

Article

Test Suite Prioritization Based on Optimization Approach Using Reinforcement Learning

Muhammad Waqar ¹, Imran ^{2,*} , Muhammad Atif Zaman ¹, Muhammad Muzammal ¹  and Jungsuk Kim ^{2,*}

¹ Department of Computer Science, Bahria University Islamabad, Islamabad 44000, Pakistan; waqarkhattak1993@gmail.com (M.W.); atif.apcoms@yahoo.com (M.A.Z.); mmuzammal.buic@bahria.edu.pk (M.M.)

² Department of Biomedical Engineering, Gachon University, Incheon 21936, Korea

* Correspondence: imranj@gachon.ac.kr (I.); jungsuk@gachon.ac.kr (J.K.); Tel.: +82-10-9336-9498 (I.)

Abstract: Regression testing ensures that modified software code changes have not adversely affected existing code modules. The test suite size increases with modification to the software based on the end-user requirements. Regression testing executes the complete test suite after updates in the software. Re-execution of new test cases along with existing test cases is costly. The scientific community has proposed test suite prioritization techniques for selecting and minimizing the test suite to minimize the cost of regression testing. The test suite prioritization goal is to maximize fault detection with minimum test cases. Test suite minimization reduces the test suite size by deleting less critical test cases. In this study, we present a four-fold methodology of test suite prioritization based on reinforcement learning. First, the testers' and users' log datasets are prepared using the proposed interaction recording systems for the android application. Second, the proposed reinforcement learning model is used to predict the highest future reward sequence list from the data collected in the first step. Third, the proposed prioritization algorithm signifies the prioritized test suite. Lastly, the fault seeding approach is used to validate the results from software engineering experts. The proposed reinforcement learning-based test suite optimization model is evaluated through five case study applications. The performance evaluation results show that the proposed mechanism performs better than baseline approaches based on random and t-SANT approaches, proving its importance for regression testing.

Keywords: software testing; regression testing; test suite optimization; test suite prioritization; reinforcement learning



Citation: Waqar, M.; Imran; Zaman, M.A.; Muzammal, M.; Kim, J. Test Suite Prioritization Based on Optimization Approach Using Reinforcement Learning. *Appl. Sci.* **2022**, *12*, 6772. <https://doi.org/10.3390/app12136772>

Academic Editor: José Carlos Bregieiro Ribeiro

Received: 13 May 2022

Accepted: 28 June 2022

Published: 4 July 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

With the adoption of the agile paradigm in most software companies, the interest in continuous integration environments is growing. The benefits of such environments include integrating frequent software changes and making software evolution faster and less expensive. As a result, it will effectively handle tasks such as build processes, test execution, and test result reporting. Software testing has been practiced as early as when the field of software engineering emerged. Software testing was introduced to evaluate the software quality [1]. Testing includes activities that can identify every possible error in software for rectifying them before the software product has been released to end-users. Software testing is one of the expensive phases of the development process [2]. Software testing and debugging cost is over 50 percent of the whole development cost [3,4]. The cost of regression testing depends on the complexity of the application and the size of the test suite [5]. The fundamental challenge of software testing is to improve the order of the test cases for implementation to identify maximum faults in the given test suite [6]. To overcome this challenge, three common solutions are studied in the literature: test suite selection (TSS) [7,8], test suite minimization (TSM) [9,10] and test-suite prioritization

(TSP) [1–4,6,11–15]. TSP re-orders the test cases to identify more defects in the top few test cases.

Artificial intelligence (AI) and machine learning techniques have proven their practical suitability in interdisciplinary applications [16–19]. A plethora of tools have been developed to assess the development and deployment of AI models [20,21]. Machine learning techniques are also used for TSP [4,12,15]. In [22], test cases are prioritized based on supervised machine learning. This approach deals with the test case description in natural language and test case history like black box metadata. In addition, the support vector machine for ranking (SVM Rank) algorithm is used for prioritizing the test cases [22]. Reinforcement learning (RL) is an area of AI and machine learning in which the computer agents learn from the environment through interactions [23]. The agent learns the policy of actions to maximize the reward value for its current state [24]. Q-Learning is a model-free RL based on the Markov decision process [24]. In Q-Learning, the agent learns from the Q function while performing an action on some state. The Q function is also known as action-value function [24]. At any state, the agent can take five actions: up, down, left, right, and no action [25]. The agent changes its state by taking action and updating its reward value from the policy [24]. The agent memory is accommodated in the table of Q values, calculated through the action-value function. The agent takes those possible actions through which it can maximize Q values based on policy [24].

There are many approaches used for the test case prioritization, such as data mining approaches [10,13], machine learning approaches [4,12,15,22], clustering approaches [5,26], multiple criteria-based approaches [1], weighted method for test case prioritization [3], and code coverage (CC) based TSP [7,9]. A similar study on the prioritization of test cases is done through data mining techniques called application navigation tree mining and interaction pattern mining [13]. The work is done in different phases; in the first phase, the model creates the tree with the help of user and tester interaction with the system. Then in the next phase, the sequence mining algorithm identifies the frequent interaction of the user and tester with the system. The third phase [13] identifies the longest common sub-sequence, which shows the most used workflow of the system. So finally, the researcher prioritizes the test cases based on the longest common sub-sequences using data mining. TSP can be done through different methods, for instance, assigning priority to the test cases based on some criteria and changing the order of the test cases based on the priority value [1]. As a result, test cases have a high priority to detect more faults and prioritize the order of test cases, which can increase the probability of fault detection in the beginning [3]. Furthermore, the automated process of TSP preserves more time and resources [4].

This paper presents a novel TSP technique based on the RL method named Q-learning. The first step of the proposed mechanism is data preparation and storing it in the database system. For this purpose, an interaction recording system is developed to record users' and testers' interactions with the application. For example, test cases are written in natural language, and the tester needs to follow these test cases to test the software system. Therefore, the tester's interaction with the system during system testing is imperative for prioritizing the test cases. Similarly, recording the user interaction with the application after its beta releases is needed. For instance, when any user wants to use the application, the user interacts with the view part, which is the graphical user interface (GUI). Each activity and button of the GUI application has a unique name or ID. For example, when a user clicks on a button, its activity name and the ID of the clicked button are added to the list. The list ends when the user logs out from the application, or the application is forcefully terminated. It gives a complete user interaction path from the log-in state to the log-out state. After using applications, many user interaction logs are recorded. Similarly, the testers' log is recorded when the tester interacts with the applications to execute the test cases. A conceptual model of the proposed test suite prioritization technique is presented in Figure 1.

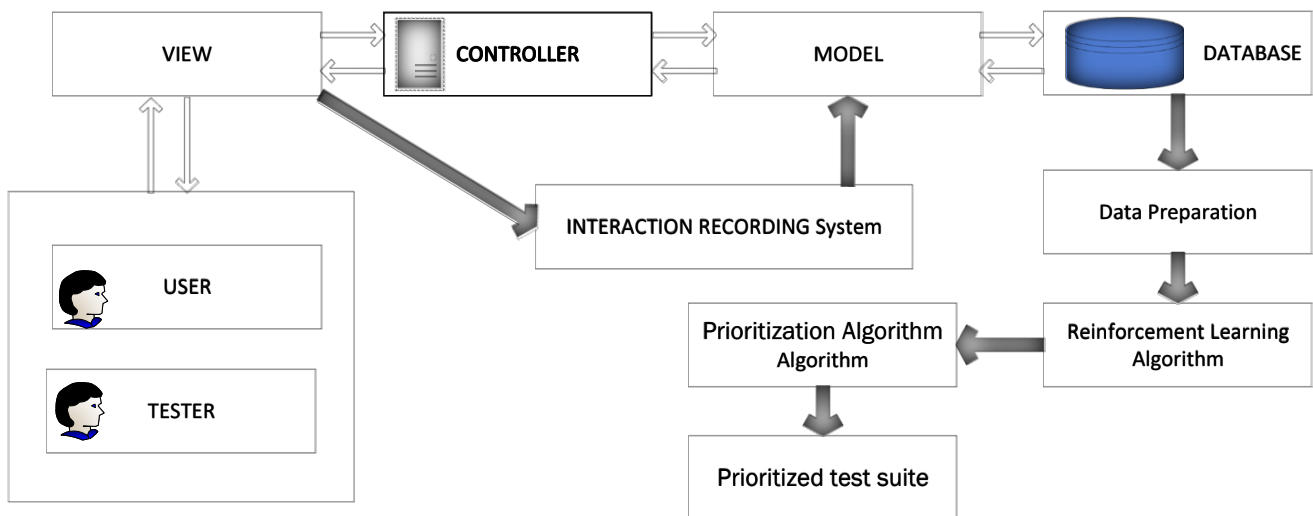


Figure 1. Conceptual model of the proposed test suite prioritization mechanism.

