electronics

MDPI

Article

A Reinforcement Learning Approach to Guide Web Crawler to Explore Web Applications for Improving Code Coverage

Chien-Hung Liu [1,*], Shingchern D. You [1] and Ying-Chieh Chiu [2]

[1] Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei 106344, Taiwan; scyou@ntut.edu.tw
[2] Phison Electronics Corp., No. 1, Qun Yi Rd., Jhunan, Miaoli County 350, Taiwan; andrew_chiu@phison.com
* Correspondence: cliu@ntut.edu.tw

Abstract: Web crawlers are widely used to automatically explore and test web applications. However, navigating the pages of a web application can be difficult due to dynamic page generation. In particular, the inputs for the web form fields can affect the resulting pages and subsequent navigation. Therefore, choosing the inputs and the order of clicks on a web page is essential for an effective web crawler to achieve high code coverage. This paper proposes a set of actions to quickly fill in web form fields and uses reinforcement learning algorithms to train a convolutional neural network (CNN). The trained agent, named iRobot, can autonomously select actions to guide the web crawler to maximize code coverage. We experimentally compared different reinforcement learning algorithms, neural networks, and actions. The results show that our CNN network with the proposed actions performs better than other neural networks in terms of branch coverage using the Deep Q-learning (DQN) or proximal policy optimization (PPO) algorithm. Furthermore, compared to previous studies, iRobot can increase branch coverage by about 1.7% while reducing training time to 12.54%.

Keywords: web crawler; reinforcement learning; software testing; code coverage

Citation: Liu, C.-H.; You, S.D.; Chiu, Y.-C. A Reinforcement Learning Approach to Guide Web Crawler to Explore Web Applications for Improving Code Coverage. *Electronics* **2024**, *13*, 427. https://doi.org/10.3390/electronics13020427

Academic Editor: Mohamed Wiem Mkaouer

Received: 2 November 2023
Revised: 17 January 2024
Accepted: 17 January 2024
Published: 19 January 2024

1. Introduction

Crawl-based approaches are commonly used for automatically crawling and testing web applications. It has been applied to various types of web application testing, such as regression testing, compatibility testing, and security testing for web applications [1]. Particularly, the approach uses a crawler such as Crawljax [2] that can dynamically interact with a web application, exercise the user interface elements of the web application, and generate a state-based model representing potential user interactions to validate desired properties of the application [3].

While crawl-based approaches hold promise for testing web applications, the input data required by the crawler to explore dynamic web applications is often generated randomly or prepared manually, leading to inefficiency and high costs. This becomes particularly challenging when considering the importance of code coverage in software testing of an application under test (AUT). Code coverage is a key metric in software testing, indicating the extent to which the source code of the AUT is executed during testing. According to Wikipedia, "A program with high test coverage has more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage" [4]. Specifically, statement and branch coverage are two widely-used metrics in software testing, representing the percentage of executed statements or branches of control structures in the AUT. Branch coverage is the percentage of branches (decision points) that are executed by a test suite. Branch coverage is stronger and subsumes statement coverage. Addressing how to generate and select test inputs for a web crawler to enhance code coverage of the explored AUT becomes a significant challenge in this context.

Consider a web application with the simple page in Figure 1. The page has two user-interacting elements: an age input field and an initially disabled NEXT button. When the user enters a valid age value, the NEXT button is enabled so that the user can navigate to the next page by clicking the button (Figure 2). If the user enters an invalid age value, such as −1 (Figure 3) or a non-integer, an error message is prompted, and the NEXT button remains disabled. To navigate this page, it is sufficient if the crawler enters a valid value for the age field and then clicks the NEXT button, as two user-interacting elements of the page will be performed. However, from the perspective of testing, the valid and invalid age values and the essential combination of the age input and click event of NEXT button must be exercised to ensure that the code that handles the input age value is adequately covered (i.e., executed). Therefore, to achieve better code coverage, the crawler may need to interact with the page via multiple sequences of user actions, such as < age = 25, NEXT: clickable = true >; < age = −1, NEXT: clickable = false >; < age = "String", NEXT: clickable = false > in order to cover as many programming logics in the source code as possible.

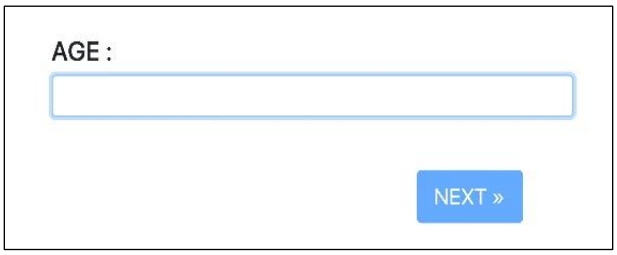

Figure 1. A simple web page in the initial state.

Figure 2. A simple web page in a state after receiving a valid input.

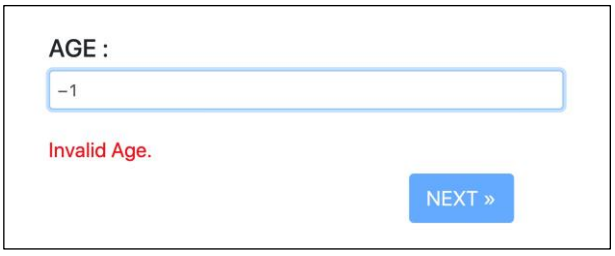

Figure 3. A simple web page in a state after receiving an invalid input.

Existing web crawlers rely on randomly generated or manually prepared inputs, leading to inefficiency and high costs in exploring and testing dynamic web applications. This study is motivated by the need for an improved approach, utilizing a reinforcement learning (RL) [5] agent capable of providing input actions to guide web crawlers in interacting with web pages. The goal is to automate and enhance the exploration of web applications under test efficiently, ultimately increasing code coverage during crawling and thereby achieving better test adequacy.

To realize this objective, an RL agent, which we call iRobot, is utilized to select a sequence of actions to guide crawlers' interactions with web pages, aiming to maximize the code coverage of the AUT. Specifically, a design of actions is proposed for iRobot to fill

all input fields of a web form using a single action to simulate the behavior of human tester and help the training converge faster. Additionally, the design of actions enables iRobot to automatically select both valid or invalid values for input fields to increase crawling code coverage, rather than randomly generating input values or manually selecting the inputs in advance. Furthermore, a convolutional neural network (CNN) is also presented to train the iRobot. Specifically, the CNN takes the Document Object Model (DOM) source code of a web page as input and generates corresponding actions to guide the web crawler in exploring the AUT. Moreover, an environment has been specifically designed for iRobot to support different reinforcement learning algorithms and neural networks so that we can study their effects on improving the code coverage of the crawler.

To evaluate the effectiveness of the proposed approach, several experiments are conducted using the design of actions and CNN network. The experimental results indicate that the proposed approach is promising. The environment supporting multiple reinforcement learning algorithms and neural networks proves to be useful. Additionally, the study demonstrates the effectiveness of the design of action combined with our CNN network in helping the crawler achieve higher code coverage compared to earlier work [6].

The rest of the paper is organized as follows: Section 2 briefly reviews related work. Section 3 presents the proposed approach and the design of iRobot. Section 4 describes and discusses the experimental results. Concluding remarks and future work are presented in Section 5.

2. Related Work

Reinforcement learning techniques have been successfully applied to various fields [7–10]. In software testing, Waqar et al. [11] proposed a reinforcement learning-based methodology for test suite prioritization. Their results show promise in detecting faults in regression testing. However, the application of RL to web application testing is still in its early stages. Existing research is still very limited. The studies related to this work are briefly reviewed below.

Lin et al. [12] proposed a natural-language approach for crawling-based web application testing. Basically, the method extracts and represents the attributes of a DOM element and its nearby labels as a vector. The vector is transformed into a multi-dimensional real-number vector by using a series of natural-language processing algorithms such as bag-of-words. The approach then uses the semantic similarity between the training corpus and the transformed vector to identify an input topic for the DOM element. Based on the identified topic, the input value of the element can be selected from a pre-established databank. The experimental results show that the proposed approach has comparable or better performance compared to traditional rule-based techniques.

Groce [13] used an adaptation-based programming (ABP) approach that utilizes reinforcement learning to automatically generating test inputs. Specifically, the approach generates test inputs for a Java program under test (PUT) by calling the ABP library to expose new behavior of the PUT, with the goal of optimizing the reward based on increases in test coverage. Compared with random testing and shape abstraction for testing container classes, the experimental results show that the proposed approach is quite competitive.

Carino and Andrews [14] proposed an automated approach based on ant colony optimization (ACO) to test application GUIs. Specifically, the approach presents an ant colony algorithm combined with Q-learning, called AntQ. It generates event sequences to traverse the GUIs and uses the number of GUI state changes caused by the events as the optimization goal. The experimental results show that, compared with random testing and the normal ant colony algorithm, AntQ can achieve better statement coverage and exhibits better fault-finding ability.

