January 2024

Reinforcement Learning to Improve Coverage in Software Testing

Shu-Wei Cheng

Reinforcement Learning to Improve Coverage in Software Testing

ABSTRACT

Automated test procedures to test mobile applications or other software may repeatedly revisit the same user interface screens of the app-under-test in an attempt to improve coverage. Complete traversal of all screens of the app-under-test for regression or exploratory testing invariably takes a long time, is inefficient, and is sometimes ineffective. This disclosure describes a crawler that leverages reinforcement learning (RL) to efficiently traverse the screens of a mobile app-under-test or other software, and to form a spanning tree interconnecting the screens of the mobile app-under-test. In an exploratory phase, newly added parts of the mobile app-under-test are discovered and mapped. In an exploitation phase, the mapped parts of the mobile app are explored and regression-tested. The crawler does not require prior knowledge or human pre-encoding of the app-under-test. Rather, it automatically discovers and maps the app-under-test, enabling rapid, scalable, and comprehensive testing of newly released software or applications.

KEYWORDS

- Reinforcement learning (RL)
- Software testing
- App testing
- Regression testing
- Exploratory testing
- Quality assurance
- Q-learning
- Bellman optimality equation
- Spanning tree
- Graph discovery

BACKGROUND

The testing of software, including mobile applications, prior to release includes regression testing to ensure that existing features continue to work as intended and exploratory testing to ensure that new features work as intended. Currently, both exploratory and regression testing is carried out by quality assurance personnel, making testing labor-intensive and expensive.