The RL model is used to train the agent to interact with the environment and maximize reward from state actions. The policy tells the agent which action to take in various circumstances. The policy is based on the finite Markov decision process (FMDP). Markov's decision process calculates the reward value based on the current and the next state. After learning the environment, the agent follows that sequence of states to maximize its reward from starting state to the final state. The proposed model is evaluated on datasets from the five case study applications. The case studies include the health care case study, intelligent vehicle assistance system (IVAS), real estate case study, mobile shop POS app, and jobs portal case study. For evaluation, the fault seeding [27–30] mechanism is used to measure the fault detection after employing the prioritization algorithm. In summary, the main contributions of this study are as follows:

- A four-fold methodology of test suite prioritization based on reinforcement learning.
- Regression testing tool for android applications based on the real-time interaction of users and testers with the application.
- Automatic test suite prioritization based on Q-Agent.
- Five case study applications, including health care, intelligent vehicle assistance system (IVAS), real estate case study, mobile shop POS, and jobs portal.
- Fault seeding-based approach for the validation of the proposed RL model.

The rest of the paper is organized as follows: Section 2 presents the literature review. The proposed methodology is discussed in Section 3. Section 4 presents the results, proof of concept, and performance evaluation. Finally, the conclusions and future work directions are presented in Section 5.

2. Related Work

This section discusses the literature on the TSP and various related techniques proposed by the scientific community. Robeala Abid and Aamer Nadeem [1] presented multiple criteria-based techniques using the additional strategy for test case prioritization in their research. One is recognized as primary criteria, and the other one is recognized as secondary criteria. Primary criteria prioritize the test cases, and the secondary criteria break the attachment between the test cases when more than one test case gives the same coverage in the first criteria. Rubbing Huang presented a new fixed-strength interaction coverage-based prioritization (FICBP) method [11]. At the time of prioritization, authors frequently use base choice coverage so that it can enhance testing efficiency. The algorithm chooses those test cases covered by the greater one-wise factor level combination. These

include test cases that have not been considered in the base choice coverage. The evaluation is performed in terms of the time complexity of the FICBP algorithm.

The authors in [9] discussed the test-suite reduction and prioritization framework called test optimizer. The test optimizer framework is based on the CC. The test optimizer consists of a test planner and a test manager. The test leader provides inputs to the test planner, and the inputs are allotted optimization time, test suites, and test repository. The test manager generates the test cases using coverage-based test suite reduction and test-suite prioritization techniques by observing the allocated optimization time. The results are then stored in the test repository. The timer and optimizer are the main sub-components of the test manager. Test case prioritization can also be accomplished using the cuckoo search (CS) algorithm [8]. The CS algorithm is based on the nature of cuckoo birds. The CS algorithm prioritizes the test cases on fault statistics, such as the more significant number of fault detection in less time. Artificial bee colony (ABC) is also used in [4] for TSP. Moreover, swarm intelligence (SI) have been adopted to prioritize test cases based on CC. SI or particle swarm optimization (PSO) are algorithms developed based on biologically inspired artificial intelligence models of social insects, such as bees and ants [4].

Helge Spieker [15] used RL for test case prioritization and test case selection. In [15], the researchers introduced a method called RETECS for learning test case prioritization and test case selection in continuous integration (CI) to reduce the round trip time between the developer feedback and code commits on failed test cases. RETECS uses RL to prioritize and select test cases based on the last execution, failure history, and duration. In [5], the researcher presented a test case selection technique for applications connected to databases. Raul H. Rosero and Omar S. Gómez presented unsupervised clustering with unit tests and random values to regulate those test cases associated with modifications and connected to databases [5]. The authors in [14] proposed a novel approach called REPiR for TSP. The REPiR approach combines regression test prioritization (RTP) and information retrieval that primarily deals with the ranking problems. RTP analyzes the test cases written in a programming language, and IR looks into the documents written in natural language.

Jinfu Chen [26] presented an adaptive random sequences clustering technique. This technique uses white box information such as CC and black box information such as input data for testing and the expected output against input values. In [31], the researcher presented ant colony optimization (ACO) for TSP, and that technique is acquired from natural ant behavior. Their technique considers test cases and faults related to software as an input with the objective function. The ants navigate from the initial fault node to the final fault node, and the initial edges node to the final fault node are chosen randomly between ants. They use a heuristic function to authorize the memory information related to paths for fault coverage. The ants' movement is recorded, and paths are updated until or unless no ant moves on any path. Then they prioritize the test suite based on these paths. The test cases covering all faults in minimum time are given high priority. Mills introduced fault seeding in 1972 [27], in which the effectiveness of the test suite is presented using a comparison of the total seeded faults, and total detected faults. There are different software testing techniques for injecting faults [24,28–30].

Table 1 presents a comparative analysis of the proposed approach with existing TSP techniques. The summary is based on the type of methodology, test suite minimization (TSM), TSP, test suite selection criterion, the contribution of the new software platform, the use of AI-based models, and relative demerits. The test suite selection criterion includes the analysis approach based on fault detection effectiveness (FDE) and CC.

Table 1. Comparative analysis of proposed approach with existing TSP techniques.

S.No	Methodology	TSM	TSP	Test Suite Selection Criterion	Software Platform Developed	AI-Based Model	Optimization	Relative Demerits
1	Modification based approach [32]	yes	Yes	Correlation index based on FDE and CC	No	No	No	Both techniques can give good results
2	Identify the code lines affected by the modification [33]	No	Yes		Yes	No	No	Proposed TSP approach can identify errors and test suite is prioritize test suites.
3	Analysis based on 9 mechanisms [34]	No	Yes	Fault detection percentage	yes	Yes	No	Optimization based TSP performs better.
4	Coverage data [35]	Yes	No	Analyzing dataflows and tracking executions	Yes	No	No	Regression tests are reduced, but no test suites are prioritized.
5	Integration of 5 algorithms [36]	No	Yes	CC	No	Yes	Yes	GA perform well, while greedy approaches are surprisingly effective
6	Methodology based on ReduceUSession and GA [37]	No	Yes	Finding traces from user session data and selection using GA	No	yes	Yes	Decrease execution time but no increase in fault detection rate.
7	CC and fitness function [38]	Yes	No	CC and runtime of test cases	No	yes	Yes	Proof of concepts on 3 projects, performance analysis shows an increase in the granularity of fitness function, leading to TSM.
8	Mean percentage of fault detection rate [39]	No	Yes	CC and Runtime	Yes	yes	Yes	Applied to 5 case studies, the fault detection rate was improved.
9	Partial CC [39]			path building and optimal path selection.		yes	Yes	Solve the triangle problem and reduces test suite.
10	Multi-objective optimization [40]	No	Yes	Requirement coverage and cost	No	yes	Yes	Minimize the cost and maximize the coverage
11	Fuzzy system[41]	No	Yes	selection based on defects and test coverage	Yes	Yes	No	Case studies based on the GSM test database and the proposed defects detection rate in a short time.
12	Optimization and fuzzy logic-based expert system [42]	Yes	No	Selection based on performance, throughput and CC.	No	Yes	Yes	The proposed expert system is best suited for multi-objective optimization based on test suite reduction.