Kim et al. [15] proposed an approach using reinforcement learning to replace human designed metaheuristic algorithms in search-based software testing (SBST) method. Basically, the SBST algorithms try to find an optimal solution for test data generation based on feedback from the fitness function. The researchers formulate a search-based test data generation problem as an RL environment and train an RL agent using double deep Q-

networks (DDQN). The fitness value is used as the reward for the agent. Therefore, when the agent makes an action by creating a new candidate solution to maximize the cumulative reward, the fitness value of the solution can be minimized. The experimental results show that the proposed approach is feasible and can achieve 100% branch coverage for training functions written in C language.

Liu et al. [16] proposed an incremental and interactive web crawler called GUIDE that can be guided by user-supplied directives to iteratively explore a web application. Specifically, GUIDE actively asks the user for directives to explore web pages when it finds an input field rather than passively accepting the user's instructions. The experimental results show that GUIDE can increase code coverage compared to traditional web crawlers. However, GUIDE still requires human intervention to provide inputs during crawling. This work is an attempt to use reinforcement learning techniques to train an agent to provide inputs and guide web crawlers to achieve better code coverage.

Zheng et al. [17] presented an automatic end-to-end web testing framework named WebExplor to enable adaptive exploration of web applications. In particular, WebExplor adopts reinforcement learning to generate different action sequences to discover new web pages. A curiosity-driven reward function and a DFA are used to provide low-level and high-level guidance for RL exploration, respectively. The DFA is a deterministic finite automaton that records global visit information during exploration. If WebExplor cannot discover a new state within a certain amount of time, it selects a path from the DFA based on curiosity and resumes exploration. The experimental results show that WebExplor can significantly improve fault detection rate, code coverage, and efficiency compared to state-of-the-art techniques.

Liu et al. [18] proposed an RL approach for workflow-guided exploration. The approach aims to alleviate the overfitting problem when training an RL agent to perform web-based tasks [19] such as booking a flight by mimicking expert demonstrations. Particularly, the approach includes high-level workflows that can limit the allowable actions at each time step by pruning those bad exploration directions. This allows the agent to discover sparse rewards faster while avoiding overfitting. The experimental results show that the proposed approach can achieve higher success rates and significantly improve sample efficiency compared to existing methods.

Sunman et al. [20] propose a semi-automatic method and tool called AWET that combines exploratory testing (ET) with crawler-based automated testing and apply it to web application testing. The tool records a set of test cases by performing ET manually beforehand, and then uses these test cases as the basis to explore and generate test cases for a web application.

Liu et al. [21] proposed a model-based representational state transfer (RESTful) API testing model to dynamically update the built property graph. They claimed that their model could detect more lines of code and more bugs than state-of-the-art methods. Yandrapally et al. [22] applied a model-based test generation technique to analyze page fragments finely and to create test oracles. Their experiments showed that their approach outperformed feeding the whole webpage. Sherin et al. [23] proposed a Q-learning inspired dynamic exploration approach that uses guided searches to systematically explore dynamic web applications with little prior knowledge about the applications. Their results show that QExplore outperforms the Crawljax and WebExplor tools in achieving higher coverage and more diverse DOM.

Another attempt is the earlier work by authors of this paper [6], which proposes a reinforcement learning agent trained with DQN to guide crawlers to explore web applications to increase code coverage. The experimental results show that the agent can guide the crawler to achieve better code coverage than traditional web crawlers. This paper is an extension of this earlier work with a new design of actions, state model, and reward function to further improve the code coverage of web crawlers. Additionally, an environment is designed to support agent training using different reinforcement learning algorithms and neural networks.

3. The Proposed Approach

This section describes the proposed approach, including an overview of the proposed approach for web application exploration, the design of actions, the state model, the reward function, the neural network architecture, and the design of environment to support different RL algorithms.

3.1. Overview of the Proposed Approach

Figure 4 shows a schematic diagram of the proposed approach. The iRobot interacts with an environment consisting of a (modified) Crawljax and the web application under test (i.e., WebApp). At each time step t, iRobot receives the current state of the environment s_t and the reward r_t from the environment. The iRobot then selects an action a_t from a set of available actions based on r_t according to the reinforcement learning algorithm and sends a_t to guide Crawljax to explore the WebApp. The exploration of the WebApp caused by a_t can change the state of the environment to s_{t+1} and generate a new reward r_{t+1}, both of which are returned to iRobot. The process will continue until a predefined time step is reached. The goal of the process is to train iRobot to maximize a cumulative reward proportional to code coverage of the WebApp.

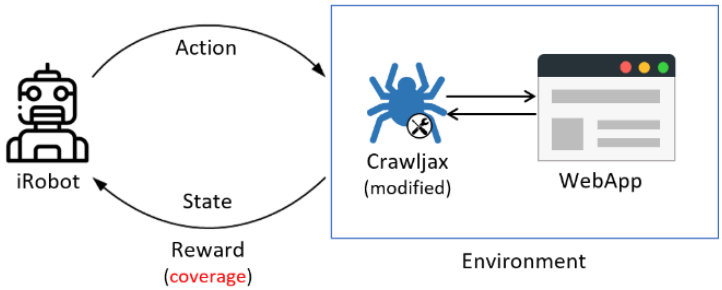

Figure 4. Schematic diagram of the proposed approach.

3.2. The Design of Actions

In reinforcement learning, the agent selects an action from a set of available actions and sends the selected action to the environment. Usually, the set of available actions is finite so that reinforcement learning algorithms can converge. However, this can be a challenge when designing actions to explore web applications, as a web page can contain various input fields, buttons, and hyperlinks that users can interact with. Also, the number of input fields, buttons, and hyperlinks in each generated web page of a given web application is often different. Therefore, identifying a finite set of actions required to train the RL agent to interact with a web application is not trivial.

For crawling and testing of web applications, in the earlier work of this paper [6], a set of primitive actions including clicking, changing focus, and entering text with different values was used, as shown in Table 1. In this action design, $n + 2$ actions were used, where n is the number of test input values used to explore the application under test. Specifically, a_0 and a_1 are click and change-focus actions, respectively, while a_2 to a_{n+2}, are input actions with associated input values $v_1, \ldots, v_n$. Using this set of actions, it is sufficient to perform the actions required to interact with a web page, such as clicking buttons or hyperlinks, changing the focus of widgets, and populating all input fields with a set of pre-defined values.

However, for most web application user scenarios, the user typically fills in all input fields of a web page form (such as registering a user account) and then submits the form. When utilizing click, change-focus, and input-text actions as the available set of actions in the environment, the agent requires numerous actions to navigate, complete, and submit the form. Consequently, the action search space can be huge.

To narrow the action search space to improve RL agent training, this paper proposes a new design of actions for exploring and testing web applications. Specifically, the proposed design mainly focuses on actions related to input fields and button clicks, as the web crawler

itself can automatically navigate and test hyperlinks. The proposed design, specifically, incorporates actions aimed at populating all fields of a web form, simulating the behavior of a human tester who typically fills the input fields with test data before submitting the form. This action design can minimize the size of action search space, resulting in faster convergence during training.

Table 1. The action design in the authors' earlier work.

No	Action Type	Input Value
a_0	Click	–
a_1	Change-Focus	–
a_2	Input	v_1
⋮	⋮	⋮
a_{n+2}	Input	v_n

For example, consider the simple web page shown in Figure 5, which consists of two input fields and one button. This web page requires at least three actions, including two inputs and one click, for a web crawler to complete and submit the form. However, even if the input values of these two fields provided by crawler are all valid, the probability that the crawler will successfully complete and submit the form is $1/3 \times 1/3 \times 1/3 = 1/27$. Nevertheless, if the crawler has an action to populate all fields with valid inputs. It takes one action to fill in the input fields, and one action to click the button to complete and submit the form. Therefore, the probability of successfully completing and submitting the form can be increased to $1/2 \times 1/2 = 1/4$. Consequently, the training speed of the agent can be increased, and the neural network can be expected to converge more quickly.

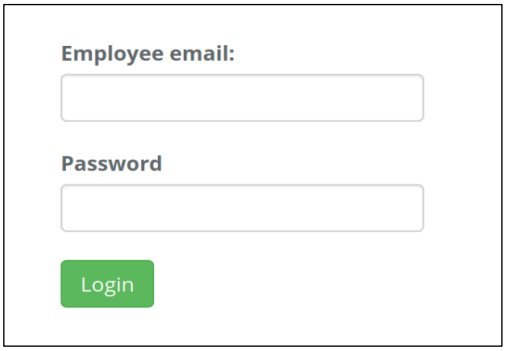

Figure 5. A simple login webpage.

Also, as mentioned earlier, both valid and invalid values of the input fields need to be tested and covered for testing purposes. Verifying the responses of the web application under test to invalid inputs may also increase the code coverage of the web crawler. Therefore, in the proposed action design, we also include actions that fill invalid values. Note that all input fields need to be filled with valid values in order to test the happy path (i.e., no exceptions or error conditions). However, in order to test an unhappy path (or exception path), it is not necessary to fill all input fields with invalid values. Specifically, to reveal faults, it is necessary that each input field is individually tested with an invalid value.

Table 2 shows the proposed design of actions for iRobot. The design consists of 8 actions, where actions $a_0 - a_2$ are related to input values, and actions $a_3 - a_7$ are associated with button clicks. This design can handle different types of AUTs with varying numbers of input fields and buttons. It fills all input fields with values in one action, instead of selecting one field at a time, for higher efficiency. It also groups buttons into at most four per group

to accommodate webpages with many buttons. If there are less than four buttons, such as two, then there is only one button group, and some actions have no effect (see also below).

