BADGE: Prioritizing UI Events with Hierarchical Multi-Armed Bandits for Automated UI Testing

Dezhi Ran
School of Computer Science
Peking University, China
dezhiran@pku.edu.cn

Hao Wang
School of EECS
Peking University, China
tony.wanghao@stu.pku.edu.cn

Wenyu Wang
University of Illinois
Urbana-Champaign, USA
wenyu2@illinois.edu

Tao Xie[§]
School of Computer Science
Peking University, China
taoxie@pku.edu.cn

***Abstract*—To assure high quality of mobile applications (*apps* for short), automated UI testing triggers events (associated with UI elements on app UIs) without human intervention, aiming to maximize code coverage and find unique crashes. To achieve high test effectiveness, automated UI testing prioritizes a UI event based on its exploration value (e.g., the increased code coverage of future exploration rooted from the UI event). Various strategies have been proposed to estimate the exploration value of a UI event without considering its exploration diversity (reflecting the variance of covered code entities achieved by explorations rooted from this UI event across its different triggerings), resulting in low test effectiveness, especially on complex mobile apps. To address the preceding problem, in this paper, we propose a new approach named BADGE to prioritize UI events considering both their exploration values and exploration diversity for effective automated UI testing. In particular, we design a hierarchical multi-armed bandit model to effectively estimate the exploration value and exploration diversity of a UI event based on its historical explorations along with historical explorations rooted from UI events in the same UI group. We evaluate BADGE on 21 highly popular industrial apps widely used by previous related work. Experimental results show that BADGE outperforms state-of-the-art/practice tools with 18%-146% relative code coverage improvement and finding 1.19-5.20x unique crashes, demonstrating the effectiveness of BADGE. Further experimental studies confirm the benefits brought by BADGE's individual algorithms.**

***Index Terms*—GUI testing, mobile testing, mobile app, Android, multi-armed bandits, reinforcement learning**

I. INTRODUCTION

Mobile applications (*apps* in short) have been an indispensable part of people's daily work and life [1]. The number of global smartphone users is estimated at 6.6 billion [2] in 2022 and users spend nearly 5 hours [3] on apps daily. Consequently, ensuring good user experiences has become unprecedentedly important for mobile app vendors. While manual UI testing is usually conducted to test complex app functionalities, it can be notoriously expensive and tedious to satisfy frequent and rapid shipments of mobile apps [4].

To alleviate the cost of manual UI testing, *automated UI testing* [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19] requires no human intervention and automatically selects *UI events*[1] to trigger based on their *exploration values*. In this paper, the exploration value of UI event e is denoted as the increased code coverage (e.g., activity coverage and method coverage) achieved by future exploration rooted from e (beyond the code coverage achieved during the exploration process before the triggering of e) in the directed acyclic activity transition graph [20] of the AUT. As shown in Figure 1, a click on the search tab element leads to search-related functionalities. The exploration value of this click event is the estimated increased code coverage from triggering this event and its subsequent UI events. Since the goal of automated UI testing is usually to maximize the coverage of the App Under Test (*AUT* for short), it is more desirable to trigger UI events estimated to result in more code coverage gain in future exploration, ideally having higher priorities over other UI events to be performed on the AUT.

To estimate exploration values of UI events, various approaches have been proposed, falling into four main categories. *Random* approaches [21], [8], [9] use naïve/adaptive random heuristics to estimate the exploration values of UI events. *Systematic* approaches [11], [12], [13] prioritize UI events that target hard-to-cover code. *Model-based* approaches [5], [6], [16], [10], [17] use a UI transition model to determine the current test progress and prioritize UI events that target not-yet-explored functionalities. *Machine-learning-based* approaches [22], [23], [19], [24] prioritize UI events based on their exploration values estimated by deep learning or reinforcement learning algorithms.

Despite various proposed strategies, estimating exploration values of UI events in industrial apps still faces two major challenges. First, treating the UI events in an extendable list as equivalent UI events will incur the loss of code coverage. As shown in Figure 1, exploring the UI events (denoted as E) associated with the UI elements U in the extendable list can incur different increased code coverages since U correspond to different cards. If a tool treats the UI events in the extendable list as equivalent UI events, the tool may click on only one UI element in U as shown in Figure 1(a), and estimate the

[§]Corresponding author.