3. Materials and Methods

This section presents the proposed methodology for TSP based on a four-fold methodology. Figure 2 shows the block diagram of the proposed methodology. The proposed four-fold methodology of test suite prioritization is based on reinforcement learning.

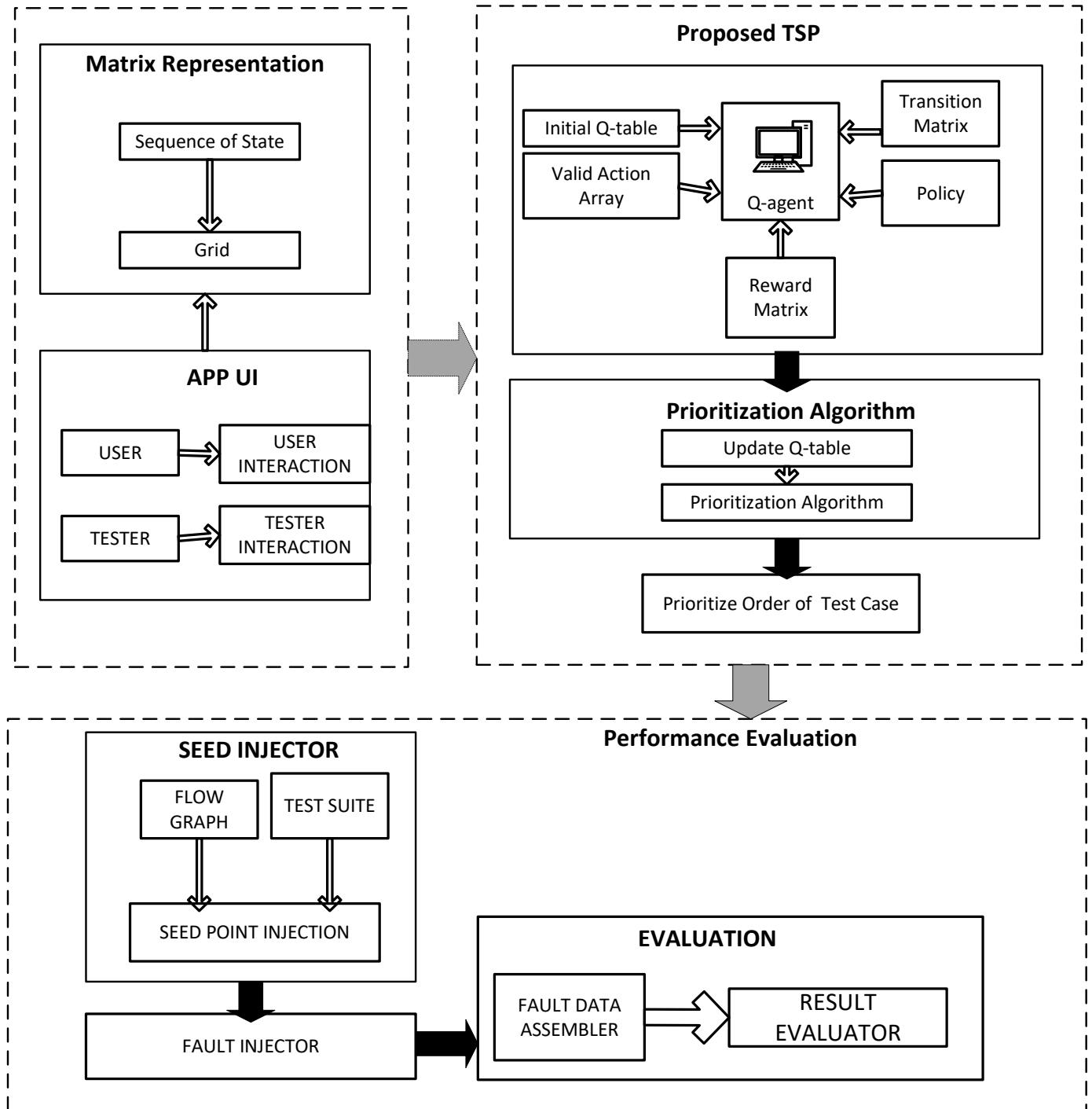


Figure 2. Proposed methodology for test suite prioritization based on RL based optimization.

First, the testers' and users' log datasets are prepared using the proposed interaction recording systems for the android application. Second, the proposed reinforcement learning model is used to predict the highest future reward sequence list from the data collected in the first step. Third, the proposed prioritization algorithm signifies the prioritized test suite.

Lastly, the fault seeding approach is used to validate the results from software engineering experts. In the proposed interaction recording systems, the testers' and users' interactions are recorded when the user or tester interacts with the system. Then, the sequence of activities or states is extracted from those users' and testers' logs. Next, the number of states from the testers' log is computed for the environment construction of the proposed RL model. Next, the agent is trained on the matrix representing the testing environment using the Q-learning algorithm. During learning, the agent updates and maintains its Q-table. The final updated Q-table is obtained when the agent learns the complete environment. In the next step, the algorithm identifies the sequence of states with the highest reward values from starting state to the final state. The prioritization algorithm is then applied based on the sequence with the highest reward values.

For prioritization of the user and tester interactions, the system records the interaction when users and testers use the application. User interactions are recorded when the user interacts with the GUI of the application. An interaction list is generated when the user logs in to the application or performs any other application activity, such as 'Sign up' and 'Forget password'. The list maintains and adds all actions performed by the user in the GUI of the application. At each instance or button click, the activity name and the button ID are stored in the list. The addition to the list ends when the user logs out from the system, or the application is terminated forcefully. The final list is added to the database for persistence. Tester Log is also recorded in the same way when the tester interacts with the system to implement the test cases for testing the system. A tester list is generated by clicking buttons on the application interface while testing the application. By executing all test cases, the logs of every test case according to its type are collected. The user and tester logs are then processed to get the sequence of states visited by users or testers. Next, all possible states are extracted from the tester logs for the construction of the matrix representing the testing environment and computing frequency count and complexity from the list of user logs. Finally, the mean, median, and complexity of the states are calculated from the list, excluding the initial and final states for the reward matrix calculation.

Faults are seeded or injected through software. In the first step, frequent and rare paths are extracted by sequential pattern mining [13,27] method. Sequence mining is a handy data mining technique for understanding the user and tester behavior with the system [13]. After applying sequential pattern mining, the system gets frequent paths and rare paths from the test suite. In the second step, the program is given to the software in which we want to inject the faults for evaluation [30]. The software scans the source code and generates the control flow graph according to the program [28]. The software gets the program's seed point injection locations and workflow in the third step. Faults are injected in particular locations in the workflow through the fault injector, and then for evaluation, test the program after faults are injected in the program [30]. The fault data assembler assembles the related data for analysis, such as the number of injected faults, detected faults, and undetected faults. The result evaluator obtains the data from the fault data assembler and generates the results according to the data. In Q-learning, the agent learns from its environment. Figure 3 presents the workflow of agent learning in the proposed approach.

