Can Cooperative Multi-Agent Reinforcement Learning Boost Automatic Web Testing? An Exploratory Study

Yujia Fan[1,2], Sinan Wang[1], Zebang Fei[2], Yao Qin[2], Huaxuan Li[2], Yepang Liu[1,2,*]
[1]Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
[2]Department of Computer Science and Engineering, Southern University of Science and Technology
Shenzhen, China
{12431253,wangsn,12110608,12112016,12112045}@mail.sustech.edu.cn
liuyp1@sustech.edu

Abstract

Reinforcement learning (RL)-based web GUI testing techniques have attracted significant attention in both academia and industry due to their ability to facilitate automatic and intelligent exploration of websites under test. Yet, the existing approaches that leverage a single RL agent often struggle to comprehensively explore the vast state space of large-scale websites with complex structures and dynamic content. Observing this phenomenon and recognizing the benefit of multiple agents, we explore the use of Multi-Agent RL (MARL) algorithms for automatic web GUI testing, aiming to improve test efficiency and coverage. However, how to share information among different agents to avoid redundant actions and achieve effective cooperation is a non-trivial problem. To address the challenge, we propose the first **MAR**L-based web **G**UI testing system, MARG, which coordinates multiple testing agents to efficiently explore a website under test. To share testing experience among different agents, we have designed two data sharing schemes: one centralized scheme with a shared Q-table to facilitate efficient communication, and another distributed scheme with data exchange to decrease the overhead of maintaining Q-tables. We have evaluated MARG on nine popular real-world websites. When configuring with five agents, MARG achieves an average increase of 4.34 and 3.89 times in the number of explored states, as well as a corresponding increase of 4.03 and 3.76 times in the number of detected failures, respectively, when compared to two state-of-the-art approaches. Additionally, compared to independently running the same number of agents, MARG can explore 36.42% more unique web states. These results demonstrate the usefulness of MARL in enhancing the efficiency and performance of web GUI testing tasks.

Keywords

Web Testing, Multi-Agent Reinforcement Learning, Automatic GUI Testing, Information Sharing

*Yepang Liu is the corresponding author of the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694983

ACM Reference Format:
Yujia Fan[1,2], Sinan Wang[1], Zebang Fei[2], Yao Qin[2], Huaxuan Li[2], Yepang Liu[1,2]. 2024. Can Cooperative Multi-Agent Reinforcement Learning Boost Automatic Web Testing? An Exploratory Study. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3694983

1 Introduction

Web applications offer users convenient browser-based access to software services across different platforms [49] and are widely-used for various purposes. It is essential to ensure the software quality of web applications. Automatic web GUI testing aims at exploring a given website under test (WUT) without human intervention to achieve a comprehensive functional coverage and detect potential misbehaviours of the website [11]. By employing various meta-heuristic strategies, a *testing agent* can autonomously explore the website to meet the pre-defined testing objectives.

Reinforcement learning (RL) is a popular technique for driving the GUI testing of web applications [11]. For example, *WebExplor* [52] formulates web testing as a Markov decision process (MDP) and solves it with a value-based RL algorithm, Q-learning. By employing a curiosity-based reward function, the Q-learning algorithm guides *WebExplor* to efficiently explore diverse behaviors of the WUT. *QExplore* [38] is another Q-learning-based web GUI testing technique, which adopts a different state representation method than *WebExplor*. The effectiveness of RL-based web GUI testing approaches mostly relies on the exploration abilities of the RL agents in the dynamic web GUI contexts with the goal of achieving higher coverage and detecting more faults. Besides the outstanding results achieved on web GUI testing, RL algorithms also outperform traditional testing approaches for many other types of software applications, such as mobile apps [19, 20, 35, 44, 45] and desktop applications [7, 10, 25, 26].

Recently, Fan et al. conducted a large-scale empirical study to analyze the performance of Q-learning-based web GUI testing approaches [11]. They investigated the combinations of various RL components and Q-learning parameters, with a total of 216 different configurations. Their empirical findings reveal that none of the explored RL configurations can fit all the websites used in the study. Moreover, they showed that Q-learning can quickly reach a saturation status when exploring small-sized websites, whereas exploring large-scale websites inevitably requires much more testing resources. Their research calls for the enhancement on the current single-agent RL-based GUI testing techniques.

Observing the potential performance upper bound of a single-agent approach, we in this paper explore the use of multiple agents for efficient and effective web GUI testing. However, designing a practically useful multi-agent system to achieve a comprehensive testing of web applications is a non-trivial task. A straightforward solution is to run multiple single-agent processes in parallel, as proposed in GT^{PQL}[32]. Given the inherent randomness brought by an ε-greedy strategy [42], it is possible for multiple agents to cover different functional points in the same website. However, as we will show in a motivational study (Section 3), under this simple setting, only a small amount of web states can be exclusively visited by each single agent (i.e., the agents visit a large number of identical states), leading to insignificant performance improvement and waste of testing resources. To address the problem, GT^{PQL} synchronizes the Q-models of parallel Q-learning agents at the end of each epoch to facilitate information communication. Nevertheless, the complexities of epoch division, the lack of communications among agents within each epoch, and the overhead of model synchronization may diminish the practical utility of the tool. A more effective agent cooperation mechanism is needed to mitigate the redundancies in the webpage exploration of multiple agents and improve the overall performance of web GUI testing.

Motivated by the intrinsic limitations of existing single-agent RL-based testing approaches, and the weaknesses of the naive parallelism approach, we strongly advocate the necessity of an efficient multi-agent reinforcement learning (MARL) based approach for improving the efficiency and effectiveness of web GUI testing. To this end, we made the first attempt of utilizing MARL algorithms for coordinating multiple Q-learning testing agents to collectively explore the same WUT. Our prototype tool, named MARG, is an **MAR**L-based web **G**UI testing system that allows a configurable number of Q-learning agents to simultaneously test the same website without human intervention. Specifically, MARG adopts a client-server architecture, in which the agents interact with different webpages and the controller coordinates the testing process. To facilitate the exchange of experiences among various Q-learning agents within the MARL system, we have developed two data sharing schemes. These schemes operate under both centralized and decentralized settings, allowing agents to share Q-values derived from their policies.

We have implemented MARG and evaluated it on nine commercial websites (such as YouTube and GitHub). By comparing MARG against two state-of-the-art RL-based techniques, *WebExplor* [52] and *QExplore* [38], we show that an MARL-based approach that leverages plain Q-learning algorithms can already significantly outperform existing single-agent techniques that leverage extra methods (such as using a DFA to provide high-level guidance [52] and a contextual data input method to generate textual inputs [38]) beyond RL. When running five agents, MARG can explore 4.34 and 3.89 times more states, and detect 4.03 and 3.76 times more failures, respectively, than *WebExplor* and *QExplore*. It can also explore 36.42% more states than a non-cooperative approach that simply runs five parallel Q-learning agents. Additionally, by increasing the number of agents, which can be constrained by computational resources, the performance of MARG can be further increased. These promising results demonstrate the great potential of applying MARL to boost the performance of web GUI testing.

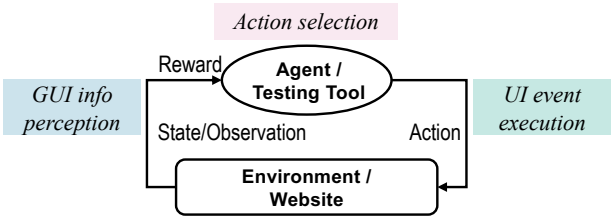


Figure 1: A web testing loop under the MDP formulation

The contributions of this paper are summarized as follows:

- We propose a new web GUI testing approach that employs asynchronous MARL to drive multiple Q-learning agents to collaboratively test web applications. To the best of our knowledge, we make the first attempt in devising efficient agent communication mechanisms to enable effective multi-agent web GUI testing.
- We have implemented our approach in a prototype system MARG with two types of data sharing schemes to support both centralized and decentralized settings, which can provide references for future research endeavors.
- We have evaluated MARG using real-world commercial websites. Our experiments show that MARG can significantly outperform two state-of-the-art RL-based methods, *WebExplor* and *QExplore*, as well as a non-cooperative multi-agent testing approach that runs Q-learning agents in parallel.

