Regression Testing of Massively Multiplayer Online Role-Playing Games

Yuechen Wu*
Fuxi AI Lab, Netease, Inc.

Yingfeng Chen*
Fuxi AI Lab, Netease, Inc.

Xiaofei Xie
Nanyang Technological University

Bing Yu
Kyushu University

Changjie Fan
Fuxi AI Lab, Netease, Inc.

Lei Ma
Kyushu University

***Abstract*—Regression testing aims to check the functionality consistency during software evolution. Although general regression testing has been extensively studied, regression testing in the context of video games, especially Massively Multiplayer Online Role-Playing Games (MMORPGs), is largely untouched so far. One big challenge is that game testing requires a certain level of intelligence in generating suitable action sequences among the huge search space, to accomplish complex tasks in the MMORPG. Existing game testing mainly relies on either the manual playing or manual scripting, which are labor-intensive and time-consuming. Even worse, it is often unable to satisfy the frequent industrial game evolution. The recent process in machine learning brings new opportunities for automatic game playing and testing. In this paper, we propose a reinforcement learning-based regression testing technique that explores differential behaviors between multiple versions of an MMORPGs such that the potential regression bugs could be detected. The preliminary evaluation on real industrial MMORPGs demonstrates the promising of our technique.**

I. Introduction

Regression testing is an important activity for software quality assurance during software evolution, and maintenance. Changes during software updates could be a common source of introducing bugs, making regression testing of great importance. As a typical software, the video game software, especially the industrial Multiplayer Online Role-Playing Game (MMORPG), often performs fast and frequent version updates, to fix bugs, add new features, enhance the performance, *etc.*. Rapid version changes may introduce new bugs in the new version, especially under the fast development and release pressure. Reducing the buggy code risk is very important for the game company as the bugs can affect the user experience, resulting in the loss of users and even leading to security risks and economic loss In practice, regression testing is commonly adopted in the video game industry and plays an important role in quality assurance during game software evolution.

Although extensive studies have been performed for regression testing, the regression testing of video games, especially MMORPGs (the focus of this paper), is largely untouched so far due to its characteristics. To better understand the regression testing activities in industrial MMORPGs, we conduct an empirical study on three mainstream popular commercial games at Netease. Table I summarizes the information (from the internal game development report) about the statistics of three large-scale industrial MMORPGs (*i.e.*, *Justice*, *Revelation*, *Ghost*). We can see that the industrial games are often quite complex in terms of installation size and game content (*i.e.*, containing many quests). To accomplish these quests, a shortcut strategy still needs at least 200+ hours. In particular, these games are also constantly updated in a very fast way due to that, 1) the complicated quests may contain bugs and need to fix very quickly, 2) the game needs to add new features or maintain balance to avoid the loss of users. In Netease, each game is often updated so frequently, with 3 internal versions per day (*i.e.*, the version at 2:00AM, 12:00AM and 6:00PM). Each version may contain hundreds of commits (*e.g.*, 400+, 300+, and 100+ commits). To guarantee the quality of the game in the fast evolution pace, Netease employed many professional human testers, especially for some timely and hot games (*e.g.*, the game *Justice* alone has more than 30 professional testers). Even though, due to the huge space of MMORPGs, only the main story of the game could be tested at a certain level. Many bugs still remain in the code, which are often discovered by the users after releasing.

In summary, the fast evolution of industrial MMORPGs brings several challenges for regression testing.

- The game space is huge. To accomplish non-trivial tasks, many user interactions (*e.g.*, use the right skill) are required. Existing testing techniques are often hard to generate valid test cases (*i.e.*, a valid sequence of operations) for games, making the game testing largely rely on the manual play or written scripts, where scripts can also often become unusable when the game is updated.
- Existing white-box testing techniques [3], [14] (*e.g.*, symbolic execution) could capture the software differential behaviors but they can be difficult to be applied for game testing. They are often not scalable in the complex MMORPGs, and some relevant states (*e.g.*, timeout) or randomness in MMORPGs could only be triggered at runtime.
- Moreover, for confidentiality reasons, Netease has strict control over the access of the source code, which testers often do not have permission to access. Thus, the game testing process is mostly black-box.

*Equal contribution.

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00074

- Game updates are rather frequent and require efficient methods for testing the change. For example, although there are quite a few professional testers employed by Netease, the regression testing is still a pain point, leaving the released game under risks in some cases. Fully automated regression testing of games is highly demanding.