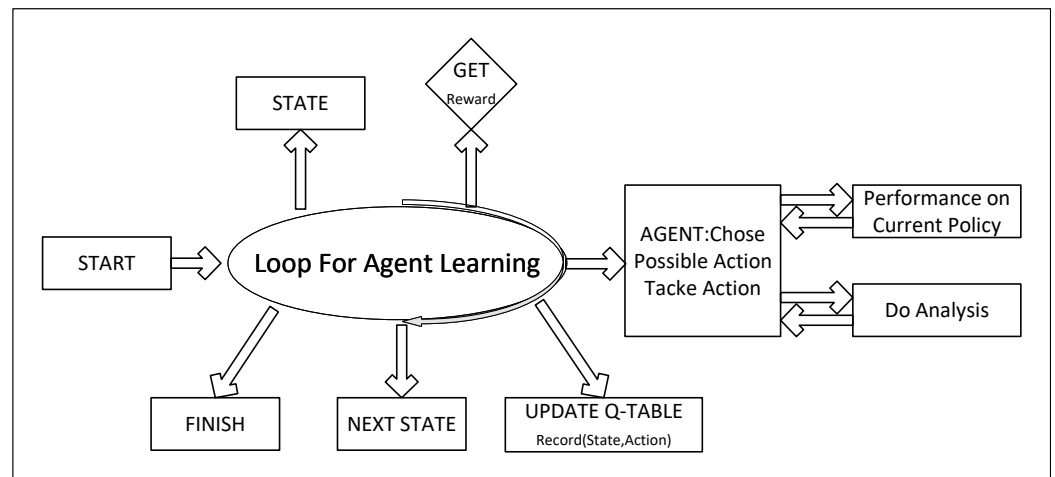


Figure 3. Flow of the learning methodology of the proposed for test suite prioritization.

The agent starts from the start state that becomes the current state, and the agent receives a reward from the current state. Then, the agent analyzes valid actions from the valid action array for the current state and the reward values against valid actions from the reward matrix. After analysis, the agent takes action, moves to the next state, and updates its Q-Table. Then, the next state becomes the current state, and the entire process is repeated according to the defined episodes. For instance, in our experimental configuration, the agent repeats the episode 1000 times, and the agent navigates from the initial state to the final state in the matrix representing the testing environment. The more agents explore the environment, the better the Q value achieved.

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

Equation (1) (known as the Bellman equation) is used to compute the maximum future reward. It is the reward the agent received for entering the current state s plus the maximum future reward for the next state s' . The Q value is iteratively approximated based on the Bellman equation. The Q-learning equation is given by

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \quad (2)$$

In Equation (2), α is the learning rate that controls the difference between previous and new Q values. As the agent explores more and more of the environment, and the approximated Q values start to converge to Q. The final updated Q table is obtained from 1000 episodes of training from the start state to the end state. After obtaining the updated Q-Table, the algorithm extracts the sequence of states from the start state to the final state with maximum reward. Finally, the proposed prioritization algorithm prioritizes the test cases based on this extracted sequence. For instance, if two or more test cases have the same similarity index, the length of test cases is another selection parameter. The test case with greater length is given priority among test cases with the same similarity index.

The proposed Q learning algorithm-based TSP mechanism initializes Q-matrix with zeros. The value of γ representing the learning rate is set between 0 and 1. The reward matrix is constructed and initialized with the reward value of each state. To make the agent understand its environment well, the agent is trained with more episodes as they say, the more the agent explores the environment, the better the Q value obtained. For each episode, the agent starts from state-0, navigates to the goal state with a random navigation path, and selects one action among all possible actions for the current state (S) with the highest Q value. After performing an action on the state (S), the agent moves to the next state (S'), and the Q-table is updated using Equation (1). The proposed Q-Learning based TCP algorithm is presented in in Algorithm 1.

Algorithm 1 The proposed learning algorithm for test suite selection.

Input: Initial Q-Table, Valid Action Array, Reward Matrix, Transition Matrix, Policy, and testing environment matrix .

Output: Updated Q-Table.

```

function GETVALIDACTION(current-state)
    action=random.choice(valid-actions[current-state])
    IF action = (−1):
        getvalidation(current-state);
    ENDIF
    RETURN action
end function
for all episode i in 1000 episodes: do
    start-state=0
    current-state=start-state
    while current-state!=goalstate: do
        action=getvalidation(current-state)
        next-state=transition-matrix[current-state][action]
        future-rewards=[]
        for all action-next in valid-actions[next-state]: do
            Add to future-rewards q-matrix[next-state][action-next])
        q-state=reward[current-state][action]+gamma*max(future-rewards)
        q-matrix[current-state][action]=q-state
        current-state=next-state
        end for
    end while
end for
    OUTPUT q-matrix

```

The sequence extraction algorithm creates a state sequence from the initial to the final state from the final updated Q-table based on the greater Q value. This sequence is called the highest future reward sequence. Algorithm 2 presents the proposed highest future reward sequence extraction algorithm.

Algorithm 2 The Sequence extraction algorithm.

Input: Updated Q-Table and testing environment matrix

Output: Sequence of states with highest future reward

$a \leftarrow \text{Length of Updated Q-Table}$

$A \leftarrow \text{Array}$

$c \leftarrow \text{Actions}$

$n = 0$

$m = 0$

while $n \neq a$ **do**

$A[m] = \text{state}$

for all $C_{i+1} \in c$ **do**

start from state n, select action highest reward

Perform action on the testing environment matrix and move to the next state

update n

variable to the next state

$n = \text{Next State}$

end for

$m = m + 1$

end while

Output: print A[m]

The prioritization algorithm takes tester logs and the highest future reward sequence from the Q-Learning algorithm as the input. The proposed prioritization algorithm is presented in Algorithm 3.

Algorithm 3 The proposed prioritization algorithm.

Input: Highest future reward sequence and test suite T

Output: prioritized order test-suite

$L \leftarrow \text{Length of Test suite } T$

$F \leftarrow \text{Highest future reward sequence}$

$S = \text{Hash table with two values (length, similarity)}$

$i = 0$

$k = 0$

for all $T_i \in L$ **do**

$S[\text{similarity}] = \text{compute similarity index of } T_i$

$S[\text{Length}] = \text{Length of } T_i$

 sort S by similarity DESC

for all $k \in S$ **do**

 if similarity $S[k] == \text{similarity of } S[k + 1]$

 swap item of S with maximum length first

end for

end for

 Output: Print hash table S

The highest future reward sequence is mapped into the tester log suite, and the cosine similarity of the future reward sequence with each tester log is calculated using Equation (3):

$$\cos(\Theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}} \quad (3)$$

Tester logs with the highest cosine similarity value are prioritized high and vice versa. If test cases have the same cosine similarity index, then the test case's length is considered as secondary parameter and Test case with greater length value is prioritized high.

4. Results and Discussion

In this section, the results of the proposed methodology are discussed. First, we present a TSP example for the proof of concept of the proposed methodology. Second, we present the case study based assessment of the proposed methodology. Finally, we present the proposed technique's significance and comparative analysis, concluding remarks on threats to validity.

4.1. TSP Example as Proof of Concept

In this sub-section, the proof of concepts of the proposed model for TSP are presented for a clear understanding of the proposed TSP mechanism using the theorem. In this example, we consider an application that utilizes some users' and testers' logs. First, the Q-learning algorithm takes users' and testers' logs, testing environment matrix, reward matrix, transition matrix, initial Q-Table, and valid action matrix as the input and calculates the updated Q-Table. Next, another algorithm extracts a sequence of states with the highest reward value from the initial state to the final state. Finally, the prioritization algorithm prioritizes the test cases based on the sequence with the highest reward value. Testers' logs are recorded while implementing the test cases; the test case log is "S0(C), S3(A), S8(A), S10," where S0(C) means the tester performs action C on the state S0 which leads to state S3 and then tester performs action A on the state S3 which leads to state S8 and so on. After tester logs collection, the actions from states are separated to obtain a sequence of states for constructing the testing environment matrix. Table 2 presents an example of tester logs based on the states S.

Table 2. Tester logs states sequences.

Test Case No.	Sequence
1	S0, S1.
2	S0, S1, S5.
3	S0, S2, S6.
4	S0, S1, S4, S10.
5	S0, S1, S5, S10.
6	S0, S3, S8, S10.

After the testers' logs collection, the user logs are collected. For instance, different user logs are automatically recorded using case study applications. Each entry in the user log consists of state numbers, and action values such as "S0(B), S3(B), S8(A), S10", where S0 is the state name, B is the action, and ',' separates the events of user logs. Then preprocessing function is applied to the user logs, separating actions from activity names of all the user logs. After preprocessing and data preparation, the sequence of states of user logs is prepared. The user logs state sequences is presented in Table 3.