The remainder of this paper is organized as follows. Section 2 introduces background knowledge about RL for GUI testing and MARL algorithm. In Section 3, we demonstrate the limitation of single-agent RL-based approaches to motivate the necessity of MARL-based web GUI testing. Section 4 depicts the detailed design of our proposed multi-agent approach. Section 5 presents the experimental setup for evaluating our approach and the results are discussed in Section 6. We review the related work in Section 7 and finally conclude the paper in Section 8.

2 Preliminaries

2.1 Reinforcement Learning for GUI Testing

Formally, an MDP instance is defined by a 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ [5]. In the context of web GUI testing, we call the software under test, i.e., the website, as an *environment* while the testing process is carried out by an *agent*. The interaction between the environment and the agent is depicted in Figure 1. In the 5-tuple, a *state* $s \in \mathcal{S}$ represents how the testing agent observes information from the current webpage. At each state, the *action space* $\mathcal{A}$ encompasses human-like interactions with the web GUI elements, such as clicking buttons or links, filling input boxes, and so on. After an *action* $a \in \mathcal{A}$ is performed, there will be a transformation of webpages that follows the *transition function* $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, which depends on the functionalities and business logic of the website. After an action is applied at time t, the agent will be provided with an immediate reward r_t based on a predefined reward function $\mathcal{R}$ to guide better GUI exploration. Such reward is typically designed based on state changes [25, 26] or the execution frequencies of actions [35, 52]. Additionally, it also needs to define a *discount factor* $\gamma \in [0, 1]$.

Lower values of γ prioritize immediate rewards, while higher values consider long-term rewards. With such a formulation, the goal of an RL agent is to learn to maximize the accumulated rewards during its exploration in the environment: $r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \cdots$.

2.2 Multi-Agent Reinforcement Learning

An MARL system can be *competitive* or *cooperative* [50]. In competitive MARL, the agents have conflicting objectives, in which the gross reward among agents is usually zero-summed [23]. Most research work on competitive MARL focuses on two-player games, such as AlphaGo [39, 40], a representative breakthrough in the game of Go. In cooperative MARL, agents collaborate and coordinate their actions to achieve a common goal, thus maximizing the overall team reward. This is applicable in team-based robotics [9], traffic management [4], etc. During web GUI testing, the testing agents should be coordinated to maximize test efficiency and collaboratively explore as many diversified webpages as possible. As such, it is a typical application scenario of cooperative MARL.

A key challenge of implementing an efficient MARL-based system is to properly design the system's information structure and the communication mechanism between agents. Generally, an MARL system can be classified into three types according to their communication styles as shown in Figure 2, namely, fully decentralized, decentralized with networked agents, and centralized settings [50]:

(a) In a *fully decentralized setting* (Figure 2a), agents autonomously make decisions and interact with the environment based on their individual observations and their own policies. This decentralized architecture simplifies system design and reduces the complexity of coordination. However, the lack of information exchange prevents agents from utilizing each other's knowledge and experience, which limits the overall performance [43].
(b) When the agents are connected through a time-varying communication network, it becomes possible to disseminate local information of an agent over the entire network [24, 46, 51]. It is called a *decentralized setting with networked agents* (Figure 2b). However, within this information structure, not every pair of agents is required to exchange information at every instant.
(c) In situations where it is necessary for all agents to consistently share information, a central controller can be employed, resulting in the *centralized setting* (Figure 2c). This controller aggregates information from the agents and makes decisions for all agents on behalf of the entire MARL system [13–16].

For the RL agents in an MARL system, they shall update their own policies based on their observed state transitions. This can be done in a *synchronous* manner, where the agents temporarily halt their explorations, sequentially update their policies, and then resume the next exploration step. In comparison, in an *asynchronous* MARL system, agents update their policies independently and asynchronously without waiting for other agents to update theirs [30]. In web GUI testing, the RL agents explore different paths and interact with the web application using their dedicated browser instances. In such a scenario, asynchronous exploration allows the agents to cover a wider range of the web application's functionalities and achieve a better overall coverage.

3 Motivational Study

Recently, an empirical study conducted by Fan et al. analyzed the effectiveness of Q-learning-based web GUI testing approaches [11]. The study pointed out a performance limit for single-agent Q-learning approaches: it is difficult for a single agent to achieve high functional coverage for the WUT, especially when the testing resources are constrained. To tackle this problem, i.e., improving efficiency and achieving higher coverage, a possible solution is to run multiple testing agents simultaneously. To investigate the feasibility of the idea and understand the challenges during the process, we performed a pilot study to see whether the parallelism of agents can effectively improve the test coverage of web applications.

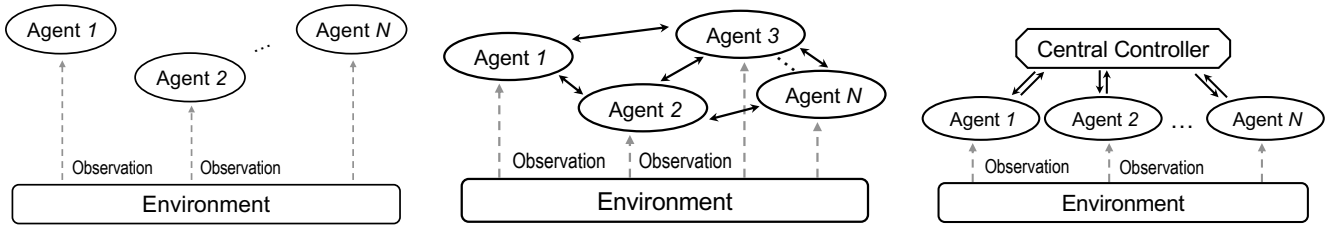
• **Implementation**: According to the Q-learning-based web GUI testing framework proposed in Fan et al.'s work [11], we implemented our own web testing agents on top of Selenium [2]. Specifically, following Fan et al.'s approach, we abstracted a web state as the set of all GUI elements on the corresponding webpage and selected actions according to an ε-greedy scheme. The Q-learning agents will obtain a numerical reward based on curiosity, which has been shown to be an effective choice for defining reward functions in existing studies [7, 10, 38, 44, 52].

• **Setup**: For the pilot study, we ran our implemented Q-learning agents on a complex commercial website[1]. This commercial website is a web portal that contains more than one hundred hyperlinks, each leading to a new subpage providing different web services. Moreover, most of these subpages do not require user login before visiting, making the background status identical for all parallel agents. Besides, this website has a sufficient number of different web states, which allows the measurement and comparison of the testing adequacy achieved by each agent, as well as assessing their collective performance. For the experiment, we ran three testing agents in parallel for two hours and compared their visited states at the end.

• **Result**: Figure 3 presents the number of visited states achieved by each of the three Q-learning agents. It shows that within a two-hour testing period, the three agents discovered 564 different web states, while each contributed to 59%, 76%, and 71% of the total states, respectively. Besides, from the Venn diagram, we can make two more observations. First, among the 564 web states, 36% (=(93+72+39)/564) of them were discovered exclusively by a single agent. Second, 42% (=236/564) of the states were visited by all three agents. From the above observations, we can see that it is indeed feasible to leverage a multi-agent approach to improve the coverage of web GUI testing, so as to increase the possibility of detecting potential faults. However, without effective agent communications, there would a considerable amount of redundant exploration among the parallel agents, resulting in the waste of testing resources.

• **Conclusion**: Based on the above analysis, it is evident that reducing the redundancies in the states visited by the testing agents has a great potential of improving the overall performance of web GUI testing. Motivated by this, in our work, we will focus on harnessing multi-agent approaches for web GUI testing and devising practical strategies to facilitate the cooperation among the testing agents, enabling them to efficiently share information and collaboratively explore diversified webpages to quickly achieve high test coverage.

[1]For commercial reasons, the website is anonymized.