Towards addressing these challenges, in this paper, we propose a reinforcement learning (RL) based regression testing technique for MMORPGs, aiming at effectively exploring the changes of two versions of the MMORPG. The intuition is that capturing more divergent behaviors of two versions of a game is more likely to identify regression bugs. The essence of our method is to train an intelligent RL policy that considers both exploring different states between the old version and new version (*i.e.*, capturing the divergence) and exploiting task accomplishment in the new version (*i.e.*, reaching deep states). We performed a preliminary study to demonstrate the promising of our approach on industrial MMORPG of Netease. To the best of our knowledge, this is the first attempt to explore automated regression testing techniques for industrial MMORPG game software.

II. Methodology

A. Game Playing and Regression Testing

Game playing could be well modeled as a Markov Decision Process (MDP). At each discrete time step t, the agent observes the current state s_t (*e.g.*, the health points and magic points) and chooses an action a_t (*e.g.*, moving forward or using some skills). After the execution of action a_t, the game environment is updated to a new state s_{t+1} (*i.e.*, the effect of the action a_t). A reward r_t with regards to a specific goal (*e.g.*, accomplishing the task as soon as possible or losing health points as little as possible) is calculated for the current action a_t at state s_t. The process continues until the agent finishes the given task or the allocated budget exhausts. The game playing process can be represented by a sequence $(s_0, a_0, s_1, \ldots, a_{n-1}, s_n)$. More specifically, for a game, we define its MDP $\mathcal{M}$ as a tuple $(\mathcal{S}, \mathcal{A}, r, \gamma)$, where $\mathcal{S}$ is a set of states of the game, $\mathcal{A}$ is a set of actions, r denotes the reward function and $\gamma \in (0, 1]$ is a discount factor.

Definition 1 (Game Regression Testing): Given a game $\mathcal{M}(\mathcal{S}, \mathcal{A}, r, \gamma)$ and its updated version $\mathcal{M}'(\mathcal{S}', \mathcal{A}', r', \gamma')$, regression testing aims to generate test cases that capture the differences between the two versions as many as possible such that the regression bugs are more likely to be detected.

Intuitively, regression testing aims at maximizing the exploration of the differential behaviors of two versions of the game. We define two forms of game version divergence between M and M' as follows:

Definition 2 (State Divergence): A state s is divergent between M and M' if $s \in \mathcal{S} \wedge s \notin \mathcal{S}'$ or $s \in \mathcal{S}' \wedge s \notin \mathcal{S}$.

Definition 3 (Trace Divergence): Assume the same strategy is adopted in playing two versions of a game, after the trace $(s_0, a_0, s_1, \ldots, a_{n-1}, s_n, a_n)$ is executed, the states s_{n+1} and s'_{n+1} are returned from two versions of the game M and M'. We say the trace is divergent if $s_{n+1} \neq s'_{n+1}$.

State divergence reveals the direct update, *i.e.*, adding some new features that belong to $\mathcal{S}'$ but not $\mathcal{S}$, or removing some features that belong to $\mathcal{S}$ but not $\mathcal{S}'$. For trace divergence, both s_{n+1} and s'_{n+1} can belong to $\mathcal{S}$ and $\mathcal{S}'$, *i.e.*, $s_{n+1} \in \mathcal{S} \wedge s_{n+1} \in \mathcal{S}' \wedge s'_{n+1} \in \mathcal{S} \wedge s'_{n+1} \in \mathcal{S}'$. In this case, the state divergence is not identified, but the update still changes the internal behaviors.

At each state, we check whether bugs are triggered. In this work, we mainly detect two types of bugs: the crash bugs (*i.e.*, the game crashes) and stuck bugs (*i.e.*, the state cannot be changed within a certain time), both of which could be automated.

B. Regression Testing with Reinforcement Learning

A key challenge is how to explore the states of a game such that the divergence could be captured. Existing techniques (*e.g.*, random, breadth-first search, depth-first search) often cannot effectively accomplish the task of the game, which requires a certain level of intelligence, making it difficult to cover deep states (*e.g.*, a subtask can only be triggered after completing another subtask).

Reinforcement learning [20] aims to train an agent interacting with an environment to maximize expected rewards. It generates an optimized policy to better accomplish the task of the game. Our key insight is that we could use reinforcement learning to train a specific policy, which can not only accomplish the main task but also explore more divergent behaviors between two versions of a game.

In the paper, our early attempt on game regression testing follows the Q-learning algorithm [21], which is an off-policy temporal difference (TD) control algorithm in reinforcement learning. It estimates the value of executing an action a from a given state s, which is referred to as Q-value. Q-values are learned iteratively by updating the current Q-value estimate towards the observed reward r and estimated utility of the next state s' as follows:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right) \quad (1)$$

where s' is the next state after executing the action a at state s, r is the corresponding reward and α is the learning rate. An episode of the training ends when the current state is a final or terminal state (*i.e.*, timeout, fail or success).