Table 2. The proposed design of actions.

No	Action
a_0	Fill all input fields with valid values (i.e., all valid inputs), ValidInputIndex++, FocusIndex++
a_1	Fill all input fields with valid values except for the focused field, which is filled with an invalid value (i.e., single invalid input), ValidInputIndex++, InvalidInputIndex ++, FocusIndex++
a_2	Fill the focused field with an invalid value, InvalidInputIndex++, FocusIndex++
a_3	Click the first button in Group, FocusIndex++
a_4	Click the second button in Group, FocusIndex++
a_5	Click the third button in Group, FocusIndex++
a_6	Click the fourth button in Group, FocusIndex++
a_7	GroupIndex++ (i.e., change group), FocusIndex++

In our design, action a_0 is used to fill all input fields with valid values. Action a_1 is used to fill only one input field with an invalid value and the remaining input fields with valid values. Action a_2 is used to populate the focus input field with an invalid value. Combining action a_0 with actions a_1 and a_2 enables iRobot to simulate the behavior of a human tester, testing each input field separately with invalid values (called **invalid inputs** below). Note that the design of actions uses FocusIndex as the counter index of the input widget that the application is currently focused on (i.e., the widget has a keyboard input focus), and FocusIndex++ sets the focus index to the next input widget. In the proposed action design, each action contains a FocusIndex++ to change focus to the next input widget when the action is completed. This avoids situations where the agent immediately repeats actions on the same input widget, which may further narrow the action search space.

Also, web applications can often dynamically generate different response pages when they receive different input data. For example, depending on the logged in user, the response page may be different. Therefore, different valid and invalid values for an input field are necessary for testing purposes. By doing this, the test code coverage can also be improved. To enable iRobot to select a different value for an input field, a counter index for the sets of input values called InputIndex is used in the action design. If the values of the input set is valid (or invalid), the InputIndex is called ValidInputIndex (or InvalidInputIndex). The idea of an input index is similar to that of the FocusIndex. In the proposed action design, actions a_0, a_1 and a_2 will populate input fields with a set of valid (or invalid) values according to the value of ValidInputIndex (or InvalidInputIndex), and increment the index value by 1 after completing the actions. This design allows iRobot to interact with web applications using different valid and invalid inputs.

For example, consider the simple web page shown in Figure 5. Initially, let the focus of web page be on the field labeled "Employee email". Suppose that Tables 3 and 4 are the two sets of valid and invalid inputs for a web page, respectively. Figure 6a,b show the input values of the web page when iRobot selects action a_0 with ValidInputIndex=1 and ValidInputIndex = 2, respectively. Likewise, Figure 6c shows the input values of the web page when iRobot selects action a_1 with InvalidInputIndex = 1. However, if iRobot selects action a_2 (instead of a_1) with InvalidInputIndex = 1, only the focus field is populated with an invalid value, as shown in Figure 6d. Note that the labels of the input fields in a web page form will be extracted and classified according to semantics of the labels. For example, an input field with labels "Employee email" and "Email address" could be classified into an "Email" category. Based on the category of the input field label, iRobot can find the

corresponding values for a particular input field from the indexed set of invalid (or valid) inputs. Therefore, in Figure 6d, iRobot selects the value "teacher@@ntut.edu.tw" of the "Email" category from the invalid input set {"An*rew", "teacher@@ntut.edu.tw", "-", . . .} for the focused field labeled "Employee email".

Table 3. An example of two sets of valid inputs.

Category of Input Field	ValidInputIndex	
	Set 1	**Set 2**
Name	Andrew	Peggy
Email	teacher@ntut.edu.tw	student@ntut.edu.tw
Password	Andrew0610	ab5sRsda.ad
. . .	. . .	. . .

Table 4. An example of two sets of invalid inputs.

Category of Input Field	InvalidInputIndex	
	Set 1	**Set 2**
Name	An*rew	Pe@ggy
Email	teacher@@ntut.edu.tw	student@@ntut.edu.tw
Password	-	0
. . .	. . .	. . .

Figure 6. The login webpage with input fields populated using different indexed sets: (**a**) action a_0 using the value set ValidInputIndex = 1; (**b**) action a_0 using the value set ValidInputIndex = 2; (**c**) action a_1 using the value set InvalidInputIndex = 1; and (**d**) action a_2 using the value set InvalidInputIndex = 1.

For actions related to button clicks, iRobot selects a button to click from the group cataloged by a counter index called GroupIndex. The actions a_3, a_4, a_5, and a_6 in Table 2 are used to click the first, second, third, and fourth button in the indexed button group,

respectively. If there are less than four buttons, such as two, then there is only one button group and actions a_5 and a_6 have no effect (called **invalid action** below), while a_7 is used to change the button group. For instance, let us assume that a web page has 11 buttons. In the design of actions, these 11 buttons will be arranged into 3 groups, as shown in Figure 7. When the GroupIndex is 1, actions a_3, a_4, a_5, and a_6 will select widget buttons 0 to 3 to click, respectively. Similarly, if the GroupIndex is 2, widget buttons 4 to 7 are selected. Note that if the GroupIndex is 3 and action a_6 is selected, this will result in an invalid action as there is no corresponding button to click. An invalid action here means that the agent chose a certain action, but the action has no meaning; it neither has any effect on the environment, nor does it change the state. Because it does not help the training of the agent, we should try to avoid invalid actions during the training process with the help of the reward function.

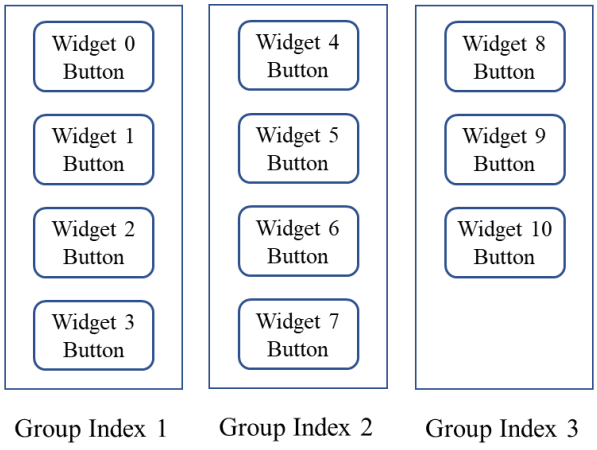

Figure 7. An example of button groups.

3.3. The Design of State Model

To represent the state of the environment in our earlier work, we used the DOM [24] of the web application under test, the branch coverage of the application, and the index vector (i.e., FocusIndex) of the focus widgets in the web pages of the application. To support the proposed new action design, the environment state also contains additional information from ValidInputIndex, InvalidInputIndex, and GroupIndex in addition to the DOM, branch coverage, and focus vectors. Equation (1) is the state vector $\widehat{s_i}$ of the environment:

$$\widehat{s_i} = < DOM(s_i), CV(s_i), FI(s_i), GI(s_i), VII(s_i), IvII(s_i) > \quad (1)$$

where $DOM(s_i)$ is a vector representing the state s_i (i.e., DOM) of the web application; $CV(s_i)$ is a vector representing the branch coverage of the application at state s_i; $FI(s_i)$ and $GI(s_i)$ are one-hot encoding vectors representing FocusIndex and GroupIndex, respectively, for the application in state s_i; and $VII(s_i)$ and $IvII(s_i)$ are vectors representing the values of ValidInputIndex and InvalidInputIndex, respectively.

Note that, instead of statement coverage, branch coverage is used to reduce the search space in the approach since the number of branches is much less than that of statements. Suppose that the number of branches in the sever-side source code of the web application is n. Equations (2) and (3) show the coverage vector $CV(s_i)$:

$$CV(s_i) = < b_1(s_i), b_2(s_i), \cdots, b_n(s_i) > \quad (2)$$

$$where\ b_j = \begin{cases} 1 & if\ branch\ j\ of\ state\ s_i\ is\ covered, \\ 0 & otherwise \end{cases}, j = 1 \cdots n \quad (3)$$

To encode the values of ValidInputIndex and InvalidInputIndex, we first encode the input index into a category vector, where the input index can be ValidInputIndex or InvalidInputIndex. Suppose there are m input values in a category and the j-th index value

is selected. Equations (4) and (5) show the one-hot encoding for the category vector. Here the encoding value of the *j*-th input in the vector is 300 instead of 1, simply because we try to avoid overlapping with the encoding value of DOM.

$$Category(s) = < e_1(s), \ldots, e_i(s), \ldots, e_m(s) > \tag{4}$$

$$\text{where } e_i = \begin{cases} 300 & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}, \; i = 1 \cdots m \tag{5}$$

With the vector for each category of input fields, the vectors $VII(s_i)$ and $IvII(s_i)$ can be encoded. Suppose the input fields of a web application has n categories. Equations (6) and (7) show the vector representations for $VII(s_i)$ and $IvII(s_i)$, respectively.

$$VII(s_i) = < Category_1(s_i), Category_2(s_i), \ldots, Category_n(s_i) > \tag{6}$$

$$IvII(s_i) = < Category_1(s_i), Category_2(s_i), \ldots, Category_n(s_i) > \tag{7}$$