(a) Fully decentralized setting (b) Decentralized setting with networked agents (c) Centralized setting

Figure 2: Three representative information structures in MARL [50]

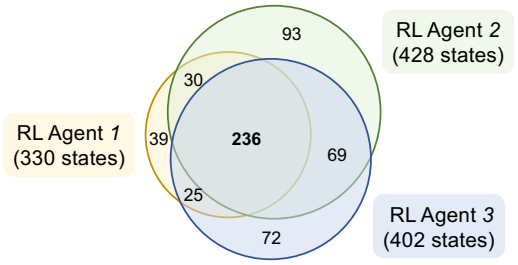


Figure 3: The Venn diagram of the sets of web states visited by the three Q-learning-based testing agents

4 Approach

4.1 Problem Formulation

MARL can be formulated as a multi-agent extension of the MDP described in Section 2.1, as denoted by $\langle N, \mathcal{S}, \{\mathcal{O}_i\}, \{\mathcal{A}_i\}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ [48], where $i \in \{1, \ldots, N\}$ corresponds to the i-th agent in the system of N agents. The $\mathcal{O}$ stands for *observation*, which refers to the information that an agent perceives about the environment. Typically, the state space $\mathcal{S}$ is defined by the Cartesian product of observation space of each agent, i.e., $\mathcal{S} = \mathcal{O}_1 \times \mathcal{O}_2 \times \cdots \times \mathcal{O}_N$. However, this formulation presents the challenge of state explosion, particularly in the context of web GUI testing, as webpage states exhibit high variability.

In an asynchronous MARL system, it is essential to consider which information the agents need to communicate in order to achieve better cooperation and coordination. Instead of coordinating the joint exploration status among all agents, the emphasis should be placed on exchanging their historical experiences to avoid redundant exploration. Therefore, we define the ***state space*** in our problem as: $\mathcal{S} = \mathcal{O}_1 \cup \mathcal{O}_2 \cup \cdots \cup \mathcal{O}_N$. Such a setting allows the MARL algorithm to prioritize the decisions that agents should make when observing specific webpages, without considering the states of the other agents at the same time. In other words, the state space in our problem consists of all states observed by all the agents in the system. To simplify presentation, we will use the symbol s to represent both the concepts of *state* and *observation*.

We extract valid UI events set as a representation of the ***state*** $s = (\text{URL}, E_1, E_2, ..., E_n)$, which also represents the ***action space*** $\mathcal{A}_s = \{E_1, E_2, ..., E_n\}$ within this observed state. Here, E_k is a UI event and n is the number of unique UI events that can be triggered on a given webpage. This abstraction method borrows the idea of combined state representation employed by *WebExplor* [52], while we replace its tag sequence as the set of actions, which has been shown to be more effective [11]. It provides the benefit of simplifying the state space while retaining essential information. Moreover, it aligns with the state-action pairs in the tabular Q-learning framework.

During testing, the state transition function $\mathcal{P}$ is implicit as the agent lacks direct access to navigation information. Instead, the agent learns an approximate $\mathcal{P}$ by observing state transitions through interactions with the website. When an agent in the multi-agent system executes an action $a \in \mathcal{A}_s$ based on the state $s \in \mathcal{S}$, the webpage undergoes changes, resulting in a new state $s' \in \mathcal{S}$ and an immediate ***reward*** r based on the reward function $\mathcal{R}$:

$$r = \mathcal{R}(a) = \frac{1}{Count(a)} \tag{1}$$

This *curiosity* reward function has been demonstrated to be effective [11, 38, 52], where $Count(a)$ represents the number of times the action a has been applied within the multi-agent system during the testing session.

4.2 An Overview of MARG

Figure 4 shows the overview of MARG. In this system, each agent can perform automatic GUI testing of the WUT on its own browser instance. Each testing step of an individual agent is divided into four phases: ❶ GUI information perception, ❷ policy optimization, ❸ action selection, and ❹ UI event execution. The phases of GUI information perception and UI event execution are performed by the *Interaction* component, which is handled independently by each agent. On the other hand, the phases of policy optimization and action selection are grouped as a *Decision-making* component managed by a controller that facilitates experience sharing. To achieve reliable and asynchronous information transmission between the agents and the controller, our system employs a client-server architecture [21] and utilizes the HTTP protocol [12] for data transfer.

During a testing session, each agent independently detects interactable elements on its observed webpage. Such elements can be clickable buttons or links, text boxes or multiple-selection boxes. These elements form the action space, which, combined with the webpage's URL, serves as the agent's state, as defined in Section 4.1. Additionally, the agent discerns the categories of the current testing step, generates corresponding request messages, and sends the messages to the controller (i.e., the server). Different categories of steps correspond to different processing logics, which will be elaborated in Section 4.3. With the requests from the agent, the controller obtains transition information for updating global data and optimizing strategy based on extended Q-learning algorithms.

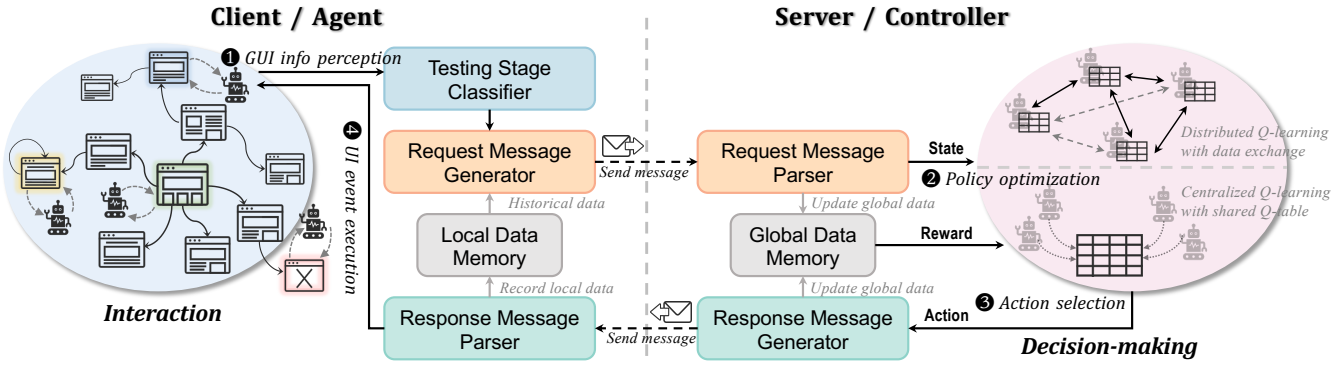


Figure 4: An overview of MARG

Meanwhile, the current GUI information, including state and action space, will be used to select an appropriate action. The detailed algorithms will be introduced in Section 4.4. Subsequently, the controller will send a response message to the corresponding agent, which contains the selected action along with other necessary information. Upon receiving the response, the agent executes the selected action, which is a UI event, updates its local data memory, and proceeds to the next step.

4.3 Testing Stages and Messages

During the continuous interaction with the WUT, a testing agent often faces different stages that should be handled with different methods. For example, at the beginning of a testing session, as the agent does not possess any knowledge about the website, it cannot properly update its action policy. As another example, when some exceptional situations occur (e.g., crashes), the agent should have means to recover from the erroneous status. In MARG, at each step in the testing loop, the agent will recognize its current *stage*. There are four stages defined in MARG: initial, normal, failure, and stagnation stages. Based on the stage, the agent sends a corresponding message to the controller to obtain optimal action in the current step. The classification of stages, request message formats, and expected response message content are shown in Table 1.

• *Initial Stage* refers to the beginning of the testing process. With no prior visited states or applied actions, the agent simply needs to provide the controller with the state and action space of the current webpage. After receiving the initial request message, the controller checks the global state list to determine if the state has appeared in the testing history of the entire system. If the state exists, its corresponding index is retrieved. Otherwise, the state is added to the global state list along with its action space information. Then, the system retrieves the corresponding index and feeds it into the policy function to obtain an action. Finally, the controller encapsulates the state index and the selected action into a response message, and returns it to the agent.

• *Normal Stage* refers to a normal interaction during testing, which involves a complete state transition. As shown in Table 1, agent sends the index of previous state and action that navigate to the current state, along with GUI information of current webpage, to the controller. After the controller acquires the state index based on the GUI information and calculates the action execution count to compute reward r using Equation (1), the tuple (s, a, s', r) can update the policy function. The process of action selection and response message generation is the same as that in the initial stage.