[1]UI events are interactions associated with UI elements. Although our work can support situations where multiple types of action can be performed on each UI element, for simplicity of presentation, in this paper, we assume that only one type of action can be performed on each UI element, and use the two terms (UI events and UI elements) interchangeably.

exploration values of clicking other UI elements in U as zero and ignore the other UI elements in U, failing to discover possibly different UI screens (i.e., different coverages) after clicking different UI elements in U. Second, treating the UI events in an extendable list as nonequivalent UI events can waste the test budget on these UI events. Suppose that a tool finds that triggering the UI events associated with the UI elements in the list results in different code coverages. In that case, the tool will allocate the test budget to exhaustively trigger all UI events in the list, resulting in low effectiveness.

The preceding challenge faced by existing strategies results from the *exploration diversity* of **UI event** e, which indicates the differences among the sets of covered code entities (such as activities and methods), in short as covered entity sets, achieved by explorations rooted from e triggered in **multiple trials**, respectively. To estimate the exploration diversity of e, we leverage the covered entity sets achieved by historical explorations rooted from e triggered in multiple trials. For example, assume that UI event e is triggered two times, and the covered entity sets are X and Y, respectively. Then the exploration diversity of UI event e can be estimated by the Jaccard Index [25] $J(X, Y) = \frac{X \cap Y}{X \cup Y}$ between sets X and Y. A lower Jaccard Index (i.e., the two sets are more different) indicates a higher difference between the different triggerings of UI event e.

In addition, we also define the exploration diversity of **UI events e_1, ..., e_n associated with the UI elements in the same *UI element group*** (in short as UI group)[2] such as an extendable list. The exploration diversity here indicates the differences among the covered entity sets achieved by explorations rooted from e_1, ..., e_n, respectively. To simplify the presentation, we also name this exploration diversity as the exploration diversity of this UI group. Similarly, to estimate exploration diversity of e_1, ..., e_n, we leverage the covered entity sets achieved by historical explorations rooted from a subset of e_1, ..., e_n, and then the exploration diversity can be estimated by the Jaccard Index [25]. Note that the second setting is based on the hierarchical organization [26] of UI elements: instead of being independent or unrelated to each other, UI elements on a UI screen are organized according to specific principles [27]. As shown in Figure 1, UI elements are organized into *UI groups* based on their alignment and proximity. The UI events associated with the elements in the same group (e.g., the extendable list in Figure 1) are strongly correlated to each other. Our insight is that higher exploration diversity of already explored UI events in the same group reflects a higher priority to explore some remaining unexplored UI events in the same group.

While the exploration diversity of UI events can substantially affect the effectiveness of automated UI testing, it is challenging to effectively quantify and exploit the exploration diversity of UI events for two reasons. First, the exploration diversity of UI events is hard to estimate. Due to the randomness of UI-event prioritization strategies and the AUT's own non-deterministic behaviors [28], [29], the covered entity set achieved by a future exploration rooted from a UI event is non-deterministic. Second, it is challenging to balance exploration diversity and exploration value when prioritizing UI events. While exploration value indicates a possibility to bring coverage gain to trigger an already explored UI event, triggering an unexplored UI event can also possibly bring coverage gain (from pure exploration value). It is not clear which one is more beneficial for automated UI testing to select between the two cases.

To improve the effectiveness of automated UI testing against exploration diversity of UI events, in this paper, we propose BADGE, a hierarchical *multi-armed bandit* [30] approach to prioritizing UI events by jointly estimating and balancing their exploration values and exploration diversity. Multi-Armed Bandits (MABs) is a reinforcement learning paradigm [31] for estimating the values of performing individual actions and balancing between two factors (exploration values and exploration diversity in our setting). By balancing exploration values and exploration diversity with MAB, BADGE avoids the two extremes of exploring an extendable list faced by existing work [16], [17], [19]. In particular, BADGE learns the exploration values and exploration diversity of UI events based on the increased coverage gained from historical trials, and exploits MAB to prioritize the UI events by jointly considering their exploration values and exploration diversity.