Table 3. User logs states sequences.

Test Case No.	Sequence
1	S0, S1.
2	S0, S1, S5.
3	S0, S2, S7, S10.
4	S0, S3.
5	S0, S1, S4, S10.
6	S0, S3, S9, S10.
7	S0, S2, S7.
8	S0, S1, S5, S10.
9	S0, S3.
10	S0, S3, S8, S10.

The agent learning environment can be presented as a grid or matrix. In the testing environment matrix, an agent can perform only a few correct actions; the rest are invalid ones. Valid action array includes left, right, up, down, and no action. In addition, the same action can be invalid in the case of border states, resulting in an agent position outside the testing environment states. From the above users' and testers' logs, there are 11 states obtained, ranging from S0 to S10. The algorithm converts them into a testing environment matrix. For 11 states, a 3×4 matrix is needed, where every cell represents a state.

Table 4 presents an example of the testing environment matrix in which every cell symbolizes a state. The total number of states is 11, but the 3×4 matrix has 12 states, as shown in the example in the Table 4. Thus the 12th state of the testing environment matrix is considered as a block state. The block state means the agent cannot move to this state.

Table 4. Agent navigation testing environment matrix.

S0	S1	S2	S3
S4	S5	S6	S7
S8	S9	S10	Block

The reward is allocated to every state of the agent navigation testing environment. The agent learns from the testing environment matrix based on these reward values. Agent prefers valid actions which maximize its reward value. The following functions are used to calculate the reward values of each state.

$$f(r) = f(x) * c \quad (4)$$

In Equations (4) and (5), x represents the state frequency value, r denotes the reward value, c corresponds to the complexity of the state defined by the programmer, me represents the mean value, and md denotes the median value. Equation (4) shows that the reward of each state is calculated using $f(x)$ multiplied by the complexity of the state.

$$f(x) = \begin{cases} 0 & \text{invalid} \\ -\lambda & < me \\ +1 & me \leq x \leq md \\ +\lambda & > md \end{cases} \quad (5)$$

Equation (5) represents that function $f(x)$ assigns 0 to invalid actions of the agent and 10 to λ value, and +1 where the frequency is in between the mean value and median value.

The block state is the state in which the agent cannot move. S0 is the initial state, and S10 is the goal or final state shown in Table 4. To encourage the agent to navigate into the final goal, the reward with a negative number means to avoid the action, and the reward in a positive number encourages to do the action to maximize the reward. For example, the final state 'S10' is given three times the highest positive reward amount to encourage the agent to reach the final state. Block states are given a reward amount of five times more negative than the highest reward to discourage the agent from moving there. Reward values of each action on a state are stored in the reward matrix. The reward matrix contains the reward values of every state; for example, the reward values of state S0 are 0, 1, 0, 10, −1.

The transition matrix is necessary to make the agent understand the transitions for each state. The value −1 in the matrix means that the transition is impossible, as it is not possible on the first state S0 for the top action. The values of state S0 in the transition matrix are {−1, S4, −1, S1, S0}. If the agent performs an action, it move to state S4. The transition matrix is constructed based on the states and actions. It shows by taking action 'a' what would be the agent's next state 's'. The initial Q-Table is initialized with zeros, and its size is $M \times N$, where M is the number of states and N is the number of actions. The agent takes an initial Q-Table to maintain the history of reward values in the learning process. When the agent takes any action in the testing environment, the reward value is stored in the initial Q-Table. After every episode, the agent updates its initial Q-Table, and the final updated Q-Table is obtained at the end of the learning process. Valid action matrix is for providing the agents with valid actions it can practice on each state during its learning episodes:

- Valid actions are (Up = 0), (Down = 1), (Left = 2), (Right = 3) and (NA = 4).
- −1 represents invalid actions that the agent cannot perform and restricts the agent while programming the agent to avoid −1.

In Q-Learning, the valid action matrix is essential because it contains valid actions for every state for the agent navigation over different states in a learning environment. For example, valid actions of State S0 are {−1, 1, −1, 3, 4}, stating that the agent cannot take up and left actions in State S0, and the agent can take only down, left, and no action in State S0. The up and left actions are invalid for the agent on the State S0, and the rest of the actions are valid.

During the learning process, the agent updates its initial Q-Table according to the reward values against actions. Then, after learning the complete environment, the agent gets an updated Q-Table. In this example, the agent learns the environment in 1000 episodes; after 1000 episodes, the agent's updated Q-Table is given in Table 5.

Table 5. Updated Q-Table.

State/Actions	0	1	2	3	4
S0	0	74.6889	0	83.6889	65.9511
S1	0	92.1111	67.9511	76.3111	72.6889
S2	0	82.8889	74.6889	69.1111	65.3111
S3	0	73.8889	67.3111	0	58.1111
S4	67.9511	61.1111	0	92.1111	72.6889
S5	83.6889	113.889	74.6889	82.8889	90.1111
S6	76.3111	141.111	92.1111	73.8889	111.889
S7	60.1111	0	0	91.1111	71.8889
S8	74.6889	0	0	113.889	90.1111
S9	92.1111	0	61.1111	141.111	111.889
S10	82.8889	0	113.889	0	90.1111

For extracting the sequence with highest reward value from updated Q-Table, the reward allocation algorithm considers the testing environment, updated Q-Table and actions. In the testing environment matrix, the initial state is S0. In the updated Q-Table for State S0, the highest reward value is on action 3, so the agent performs action 3, which denotes action to the right. After taking the right action, now the agent is on State S1 based on the sample testing environment matrix in Table 4. The agent keeps track of the previous state and maintain a list. On State S1, the highest reward value is against action 1. When the agent performs the down action from State S1, it reaches State S5, and the agent updates the list to S0, S1, and S5. On State S5, the highest reward value is again on action 1, so the agent performs the down action, reaching State S9, and the list is updated. On State S9, the reward value is maximal on action number 3, which is the right action. When an agent performs the right action from S9, it reaches State S10, the final state of the testing environment matrix. Thus the agent reached the goal state, and the final updated list of state sequences is S0, S1, S5, S9, and S10. The best state navigation sequence of the agent with maximum Q value to reach the goal is S0, S1, S5, S9, and S10.

Prioritization algorithm maps that sequence on the testers' log, computes cosine similarity on the logs shown in Equation (3) and then rearranges the test cases according to the similarity. The test case with the highest similarity is prioritized as high and vice versa. If the test cases have the same similarity index, then the length of the test case is considered another parameter. The test case with a large length value is prioritized as high. The prioritized test cases are given in Table 6.

Table 6. Prioritize tester logs with similarity.

Test Case No.	Sequence	Similarity
5	S0, S1, S5, S10.	1.0
4	S0, S1, S4, S10.	0.6
2	S0, S1, S5.	0.6
6	S0, S3, S8, S10.	0.4
1	S0, S1.	0.4
3	S0, S2, S6.	0.2

4.2. Performance Analysis

In this section, the proposed methodology of TSP is validated using five android application case studies. The list of these case studies is presented and described below:

- Health care case study.
- Intelligent vehicle assistance system (IVAS).
- Real estate case study.
- Mobile shop POS app.
- Jobs portal case study.

The evaluation of each application is done in four steps; In the first step, the testers' and users' log datasets are prepared for the application. The second step uses the Q-learning model to predict the highest future reward sequence. In the third step, the prioritization algorithm predicts the prioritized test suite. Lastly, the fault seeding approach is used to validate the results from software engineering experts. For this purpose, two datasets are prepared to evaluate each application's case study. Firstly, the dataset is generated from the test-suite execution sequences. In contrast, the second dataset is generated from the sequence of the interaction of users with the application. The details about all the apps datasets are shown in Table 7.

Table 7. Structure of apps datasets.