• *Failure Stage* encompasses situations where the agent encounters obstacles preventing smooth testing progress. It may occur when the webpage 1) lacks interactive elements, 2) is inaccessible due to broken links, server errors, or access restrictions, 3) is from external websites that are not within the defined testing domain. In these situations, the agent shares with the controller the previous state and action that resulted in current webpage, without providing detailed information about the webpage itself. Thus, the controller can penalize the Q-values to indicate that executing the action in that state is unfavorable. Different from the action selection process in normal stage, in such cases, the controller returns a URL with lowest visit count for the agent to restart from. This helps the agent recover from abnormal status and facilitates further exploration.

• *Stagnation Stage* occurs when the agent becomes trapped in a local loop, meaning that it has executed actions more than *threshold* times without triggering a new state. Similar to the failure stage, the agent will restart the testing process from the least visited webpage. Additionally, this stage involves a complete state transition, thus the information shown in Table 1 should also be synchronized to the controller for policy optimization.

The advantage of this classification method is that the controller only needs to determine the decision category of the agent based on the type of request message, without having to retain excessively detailed local information of the agent. This simplifies the task of the controller, enabling it to handle and respond to the request messages from agents more efficiently.

4.4 Multi-Agent Q-Learning

As mentioned above, the controller is responsible for two key phases: policy optimization and action selection. For single-agent GUI testing, previous studies [10, 19, 25, 35, 38, 44, 52] have demonstrated the effectiveness of the Q-learning [47] algorithm in updating the action policy:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r_t + \gamma \max_{a' \in \mathcal{A}_{s'}} Q(s',a') - Q(s,a)] \quad (2)$$

When it comes to action selection, ε-greedy is widely employed to strike a balance between exploration and exploitation [25, 35, 44].

Table 1: Attributes in the HTTP messages at different stages and the corresponding responses

Stage	Message	Agent ID	Previous State Index	Executed Action	Current Observation	Current Action Space	Current State Index	Selected Action
Request	Initial	✓			✓	✓		
	Normal	✓	✓	✓	✓	✓		
Response	-	✓					✓	UI event
Request	Failure	✓	✓	✓				
	Stagnation	✓	✓	✓	✓	✓		
Response	-	✓						restart URL

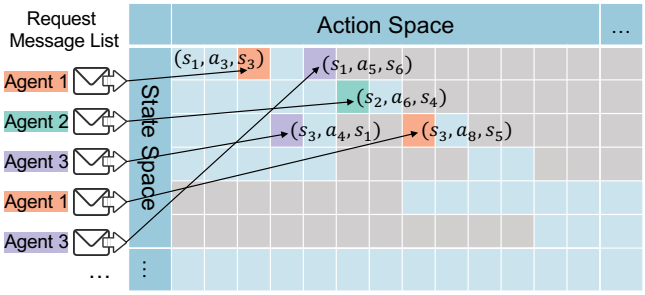


Figure 5: An illustration of the shared Q-table scheme

This strategy randomly selects an action from the action space with a probability of ε, while exploiting prior experience by choosing the highest-scored action with a probability of $1-\varepsilon$:

$$a^* = \begin{cases} \underset{a' \in \mathcal{A}_s}{\text{argmax}}\, Q(s, a') & \text{with probability } 1-\varepsilon, \\ \text{random} \in \mathcal{A}_{s'} & \text{with probability } \varepsilon. \end{cases} \tag{3}$$

4.4.1 Centralized Q-learning with a shared Q-table. When there are multiple RL agents relying on a centralized controller for decision-making, it becomes intuitive to record all state-action pairs and the corresponding Q-values into a shared Q-table. Figure 5 provides an illustration of how the controller optimizes the RL policy using a shared Q-table. Upon parsing the response message and accessing the global data memory, the controller obtains a 4-tuple (s, a, s', r), enabling it to update the value of $Q(s, a)$ according to Equation (2). Referring to Figure 5, assuming there are three agents in the system. The first request message from Agent 1 requires the controller to update $Q(s_1, a_3)$. Suppose that the updated Q-value $Q(s_1, a_3)$ is lower than its previous value, leading to $Q(s_1, a_5)$ becoming the highest Q-value. Consequently, when Agent 3 visits the state s_1 after taking action a_4 in state s_3, the probability of selecting $Q(s_1, a_5)$ becomes highest. If the controller chooses action a_5 for Agent 3, it will receive a request message from Agent 3, as depicted by the fifth message in Figure 5, which leads to an update of $Q(s_1, a_5)$.

In this way, all agents within the system can share testing experiences through the shared Q-table. When an agent receives a reward for a state-action pair, the corresponding value in Q-table is updated accordingly. As a result, when other agents visit the same state, they will be influenced by the experiences and prioritize actions with higher Q-values. In other words, they exhibit a preference for actions that have previously demonstrated higher rewards across all agents' experiences. This mechanism of experience sharing facilitates mutual learning and leverage among all agents, thereby improving the overall system performance and effectiveness.

Algorithm 1: Distributed Q-learning for the k-th agent

Input: Agent k: previous state s, executed action a, current state s', action space of current state $\mathcal{A}_{s'}$, reward r
Global: number of agents N, Q-tables $Q_1, Q_2, ...Q_N$, and hyperparameters $\alpha, \gamma, \varepsilon$
Output: Q_k, chosen action a'

1 **if** s' *not in* Q_k **then**
2 initialize $Q_k(s', \mathcal{A}_{s'})$
3 **for** $j = 1, ..., N; j \neq k$ **do**
4 **if** s' *in* Q_j **then**
5 *tempQ*.append($Q_j(s', \mathcal{A}_{s'})$)
6 **if** *tempQ is empty* **then**
7 update $Q_k(s, a)$ using Equation (2)
8 Choose a' from $\mathcal{A}_{s'}$ using Equation (3) on $Q_k(s', \cdot)$
9 **else**
10 update $Q_k(s, a)$ using Equation (4)
11 **for** a_i *in* $\mathcal{A}_{s'}$ **do**
12 $Q^*_{s'}(a_i) = \Sigma_{Q_j \in tempQ} Q_j(s', a_i)$
13 Choose a' from $\mathcal{A}_{s'}$ using Equation (3) on $Q^*_{s'}(\cdot)$

4.4.2 Distributed Q-learning with data exchange. The shared Q-table scheme is simple and easy to implement. More importantly, it allows timely updates of testing experiences among all agents. However, it may become inefficient as the scale of the Q-table grows. Therefore, we consider an alternative approach, where each agent maintains its own Q-table and updates it on demand. This approach, which we called distributed Q-learning, is shown in Algorithm 1.

When the controller receives a message from the k-th agent, it parses the message to obtain the tuple (s, a, s', r). Then, it searches the Q-table for agent k to check if there are Q-values for the current state s'. If there are no Q-values for s', the controller initializes $Q_k(s', \mathcal{A}_{s'})$ with an initial Q-value (lines 1-2).

Next, the controller optimizes the policy for agent k by updating $Q_k(s, a)$. Before calculating the new value of $Q_k(s, a)$, the controller gathers information about the current state s' from the Q-tables of other agents. Specifically, if s' is present in the Q-table of the agent j such that $j \neq k$, the value $Q_j(s', \mathcal{A}_{s'})$ is saved into a temporary table *tempQ* (lines 3-5). In case no such state s' exists (line 6), meaning that the state s' which agent k is currently visiting

has not been visited before in the testing process of the entire system, there is no experience data available. Therefore, agent k relies solely on its own Q-table to estimate the optimal action-value function, updating $Q_k(s,a)$ by following the Q-learning algorithm in Equation (2) (line 7). Additionally, the controller chooses an action from its action space $\mathcal{A}_{s'}$, by employing the ε-greedy (Equation 3) strategy on the Q-values for state s' , denoted as $Q_k(s',\cdot)$ (line 8).