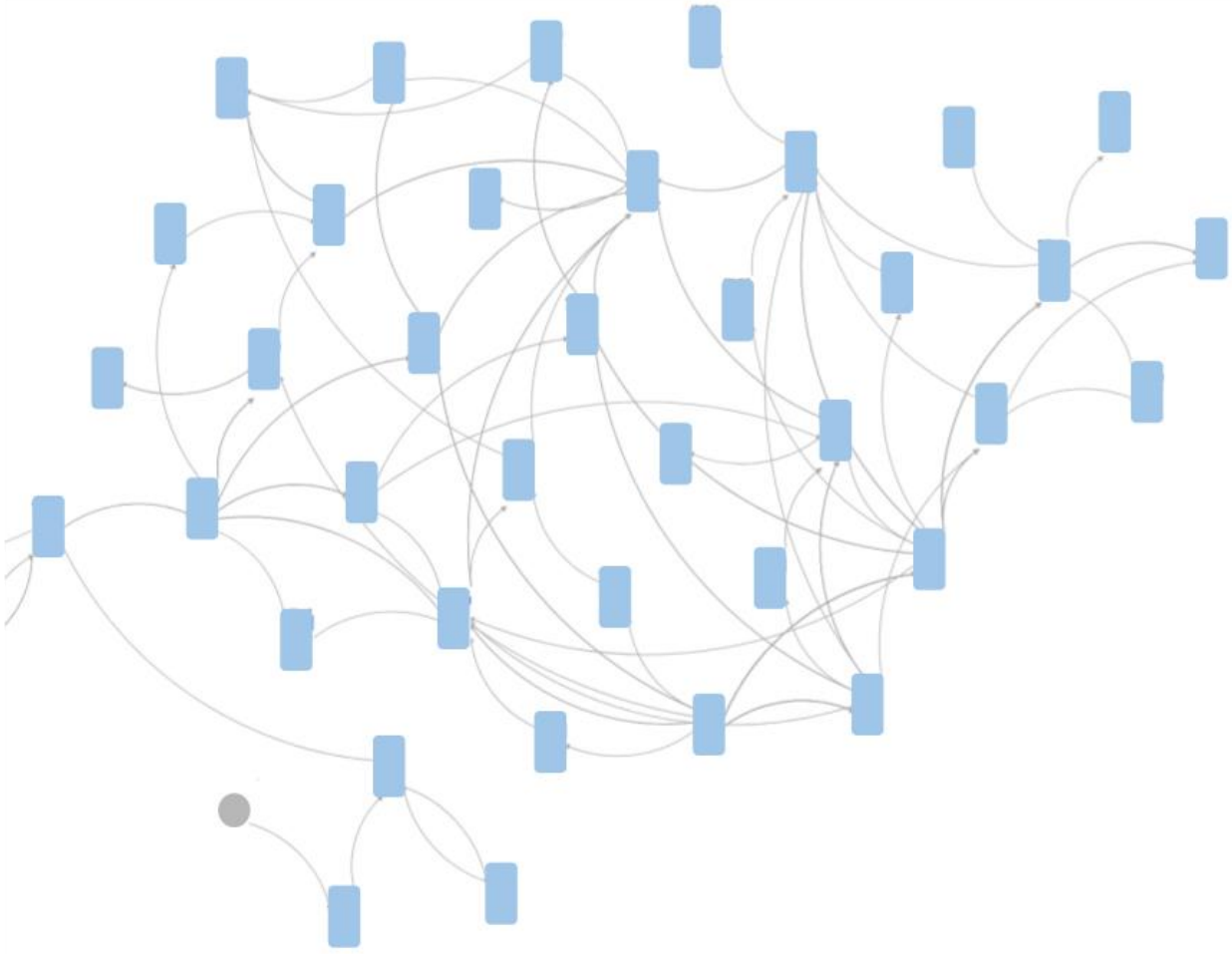

Fig. 1: Example screen-traversal graph constructed by a conventional search procedure

While automated test procedures based on depth-first, breadth-first, or random searching are available, these tend to repeatedly revisit the same screens (user interface screens) of the app-

under-test. For example, Fig. 1 illustrates an example of a screen-traversal graph constructed by a conventional search procedure. In Fig. 1, the nodes (blue) are screens of the app-under-test, and the edges are actions that trigger transition from one screen to another. The inefficiency of the search is evident from the presence of numerous loops (redundant edges) between several pairs of nodes. In an effort to exhaust all app states (nodes, or screens), the search results in visits to the same screens over and over again. Complete traversal of all screens of an app therefore takes a long time and is inefficient and/or ineffective.

Some test procedures use pre-encoded human knowledge of the app-under-test, but pre-encoded test procedures lack generality because such procedures cannot discover or map aspects of an app or release that are not pre-encoded.

DESCRIPTION

This disclosure describes a crawler that efficiently traverses the user interface screens of a mobile app or any other software. The described crawler leverages reinforcement learning (RL) to efficiently form a spanning tree of the screens of the mobile app-under-test and to rapidly traverse all screens of the mobile app.

In an exploratory phase of reinforcement learning, newly added parts of the mobile app-under-test are discovered and mapped. In an exploitation phase of reinforcement learning, the mapped parts of the mobile app are explored and regression-tested. The crawler does not require any prior knowledge or human pre-encoding of the app-under-test. Rather, it automatically discovers and maps the app-under-test. The exploratory and the exploitation phases of reinforcement learning can be alternated.

A user interface screen of a mobile app (or other software) typically includes a number of associated actions, e.g., press button, scroll, touch region, etc., that lead to other screens within

the app. Reinforcement learning is used to assign a value to an action on a visited screen of the app to reward finding as many distinct screens in one run as possible. More formally, a state-action value for each screen-action pair is constructed with a reward function aimed at maximizing the finding of distinct screens.

Under a reward framework designed to maximize the finding of new (distinct) screens, the crawler explores new screens as much as possible or revisits known screens with efficiency. Furthermore, by changing the reward function, the behavior of the crawler can be changed. For example, the crawler can be rewarded when it finds an incorrect user interface (UI) element, driving crawler behavior towards finding incorrect UI elements.