App	Nos. of Test Cases	Nos. of App Users	Test Sequence' Avg Length	User Sequence Avg Length
Health Care Case Study	123	35	9	13
IVAS Case Study	155	40	11	15
Real Estate Case Study	131	30	11	14
Mobile Shop POS Case Study	87	40	9	15
Jobs portal Case Study	106	35	8	11

The validation of the proposed prioritization technique is performed using fault seeding-based performance analysis. Fault seeding is a technique for injecting faults into the application. Then it identifies in a given test suite how many top test cases can detect the seeded faults into the program, based on the confidence level. Fault injection can be classified on different assumptions, such as fault type, test environment, test objective, and level of faults, to name a few [30]. In this study, for evaluation of this study, two types of faults were considered, namely computational and domain faults [27]. The fault injection mechanism is based on the literature, as presented in Table 8 [13,27].

Table 8. Error distribution based on literature.

Class of Errors	Percentage
Computational	13
Initialization	12
Interface	23
Data	22
Logic/Control	30

Each type of fault is seeded into specified percentage and based on the seeded percentage. After seeding faults in the application, the top 30 percent of test cases are executed to evaluate the prioritized test cases. This shows how many faults are detected because the primary goal of TSP is to reorder the test cases so that maximum faults can be detected in the minimum number of test cases. When the top 30 percent of the health care case study test cases were executed using the random approach, only 41 percent of faults were detected from the total seeded faults. Implementing the top 30 percent of test cases from t-SANT prioritize test suite, so it detects 82 percent of faults from total seeded faults. By executing the top 30 percent of test cases from Q-Learning, prioritize the test suite, and detects 85 percent faults from total seeded faults, and the total seeded faults in the health care case study are 35. The comparative fault detection percentage analysis of the proposed and baseline techniques is presented in Figure 4. Health care case study results are graphically shown in Figure 4a. The proposed approach-based TSP technique comparatively detects more faults in the top few test cases. Similarly, injecting 30 faults in IVAS and executing the top 30 percent test cases to evaluate IVAS for TSP. When executing the top 30 percent of the IVAS case study test cases using the random approach, only 51 percent of faults are

detected from the total seeded faults. On the other hand, the t-SANT detects 77 percent faults from the total seeded faults.

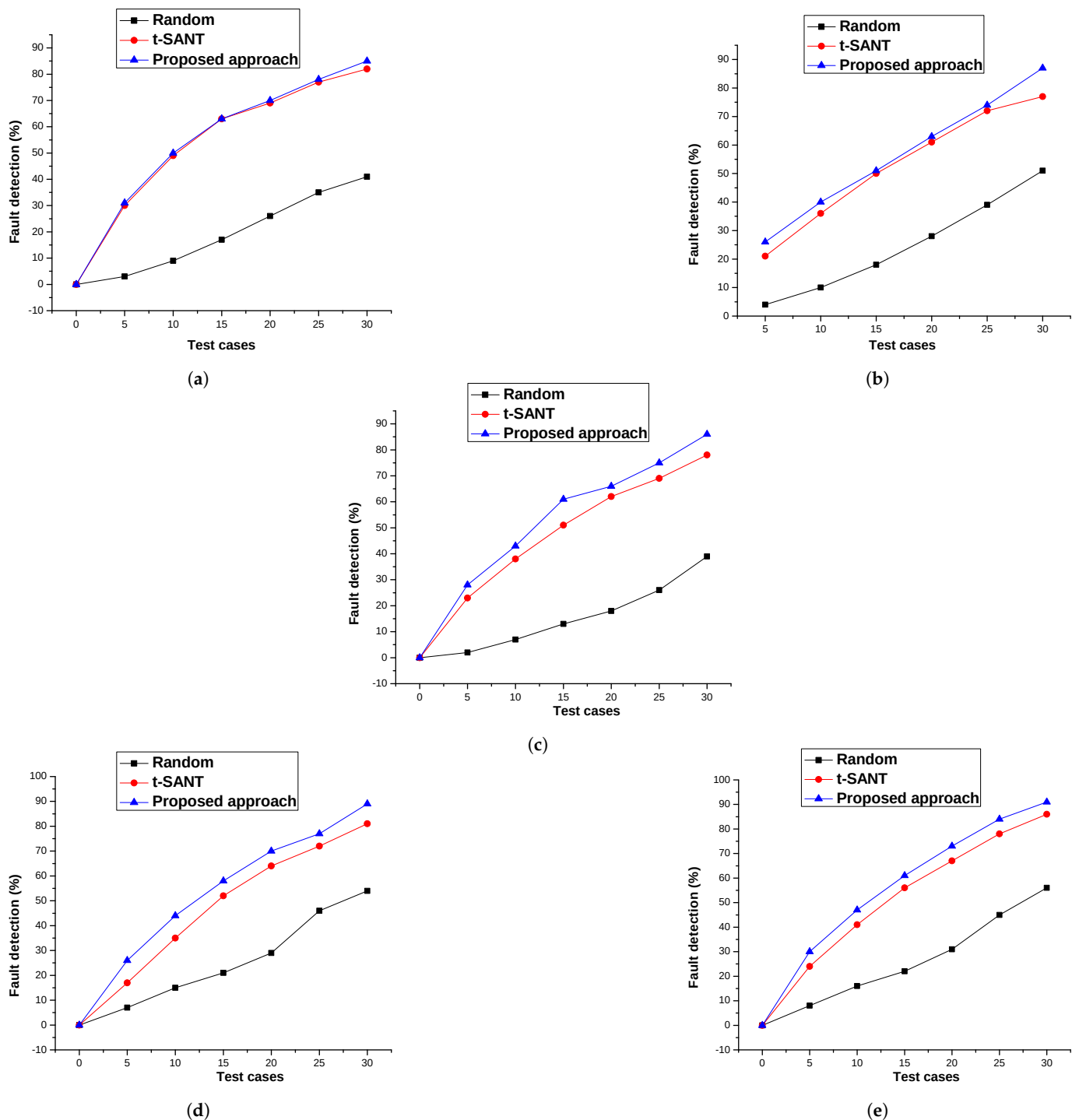


Figure 4. The comparative analysis based on fault detection percentage using the case study applications. (a) Health care case study. (b) IVAS case study. (c) Real estate case study. (d) POS Case Study. (e) Job portal case study.

Therefore, the proposed Q-Learning approach outperforms the compared approach by detecting 87 percent of faults from total seeded faults. The IVAS case study results are graphically shown in Figure 4b. Similarly, 40 faults were injected into the real estate application for TSP evaluation. After employing the random approach, 39 percent of faults are detected from total seeded faults. In contrast, the t-SANT approach detects 78 percent

of faults from total seeded faults. Finally, the proposed Q-Learning has improved the fault detection by 120% and 11% against the random and t-SANT approaches, respectively. Real estate case study results are graphically shown in Figure 4c. Thirty-five faults were injected in mobile shop POS, and then execution of the top 30 percent test cases is used for the evaluation. The random approach is able to detect 54% of faults from the total seeded faults. Implementing the t-SANT approach for prioritizing the test suite, the detection rate increased to 81%. The proposed Q-Learning detected 89 percent of faults from the total seeded faults. Mobile shop POS case study results are graphically shown in Figure 4d. In the job portal case study, the injected faults were 25, and the illustration of the faults detection results is presented in Figure 4e. As a result, the random approach has attained 56% faults detection, and the t-SANT approach detected 86% faults. Similarly, the Q-Learning approach detected 91 percent of faults.

4.3. Significance and Comparison

This section discusses the significance, comparison, and threat to the validity of the proposed TSP technique. To compare the case study applications, the same number of faults were injected in all five applications to evaluate the prioritized order of test cases using the faults detection rate. The obtained results show that the average percent of fault detection of the random technique is 48.2 percent, the average percent of fault detects of the t-SANT technique is 82.8 percent, and the average percent of fault detection of the proposed technique is 87.6 percent based on all five applications' fault detect results. These results assert that the Q-Learning algorithm is a beneficial technique for TSP, and it detects maximum faults in minimum test cases to reduce the testing efforts. The comparison of the proposed approach with other TSP approaches is presented in Figure 5.