3.4. The Design of Reward Function

To guide the crawler to explore the web application under test and increase the branch coverage during agent training, the reward for the action is calculated using the coverage vector $CV(s_i)$. Basically, the more branch coverage an action can increase, the higher the reward for that action. Since branch coverage monotonically increases during the exploration of a web application, if an action changes the state of the application from s_i to s_j, the increase in branch coverage can be computed by comparing vectors $CV(s_i)$ and $CV(s_j)$. Equation (8) shows the reward function for iRobot where K_0, K_1, K_2, K_3 are positive integers.

$$reward(a) = \begin{cases} K_0 \sum_{i=1}^{n} \left(b_i(s_j) - b_i(s_i) \right) & \text{if action } a \text{ increases overall coverage} \\ -K_1 & \text{if action } a \text{ is invalid action} \\ -K_2 & \text{if action } a \text{ changes group} \\ -K_3 & \text{else} \end{cases} \tag{8}$$

The above reward function will generate a positive reward only if the selected action can increase branch coverage. Note that coverage may increase when executing error detection codes with invalid inputs (i.e., taking actions a_1 or a_2). Recall that invalid inputs are the result of taking actions a_1 and a_2, and invalid actions are actions with no effect, such as clicking on a third (nonexistent) button in a webpage with two buttons. The reward will be negative when the selected action is an invalid action, the action only changes the button group, or the action does not increase coverage at all. We have added a sentence to clarify this point in the paragraph discussing the reward.

Note that, to prevent the agent from selecting an invalid action, a large negative reward should be given for such cases. Also, to reduce the probability that the crawler will keep changing the button group without filling in the inputs or clicking the buttons, a small negative reward should be given when selecting such an action. Additionally, to prevent the agent from selecting actions that do not further increase branch coverage, a slightly smaller negative reward can also be given for those actions. Therefore, in the proposed reward function, the parameters are deliberately designed as $K_1 > K_2 > K_3$. Furthermore, the value of K_0 can be adjusted based on the web application under test, since the increase in branch coverage for an action is implementation dependent for the web application under test. In the proposed approach, the values $K_0 = 10$, $K_1 = 0.5$, $K_2 = 0.35$, and $K_3 = 0.25$ were used.

3.5. The Architecture of the Proposed CNN Network

Figure 8 shows the convolutional neural network (CNN) architecture used to train iRobot. Basically, the inputs of the network include the DOM source code of the web page under exploration, Coverage Vector, FocusIndex, GroupIndex, and InputIndex as mentioned in Section 3.4. It is worth noting that different web pages can have different widths and heights, and users can usually scroll right and down to view the information of a web page. Therefore, if a fixed-size screenshot is used as CNN input, the quality of different web page screenshots can vary greatly, which could make the web form features difficult to recognize and extract. Thus, instead of using a web page screenshot, the DOM source code of web page is used as CNN input. Specifically, the DOM source code of a web page is converted into a one-dimensional array by concatenating each line of code. Since the length of DOM for different web pages can still be different, in order to have a fixed-length input for CNN, the maximum length of DOM source code is deliberately limited to 130,100 to accommodate most web pages. If the DOM length of a web page is less than the maximum length, padding is added to the end of the DOM. Similarly, for the length of FocusIndex (or GroupIndex), we will identify the maximum number of input fields (or buttons) in a web page for the application under test and use it as the length of the index. Again, padding is added if the number of input fields (or buttons) for a web page is less than the index length.

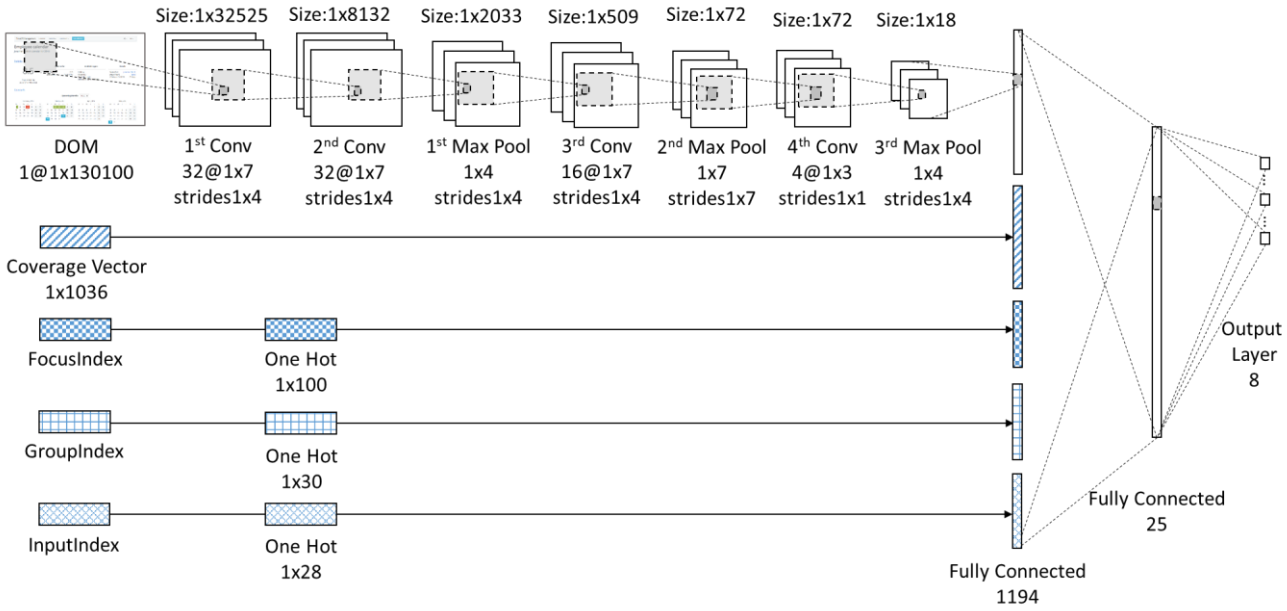

Figure 8. The convolutional neural network architecture of the agent.

In Figure 8, the CNN network takes the DOM of a web page as an input and treats it as a long picture. The structure of the CNN network contains 4 convolutional layers, 3 max pooling layers, and 2 fully connected layers. The convolutional layers are used to extract features of the DOM; the max pooling layers are responsible for reducing the size of the DOM while preserving vital information; and the fully connected layers are used to synthesize the final output. Further, in the network structure, the information of Coverage Vector, FocusIndex, GroupIndex, and InputIndex is fed directly to the first fully connected layer together with outputs from the third max pooling layer and then to the second fully connected layer. This enables the agent to learn their impacts on changes to the DOM state. Finally, the output layer produces the values of 8 actions. Moreover, the activation function of each layer is LeakyReLU, and the linear activation function is used in the output layer.

3.6. The Design and Implemetation of the Environment

To enable iRobot to use different reinforcement learning algorithms and neural networks, the approach provides an environment leveraging OpenAI Gym [25], an open-source library that supports the development and comparison of different RL algorithms. The environment has a web driver that drives a popular open-source web crawler, Crawljax [2] to interact with the web application under test. Furthermore, the environment is designed to support user-defined environment states, actions, and reward functions for reinforcement learning. In addition, the environment collects server-side code coverage of web application under test, which is used to calculate the reward for an action.

Figure 9 shows the system architecture of the proposed environment. The iRobot can interact with the WebEnvironment that implements the functions of Gym.env and the necessary APIs to provide a Gym environment for utilizing OpenAI Gym. The WebDriver is used to drive a crawler, control the web application under test (i.e., WebApp), and obtain observations from WebEnvironment. The Crawljax tool has been extended to implement the WebDriver interface. ActionStrategy is used to convert high-level actions into low-level operations that are used by WebDriver to drive Crawljax to interact with WebApp. Note that ActionStrategy is an abstract class and has to be inherited and instantiated by ConcreteActionStrategy. This design allows the environment to easily change actions by simply implementing another ConcreteActionStrategy. The State is used to hold a set of environment states retrieved from WebApp by WebEnvironment through WebDriver. Reward is responsible for calculating the reward based on the code coverage obtained from the CodeCoverageCollector after executing an action.