(a)

Screen1 ⟶ Action1 ⟶ Screen2

Screen2 ⟶ Action2 ⟶ Screen3

Screen1 ⟶ Action4 ⟶ Screen4

Screen3 ⟶ Action3 ⟶ Screen1

(b)

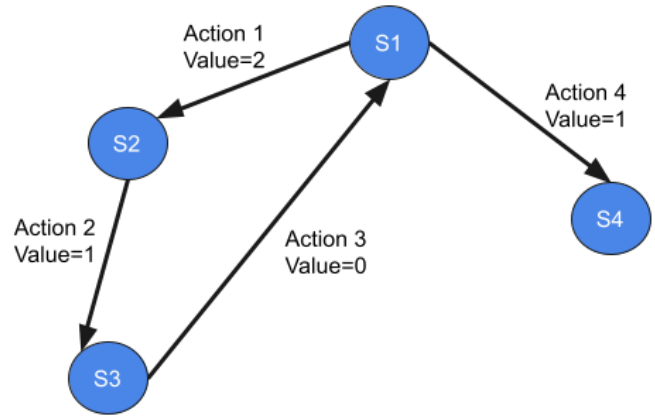

Fig. 2: Discovery of a tree interconnecting the screens of an application or software (a) Sequences of (screen, action) pairs that lead to other screens (b) A graph constructed out of the (screen, action) pairs, with values assigned to the graph-edges

Fig. 2 illustrates the discovery of a tree interconnecting the screens of an application. In an exploratory phase, various (screen, action) pairs are executed, leading to the discovery of new screens. For example, the screen action pair (Screen1, Action1) leads to Screen2 via the pathway Screen1→Action1→Screen2, as illustrated in Fig. 2(a). The various executed (screen, action) pairs are used to construct a corresponding graph, illustrated in Fig.2 (b). The nodes of the graph are the screen identities (Screen1 represented by S1, etc.) and edges are the actions that transition one screen to another.

Reward values are assigned to edges of the graph based on the reward function. For example, if the reward function is configured for the discovery of as many distinct screens as possible in the shortest amount of time, then the reward value assigned to an edge depends on whether the edge leads to previously unseen screens or not, and if the screen that the edge leads to in turn has actions that lead to more previously unseen screens or not. In Fig. 2(b), for example, the Screen1→Action1→Screen2 edge is assigned a relatively high reward value of 2, because the ending screen (Screen2) includes multiple actions that can lead to more previously unseen screens. Similarly, the Screen1→Action4→Screen4 edge is assigned a relatively high reward value of 1 because the ending screen (Screen4) is previously unseen.

On the other hand, the Screen3→Action3→Screen1 edge is assigned a low reward value of 0, because the ending screen (Screen1) is previously seen, and, furthermore, Screen1 has only two associated actions, Action1 and Action4, both of which have been previously explored, e.g., Screen1 does not lead to more previously unseen screens. Thus, in the interest of efficiency, the Screen3→Action3→Screen1 edge is traversed minimally, if at all, and the Screen3→Action3→Screen1 edge may not form part of the eventual spanning tree that interconnects the screens of the application or software.

Formally, the spanning tree interconnecting the screens of the application or software can be discovered, for example, using a procedure known as Q-learning, also known as the Bellman optimality equation. Per the Q-learning procedure, there are two types of rewards, e.g., short-term and long-term rewards. A short-term reward is awarded when a previously unseen screen is found. A long-term reward is awarded when a screen that has been landed upon, whether previously seen or unseen, has the potential to lead to more previously unseen screens.