To evaluate the effectiveness of BADGE, we instantiate BADGE with a simple activity-level UI transition model and conduct experiments on 21 highly popular industrial apps (widely used in previous related work [32], [18], [33], [34]) compared with three state-of-the-art tools namely Stoat [16], Ape [17], Q-testing [19], and two state-of-the-practice tools namely Monkey [21] and Fastbot [20]. Evaluation results demonstrate that BADGE substantially outperforms the state-of-the-art/practice tools with 18%-146% relative method coverage improvement and finding 1.19-5.20x unique crashes. To study the impact of exploration values and exploration diversity of UI events, we conduct experimental studies and the results demonstrate the usefulness of jointly balancing them in automated UI testing.

In summary, this paper makes the following main contributions:

- A novel MAB formulation of automated UI testing to prioritize UI events based on both exploration values and exploration diversity.
- An extensible approach named BADGE instantiated with novel designs. The implementation is publicly available.[3]
- Evaluations on widely used industrial apps demonstrating the effectiveness of BADGE with 18%-146% code coverage improvement and finding1.19-5.20x unique crashes.

[2]Note that for each e_i of UI events e_1, ..., e_n associated with the UI elements in the same UI group, we also have the exploration diversity of e_i as defined earlier.

[3]https://github.com/ranpku/Badge

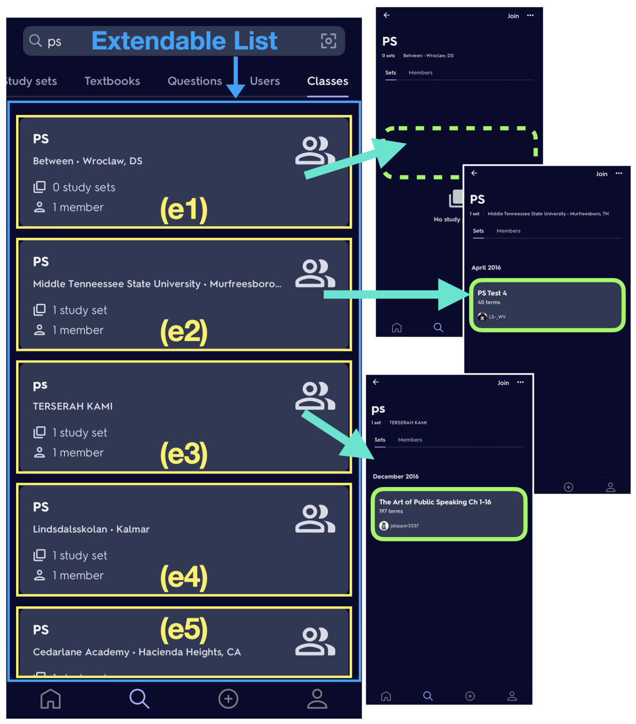

Fig. 1. A motivating example: the search tab and search result list in the "Quizlet" app.

II. Motivating Example

In this section, we introduce the concept of exploration values and exploration diversity of UI events with an illustrative example.

Figure 1 presents the search tab in the "Quizlet" app, whose main functionalities are to help users learn through specialized assistance and flashcards. A list of search results can be accessed from the search screen shown in the main figure in Figure 1. The three sub-figures on the right side are the flashcard sets after the UI elements marked with (e1), (e2), and (e3) in the search result list are clicked, with three pointers each pointing from an element to its corresponding result, respectively. While the flashcard set for (e1) contains no available flashcards, those for (e2) and (e3) contain flashcards with different items. In this example, the exploration value of clicking a different UI element in the search list can vary drastically but the exploration diversity of the search list is moderate since the covered entity sets of clicking different UI elements can be clustered into two groups (w/wo flashcards after clicking a UI element), and thus the UI elements can be classified into two types accordingly.

Automated UI testing needs to allocate a proper test budget to explore the UI elements in the search list for achieving good test effectiveness (e.g., code coverage). Stoat [16] treats all the UI elements in the search list equivalent, ignoring the differences of exploration values of clicking different UI elements and the exploration diversity of clicking UI elements in the search list, as pointed out by the work of Ape [17]. Ape [17] and Q-testing [19] can identify the differences between the exploration values of clicking UI elements (e1) and (e2). However, Ape and Q-testing treat all of the UI elements in the search list as different ones, ignoring the exploration diversity of clicking the UI elements in the search list and wasting the test budget. A desirable solution should balance the exploration values and exploration diversity of UI events to achieve good test effectiveness. In this example, a moderate test budget is expected to be allocated to the search list to explore the two types of UI elements and timely stop exploring the search list after the exploration diversity of the search list stops increasing.