When there were past experiences on state s' learned by other agents, the agent k should update its own policy. Inspired by double Q-learning [17], we update the Q-values as follows:

$$Q_k(s,a) \leftarrow \alpha[r + \frac{\gamma}{l} \sum_{Q_j \in tempQ} Q_j(s', \underset{a' \in \mathcal{A}_{s'}}{\operatorname{argmax}}(Q_k(s',a'))) - Q_k(s,a)] + Q_k(s,a) \tag{4}$$

where l is the length of the *tempQ* table. It updates the $Q_k(s,a)$ for agent k based on reward r, the discounted future rewards obtained from the other agents' experiences, and the current Q-value itself. It calculates the average of the Q-values for the actions with the highest value in $Q_k(s',\cdot)$ among the other agents' Q-tables. After updating the policy for agent k, the controller will choose an action to be executed. In order to make an informed decision, it recalculates the Q-values for each action in $\mathcal{A}_{s'}$ based on the experiences of other agents on state s' (line 12). Then, it utilizes ε-greedy strategy to choose an action on $Q^*_{s'}(\cdot)$ (line 13).

5 Experimental Setup

5.1 Implementation and Environment

MARG contains two main components (Figure 4): The client-side agents are implemented on top of Selenium-Java [2] for webpage interactions; as for the server side, the controller consists of two Q-learning algorithms implemented in pure Python, while the data is stored in a MySQL database. The HTTP communication is implemented using the Flask framework [1] as a RESTful server.

Regarding the two multi-agent Q-learning algorithms, we developed two versions of our proposed tool, namely MARG^N_C (i.e., Centralized Q-learning with a shared Q-table) and MARG^N_D (i.e., Distributed Q-learning with data exchange), which correspond to the centralized Q-learning and the distributed Q-learning schemes, respectively. Here N represents the number of agents.

We conducted all the experiments on multiple desktop devices with identical configurations, each of which has an Intel i7-13700 CPU and 32GB RAM. The devices are connected with 1Gbps Ethernet to ensure stable network conditions. During the experiments, we ran all agents in the same MARL system on the same device for better communication efficiency.

5.2 Baseline Approaches

To evaluate MARG, we first selected *WebExplor* [52] and *QExplore* [38], two state-of-the-art web GUI testing tools, as the baseline approaches. *WebExplor* [52] converts an HTML document to a sequence of tags and adopts the gestalt pattern matching algorithm for state abstraction. It uses a deterministic finite automaton (DFA) to provide high-level guidance. *QExplore* [38] defines the state of a web application as the set of actions (e.g., button clicks) on a webpage, which is similar to MARG, and involves a contextual data input method to generate textual inputs.

Besides, to evaluate the performance improvement brought by the data sharing mechanism, we also included a baseline method that runs multiple Q-learning agents without any information exchanging. We call this baseline IQL^N (Independent Q-Learning [27]), where N represents the number of agents.

It is worth explaining that we did not run multiple instances of *WebExplor* or *QExplore* in parallel for comparison due to two primary reasons. First, the main focus of our work is to devise agent cooperation strategies to improve the efficiency and performance of web GUI testing. Hence, our essential task in the evaluation is to compare MARG with a multi-agent approach without effective agent cooperation (i.e., the IQL^N approach mentioned above). Second, both *WebExplor* and *QExplore* leverage extra methods (e.g., DFA or contextual data input method) beyond RL, it would be unfair to MARG, in which each agent only leverages plain Q-learning algorithms to guide webpage exploration, if we compare it with parallelly running multiple instances of *WebExplor* or *QExplore*.

5.3 Research Questions

To evaluate MARG, we conducted experiments to investigate the following three research questions:

- **RQ1 (Tool Performance)**: How does MARG compare with the state-of-the-art web GUI testing methods? Additionally, can cooperative MARL techniques (i.e., MARG^N_C and MARG^N_D) achieve a better performance than independently executing multiple agents (i.e., IQL^N)?
- **RQ2 (Comparison of Data Sharing Schemes)**: Is there a performance difference between MARG^N_C and MARG^N_D? If so, what are the contributing factors of this difference?
- **RQ3 (Effect of Agent Numbers)**: How does the number of agents affect the performance of MARG?

5.4 Configurations

Table 2 summarizes all compared approaches. During the experiments, we followed the settings from existing work [38, 52], giving the same time budget of two hours to all these approaches. To mitigate randomness, we repeated each experiment 3 times and calculated the average results.

The parameters for *WebExplor* and *QExplore* were set according to the respective papers. As for IQL^N, MARG^N_C, and MARG^N_D, we set the parameters $\alpha = 1$, $\gamma = 0.5$, $\varepsilon = 0.5$. These parameter values have been evaluated and demonstrated good performance and stability in a recent empirical study [11]. In Algorithm 1, the initial Q-value is set to 10.0 based on a pilot experiment.

For RQ1 and RQ2, the number of agents in MARG's system was set to five. As for RQ3, we conducted experiments using different numbers of agents, with N being set to 3, 5, 8, 12 and 15, respectively.

5.5 Subject Websites

Multi-agent approaches are more suitable for testing large-scale websites (i.e., they would be an overkill for small-sized websites). For this reason, we selected eight popular real-world websites with world-wide influence from Semrush rankings [3] for our experiments. In order to comprehensively evaluate the performance of MARG on different types of websites, we randomly selected websites from different categories, with details of the eight websites

Table 2: Details of the compared approaches

Approach	Unique Feature	State	Reward	γ	ε	α
WebExplor	DFA guidance	(URL, tag sequence of *html_doc*)	$\frac{1}{\sqrt{Count(s,a,s')}}$	0.95	-	1
QExplore	Contextual data input method	$(E_1, E_2, ..., E_n)$	$\begin{cases} R_{max} & \text{if } Count(a) = 0 \\ \frac{1}{\sqrt{Count(a)}} & \text{if } Count(a) > 0 \\ R_{negative} & \text{if } s \text{ is not valid} \end{cases}$	$0.9 \times e^{-\frac{\lvert A_{s'} \rvert - 1}{10}}$	0	
IQL^N	Independently run multiple Q-learning agents	(URL, $E_1, E_2, ..., E_n$)	$\frac{1}{\sqrt{Count(a)}}$	0.5	0.5	
$MARG_C^N$	Centralized multi-agent Q-learning with a shared Q-table					
$MARG_D^N$	Distributed multi-agent Q-learning with data exchange					

Table 3: Details of subject websites

Name	Category	URL
WUT_A	*Anonymous.*	*Anonymous.*
toppr	Distance Learning	https://www.toppr.com/
Smadex	Advertising and Marketing	https://smadex.com/
Vuestic	Education	https://ui.vuestic.dev/
YouTube	Newspapers	https://youtube.com/
GitHub	Software and Development	https://github.com/
GameSpot	Computer and Video Games	https://www.gamespot.com/
EatingWell	Food and Beverages	https://www.eatingwell.com/
IKEA	Online Services	https://www.ikea.com/

listed in Table 3. We also included the website used in the pilot study (Section 3), which is anonymized and denoted as WUT_A.

5.6 Metrics

Considering that these RL-based approaches are driven by curiosity, wherein the fundamental principle revolves around exploring more webpages to increase the possibility of finding failures, we evaluated web GUI testing approaches from two perspectives: exploration capability and testing effectiveness. In terms of the exploration capability, we used metrics inspired by the evaluation of web crawling methods [29]. Specifically, we recorded the number of *explored states*, *detected actions* (including executed ones), and *executed unique actions*. Due to the different state abstractions among these compared approaches, we unified their states as (URL, $E_1, E_2, ..., E_n$) to statistically analyze the number of explored states. For evaluating testing effectiveness, we collected *detected failures*, which involves capturing and analyzing the triggered crashes and console errors during the testing process.

6 Results and Discussion

6.1 RQ1: Tool Performance

The averaged results among three repeated runs are summarized in Table 4, where bold numbers indicate the best results. It can be observed that running multiple agents significantly improves the coverage and efficiency of the tests. Specifically, based on average results, equipped with five agents, $MARG_D^5$ demonstrates excellent performance across most of the metrics. In terms of the number of explored states, it surpasses *WebExplor* and *QExplore* by factors of 4.34 (=661.5/152.4) and 3.89 (=661.5/170.1), respectively. As for executed unique actions, the factors are 3.03 (=626.1/206.3) and 2.46 (=626.1/254.4). The increase in the explored states and executed actions contributes to the improved testing outcomes: MARG detects 4.03 and 3.76 times more failures than the two baseline methods.