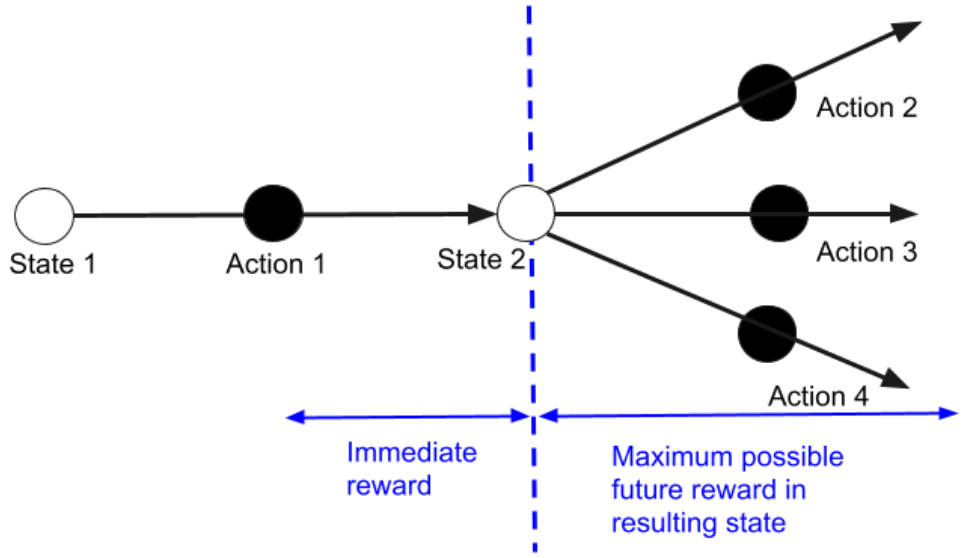

Fig. 3: Short-term and long-term rewards

The following Bellman optimality equation balances the short- and long-term rewards.

$$Q(S, A) \leftarrow Q(S, A) + \boldsymbol{\alpha} \times \{ \text{reward} + \boldsymbol{\beta} \times \max(S', A') - Q(S, A) \},$$

where:

- Q(S, A) is the action-value function, e.g., the value of taking an action A in a state S;
- $\boldsymbol{\alpha}$ is a tunable learning step-size;
- reward is the immediate (short-term) reward for taking action A in state S;
- $\boldsymbol{\beta}$ is a tunable discount factor that accounts for future uncertainty, e.g., the possibility that a screen leads in turn to more unseen screens;

- S′ is the screen that results when action A is taken in state S; and
- max(S′, A′) is the maximum return in state S′ over the actions A′ that can be taken from the screen S′.

In the above equation, the quantity (reward + $\boldsymbol{\beta}$ × max(S′, A′)) is the latest estimate of Q(S, A), and the symbol ←represents an update operator, e.g., the left-hand side Q(S, A) is updated by the right-hand side at each iteration.