On one hand, the exploration values and exploration diversity of UI events are learned from historical trials, and more trials result in a more accurate estimation of exploration values and exploration diversity. On the other hand, more trials can lead to possible waste of test budgets. Badge addresses the challenge of estimating exploration values and exploration diversity with the hierarchical MAB model [30]. In particular, the MAB model determines the optimal trials to obtain an accurate estimation of exploration values and exploration diversity, and the hierarchical model simultaneously estimates the exploration diversity of UI events and UI groups.

III. Preliminary

In this section, we present basic concepts necessary for introducing Badge.

A. Android UI System

Activity. *Activities* are one of the major types of components [35] for building Android apps. As the entry points to app functionalities, activities present UIs that users can interact with. Each activity internally maintains the data structures that describe the properties and hierarchical information of its current UI.

UI Hierarchy. Android UIs can be structurally represented by *UI hierarchies*. Each UI hierarchy consists of *UI properties* (e.g., location, size) for individual *UI elements* (e.g., buttons, text boxes) and hierarchical relations among UI elements. Typically, UI elements are implemented with `View` [36] subclass instances, and hierarchical relations are represented by children `View`s of `ViewGroup` [37] subclass instances.

UI event. A UI event [38] represents a user's interaction with a UI element. An event e can be represented as a triple (w, a, v), where w denotes a UI element, a denotes an associated action to perform on the UI element, and v represents optional parameters of the action. To handle an event, a UI element typically needs a corresponding *event listener*, a callback object that responds to UI events targeted by itself. For example, Android apps typically implement `View.OnClickListener` and attach the callback objects to `Button` instances to handle button clicks. For simplicity of presentation, by following previous work [17], we assume that each UI element can handle only one UI event, and we use UI element and UI event interchangeably to refer to the UI event associated with the single UI element.

B. Multi-Armed Bandit

Multi-Armed Bandit (MAB) [30] is a classic family of reinforcement learning algorithms [31]. In a basic MAB

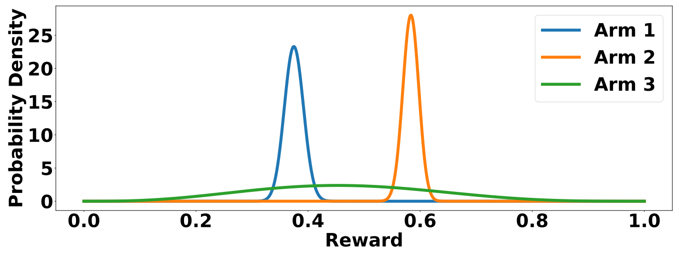

Fig. 2. An illustration of Thompson sampling. Curves represent the probability density function of the rewards of three arms. The more concentrated the curve is, the more confident we are with the arm's actual reward. The higher the curve is, the higher reward that arm is estimated to have.

formulation, the MAB algorithm chooses an *arm* (i.e., an action) from K possible arms and collects a reward drawn from a distribution for the chosen arm. When selecting arms, the MAB algorithm faces a trade-off between *exploration* and *exploitation*. On one hand, choosing arms that are not explored (denoted as *exploration*) may bring higher rewards than those of already explored arms. On the other hand, choosing already explored arms that are known to bring high rewards (denoted as *exploitation*) can be better than risking exploring unexplored arms.

In automated UI testing, the arms are UI events or UI groups (choosing an arm of UI group indicates choosing UI events within the UI group, and triggering a UI group indicates triggering an event within the UI group). The reward of a UI event/UI group is the exploration values of the UI event/UI group. Given the existence of exploration diversity of the UI event/UI group, the reward is a stochastic distribution. For a UI event/UI group that has been explored more times, we are more confident with its exploration value and its exploration diversity observed from historical trials.

It is proved that the random exploration and exploration-first strategies in MAB yield sub-optimal cumulative rewards [39]. Consequently, MAB algorithms strive to learn which arms are the best (approximately), while not spending too much time exploring.