When comparing $MARG_C^5$ and $MARG_D^5$ with the baseline approach IQL^5, it shows that the performance of MARG surpasses that of the baseline on most websites. For instance, there is a 36.42% (=(661.5-484.9)/484.9) improvement in the capability of state exploration from the average results. The improvement is particularly significant on the Vuestic and Toppr websites, exhibiting an impressive growth of 331% and 550% in the explored states, respectively. We did a deeper investigation to understand the underlying reason for such significant differences and found that both of the two websites have numerous external links, which, due to network conditions, can delay agent exploration significantly. In our experiment, such "risky actions" can waste time and lead to a *failure stage*. However, **cooperative MARL algorithms can decrease the probability of selecting risky actions**. When an agent selects an action that may result in clicking on an external link, it propagates this information through Q-value updates to other agents, greatly reducing the chance of re-clicking such links and thereby enhancing the overall performance of MARG.

While $MARG_C^5$ and $MARG_D^5$ demonstrated superior performance than IQL^5 in terms of state exploration, IQL^5 executed more unique actions on a few websites, such as GitHub. However, the actual testing effectiveness of IQL^5 still falls short of $MARG_C^5$ and $MARG_D^5$, as shown by the number of detected failures.

> ***Answer to RQ1***: Compared to the two state-of-the-art tools, *WebExplor* and *QExplore*, $MARG_D^5$ surpasses them by **4.34** and **3.89** times, respectively, in the number of explored states, leading to the detection of approximately three times more failures. Moreover, cooperative MARL helps avoid unnecessary repetitive behaviors of web testing agents, resulting in a **36.42%** increase in the number of explored unique web states, compared to independently running multiple agents.

6.2 RQ2: Comparison of Data Sharing Schemes

Based on the data presented in the Table 4, distinct performance advantages can be observed for $MARG_C^5$ and $MARG_D^5$ across different subject websites. To further investigate the impact of the two data sharing schemes on the performance of MARG, we compare them from the perspectives of *communication overhead* and *data propagation capabilities*.

Table 4: Comparison of *WebExplor*, *QExplore*, IQL^5, MARG^5_C, and MARG^5_D

	WebExplor	*QExplore*	IQL^5	MARG^5_C	MARG^5_D	*WebExplor*	*QExplore*	IQL^5	MARG^5_C	MARG^5_D
	Explored States (#)					Detected Actions (#)				
WUT$_A$	380.0	153.0	836.0	701.0	**912.0**	1334.0	1538.0	**3442.7**	3126.0	3169.0
toppr	13.0	7.0	12.3	**53.0**	46.0	239.0	207.0	292.0	**852.0**	692.0
Smadex	53.0	101.0	322.3	**703.3**	590.7	407.0	555.0	539.3	556.7	**562.3**
Vuestic	83.3	26.7	42.7	273.0	**283.3**	347.3	194.7	264.0	1122.0	**1169.3**
YouTube	348.7	288.3	442.3	425.7	**587.3**	1946.0	1573.0	2261.0	2235.0	**2382.3**
GitHub	37.7	387.3	631.0	**851.3**	823.3	199.0	3718.3	10949.7	**14611.6**	14487.8
GameSpot	43.7	58.0	197.3	**281.7**	227.3	260.3	373.7	**1132.5**	1129.3	1062.7
EatingWell	381.0	427.7	1082.0	**1296.7**	1155.0	1319.3	1330.7	2133.3	**2166.3**	1909.5
IKEA	31.0	82.0	798.3	1318.7	**1329**	65.0	223.0	**3008**	2648.0	2786.3
Average	152.4	170.1	484.9	656.0	**661.5**	679.7	1079.3	2669.2	**3160.8**	3135.7
	Executed Unique Actions (#)					Detected Failures (#)				
WUT$_A$	343.5	341.0	1112.3	782.0	**1168.0**	13.0	8.3	28.0	30.3	**42.0**
toppr	80.5	63.0	72.7	**291.0**	290.7	1.0	0.7	6.0	**7.3**	6.7
Smadex	120.0	332.0	391.3	376.3	**405.3**	1.0	2.0	3.0	**4.0**	1.7
Vuestic	201.7	89.0	145.3	616.0	**695.0**	6.7	4.7	25.3	29.3	**39.7**
YouTube	550.0	239.3	850.3	800.3	**1085.7**	35.3	33.0	71.0	59.3	**72.7**
GitHub	84.3	444.7	**856.7**	369.3	430.7	3.0	7.3	34.3	40.7	**44.3**
GameSpot	108.0	127.7	393.0	**477.3**	400.0	12.0	18.7	73.0	**77.3**	76.0
EatingWell	311.0	494.0	**682.3**	471.0	511.5	13.3	17.0	27.0	28.0	**32.5**
IKEA	58.0	160.0	**874.7**	596.7	648.0	3.0	3.0	31.0	32.0	**39.7**
Average	206.3	254.5	597.6	531.1	**626.1**	9.8	10.5	33.2	34.2	**39.5**

To compare the communication overhead, we collected the number of actions executed by MARG during each experiment. The results are shown in Figure 6. For instance, in three repeated experiments, the average total number of actions executed by the entire system of five agents in MARG^5_C is 5,999, while that of MARG^5_D is 8,219. It can be observed that MARG^5_D generally has a higher number of action execution compared to the MARG^5_C. This is because there is only a single global Q-table in MARG^5_C, where the controller performs action selection for each agent and updates the policy through read and write operations on this Q-table. In contrast, when using distributed Q-learning with data exchange (i.e., Algorithm 1), the controller maintains corresponding Q-tables for each individual agent, which provides several advantages: first, regarding write operations, the controller only needs to update the Q-table associated with the agent without modifying other Q-tables; second, for read operations, it solely accesses the Q-tables of other agents that involve relevant states, rather than accessing all Q-tables. By reducing the need for global communication, **MARG^5_D can perform policy optimization and action selection more efficiently**, resulting in a higher number of executed actions.

To compare the data propagation capabilities, we calculated the ratio of the number of explored states to the number of executed actions, and presented the results in Figure 7. The higher the ratio, the greater the number of states explored under the same number of actions (i.e., better data propagation capabilities and exploration efficiency). Figure 7 shows that these ratios of MARG^5_C are higher compared to MARG^5_D. This is because MARG^5_C utilizes a centralized global Q-table, which stores the collective historical experiences of the group of agents. Therefore, for any agent, the transition

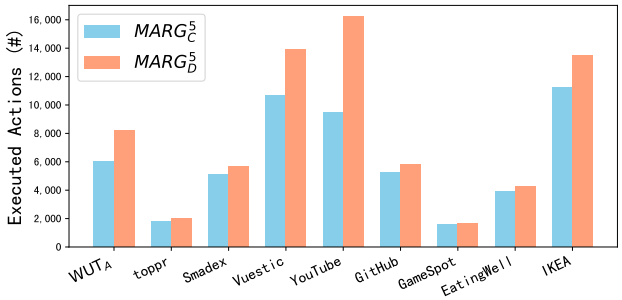


Figure 6: Comparison of total executed actions of MARG^5_C and MARG^5_D

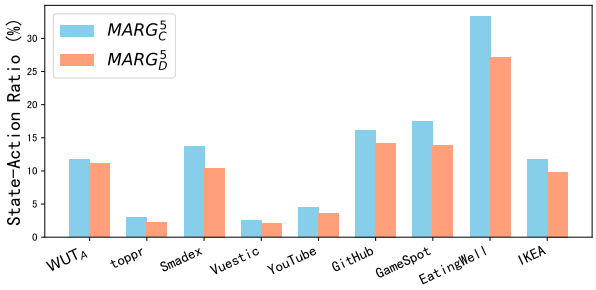


Figure 7: State-to-action ratios of MARG^5_C and MARG^5_D

data from other agents can be treated as if it has been acquired by itself, as agents can directly access the experiences of other agents through the shared Q-table. On the contrary, MARG^5_D exhibits a

gradual decrease in the efficacy of information propagation during the information dissemination process.