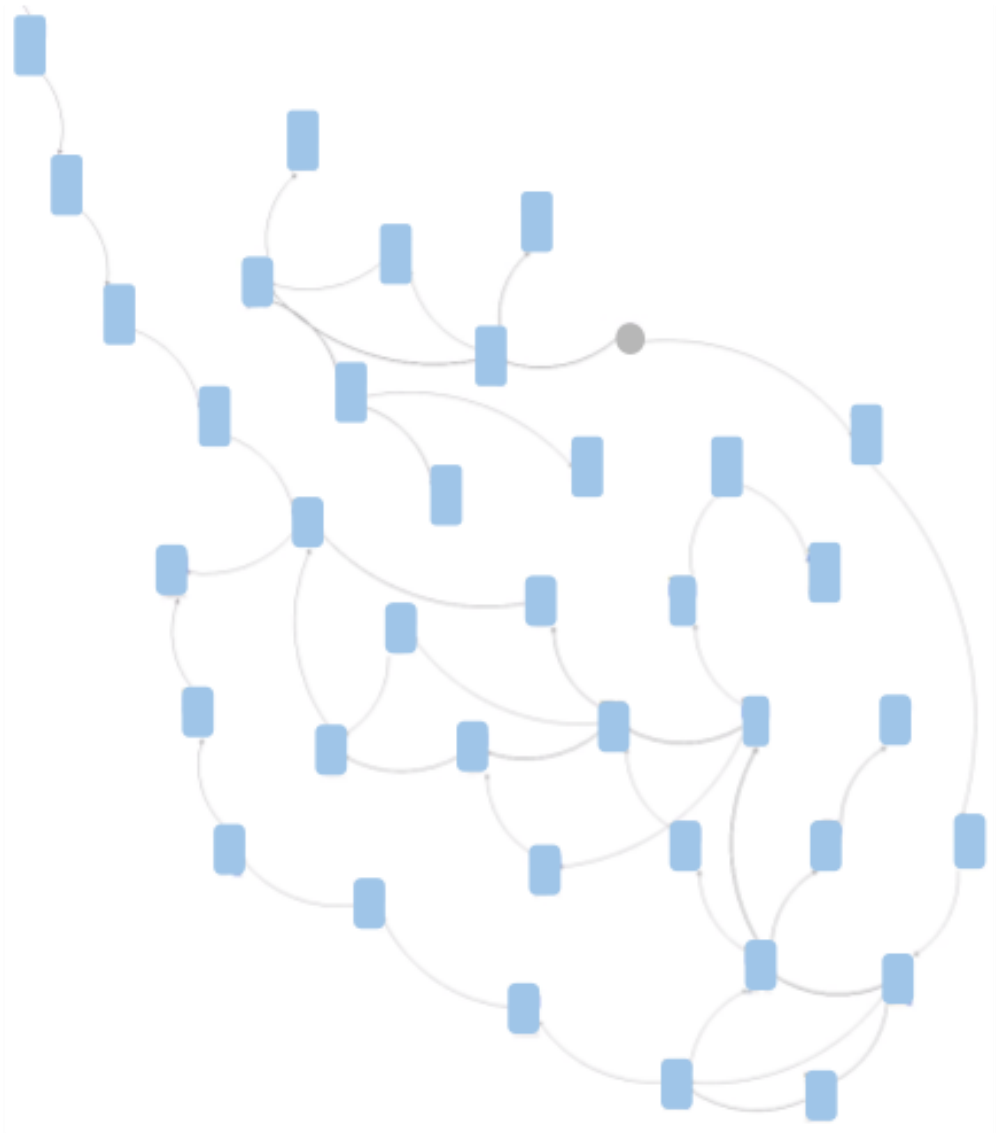

Fig. 4: Example of a screen-traversal graph constructed using reinforcement learning

Fig. 4 illustrates an example of a screen-traversal graph constructed using the described reinforcement-learning-based techniques. Compared to Fig. 1, which is the screen-traversal graph constructed by conventional techniques, Fig. 4 has substantially fewer (nearly none) redundant loops. A screen is visited generally only once, and in most cases, not more than once. The discovery of an almost spanning tree interconnecting the screens of the mobile app enables complete traversal of the screens of the mobile app in a rapid and effective manner.

The state-action and reward framework enables the described crawler to be switched between exploratory and regression testing by a simple change in configuration. For example, exploratory testing can be executed by rewarding the crawler to take actions or pathways not previously taken. Doing so, the crawler discovers new screens and rapidly constructs a spanning-tree graph interconnecting the screens of the app. Conversely, regression testing can be executed by rewarding the crawler to traverse the screens of the app as quickly as possible. With multiple instances of the crawler running simultaneously, exploratory and regression testing can be carried out simultaneously such that some instances explore and map the app-under-test while other instances regression-test it. Such combined exploratory and regression testing provides coverage of new and existing screens of the app-under-test and can improve the velocity of trapping bugs. The described crawler can find problems in deep features, e.g., where an application crash occurs after a large number of steps.

By efficiently, effectively, and automatically mapping and traversing through all screens of a software application, the described techniques enable rapid, scalable, and comprehensive testing of newly released software or applications without prior knowledge of the way in which screens transition occur in the app and without the need for human pre-encoding or intervention.

The crawler can identify release-blocking bugs sooner. The crawler can be provided to developers for use during code review.

CONCLUSION

This disclosure describes a crawler that leverages reinforcement learning (RL) to efficiently traverse the screens of a mobile app-under-test or other software, and to form a spanning tree interconnecting the screens of the mobile app-under-test. In an exploratory phase, newly added parts of the mobile app-under-test are discovered and mapped. In an exploitation phase, the mapped parts of the mobile app are explored and regression-tested. The crawler does not require prior knowledge or human pre-encoding of the app-under-test. Rather, it automatically discovers and maps the app-under-test, enabling rapid, scalable, and comprehensive testing of newly released software or applications.