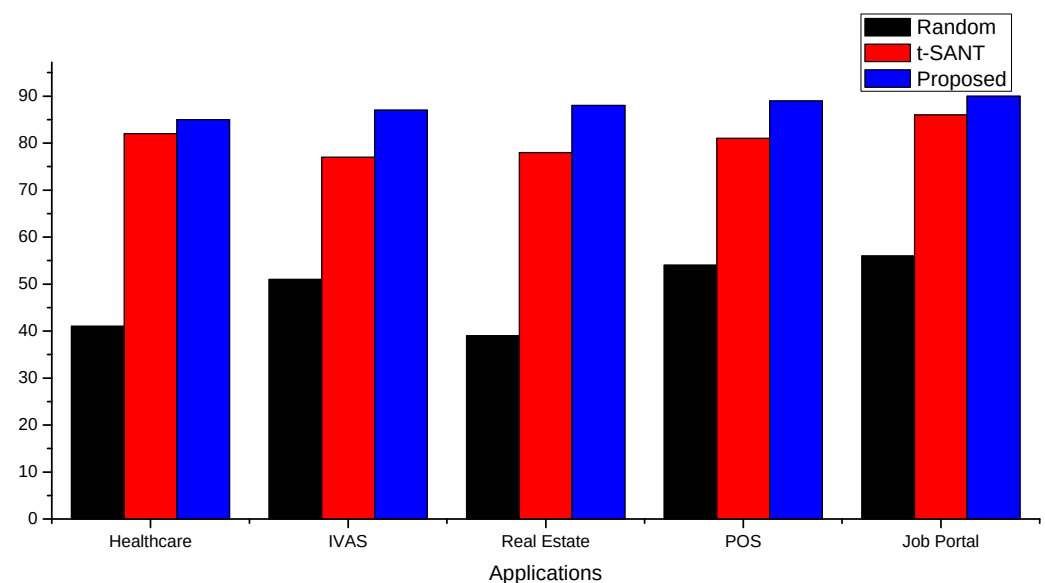


Figure 5. Comparison of the proposed TSP approach with baseline approaches.

The Threats to the Validity

Our method of TSP based on RL is designed for the android-based case study application. It relies on converting sequence patterns extracted from the user and tester logs to a matrix. If the RL model's testing accuracy is low, the testing environment matrix obtained from the sequences needs improvement. In this case, the prioritized sequence may not capture the characteristics of the specified logs. Hence, it leads to the poor performance of TSP. In the experimentation, we adopt five datasets prepared from android applications to evaluate the performance of the proposed TSP method. The performance of the proposed method may be different for various datasets. However, it is more stable than other TSP techniques used for the comparison. We incorporated several open-source software codes to support the main proposal of this study. The advantages of open-source software are

broad applications and continuous updates. In addition, evaluation results of the proposed TSP approach were thoroughly reviewed by experts in the field. Previous studies have broadly utilized the evaluation metrics involved in our experiments. For instance, the fault detection rate is a prominent metric for assessing the significance of TSP techniques.

Moreover, we compared two widely used baseline methods in the literature for the TSP. Because we have not yet found any prioritization techniques specifically designed for testing android source code. Therefore, we compared the proposed methodology with random and t-SANT prioritization algorithms. The proposed RL-based TSP approach is a step toward a promising research direction to apply deep Q learning (DQN) and other reinforcement learning algorithm with convergence in existing TSP and TSM techniques. As future work, incorporating more complicated software programs can be used for the validation to prove the practicality of the proposed TSP technique. Moreover, the case study applications were chosen based on their availability and open-source code. Our experiments used a relatively small number of fault seedings, and various thresholds can be considered to prove the sustainability of the proposed approach. Although our experiment did not involve large-scale programs with millions of lines of code, we believe that our methodology can be generalized to other complex applications.

5. Conclusions

AI and machine learning methods have proven their practical suitability in interdisciplinary applications. A wealth of tools have been developed to assess the development and deployment of AI models. This study proposes a reinforcement learning-based fourfold methodology for TSP. The proposed method prioritises the test cases automatically by minimising the faults and thus reduces the cost of software testing. First, the RL model is trained based on the datasets prepared from the user and tester interaction system. The RL model is based on Q-agent and utilises a reward function for assigning rewards to every state of the testing environment. The agent optimises the reward function based on fault detection and test suite minimisation. Second, the proposed reinforcement learning model is used to predict the highest future reward sequence list from the data collected in the first step. The proposed sequence extraction algorithm is used to extract the highest reward value sequence from the initial to the final state. Third, the sequence of test cases with maximum reward is input into the prioritisation algorithm to minimise and prioritise the test suite. Lastly, the fault seeding approach is used to validate the results from software engineering experts. As proof of concept, the proposed methodology is validated using five case study applications. As a future work, we will use deep reinforcement learning to automate test suite selection and prioritisation policies.

Author Contributions: Conceptualization, I. and M.M.; Data curation, M.A.Z.; Formal analysis, M.A.Z.; Methodology, M.W., M.M. and I.; Supervision, M.M. and J.K.; Writing—original draft, M.W. M.M. and M.A.Z.; Writing—review & editing, I. and J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This study was supported by the National Research Foundation of Korea grant (NRF-2022R1A2C1012037).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Abid, R.; Nadeem, A. A novel approach to multiple criteria based test case prioritization. In Proceedings of the 2017 13th International Conference on Emerging Technologies (ICET), Islamabad, Pakistan, 27–28 December 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 1–6.
2. Khatibsyarbini, M.; Isa, M.A.; Jawawi, D.N.; Tumeng, R. Test case prioritization approaches in regression testing: A systematic literature review. *Inf. Softw. Technol.* **2017**, *93*, 74–93. [[CrossRef](#)]
3. Ammar, A.; Baharom, S.; Ghani, A.A.A.; Din, J. Enhanced Weighted Method for Test Case Prioritization in Regression Testing Using Unique Priority Value. In Proceedings of the 2016 International Conference on Information Science and Security (ICISS), Pattaya, Thailand, 19–22 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–6.