A practical implication of the above finding is that practitioners and researchers should consider the trade-off between information propagation capabilities and communication overhead when choosing between the centralized and distributed data sharing schemes. While the former possesses better information propagation capabilities, it also incurs higher communication overhead. On the other hand, within the same time budget, the latter demonstrates superior overall performance, as indicated by the average results.

Answer to RQ2: MARG_C^N, with its centralized nature, exhibits superior information propagation capabilities. However, it incurs a higher communication overhead during the testing process. In contrast, the MARG_D^N algorithm exhibits relatively lower overhead while upholding a good overall performance.

6.3 RQ3: Effect of Agent Numbers

Utilizing a multi-agent system for web GUI testing tasks introduces a new configuration item: the number of agents N. We configured varying N to test WUT_A in order to investigate the impact of the number of agents on MARG's performance. The selection of the subject website WUT_A was based on its complexity, as shown in Table 4 with relatively large numbers of detected states and actions.

As depicted in the Figure 8, in most cases, **the testing performance of MARG exhibits an upward trend as the number of agents increases**. However, the improvement ratios are not directly proportional to the number of agents. For example, as the number of agents N increases from 3 to 5 (↑67%), 8 (↑167%), 12 (↑300%), and 15 (↑400%), the quantity of explored states of MARG_C^N increases by 20%, 138%, 140%, and 146%, respectively. Similarly, the explored states of MARG_D^N exhibited increments of 63%, 138%, 276%, and 211% correspondingly. Regarding the results of detected failures, as N increases from 5 to 8, the number of detected failures by MARG_D^N slightly decreases. This is understandable as MARG may explore different states with different numbers of agents and faults are typically not evenly distributed within web applications. When N further increases from 8, the number of detected failures also increases. These results indicate that increasing the number of agents generally brings performance improvement. However, due to the huge search space and the inherent randomness in the behaviors of the RL agents, a greater number of agents does not always lead to significant improvements in testing performance and the extent of improvement may vary across different metrics.

Additionally, in the Figure 8, we can observe that as the value of N varies from 3 to 8, the exploration capability of MARG_C^N increases significantly. However, as N further increases to 12 and 15, the rate of improvement in exploration capability gradually slows down. On the other hand, the exploration capability of MARG_D^N is relatively stable when N changes. This indicates that as the number of agents reaches a certain threshold, along with a large number of explored states and the significantly expanded Q-table, the information sharing effectiveness of MARG_C^N may decline. As discussed in Section 6.2, maintaining a large Q-table in MARG_C^N results in increased overhead associated with utilizing and updating the Q-table. We further investigated the total number of executed actions of MARG_C^{15} and MARG_D^{15} within the two-hour timeframe. We found that the total number of actions executed by MARG_C^{15} is 72% of that by MARG_D^{15}. This observation suggests that **MARG_D^N is better suited for scenarios involving a larger number of agents, especially when testing dynamic and complex websites where efficient agent collaboration is required**.

Lastly, we also attempted to configure a larger number of agents. However, **the number of agents will be constrained by the available computational resources**. We encountered an out-of-memory error when running 20 agents within a 2-hour time budget.

Answer to RQ3: As the system is configured with more agents, its testing performance generally improves. Due to the low overhead of maintaining Q-tables, MARG_D^N showcases superior results than MARG_C^N. Nevertheless, the expansion of agent numbers is limited by the available computational resources.

6.4 Threats to Validity

The experimental results are subject to uncertainties arising from network issues and inherent randomness in the algorithm (e.g., randomization in ε-greedy). To address this issue, each experiment for every subject website was conducted on the same machine, and we attempted to ensure consistency in network conditions during testing. Moreover, each experiment was conducted three times.

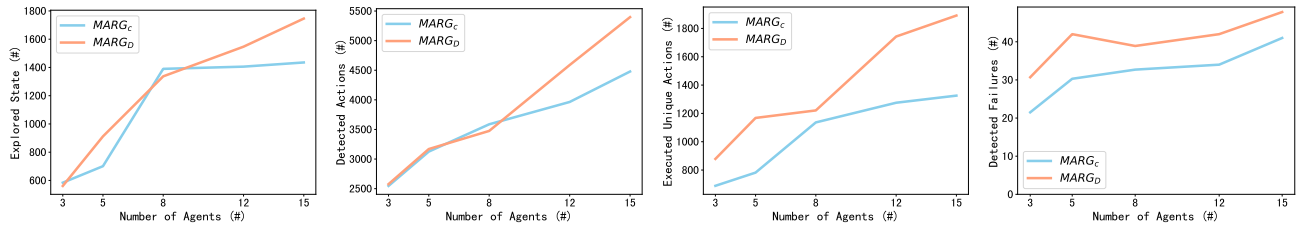
The evaluation of MARG was conducted on only nine commercial websites, which may not be representative of all possible web applications. There is a potential threat that our findings may not generalize across all real-world web applications. To mitigate this threat, we selected different types of websites, many of which are large-scale and complex (e.g., YouTube, GitHub, and IKEA). Our exploratory study demonstrates the potential of MARL in enhancing the efficiency and performance of web GUI testing. Moving forward, we plan to run our system on more diversified and complex websites to further assess and enhance its performance.

Furthermore, we followed an existing empirical study [11] and employed a single configuration to set the parameters of the Q-learning algorithm across all tools. While this ensured consistency, it could restrict the thorough evaluation of tool performances because of the potential impacts of varying parameter settings.

7 Related Work

7.1 Reinforcement Learning for GUI Testing

RL-based techniques are capable of learning and optimizing the exploration strategies through continuous interactions with the environment [33], which makes them promising to facilitate intelligent GUI testing. AutoBlackTest [25] is an RL-based testing technique for desktop applications. It treats the collection of GUI elements as its abstracted state and calculates reward by GUI changes. Bauersfeld et al. [7] adopted a similar approach, but with a minor difference, they used the inverse of action frequencies as a reward. This calculation method has been used in recent works (e.g., *WebExplor* [52] and *QExplore* [38]), which they referred to as "curiosity". Fan et al. summarized GUI testing techniques driven by Q-learning [11]. They highlighted the effectiveness of configurations such as using element collections as states and curiosity as rewards. They also



(a) Explored States **(b) Detected Actions** **(c) Executed Unique Actions** **(d) Detected Failures**

Figure 8: Performance of MARG with different numbers of testing agents

pointed out the limitation of a single Q-learning agent in testing a large-scale website. To address the slow exploration challenge, Mobilio et al. proposed GT^{PQL} [32], where parallel Q-learning agents run independently and synchronize their Q-models periodically.

With the continuous development of deep learning technology, tools that utilize deep reinforcement learning (DRL) [22, 30, 31] for GUI testing have emerged. A challenge addressed by DRL-based techniques is abstracting complex GUI states, a.k.a., GUI embedding. QDroid [45] creates a vector of length 4 where the elements hold the number of components: Input, Navigation, List and Button, which is an input of the Deep Q-Network. DQT [20] preserves widgets' structural and semantic information with graph embedding techniques, building a foundation for identifying similar states or actions and distinguishing different ones.

7.2 Multi-Agent Systems

Recently, cooperative MARL algorithms have gained wide applications in various fields, while most of them focus on the collaboration of multiple Q-learning agents. Melo et al. [28] leveraged the characteristics of sparse interactions to minimize the coupling between different Q-learning agents. Specifically, each agent employs a "global" Q-learning approach in its own responsible exploration domain, and uses a "local" Q-learning approach when coordinating with other agents. To facilitate such coordination, they introduced an additional "coordination" action in the agent's action space. Pham et al. [36] proposed a distributed MARL algorithm for unmanned aerial vehicle (UAV) teams. This algorithm enables cooperative learning among UAVs to achieve comprehensive coverage of unknown interest areas while minimizing the overlap between their fields of view. For this purpose, the reward for an individual agent will be penalized by the number of overlapping cells with the other agents. To address the issue of state explosion, they employed effective function approximation techniques to handle the representation of high-dimensional state spaces.