The state abstraction and the reward design are the two key components of our reinforcement learning-based game regression testing.

C. State Abstraction

Although we can construct the states with the information of the game, an MMORPG often provides a large set of game information, which is high-dimensional and noisy. It is also impractical to learn directly from the raw information that may include useless or irrelevant information for an agent to complete the task (*e.g.*, the wild neutral monsters, the dialogue information between the player and the NPCs).

Following the human experience in game playing, we design a denoising strategy to collect the key information from

TABLE I: Complexity and Regression of Two Industrial MMMORPGs

Game Name	#Registered Players	Size	#Quests	#time(h)	Update Freq.	# Change	#Testers
Justice (PC)	30 millions	82.4GB	15,147	200h+	3 versions/day	400+ commits/day	30
Revelation (PC)	16 millions	27.44GB	11,000	100h+	3 versions/day	300+ commits/day	12
Ghost (Mobile)	100 millions	2.6GB	1,000	20h+	3 versions/day	100+ commits/day	21

the raw data, which includes three key steps: 1) omitting task-irrelevant information with some domain knowledge; 2) reducing dimensions by transforming feature expression; 3) performing the interval abstraction to discretize values (*e.g.*, health points). Currently, some domain knowledge are leveraged in our current state abstraction, which needs to be customized for different games. For example, we manually determine which information is irrelevant. We leave the automatic state abstraction as the future work.

D. Rewards for Regression Testing

For regression testing, the key challenge is how to define rewards, which determine the interaction strategy of reinforcement learning in the game. In general, the goal of regression testing is to explore more divergent behaviors between two versions. To achieve this goal, the policy at least needs to accomplish the complex task such that deep states can be covered. However, only accomplishing the task is not enough, we may train a policy that tends to identify an optimal way to accomplish the task but other diverse states may be missed. To mitigate this problem, the policy needs to explore more diverse behaviors of the game such that more divergent behaviors are more likely to be reached. Due to the huge space of the game states, improving diversity may reduce the efficiency of detecting divergence, which is the main goal of regression testing. In other words, the diversity-based strategy may explore many states, which are the same between the old and the new version. Thus, we need to give more rewards when the state divergence or trace divergence is discovered. To address such challenges, we propose a multi-reward fusion method that considers the task accomplishment, the diversity (more diverse states) and the divergence (inconsistent states) exploration.

1) Accomplishment Reward: The *accomplishment reward* considers the capability of the task accomplishment, which is calculated after the completion of each episode. The reward of each action a at state s is calculated as:

$$r_{accomp}(s,a) = \beta * m - n * l \tag{2}$$

where β is 0 if the task is not completed in the episode (*e.g.*, timeout or fail) and 1 otherwise. m is a large reward for the success of the task, n is a tiny penalty (*i.e.*, 0.01)and l is the total steps of the episode. Intuitively, if the task is completed, a large reward is given. Meanwhile, there is a tiny penalty for each step, *i.e.*, the fewer steps to complete the task, the higher the reward is.

2) Diversity Reward: Curiosity [15] has been widely used to explore the environment and discover novel states. Here, we compute the cumulative visited times of each state in the whole train process instead of each episode, as the curiosity. Thus the policy is optimized to choose the actions that could lead to rare states. The curiosity-based reward is defined as:

$$r_{curi}(s,a) = \begin{cases} 1 & times(s') = 0 \\ \frac{1}{times(s')} & otherwise \end{cases} \tag{3}$$

where s' is the next state after executing the action a and $times$ denotes the cumulative visited time of the state s'. The diversity reward encourages to explore new states.

3) Divergence Reward: After each episode on the new version of the game, we obtain a trajectory that is a sequence of visited states by actions, *i.e.*, $(s_0, a_0, s_1, \ldots, s_n, a_n, s_{n+1}, \ldots)$. To check the divergence between two versions, we try to replay the trajectory in the old version by using the same action step by step. A divergence is triggered when either an inconsistent state or invalid action in the old version is reached. After a divergence is detected, we are unable to further replay the original trajectory and the policy is then adopted to play the games until the task is completed or failed. Finally, we obtain another trajectory from the older version, *i.e.*, $(s_0, a_0, s_1, \ldots, s_n, a_n, s'_{n+1}, \ldots)$, where $s'_{n+1} \neq s_{n+1}$. We calculate the divergence reward with the Jaccard similarity as:

$$r_{div}(s_i, a_i) = \begin{cases} Jaccard(tr^{new}(s_i), tr^{old}(s_i)) & i \leq n \\ 0 & i > n \end{cases} \tag{4}$$

where $tr(s_i)$ shows the sub-sequence of states after the state s_i in the new or old trajectory. Intuitively, we assign a high reward to the previous states from s_0 to s_n since the divergence is triggered after these states. The following divergent states are rewarded as 0 as such states are expected to be tested diversely (guided by diversity reward).