4. Konsaard, P.; Ramingwong, L. Using artificial bee colony for code coverage based test suite prioritization. In Proceedings of the 2015 2nd International Conference on Information Science and Security (ICISS), Seoul, Korea, 14–16 December 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 1–4.
5. Rosero, R.H.; Gómez, O.S.; Rodríguez, G. Regression Testing of Database Applications Under an Incremental Software Development Setting. *IEEE Access* **2017**, *5*, 18419–18428. [[CrossRef](#)]
6. Hemmati, H.; Fang, Z.; Mantyla, M.V. Prioritizing manual test cases in traditional and rapid release environments. In Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 13–17 April 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 1–10.
7. Knauss, E.; Staron, M.; Meding, W.; Söder, O.; Nilsson, A.; Castell, M. Supporting continuous integration by code-churn based test selection. In Proceedings of the Second International Workshop on Rapid Continuous Software Engineering, Florence, Italy, 23 May 2015; IEEE Press: Piscataway, NJ, USA, 2015; pp. 19–25.
8. Nagar, R.; Kumar, A.; Singh, G.P.; Kumar, S. Test case selection and prioritization using cuckoos search algorithm. In Proceedings of the 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), Noida, India, 25–27 February 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 283–288.
9. Khan, S.U.R.; Lee, S.P.; Parizi, R.M.; Elahi, M. A code coverage-based test suite reduction and prioritization framework. In Proceedings of the 2014 Fourth World Congress on Information and Communication Technologies (WICT), Malacca, Malaysia, 8–11 December 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 229–234.
10. Saifan, A.A. Test Case Reduction Using Data Mining Classifier Techniques. *JSW* **2016**, *11*, 656–663. [[CrossRef](#)]
11. Huang, R.; Zong, W.; Chen, J.; Towey, D.; Zhou, Y.; Chen, D. Prioritizing Interaction Test Suites Using Repeated Base Choice Coverage. In Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA, 10–14 June 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 174–184.
12. Morozov, A.; Ding, K.; Chen, T.; Janschek, K. Test Suite Prioritization for Efficient Regression Testing of Model-Based Automotive Software. In Proceedings of the 2017 International Conference on Software Analysis, Testing and Evolution (SATE), Harbin, China, 3–4 November 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 20–29.
13. Muzammal, M. Test-Suite Prioritisation by Application Navigation Tree Mining. In Proceedings of the 2016 International Conference on Frontiers of Information Technology (FIT), Islamabad, Pakistan, 19–21 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 205–210.
14. Saha, R.K.; Zhang, L.; Khurshid, S.; Perry, D.E. An information retrieval approach for regression test prioritization based on program changes. In Proceedings of the 37th International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; IEEE Press: Piscataway, NJ, USA, 2015; Volume 1, pp. 268–279.
15. Spieker, H.; Gotlieb, A.; Marijan, D.; Mossige, M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 June 2017; ACM: New York, NY, USA, 2017; pp. 12–22.
16. Iqbal, N.; Kim, D.H. Iot task management mechanism based on predictive optimization for efficient energy consumption in smart residential buildings. *Energy Build.* **2022**, *257*, 111762.
17. Imran, M.; Zaman, U.; Imtiaz, J.; Fayaz, M.; Gwak, J. Comprehensive survey of iot, machine learning, and blockchain for health care applications: A topical assessment for pandemic preparedness, challenges, and solutions. *Electronics* **2021**, *10*, 2501. [[CrossRef](#)]
18. Kim, D.H. Artificial intelligence-based Modeling Mechanisms for Material Analysis and Discovery. *J. Intell. Pervasive Soft Comput.* **2022**, *1*, 10–15.
19. Qayyum, F.; Kim, D.H.; Bong, S.J.; Chi, S.Y.; Choi, Y.H. A Survey of Datasets, Preprocessing, Modeling Mechanisms, and Simulation Tools Based on AI for Material Analysis and Discovery. *Materials* **2022**, *15*, 1428.
20. Jamil, F.; Kim, D. An ensemble of prediction and learning mechanism for improving accuracy of anomaly detection in network intrusion environments. *Sustainability* **2021**, *13*, 10057.
21. Zaman, U.; Mehmood, F.; Iqbal, N.; Kim, J.; Ibrahim, M. Towards Secure and Intelligent Internet of Health Things: A Survey of Enabling Technologies and Applications. *Electronics* **2022**, *11*, 1893. [[CrossRef](#)]
22. Lachmann, R.; Schulze, S.; Nieke, M.; Seidl, C.; Schaefer, I. System-Level Test Case Prioritization Using Machine Learning. In Proceedings of the 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), Anaheim, CA, USA, 18–20 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 361–368.
23. de Oliveira Paiva, É.; Guelpele, M.V.C. Improvement of Q-Learning Algorithm Convergence with Intelligent Tutoring Systems and Tabu Search. *Int. J. Mod. Educ. Res.* **2016**, *3*, 1–5.
24. Hwang, K.S.; Lin, J.L.; Yeh, K.H. Learning to adjust and refine gait patterns for a biped robot. *IEEE Trans. Syst. Man, Cybern. Syst.* **2015**, *45*, 1481–1490. [[CrossRef](#)]
25. Sun, C. Fundamental Q-learning Algorithm in Finding Optimal Policy. In Proceedings of the 2017 International Conference on Smart Grid and Electrical Automation (ICSGEA), Changsha, China, 27–28 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 243–246.
26. Chen, J.; Zhu, L.; Chen, T.Y.; Towey, D.; Kuo, F.C.; Huang, R.; Guo, Y. Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering. *J. Syst. Softw.* **2018**, *135*, 107–125. [[CrossRef](#)]

27. Khan, T.A.; Muzammal, M.; Ijaz, A. On effectiveness of fault-seeding using interaction patterns. In Proceedings of the 2015 13th International Conference on Frontiers of Information Technology (FIT), Islamabad, Pakistan, 14–16 December 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 119–124.
28. Kooli, M.; Bosio, A.; Benoit, P.; Torres, L. Software testing and software fault injection. In Proceedings of the DTIS: Design and Technology of Integrated Systems in Nanoscale Era, Napoli, Italy, 21–23 April 2015.
29. Voas, J.M.; Miller, K.W. Using fault injection to assess software engineering standards. In Proceedings of the Software Engineering Standards Symposium, 1995.(ISESS'95)'Experience and Practice', Proceedings., Second IEEE International, Montreal, QC, Canada; 21–25 August 1995; IEEE: Washington, DC, USA, 1995; pp. 139–145.
30. Yin, L.; Ri-huang, Y.; Jian-wei, B.; Chun-hui, Y. Research on a Software Fault Injection Model Based on Program Mutation. In Proceedings of the 2015 2nd International Conference on Information Science and Control Engineering (ICISCE), Shanghai, China, 24–26 April 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 419–423.
31. Elanthiraiyan, N.; Arumugam, C. Parallelized ACO algorithm for regression testing prioritization in hadoop framework. In Proceedings of the 2014 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), Ramanathapuram, India, 8–10 May 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 1568–1571.
32. Wong, W.E.; Horgan, J.R.; London, S.; Agrawal, H. A study of effective regression testing in practice. In Proceedings of the PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, Albuquerque, NM, USA, 2–5 November 1997; IEEE: Piscataway, NJ, USA, 1997; pp. 264–274.
33. Malhotra, R.; Kaur, A.; Singh, Y. A regression test selection and prioritization technique. *J. Inf. Process. Syst.* **2010**, *6*, 235–252. [[CrossRef](#)]
34. Upadhyay, A.K.; Misra, A. Prioritizing test suites using clustering approach in software testing. *Int. J. Soft Comput. Eng. IJSCE* **2012**, *2*, 222–226.
35. SivaPrasad, M. An Efficient Regression Testing By Computing Coverage Data For Software Evolution. *Int. J. Comput. Sci. Inform.* **2012**, *1*, 7. [[CrossRef](#)]
36. Li, Z.; Harman, M.; Hierons, R.M. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.* **2007**, *33*, 225–237. [[CrossRef](#)]
37. Zhongsheng, Q. Test case generation and optimization for user session-based web application testing. *J. Comput.* **2010**, *5*, 1655–1662.
38. Nachiyappan, S.; Vimaladevi, A.; SelvaLakshmi, C. An evolutionary algorithm for regression test suite reduction. In Proceedings of the 2010 International Conference on Communication and Computational Intelligence (INCOCCI), Erode, India, 27–29 December 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 503–508.
39. Kaur, A.; Goyal, S. A genetic algorithm for fault based regression test case prioritization. *Int. J. Comput. Appl.* **2011**, *32*, 975–8887.
40. De Souza, L.S.; Prudencio, R.B.; Barros, F.d.A. A hybrid binary multi-objective particle swarm optimization with local search for test case selection. In Proceedings of the 2014 Brazilian Conference on Intelligent Systems, Sao Paulo, Brazil, 18–22 October 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 414–419.
41. Xu, Z.; Gao, K.; Khoshgoftaar, T.M. Application of fuzzy expert system in test case selection for system regression test. In Proceedings of the IRI-2005 IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, USA, 15–17 August 2005; IEEE: Piscataway, NJ, USA, 2005; pp. 120–125.
42. Haider, A.A.; Rafiq, S.; Nadeem, A. Test suite optimization using fuzzy logic. In Proceedings of the 2012 International Conference on Emerging Technologies, Bucharest, Romania, 19–21 September 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 1–6.