Thompson Sampling [40] is an effective heuristic for tackling MAB problems. Given the average reward α_i and the exploration count β_i of the arm i (i.e., the number of times of choosing the arm), Thompson sampling estimates the actual reward θ of choosing the arm i as $\theta_i \sim Beta(\alpha_i, \beta_i)$, where $Beta(\alpha, \beta)$ is the Beta distribution [41]. A larger α indicates a higher reward based on the estimation from previous observations, and the larger β (i.e., higher exploration count) indicates a more confident estimation of the current reward.

Figure 2 presents an illustrative example of how Thompson sampling balances exploration and exploitation. For arms with higher exploration counts, we are more confident with their rewards based on historical observations and select arms based on their rewards. For example, we prefer Arm 2 (with a higher reward) than Arm 1. For arms with lower exploration counts, we are less confident with the current estimation of their rewards and their actual rewards can be higher than those estimated based on historical observations. Consequently, we would still select Arm 3 instead of Arm 2 with some probability although the current reward of Arm 3 is lower than that of Arm 2.

Algorithm 1 Main Algorithm of BADGE.

```
Require: $\mathtt{T}, \mathtt{r}_{\mathtt{Thresh}}$
 1: function Badge
 2:     $\mathtt{regret} \leftarrow 0$
 3:     $\mathtt{groups}, \mathtt{arms} \leftarrow \mathtt{mabConstruction}()$
 4:     for $\mathtt{t} \in \{1, 2 \ldots \mathtt{T}\}$ do
 5:         /*Generate score for each arm*/
 6:         $\mathtt{arms} \leftarrow \mathtt{MCTSTAR}(\mathtt{groups}, \mathtt{arms})$

 7:         /*Sample event $e$ with relative probability $Score(e)$*/
 8:         $\mathtt{bestEvent} \leftarrow \mathtt{sampleFrom}(\mathtt{arms}, \mathtt{key} = \mathtt{score})$
 9:         $\mathtt{r}_{\mathtt{simulated}} \leftarrow \mathtt{bestEvent.score}$
10:         $\mathtt{UIAutomator.act}(\mathtt{bestEvent})$
11:         $\mathtt{groups}, \mathtt{arms} \leftarrow \mathtt{mabConstruction}()$

12:         /*Restart if cumulative regret exceeds threshold*/
13:         $\mathtt{r}_{\mathtt{actual}} \leftarrow \mathtt{getActualReward}(\mathtt{arms})$
14:         $\mathtt{regret} \leftarrow \mathtt{regret} + \mathtt{max}(\mathtt{r}_{\mathtt{simulated}} - \mathtt{r}_{\mathtt{actual}}, 0)$
15:         if $\mathtt{regret} \geq \mathtt{r}_{\mathtt{Thresh}}$ then
16:             $\mathtt{UIAutomator.restartApp}()$
17:             $\mathtt{regret} \leftarrow 0$
```

IV. Approach

In this section, we first present the BADGE approach with MAB-based formulation of automated UI testing. Then we introduce the design and techniques to instantiate BADGE into a practical automated UI testing tool.

Algorithm 1 presents the main algorithm of BADGE. BADGE iteratively interacts with the App Under Test (AUT) until the test budget runs out. In each iteration, BADGE first acquires the interactable UI events on the screen from the MAB model and constructs a hierarchical MAB model of UI events (detailed in Section IV-A). Then BADGE uses the MCTSTAR algorithm (detailed in Section IV-B) to estimate rewards (i.e., exploration values and exploration diversity) for UI events and UI groups. Finally, BADGE prioritizes UI events based on their rewards and selects a UI event to trigger on the AUT based on their prioritization. Since the rewards are estimated and sometimes do not conform with the actual cases, BADGE monitors the testing progress and restarts the AUT when the actual rewards are less than the estimated rewards (denoted as *regrets*) for a given threshold (Lines 12-17) to avoid being stuck or trapped in exploration tarpits [34].