Besides the above studies, researchers have also explored testing software with multi-agent systems. Cai et al. proposed Fastbot [8], in which multiple agents are responsible for constructing a DAG model of an Android app to support model-based testing. Huo et al. [18] proposed a multi-agent testing environment for web applications. They divided the task of web testing into several subtasks, such as web page retrieval, information extraction, etc. In their system, the coordination and scheduling among agents were conducted by separate agents, namely, brokers. Similar task decomposition ideas have also been applied to testing tasks involving dynamic and collaborative web services. For example, Bai et al. [6] proposed a multi-agent framework, MAST, aiming to facilitate web service testing in a coordinated and distributed environment. Different from these existing studies, our work explores the use of multi-agent reinforcement learning methods to improve the efficiency and effectiveness of web GUI testing. Our main focus is to devise practical agent cooperation mechanisms to enable data sharing among multiple asynchronous RL agents.

8 Conclusion and Future Work

The primary goal of web GUI testing is to explore different page states and achieve a high coverage so as to increase the chance of detecting bugs. As single-agent testing techniques struggle to achieve comprehensive coverage, while merely parallelizing multiple agents can lead to redundancy in visited states, it highlights the demand for efficient communication and coordination mechanisms among the testing agents. To this end, we have designed MARG, the first automatic web GUI testing system driven by multi-agent Q-learning algorithms. Our experiments demonstrate that MARG can outperform two state-of-the-art RL-based web testing techniques, *WebExplor* and *QExplore*, showing promising results of MARL-based GUI testing.

In the future, we plan to enhance MARG from three aspects. First, considering the complex and huge state space of dynamic webpages, we plan to replace Q-learning algorithms in MARG with DRL algorithms [34] to improve state abstraction and value estimation. Second, besides our current collaborative strategies, the agents in MARG can be coordinated with other advanced MARL algorithms, such as VDN [41] or QMIX [37], to enhance overall performance. Third, the users of complex websites may have different roles (e.g., administrators, store managers, and customers of an E-commerce website). Testing such websites with multiple agents of the same role may not be able to trigger certain intricate business logic. We also plan to investigate how to coordinate multiple agents with diversified roles to further improve MARG's performance of testing web applications in complex real-world scenarios.

Acknowledgments

We would like to thank the ASE 2024 reviewers for their constructive comments on this paper. This work is supported by the National Natural Science Foundation of China (Grant Nos. 61932021, 62372219) and the National Key Research and Development Program of China (Grant No. 2019YFE0198100).

References

[1] [n. d.]. Flask Documentation — flask.palletsprojects.com. https://flask.palletsprojects.com/. [Accessed 27-03-2024].

[2] [n. d.]. SeleniumHQ/selenium: A browser automation framework and ecosystem. — github.com. https://github.com/SeleniumHQ/selenium/. [Accessed 27-03-2024].

[3] [n. d.]. Top Websites in the World - Most Visited & Popular Rankings - Semrush — semrush.com. https://semrush.com/website/top/. [Accessed 01-04-2024].

[4] Jeffrey L Adler and Victor J Blue. 2002. A cooperative multi-agent transportation management and route guidance system. *Transportation Research Part C: Emerging Technologies* 10, 5-6 (2002), 433–454.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).

[6] Xiaoying Bai, Guilan Dai, Dezheng Xu, and Wei-Tek Tsai. 2006. A multi-agent based framework for collaborative testing on web services. In *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*. IEEE, 6–pp.

[7] Sebastian Bauersfeld and Tanja Vos. 2012. A reinforcement learning approach to automated gui robustness testing. In *Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012)*. 7–12.

[8] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd.. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 93–96.

[9] Peter Corke, Ron Peterson, and Daniela Rus. 2005. Networked robots: Flying robot navigation using a sensor net. In *Robotics Research. The Eleventh International Symposium: With 303 Figures*. Springer, 234–243.

[10] Anna I Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda, and T Vos. 2016. Q-learning strategies for action selection in the TESTAR automated testing tool. *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)* (2016), 130–137.

[11] Yujia Fan, Siyi Wang, Sinan Wang, Yepang Liu, Guoyao Wen, and Qi Rong. 2023. A Comprehensive Evaluation of Q-Learning Based Automatic Web GUI Testing. In *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 12–23.

[12] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. *Hypertext transfer protocol–HTTP/1.1.* Technical Report.

[13] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[14] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. 2017. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*. PMLR, 1146–1155.

[15] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. 2017. Cooperative multi-agent control using deep reinforcement learning. In *Autonomous Agents and Multiagent Systems: AAMAS 2017 Workshops, Best Papers, São Paulo, Brazil, May 8-12, 2017, Revised Selected Papers 16*. Springer, 66–83.

[16] Eric A Hansen, Daniel S Bernstein, and Shlomo Zilberstein. 2004. Dynamic programming for partially observable stochastic games. In *AAAI*, Vol. 4. 709–715.

[17] Hado Hasselt. 2010. Double Q-learning. *Advances in neural information processing systems* 23 (2010).

[18] Qingning Huo, Hong Zhu, and Sue Greenwood. 2003. A multi-agent software engineering environment for testing Web-based applications. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003.* IEEE, 210–215.

[19] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 105–115.

[20] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[21] Scott M Lewandowski. 1998. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)* 30, 1 (1998), 3–27.

[22] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[23] Michael L Littman. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*. Elsevier, 157–163.

[24] Sergio Valcarcel Macua, Jianshu Chen, Santiago Zazo, and Ali H Sayed. 2014. Distributed policy evaluation under multiple behavior strategies. *IEEE Trans. Automat. Control* 60, 5 (2014), 1260–1274.

[25] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2011. AutoBlackTest: a tool for automatic black-box testing. In *Proceedings of the 33rd international conference on software engineering*. 1013–1015.

[26] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2014. Automatic testing of GUI-based applications. *Software Testing, Verification and Reliability* 24, 5 (2014), 341–366.

[27] Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. 2012. Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review* 27, 1 (2012), 1–31.

[28] Francisco S Melo and Manuela Veloso. 2009. Learning of coordination: Exploiting sparse interactions in multiagent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. Citeseer, 773–780.

[29] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. 2008. Crawling Ajax by inferring user interface state changes. In *2008 eighth international conference on web engineering*. IEEE, 122–134.

[30] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[32] Marco Mobilio, Diego Clerissi, Giovanni Denaro, and Leonardo Mariani. 2023. GUI Testing to the Power of Parallel Q-Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 55–59.

[33] Miguel Morales. 2020. *Grokking deep reinforcement learning*. Manning Publications.

[34] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. *Advances in neural information processing systems* 29 (2016).

[35] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164.

[36] Huy Xuan Pham, Hung Manh La, David Feil-Seifer, and Aria Nefian. 2018. Cooperative and distributed reinforcement learning of drones for field coverage. *arXiv preprint arXiv:1803.07250* (2018).

[37] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2020. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research* 21, 178 (2020), 1–51.

[38] Salman Sherin, Asmar Muqeet, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. 2023. QExplore: An exploration strategy for dynamic web applications using guided search. *Journal of Systems and Software* 195 (2023), 111512.

[39] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[40] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[41] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. 2017. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296* (2017).

[42] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[43] Ming Tan. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*. 330–337.

[44] Thi Anh Tuyet Vuong and Shingo Takada. 2018. 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*. 31–37.

[45] Thi Anh Tuyet Vuong and Shingo Takada. 2019. Semantic Analysis for Deep Q-Network in Android GUI Testing.. In *SEKE*. 123–170.

[46] Hoi-To Wai, Zhuoran Yang, Zhaoran Wang, and Mingyi Hong. 2018. Multi-agent reinforcement learning via double averaging primal-dual optimization. *Advances in Neural Information Processing Systems* 31 (2018).

[47] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[48] Yaodong Yang and Jun Wang. 2020. An overview of multi-agent reinforcement learning from game theoretical perspective. *arXiv preprint arXiv:2011.00583* (2020).

[49] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and Image Recognition Driving Cross-Platform Automated Mobile Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1561–1571. https://doi.org/10.1109/ICSE43902.2021.00139

[50] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control* (2021), 321–384.

[51] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. 2018. Fully decentralized multi-agent reinforcement learning with networked agents. In *International Conference on Machine Learning*. PMLR, 5872–5881.

[52] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 423–435.