Finally, three levels of rewards are fused to one reward:

$$r(s,a) = c_0 * r_{accomp}(s,a) + c_1 * r_{curi}(s,a) + c_2 * r_{div}(s,a) \tag{5}$$

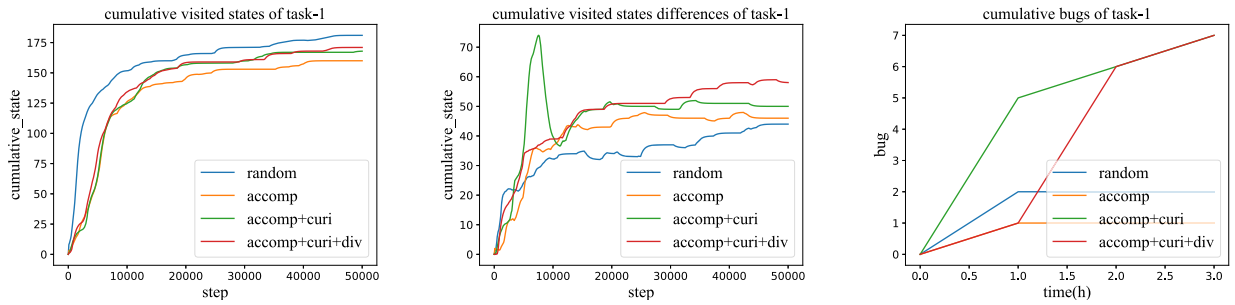
where c_0, c_1 and c_2 are the coefficients.

III. Preliminary Evaluation

A. Setting

To demonstrate the potential of our method, we select one task from a commercial MMORPG as our target, which contains 7 regression bugs. We select four baselines to perform the comparative study:

- *Random strategy,* which explores the game randomly.
- *Accomplishment strategy,* which adopts the reinforcement learning to explore the game with only the accomplishment reward.
- *Accomplishment+Diversity strategy,* which explores the game with accomplishment reward and diversity reward.
- *Accomplishment+ Diversity+Divergence strategy,* which explores the game with all three rewards.



(a) Results of cumulative visited states (b) Results of cumulative divergence (c) Results of cumulative regression bugs

Fig. 1: The results of regression testing with different strategies

The configuration details of the experiments are: the discount factor $\gamma = 0.99$, the learning rate $\alpha = 0.1$, the exploration rate $\epsilon = 0.3$. For the fusion, we empirically set the coefficients as: $c_0 = 0.1, c_1 = 0.2$ and $c_2 = 0.2$.

B. Metrics

In the experiment, we use the following four metrics for measurement:

- *Total Visited States.* The number of the unique states visited in the new version of the game.
- *Total Differential States.* The number of unique states (in the new version) that could not be covered in the older version of the game.
- *Total Bugs.* The number of regression bugs triggered in the new version of the game.

C. Results

Figure 1 summarizes the results of different strategies. Notably, each strategy is run with 50000 steps. One step represents that the game transits from one state to another one. During the game playing, if the target task completes or fails, we rerun the game. The horizontal axis represents the total number of steps.

Figure 1a shows the results of the cumulative visited states, including the common states and differential states between the new version and its previous version. We observe that random strategy covers more states of the game than others. This is reasonable since random strategy often explores meaningless states. For example, there can be many NPCs or game zones that are not relevant to the task. In addition, the strategy with only accomplishment reward captures the minimum number of states as it only considers the shortcut to complete this task. Finally, those states, which helpful for accomplishing the task, are assigned with high Q-values.

Figure 1b shows the results of the cumulative differential states. Note that, the game state space is often quite huge, it is not possible to determine whether a state is divergent (See Definition 2). Thus, we detect the potential divergent states by checking whether this state has been seen before. In this way, some states may be divergent in the early stage because it was seen in one version but not seen in a newer version. However, they may be explored in both of the versions in the late stage and are regarded as common states. This could be the reason that the curve decreases sometime. The results show that the performance of the random strategy is quite limited in detecting differential states although it performs the best in exploring the total states (Figure 1a). Driven by curiosity and diversity rewards, the trained RL policy is more effective in exploring the differential states (*i.e.*, the red and green lines).

