms for debugging presentation failures in web applications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 191–201.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[25] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Robula+: An algorithm for generating robust xpath locators for web testing," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 177–204, 2016.

[26] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.

[27] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *International conference on machine learning*. PMLR, 2017, pp. 2778–2787.

[28] "Monkey," 2018. [Online]. Available: https://developer.android.com/

[29] E. Even-Dar, Y. Mansour, and P. Bartlett, "Learning rates for q-learning." *Journal of machine learning Research*, vol. 5, no. 1, 2003.

[30] SeleniumHQ. selenium: A browser automation framework and ecosystem. [Online]. Available: https://github.com/SeleniumHQ/selenium/

[31] W3c working draft: Ui events. [Online]. Available: https://www.w3.org/TR/uievents/

[32] Splittypie: easy expense splitting. [Online]. Available: https://github.com/tsubik/splittypie/

[33] Angular 11 version of the spring petclinic sample application. [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-angular

[34] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 142–153.

[35] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[36] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *Handbook of reinforcement learning and control*, pp. 321–384, 2021.

[37] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 186–197.