A. Hierarchical MAB Construction

Algorithm 2 presents the algorithms of parsing a UI hierarchy into the hierarchical MAB model. The UI elements on the screen of mobile apps are organized in a tree structure called UI hierarchy (detailed in Section III) that can be obtained with UIAutomator [42] for Android. Each UI element in the UI hierarchy can be uniquely identified by its *element identifier* (i.e., attribute path defined in previous work [17]). Each UI element possesses attributes indicating its type and

Algorithm 2 MAB construction

```
1: /*Trim non-interactive elements*/
2: function dfsTrimHierarchy(node)
3:     for childNode $\in$ node.childs do
4:         dfsTrimHierarchy(childNode)
5:     if !isInteractable(node) then
6:         node.removeSelf()

7: /*Compress non-interactive chains*/
8: function compressPath(uiHierarchy)
9:     for node $\in$ uiHierarchy do
10:        if isInteractable(node) then
11:            continue
12:        else if node.childs.size $\leq 1$ then
13:            uiHierarchy.remove(node)

14: function getLocalID(node)
15:     return node.class + node.resource_id

16: function uiHierarchyModeler(uiHierarchy)
17:     dfsTrimHierarchy(uiHierarchy.root)
18:     compressPath(uiHierarchy)
19:     elements $\leftarrow \emptyset$
20:     for node $\in$ uiHierarchy do
21:         if isInteractable(node) then
22:             node.localID $\leftarrow$ getLocalID(node)

23:             /*Join the IDs of ancestors to obtain group ID*/
24:             pathToFather $\leftarrow$ uiHierarchy.path(node.father)
25:             ancestorIDs $\leftarrow$ pathToNode.map(getLocalID)
26:             node.groupID $\leftarrow$ join(ancestorIDs)
27:             elements.add(node)
28:     return elements

29: function mabConstruction( )
30:     S $\leftarrow$ UIAutomator.dump
31:     elements $\leftarrow$ UI_Hierarchy_Modeler(S)

32:     /*Put arms with the same IDs in the same group*/
33:     arms $\leftarrow$ elements.groupby(key = 'localID')
34:     groups $\leftarrow$ elements.groupby(key = 'groupID')
35:     rewards $\leftarrow$ MCTSTAR(groups, arms)
36:     return rewards, arms
```

content such as class, resource-id, index, text, content-description, and position. We use class and resource-id of a UI element as its *local identifier* since the two attributes of a UI element do not change during testing.

We denote the concatenation of the local identifiers of all the ancestor UI elements (in the depth-first traversal of the UI hierarchy tree) of a UI element as its *group identifier*. Then the element identifier of a UI element is the concatenation of its group identifier and its own local identifier. For a UI element supporting multiple interactions, we concatenate each interaction (indicated by the clickable, long-clickable, scrollable attributes of the UI element) to its element identifier to distinguish between the different types of interactions with the UI element. There can be some discrepancies between UI groups from the users' perspective and the groups according to elements' group identifiers due to some invisible layout UI elements. Thus, we trim the original UI hierarchy dumped from UIAutomator by removing all UI elements that contain only one interactable element and are invisible/non-interactable. When a UI element and its children elements are both interactable, we retain only its children elements in the trimmed UI hierarchy. The group identifiers of interactable elements are calculated in the trimmed UI hierarchy.

After obtaining a list of UI element identifiers and group identifiers, for each activity, BADGE builds the hierarchical MAB model whose arms represent UI groups and UI elements in the activity. For each arm, the MAB model records the count that the arm has been chosen so far, and the reward of selecting this arm. For an arm representing a UI group, its exploration count is the sum of the exploration counts of the UI elements in the UI group.

B. Monte Carlo Tree Search for Reward Generation

In order to estimate the exploration values and exploration diversity of UI elements/groups, we design an algorithm of Monte Carlo Tree Search (*MCTS*) [43], [44] based Test rewArd geneRation (*MCTSTAR* for short), being a type of reinforcement learning (RL) algorithm [31]), to estimate the code coverage achieved by future exploration rooted from a UI event. Our algorithm plans over the exploration history and estimates the exploration values and exploration diversity of UI elements/groups based on the coverages already achieved during the testing process.

For our approach, we design an MCTS algorithm instead of the standard Q-learning algorithm (a type of RL algorithm that has been popularly adopted in automated UI testing [22], [19], [24]) due to its undesirable design. In the standard Q-learning algorithm, the Q values eventually converge to a fixed value and Q-learning will greedily select the action that maximizes this fixed value of the current state (i.e., eventually learns a **deterministic** optimal policy)[4]. This design in the standard Q-learning algorithm is undesirable for automated UI testing since the exploration values and exploration diversity of UI elements/groups change during the testing process with the AUT's unexplored parts changing during the testing process.