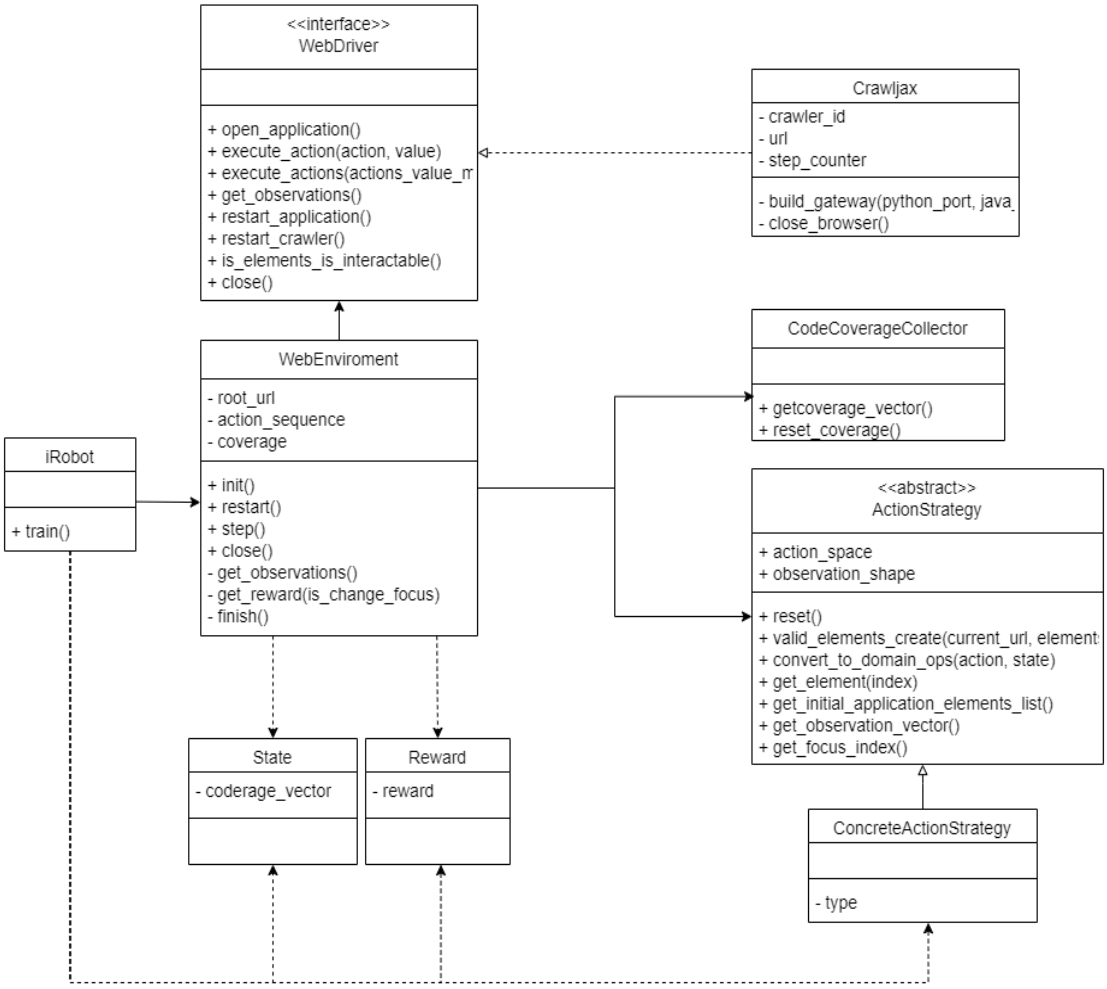

Figure 9. Class diagram of the reinforcement learning environment.

4. Experiments and Results

To evaluate the usefulness of the proposed approach and the effectiveness of the design of actions, three experiments were conducted, and the following four research questions were addressed.

RQ1. In the proposed approach, what is the most suitable neural network for web crawling?
RQ2. What is the most suitable number of episodes to train iRobot?
RQ3. Which RL algorithm achieves better code coverage in web crawling?
RQ4. Can the new iRobot improve crawling code coverage compared with our earlier work?

We conducted three experiments to answer these questions. The first experiment addressed RQ1. The second experiment answered RQ2 and RQ3. The third experiment tackled RQ4. In the experiments, we used branch coverage as the performance criterion, which is an important measure in software testing, as described in the Introduction section. Therefore, branch coverage is a suitable and useful measure for our problem.

4.1. The Experimental Environment and Subject Application

For the experiments, three computers with NVIDIA graphic cards were used for agent training. The specifications of the computers are listed in Table 5. The programs for the iRobot RL agent were written in Python with Tensorflow [26] and the OpenAI Gym framework [25]. Programs for the iRobot environment described in Section 3.6 were written in Java with Crawljax and its plugins. Versions of the tools and learning framework are listed in Table 6.

Table 5. Specifications of hardware used in the experiments.

CPU	Intel Core i7-7700
RAM	32 GB
GPU	NVIDIA GeForce RTX 2070
OS	Ubuntu 18.04

Table 6. The versions of software used in experiments.

Framework	Tensorflow v1.10
OpenAI Stable-baselines	v2.10.0
Python	v3.6.5
Java	v1.8.0
Crawljax	zaproxy v3.7
CUDA	v10.0
cuDNN	v7.2.1

The web application used in the experiments was TimeOff.Management [27], an open-source application for small or medium size companies to manage employee absences. The application is complex and contains many links and web forms that require user input. Table 7 shows some essential attributes of TimeOff.Management, including the version of the application used in experiments, the total lines of code (LOC) of the application, the number of branches in the source code, etc. It was selected as the target application for the following reasons:

- TimeOff.Management is a popular open-source web application available for public use. Therefore, it can be used by related studies to compare results with our experimental results.
- The size of the application is moderate, suitable for verifying the feasibility of the proposed approach with an acceptable training time.

- The web pages of the application contain many hyperlinks, buttons, and web forms with various input fields for filling in different kinds of data such as login, registration, and employee. Therefore, this application is suitable for training an RL agent to select input values.

Table 7. Some essential attributes of the target application.

Web application name	TimeOff.Management
Type	Employee absence management
Number of stars on GitHub	708
Version	v0.10.0
Total line of code (LOC)	2698
Number of branches in code	1036

To explore TimeOff.Management, the crawler first needs to provide appropriate values for the various input fields on the registration page, including company name, supervisor name, email, password, and confirm password, etc., to successfully register a company, as shown in Figure 10. Once a company is registered, the user can edit the employee information, configure options that include leave type, book a new leave request, or check the absences using a calendar view, and more. To improve crawling code coverage, both valid and invalid values are used for the input fields of TimeOff.Management. Also, code coverage of the application is collected using Istanbul v0.45 [28]. The tool can collect code coverage for ES5 and ES2015+ JavaScript codes.

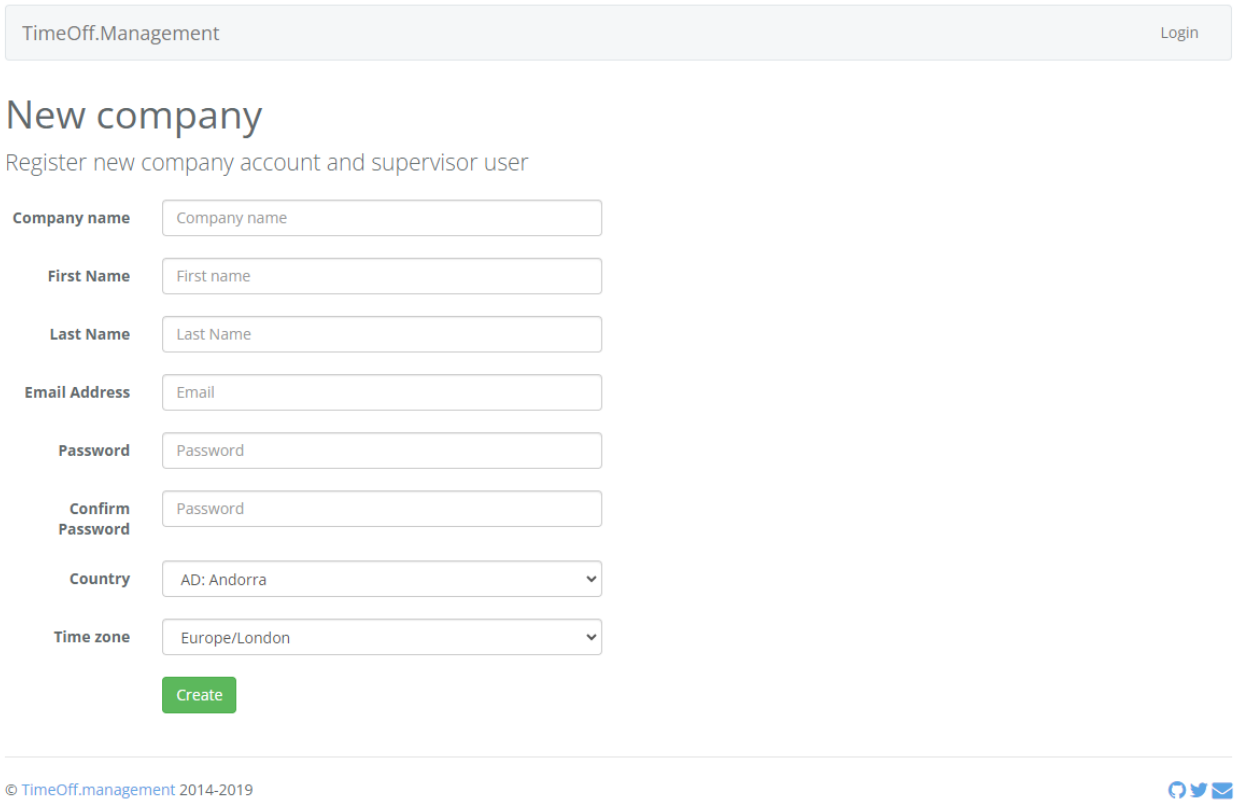

Figure 10. The registration page of TimeOff.Management.