Figure 1c shows the results of bug detection, which confirms the advantage of strategies guided by curiosity reward and the diversity reward, in detecting more bugs and exploring more differential states.

IV. Related Work

Regression testing has been extensively studied, including regression test selection [25], [4], [6], [8], [17], [28], regression test prioritization [7], [10], [12], [18], [26] and regression test generation [3], [14], [24], [13], [9], [16], *etc.*. Due to a large number of regression tests, running the entire test suite is time-consuming and expensive. Regression Test Selection (RTS) selects and runs only the relevant test cases that are likely to reveal the code changes. The test cases could be selected based on changes of different levels, *e.g.*, basic-block-level [8], [17], method-level [27], file-level [6] or combination of them [28]. Regression test prioritization [7], [10], [12], [18], [26] orders the test suite based on some criteria such that the test cases with higher priority are run earlier. Usually, the test cases, which could achieve higher code coverage or capture more differential behaviors, are given higher priority. For game testing, there are often fewer regression test cases as the test cases are manually designed by testers. Thus, this paper mainly focuses on the automatic test case generation for version inconsistent behavior detection during evolution. Existing regression testing techniques focus on generating test cases that can capture divergent behaviors between two versions. Most of them are based on symbolic execution [3], [14], [24] or dynamic fuzzing [13], [9], [16]. However, the symbolic execution based techniques may suffer from scalability issues while the fuzzing techniques could not generate high-quality test cases for testing games. It is not obvious how to adapt these techniques to the game context either.

Previous works [11], [1] have shown that automated game testing is quite challenging and still at an early state. Although some techniques [2], [5], [19] are proposed for testing GUI-based applications, they are still not effective for games due to that game playing has to accomplish some hard tasks, requiring a certain level of intelligence. Inspired by the successful application of reinforcement learning in game playing [23] and robot navigation [22], some recent work Wuji [29] adopts deep reinforcement learning (DRL) to test games. Wuji, based on the evolutionary DRL, trains multiple policies at once, which is expensive and challenging for testing the rapid evolution of games. Moreover, Wuji is a general-purpose technique for testing one single version of the game while our method focuses on testing the differential behaviors between multiple versions, towards addressing the quality challenges during the fast evolution of industrial game development.

V. Conclusion

This paper presented a learning-based automated regression testing for game software, which leverages multiple rewards to facilitate diverse and divergent behavior detection of two game versions. The preliminary results demonstrated its potential in capturing differential states and regression bugs. Our technique also has the potential to generalize to a wide range of applications that intrinsically requires complex interactions.

Acknowledgments

This work was supported in part by JSPS KAKENHI Grant No. 20H04168, 19K24348, 19H04086, and JST-Mirai Program Grant No. JPMJMI18BB of Japan.

References

[1] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. Critical success factors to improve the game development process from a developer's perspective. *J. Comput. Sci. Technol.*, 31(5):925–950, 2016.

[2] Ishan Banerjee, Bao N. Nguyen, Vahid Garousi, and Atif M. Memon. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information & Software Technology*, 55(10):1679–1694, 2013.

[3] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 334–344, New York, NY, USA, 2013. Association for Computing Machinery.

[4] L. C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 252–261, 2002.

[5] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 429–440, 2015.

[6] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 211–222, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Trans. Softw. Eng. Methodol.*, 24(2), December 2014.

[8] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *The 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and App.*, OOPSLA '01, page 312–326, 2001.

[9] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 137–146, 2010.

[10] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[11] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, 22(4):2095–2126, 2017.

[12] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 535–546, 2016.

[13] Alessandro Orso and Tao Xie. Bert: Behavioral regression testing. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, page 36–42, New York, NY, USA, 2008. Association for Computing Machinery.

[14] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: Testing for divergences between software versions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1181–1192, 2016.

[15] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17, 2017.

[16] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.

[17] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[18] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 268–279, 2015.

[19] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 245–256, 2017.

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

[21] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[22] Yuechen Wu, Zhenhuan Rao, Wei Zhang, Shijian Lu, Weizhi Lu, and Zheng-Jun Zha. Exploring the task cooperation in multi-goal visual navigation. In *IJCAI*, pages 609–615, 2019.

[23] Yuechen Wu, Wei Zhang, and Ke Song. Master-slave curriculum design for reinforcement learning. In *IJCAI*, pages 1523–1529, 2018.

[24] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 24(1), October 2014.

[25] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, 2012.

[26] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 192–201, 2013.

[27] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 23–32, 2011.

[28] Lingming Zhang. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 199–209, 2018.

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