Algorithm 3 presents the workflow of generating rewards for arms in the MAB model and prioritizing UI elements with MCTSTAR, consisting of three components: (1) *exploration diversity estimation*, (2) *exploration value estimation*, and (3) *Thompson sampling*.

Exploration diversity estimation. MCTSTAR estimates the exploration diversity of a *UI group* by measuring the differences among the covered entity sets achieved by explorations rooted from the explored UI elements in the UI group. Let us denote S_e as the set of UI elements encountered[5]

[4]We talk about only Q-learning algorithms under the Markov Decision Process (MDP) formulation used by existing automated UI testing tools, and do not discuss recent advances of memory-based Q-learning under Non-Markovian Decision Process formulations.

[5]We denote that UI element e_x is encountered after e is triggered if e_x is contained by a UI hierarchy explored by historical explorations rooted from e.

Algorithm 3 MCTS-based Test Reward Generation

```
Require: uiTransitions, $\eta$
 1: function getDiversityScore(group)
 2:     clusterNum $\leftarrow$ cluster(group,
 3:         key = mutualInformation).size
 4:     return $\frac{\texttt{clusterNum}}{\texttt{group.size}}$

 5: function getArmScore(root)
 6:     score $\leftarrow$ 0
 7:     for element $\in$ uiTransitions do
 8:         if uiTransitions.connected(root, element) then
 9:             dist $\leftarrow$ uitransitions.dist(root, element)
10:             score $\leftarrow$ score + element.score $\cdot 2^{-\texttt{dist}}$
11:     return score

12: function getActualReward(arms)
13:     reward $\leftarrow$ 0
14:     for arm $\in$ arms do
15:         reward $\leftarrow$ reward + getArmScore(arm)
16:     return reward

17: function MCTSTAR(groups, arms)
18:     for group $\in$ groups do
19:         group.score $\leftarrow$ getDiversityScore(group)
20:     for arm $\in$ arms do
21:         r = getArmScore(arm) $\cdot$ (1 + arm.group.score)
22:         $\alpha \leftarrow$ r $\cdot$ arm.explorationCount
23:         $\beta \leftarrow$ arm.explorationCount $- \alpha$
24:         arm.score $\leftarrow$ thompsonSampling($\alpha, \beta$)
25:     return arms.score
```

after triggering UI element e in the *directed acyclic activity transition graph* [20]. For any two UI elements e_1 and e_2 in the UI group, we calculate a weighted Jaccard index (*WJ* for short) [25] between S_{e_1} and S_{e_2}:

$$WJ(S_{e_1}, S_{e_2}) = \frac{\sum_{e_x \in S_{e_1} \cap S_{e_2}} \max\{w(e_1, e_x), w(e_2, e_x)\}}{\sum_{e_x \in S_{e_1} \cup S_{e_2}} \max\{w(e_1, e_x), w(e_2, e_x)\}} \quad (1)$$

where weight $w(e_i, e_x) = 2^{1-dist(e_i, e_x)}$ (for each encountered UI element e_x) is determined by the distance $dist(e_i, e_x)$ between UI elements e_i and e_x, i.e., the minimum number of events needed to reach e_x by starting with e_i. When $e_x \notin S_{e_i}$, $w(e_i, e_x) = 0$.

Assume that UI group g contains n UI elements (e_1, ..., e_n). MCTSTAR treats any two UI elements $e_i, e_j \in g$ to be in different clusters if the weighted Jaccard Index of S_{e_i} and S_{e_j} exceeds a threshold $\mathcal{T}$. Then MCTSTAR uses the number of different clusters in g to estimate the exploration diversity of g with the following equation:

$$GroupDiv(g) = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} I(WJ(S_{e_i}, S_{e_j}) \geq \mathcal{T})}{n * (n-1)/2} \quad (2)$$

where $I(x)$ is an indicator function ($I(x) = 1$ if x is true; otherwise, $I(x) = 0$), and $\mathcal{T}$ is a parameter controlling the threshold of distinguishing two sets of UI elements as different. For a UI group that has not been explored, we set its exploration diversity to be a high value v_{init} (e.g., 10) to encourage exploring the UI group.

MCTSTAR estimates the exploration diversity of a *UI element* based on whether it has been explored or not. For a UI element that has not been explored, MCTSTAR takes the exploration diversity of its belonging UI group as its exploration diversity. For a UI element that has been explored, MCTSTAR estimates its exploration diversity by calculating the weighted Jaccard Index of the sets of entities that it covers in different historical explorations (the calculation equation can be easily derived by extending Equation 1).