In the following experiments, we used two RL algorithms, DQN [29] and PPO [30] to conduct the experiments. In addition, Monkey was used for comparison with our experimental results. In the experiments, both iRobot and Monkey used the same environment. However, Monkey selects high-level actions randomly for a given environment, while iRobot selects actions through reinforcement learning. Also, for each RL algorithm, unless specified otherwise, the experiment results are averages for 3 runs. Additionally, TimeOff.Management was used to train and validate RL algorithms. We did not have a test model since the experiences learned by an agent from crawling TimeOff.Management may not apply to exploring other web applications, as the inputs and implementations may be quite different.

4.2. Experiment 1

The first experiment addressed RQ1. In the experiment, we examined several neural networks, including the proposed CNN (see Section 3.5), CNNLSTM (CNN Long Short-Term Memory [31]), MLP (Multilayer Perceptron [32]), and MLPLSTM, to see which is the most suitable for training iRobot to achieve higher code coverage. To simplify the discussion and to make fair comparisons, the hyperparameters of the network models used default settings of the stable-baseline library without additional hyperparameter optimization.

The reason that we evaluated several different neural-network models is because researchers in the ANN field find it hard to provide good guidelines for choosing architecture. In fact, they usually determine the hyperparameters, such as the number of layers and nodes, by experiment (called the validation phase in the ANN literature). Conceptually, dynamic webpages depend on the state; therefore, a stateful architecture (such as LSTM) should be better because it has internal memory. However, our experiments showed otherwise. At time of writing of this paper, the question of which ANN model is better for form filling warrants further research.

Based on the results of this experiment, the neural network determined to be the most suitable was then used for subsequent experiments. The number of training steps was set to 10,000 steps, the number of episodes was set to 32, and the design of actions described in Section 3.2 was used. Furthermore, during model training and validation, the set of generated action sequences that achieved the highest branch coverage while having the shortest sequence length, as well as their rewards, was recorded for each episode. The code coverage achieved by such action sequences for all episodes was then used to calculate the results of the experiments.

Figure 11 shows the experimental results for different neural networks. The branch coverage is the crawling result obtained using the action sequence generated during the model validation process. For DQN, the validation is performed only once. For PPO, since it is a stochastic algorithm, we select the best crawling result from 20 independent validation runs. Also, for the DQN algorithm, only the CNN and MLP results were provided. This is because the default policy networks available for the DQN algorithm in the OpenAI stable-baseline [33] include only the CNN and MLP networks. From the results in Figure 11, it can be seen that the CNN network can achieve about 19–20% branch coverage when using DQN and PPO algorithm. The other networks, however, have even smaller code coverage (not greater than 5%). There could be several factors influencing this result, such as using the DOM as input or the structure of the network being used. Nevertheless, the custom design of our CNN network performs better than other networks. Hence, among the studied agent architectures and training algorithms, the CNN agent trained with the PPO algorithm is better, or more efficient, because it reaches the highest branch coverage with the same number of training steps.

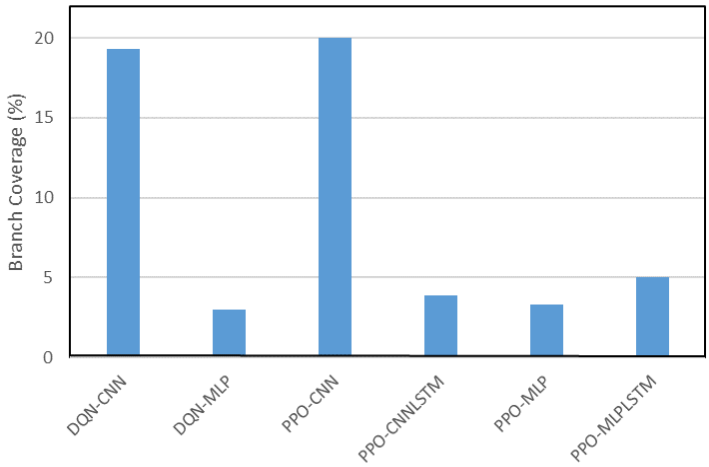

Figure 11. Comparison of results using different algorithms.

4.3. Experiment 2

The second experiment addressed RQ2 and RQ3. In the experiment, we used the proposed CNN network and changed the number of training episodes for different RL algorithms to determine the appropriate number of episodes for training iRobot with the RL algorithm. The number of episodes can significantly affect training results since, in the proposed approach, it is used to determine when to terminate the training of agent. Therefore, finding a suitable number of episodes to train iRobot is crucial. Likewise, in this experiment, for DQN (i.e., DQN-CNN in Section 4.2), PPO (i.e., PPO-CNN in Section 4.2), and Monkey, the number of training steps was set to 10,000 steps, and the number of training episodes was set to 16, 32, 48, 64, and 128.

To answer RQ2 and RQ3, we evaluated three kinds of code coverage results achieved by different RL algorithms with varying numbers of episodes, including (1) the branch coverage obtained by the shortest action sequence generated during model validation (called episode-verify); (2) the best branch coverage achieved by action sequences generated during model training for one episode (called episode-best); and (3) the total cumulative branch coverage achieved by action sequences generated throughout the entire model training process (called training-total).

The results of episode-verify, episode-best, and training-total can serve as valuable references for designing action sequences to test web applications from different perspectives. Specifically, the result of episode-verify can provide a short action sequence for quickly testing a web application while achieving better code coverage. The result of episode-best can provide a deeper or more complete action sequence than that from episode-verify to test a web application and achieve the highest code coverage. Moreover, the result of training-total can provide various action sequences that together can achieve the best overall code coverage for testing the functionality of a web application.

To obtain the result of episode-verify, similar to Experiment 1, we used the result from one validation run of DQN and the best result from 20 validation runs of PPO. Since Monkey has no training model, no validation was required. Figure 12 shows the branch coverage achieved by the shortest action sequences generated during the validation runs for DQN and PPO. The results suggest that PPO is slightly better than DQN. Furthermore, PPO achieves better code coverage at 22.30% and 23.17% for 48 and 128 episodes, respectively, while DQN shows no obvious significant differences among different numbers of episodes. In short, this part of experiment shows that the number episode affects the performance of the PPO algorithm but not that of the DQN algorithm. It is still too early to conclude if higher episode numbers improve the performance of the PPO algorithm in a more general setting.

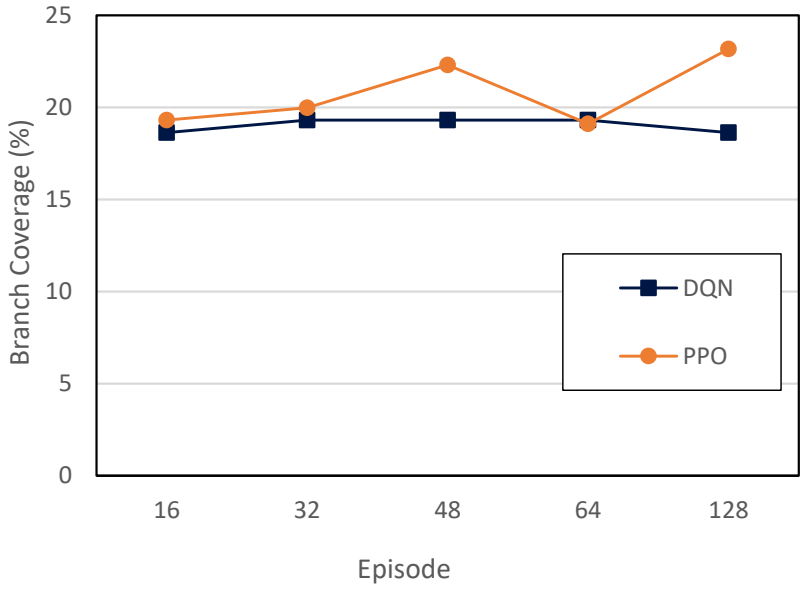

Figure 12. The results of episode-verify for different episode numbers.

Figure 13 shows the best branch coverage (i.e., episode-best) achieved by the action sequences generated in one episode during model training for different numbers of training episodes. The results show that the branch coverage of DQN in one episode is improved from 21.53 to 28.43% when the number of training episodes increases from 16 to 128. Monkey has a similar outcome, with code coverage increasing from 19.96 to 28.57%. However, PPO has higher code coverage (about 26.61%) when the number of training episodes is 48, after which the coverage decreases. Furthermore, the results also indicate that the branch coverage of DQN, PPO, and Monkey are very similar when the number of episodes is 48.

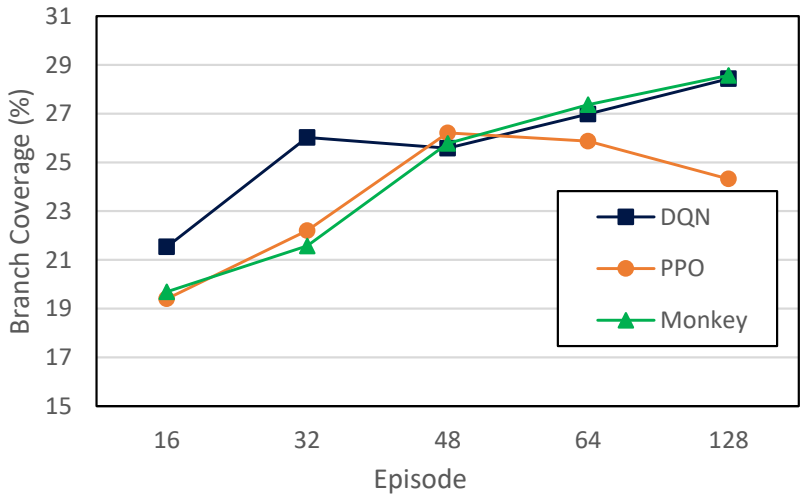

Figure 13. The results of episode-best for different episode numbers.

Notably, in contrast to the results of episode-verify in which PPO has better code coverage with 128 episodes, the results of episode-best suggest that both DQN and Monkey can achieve relatively high code coverage with 128 episodes. However, it should also be noted that as the number of episodes increases, so does training time. For example, as shown in Figure 14, the training time of DQN increases proportionally when the number of episodes increases.

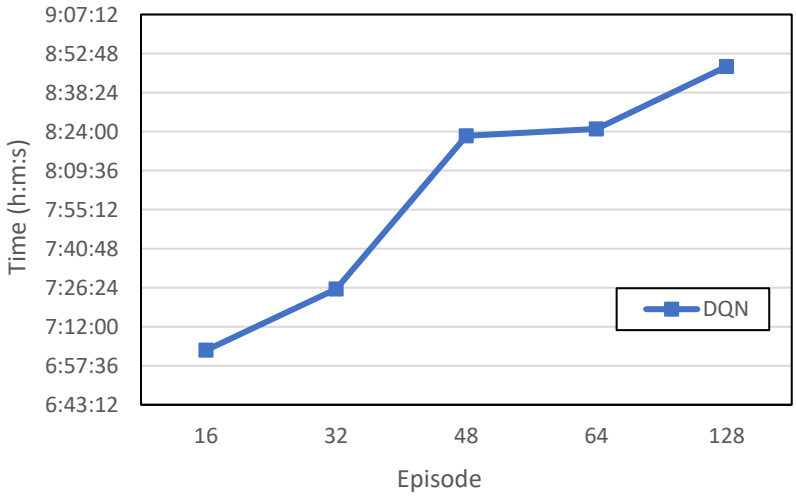

Figure 14. The training time of DQN with for different episode numbers.

It is worth noting that for episode-best, Monkey performs no worse than DQN and PPO when the number of training episodes is 64 or 128. This indicates that the Monkey can perform well when given enough time with the proposed design of actions and experimental environment. However, in terms of training-total scenario (shown below), Monkey is not as effective as the RL algorithms. Overall, it is still not a promising approach.

Figure 15 shows the total cumulative branch coverage (i.e., training-total) achieved by the action sequences generated during the entire model training process for different numbers of training episodes. The results show that for DQN, PPO, and Monkey, the total cumulative branch coverage is always greater than the coverage obtained in one episode. Furthermore, the results also indicate that DQN always outperforms the others in different numbers of episodes. Particularly, the highest cumulative branch coverage of DQN is about 44.21% with a total of 128 episodes.

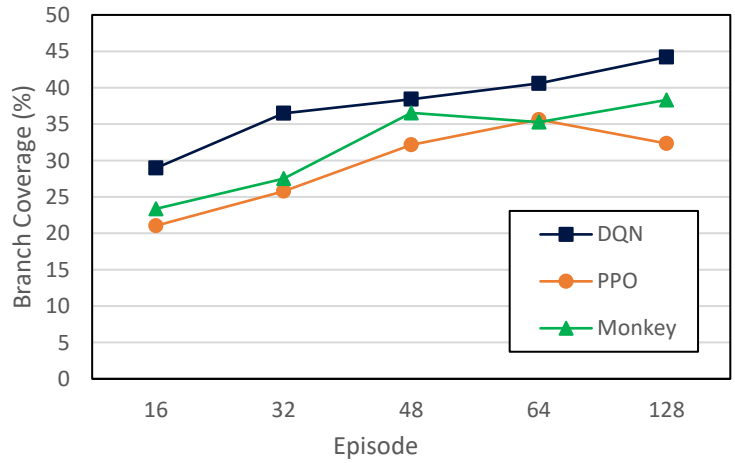

Figure 15. The results of training-total for different episode numbers.

Overall, the answer to RQ2 is that "for different RL algorithms, the suitable number of training episodes is different". For DQN and Monkey, 128 episodes can yield the highest branch coverage. However, for PPO, the training result with 48 episodes is better. Nevertheless, the more training episodes are used, the longer training time is required.

Moreover, the answer to RQ3 is that "different RL algorithms can have different performance characteristics in different cases". Particularly, PPO can get better code coverage for an action sequence when the validation model is used. On the other hand, both DQN and Monkey can achieve higher code coverage for an action sequence when

the training model is used. Furthermore, DQN performs best on the training model when considering the total cumulative code coverage of generated action sequences. As the training-total resembles the actual application of software testing, we recommend the DQN-CNN model with 128 training episodes.

4.4. Experiment 3

The third experiment addressed RQ4. In this experiment, iRobot's results were compared with those of our earlier work to evaluate iRobot's improvement resulting from the proposed design of actions, reward function, and environment. In our earlier work, DQN and CNN neural network were used to train the model and the results were evaluated using the validation model; accordingly, we used the episode-verify outcomes for iRobot with the DQN–CNN model for fair comparison.

Table 8 shows the comparisons between iRobot and our earlier work, including the number of training steps, training time, branch coverage, and differences in the target application. The results indicate that our earlier work required 500,000 steps in a single run to train the agent successfully, taking about 59 h and 8 min to achieve branch coverage of 16.5%. With the proposed action design, the training steps were significantly reduced to around 2% (10,000 steps), and the training time was also substantially decreased to approximately 12.54% (7 h and 25 min). Furthermore, there was a notable 1.7% enhancement in branch coverage, which increased from 16.5 to 18.2%.

Table 8. Comparisons between iRobot and our earlier work.

	Earlier Work	**iRobot**
Number of training steps	500 k steps	10 k steps
Training time (hh:mm:ss)	59:08:34	7:25:55
Branch coverage	16.5%	18.2%
Modification of subject application (TimeOff.Management)	The app was modified by removing the login page and corresponding hyperlinks to facilitate agent training	The app was not modified
The first page of crawling	Company registration page	Home page (i.e., login page)

Note that in our earlier work, in order to narrow the crawling scope for training the agent successfully, we deliberately removed the home page (i.e., login page) of the target application and the hyperlinks to the login page, and set the first page of crawling to the registration page. Thus, once the registration was completed successfully, the crawler could explore the target application continuously without the need to log in again. The proposed design of actions, however, requires no such restriction and enables the training to start from the home page of the target web application. This also suggests that the proposed design of actions can help solve the problem of agent training in experiments and achieve much faster convergence of the training process than our earlier work.

Overall, the answer to RQ4 is, "Yes, the new iRobot indeed can improve branch coverage by approximately 1.7% compared to our earlier work". Furthermore, it can make the training process converge faster and significantly reduce training time to 12.54%. The improvement in the training time is mainly due to the design of the actions. With the new set of actions, iRobot can explore webpages to a greater extent rather than learning which value to use for which field. Moreover, the invalid inputs help iRobot execute more error detection codes and improve code coverage.

4.5. Discussions

While our RL method can effectively guide web crawlers to improve code coverage of an application under test (AUT), it has some limitations. First, our current experiments depend on code coverage to train the agent. However, the code coverage tool is language-

dependent, as it needs to instrument the code under test. Web applications can use various programming languages such as node.js, Java, php, python, ASP.NET, etc. Therefore, we need different coverage tools for crawling web applications implemented with different languages. This observability issue is a major challenge for conducting more experiments in this study. In the future, we plan to use alternative methods besides code coverage in iRobot to explore and test web applications written in different programming languages. One possibility is to use page comparison [34] to measure the number of pages explored. This will be part of our future work.

Second, web applications may change later in the evolution or maintenance phase. Depending on how the changes are implemented, if the changes only causes minor changes to the DOM of the response pages, the iRobot training may still be applicable. Recall that the neural network, in a sense, acts as a type of nonlinear interpolator. However, if the changes modify the DOM of the response pages largely, iRobot may need to be retrained with additional actions in order to cover the new or modified code. Nevertheless, the proposed approach reduces the training time to 12.54% as compared to our previous study and thereby reduces the cost of retraining iRobot for web applications under development in the later stages.

Third, we populated the input fields with test data before submitting forms in the experiments. This arrangement might prevent some code paths from being executed in web applications. In our design, iRobot has actions to fill both valid and invalid input values. During the exploration period of RL training, iRobot chooses actions randomly. It has a certain probability of choosing actions a_1 or a_2 to provide invalid inputs. Therefore, it can also test code related to incorrect input values, albeit not completely. To test a wider variety of input types and combinations, we need to increase the entries in Tables 3 and 4. This part can be easily extended, though we do not show it in this paper.

Next, we obtained the input data encoded in the actions from a dictionary that we prepared manually for the AUT under study. For other AUTs under study, we need to identify both valid and invalid inputs for the form fields of the application to prepare an input dictionary. Although we can use the same dictionary for form fields in the same category and apply human testers' domain knowledge when preparing inputs, the cost can be high for complex web applications with many web forms and fields. Moreover, we expect that the performance of iRobot will suffer if the input dictionary becomes large, as it will need more actions to complete web forms successfully. In the future, we plan to use existing tools to automatically generate input values.

Finally, our study was based on only one AUT. It is crucial to know whether the proposed approach can be applied to other AUTs. To this point, we have conducted a preliminary study to check the applicability of our approach to other AUTs. The results are promising. As this work is still ongoing, we are unable to report our findings here.

The present design relies on the change of DOM or coverage vector to detect possible changes in dynamic webpages. Generally, when users input values into a web form and click the submit button, the DOM of the resulting page typically undergoes some degree of change to display the submission outcome, even in the case of invalid inputs. Otherwise, the user probably has no way to know the submission result. This change in the DOM may manifest as a popup alert, an error message, or a modification in the color of the invalid form field, achieved through the execution of certain source code. This is because, without changing the DOM of response page, users will find it difficult to visually see differences in the submission response. However, if the webpage uses a front-end code to detect input errors and to pop out an error message, iRobot will have no information on this situation.

Internet security and digital privacy have become important issues in e-services. In our context, we use web crawlers to automatically explore and test AUTs for code coverage. We fill in fields with artificial values, not real or confidential ones. Our approach does not gather or index the content of web pages across the Internet, so it does not raise any privacy issues. For Internet security, we test AUTs in a private and isolated network in our

experimental settings. Our approach does not depend on Internet resources. Therefore, security concerns are minimal.

5. Conclusions and Future Work

This paper introduces an innovative reinforcement learning approach aimed at guiding web crawlers in automatically selecting a sequence of input actions to maximize code coverage during exploration of a web application under test. Specifically, the proposed action design is capable of emulating human tester behavior, empowering the agent to efficiently populate input fields and enhance coverage by the crawling code. Furthermore, a convolutional neural network (CNN) is presented, and various reinforcement learning algorithms are employed to train the iRobot agent in selecting actions. Experimental results reveal that, with the proposed actions, the presented CNN network attains better code coverage compared to other neural networks when utilizing DQN or PPO algorithms. Additionally, in comparison to previous studies, iRobot demonstrates a notable increase in branch coverage by approximately 1.7% while concurrently achieving a significant reduction in training time to 12.54%.

In the future, we plan to conduct more experiments with different hyperparameter settings for the agent and to test different target applications. Additionally, we are currently investigating the use of word embedding to represent the features extracted from web pages to train iRobot to better understand the web pages and select proper input actions. Next, we also plan to extend this approach by training iRobot with a large number of web pages obtained from different applications to see if iRobot can gain some knowledge from previously trained applications. Finally, we plan to study the use of page comparison to measure the number of pages explored to replace the coverage vector in computing rewards.

Author Contributions: Conceptualization, C.-H.L. and S.D.Y.; methodology, C.-H.L. and S.D.Y.; software, Y.-C.C.; validation, C.-H.L., S.D.Y. and Y.-C.C.; formal analysis, C.-H.L. and S.D.Y.; investigation, C.-H.L. and Y.-C.C.; resources, Y.-C.C.; data curation, Y.-C.C.; writing—original draft preparation, C.-H.L.; writing—review and editing, C.-H.L. and S.D.Y.; visualization, Y.-C.C.; supervision, C.-H.L. and S.D.Y.; project administration, C.-H.L. and S.D.Y.; funding acquisition, C.-H.L. and S.D.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the National Science and Technology Council (NSTC) of Taiwan, under grant number NSTC 112-2221-E-027-049-MY2.

Data Availability Statement: The data presented in this paper are available upon request from the corresponding author.

Conflicts of Interest: Author Chiu, Y.-C. was employed by the company Phison Electronics Corp. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Deursen, A.V.; Mesbah, A.; Nederlof, A. Crawl-based Analysis of Web Applications: Prospects and Challenges. *Sci. Comput. Program.* **2015**, *97*, 173–180. [CrossRef]
2. Crawljax. Available online: https://github.com/zaproxy/crawljax (accessed on 25 October 2023).
3. Mesbah, A.; Deursen, A.V.; Lenselink, S. Crawling AJAX-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web* **2012**, *6*, 1–30. [CrossRef]
4. Wikipedia. Available online: https://en.wikipedia.org/wiki/Code_coverage (accessed on 10 January 2024).
5. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
6. Ho, W.-H. Training a Test Agent to Increase Code Coverage Based on DQN for Web Applications. Master's Thesis, National Taipei University of Technology, Taipei, Taiwan, 2018.
7. Arulkumaran, K.; Deisenroth, M.P.; Brundage, M.; Bharath, A.A. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Process. Mag.* **2017**, *34*, 26–38. [CrossRef]
8. Schmidhuber, J. Deep Learning in Neural Networks: An Overview. *Neural Netw.* **2015**, *61*, 85–117. [CrossRef] [PubMed]
9. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]

10. Sierla, S.; Ihasalo, H.; Vyatkin, V. A Review of Reinforcement Learning Applications to Control of Heating, Ventilation and Air Conditioning Systems. *Energies* **2022**, *15*, 3526. [CrossRef]
11. Waqar, M.; Imran; Zaman, M.A.; Muzammal, M.; Kim, J. Test Suite Prioritization Based on Optimization Approach Using Reinforcement Learning. *Appl. Sci.* **2022**, *12*, 6772. [CrossRef]
12. Lin, J.-W.; Wang, F.; Chu, P. Using Semantic Similarity in Crawling-Based Web Application Testing. In Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, 13–17 March 2017; pp. 138–148.
13. Alex, G. Coverage Rewarded: Test Input Generation via Adaptation-based Programming. In Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA, 6–10 November 2011; pp. 380–383.
14. Carino, S.; Andrews, J.H. Dynamically Testing GUIs Using Ant Colony Optimization. In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 138–148.
15. Kim, J.; Kwon, M.; Yoo, S. Generating Test Input with Deep Reinforcement Learning. In Proceedings of the IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST), Gothenburg, Sweden, 28–29 May 2018; pp. 51–58.
16. Liu, C.-H.; Chen, W.-K.; Sun, C.-C. GUIDE: An Interactive and Incremental Approach for Crawling Web Applications. *J. Supercomput.* **2020**, *76*, 1562–1584. [CrossRef]
17. Zheng, Y.; Liu, Y.; Xie, X.; Liu, Y.; Ma, L.; Hao, J.; Liu, Y. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In Proceedings of the 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; pp. 423–435.
18. Liu, E.Z.; Guu, K.; Pasupat, P.; Shi, T.; Liang, P. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. In Proceedings of the International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
19. Shi, T.; Karpathy, A.; Fan, L.; Hernandez, J.; Liang, P. World of Bits: An Open-Domain Platform for Web-Based Agents. In Proceedings of the 34th International Conference on Machine Learning (ICML), Sydney, Australia, 6–11 August 2017; pp. 3135–3144.
20. Sunman, N.; Soydan, Y.; Sözer, H. Automated web application testing driven by pre-recorded test cases. *J. Syst. Softw.* **2022**, *193*, 111441. [CrossRef]
21. Liu, Y.; Li, Y.; Deng, G.; Liu, Y.; Wan, R.; Wu, R.; Ji, D.; Xu, S.; Bao, M. Morest: Model-based RESTful API testing with execution feedback. In Proceedings of the 44th International Conference on Software Engineering, Pittsburhg, PA, USA, 25–27 May 2022; pp. 1406–1417.
22. Yandrapally, R.K.; Mesbah, A. Fragment-based test generation for web apps. *IEEE Trans. Softw. Eng.* **2022**, *49*, 1086–1101. [CrossRef]
23. Sherin, S.; Muqeet, A.; Khan, M.U.; Iqbal, M.Z. QExplore: An exploration strategy for dynamic web applications using guided search. *J. Syst. Softw* **2023**, *195*, 111512. [CrossRef]
24. Document Object Model (DOM) Technical Reports. Available online: https://www.w3.org/DOM/DOMTR (accessed on 20 October 2023).
25. OpenAI Gym. Available online: https://gym.openai.com/ (accessed on 18 August 2023).
26. Tensorflow. Available online: https://www.tensorflow.org/ (accessed on 15 September 2023).
27. TimeOff.Management. Available online: https://github.com/timeoff-management/application (accessed on 20 October 2023).
28. Istanbul. Available online: https://istanbul.js.org/ (accessed on 20 October 2023).
29. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level Control through Deep Reinforcement Learning. *Nature* **2015**, *518*, 529–533. [CrossRef] [PubMed]
30. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal Policy Optimization Algorithms. *arXiv* **2017**, arXiv:1707.06347v2.
31. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
32. Multilayer Perceptron. Available online: https://en.wikipedia.org/wiki/Multilayer_perceptron (accessed on 1 November 2023).
33. OpenAI Stable Baselines. Available online: https://github.com/hill-a/stable-baselines (accessed on 1 November 2023).
34. Page Compare. Available online: https://github.com/TeamHG-Memex/page-compare (accessed on 1 November 2023).