Exploration value estimation. MCTSTAR estimates the exploration value of a *UI element* based on whether it has been explored or not. For a UI element that has not been explored, MCTSTAR uses the average exploration values of the explored UI elements in its belonging UI group as its exploration value. For a UI element that has been explored, MCTSTAR estimates its exploration value from its historical explorations by the following equation:

$$ExploreValue(e) = \sum_{e_x \in S'_e} w(e_x, e) \cdot ExploreValue(e_x) \quad (3)$$

where S'_e is the set of UI elements that (1) are encountered by historical explorations rooted from e and (2) are not explored by the historical explorations.

Thompson sampling. MCTSTAR employs Thompson sampling [40] to balance exploration values and exploration diversity of UI elements. In particular, we give higher priority to UI elements with higher rewards for reflecting higher exploration values and/or higher exploration diversity. To capture the preceding intuition, Thompson sampling balances the exploration values and exploration diversity of UI elements by sampling $Score(e)$, which represents the score of UI element e from a Beta distribution [41]. Specifically, for each UI element e,

$$\begin{gathered} Score(e) \sim Beta(\alpha, \beta) \\ \alpha = ExploreValue(e) \cdot (1 + DivValue(e))) \cdot Cnt(e) \\ \beta = Cnt(e) - \alpha \end{gathered} \quad (4)$$

where $ExploreValue(e)$ and $DivValue(e)$ are the exploration value and exploration diversity of e, respectively, and $Cnt(e)$ is the exploration count of e. Recall that $Cnt(e)$ is used to indicate the confidence level for estimating the current reward (see Section III for details).

BADGE prioritize UI elements according to the sampled score $Score(e)$ and selects the UI element with the highest score to trigger on the AUT.

V. EVALUATION

To evaluate the effectiveness of BADGE, we conduct an evaluation to investigate the following three research questions:

- **RQ1:** How effective is BADGE compared with state-of-the-art/practice tools in terms of code coverage?
- **RQ2:** How effective is BADGE compared with state-of-the-art/practice tools in terms of crash revelation?
- **RQ3:** How much does BADGE benefit from its individual algorithms?

To answer RQ1, we analyze the traces and code coverage information collected from test runs of BADGE and state-of-the-art/practice automated UI testing tools. To answer RQ2, we

TABLE I
OVERVIEW OF INDUSTRIAL APPS USED FOR EVALUATION

ID	App Name	Version	Category	#Inst
A_1	Abs	4.2.0	Health & Fitness	10m+
A_2	AccuWeather	7.4.1-5	Weather	100m+
A_3	AutoScout24	9.8.6	Auto & Vehicles	10m+
A_4	Duolingo	3.75.1	Education	100m+
A_5	Evernote*	7.12	Productivity	100m+
A_6	Filters For Selfie	1.0.0	Beauty	10m+
A_7	GoodRx	5.3.6	Medical	10m+
A_8	Google Chrome	65.0.3325	Communication	10b+
A_9	Google Translate	6.5.0	Books & Reference	1b+
A_{10}	Marvel Comics	3.10.3	Comics	10m+
A_{11}	Merriam-Webster	4.1.2	Books & Reference	10m+
A_{12}	My Baby Piano	2.22.2614	Parenting	5m+
A_{13}	Quizlet*	6.6.2	Education	10m+
A_{14}	Sketch	8.0.A.0.2	Art & Design	50m+
A_{15}	Spotify*	8.4.48.497	Music & Audio	1b+
A_{16}	TripAdvisor*	25.6.1	Food & Drink	100m+
A_{17}	Trivago	4.9.4	Travel & Local	50m+
A_{18}	WEBTOON*	2.4.3	Comics	100m+
A_{19}	Yelp*	9.33.0	Food & Drink	50m+
A_{20}	Youtube	15.35.42	Video Player & Editor	10b+
A_{21}	Zedge	7.34.4	Personalization	100m+

Notes: '#Inst' denotes the approximate number of downloads. Apps with asterisks (*) after their names are the ones that require logging in to access most features.

analyze the crash reports collected from test runs of BADGE and state-of-the-art/practice automated UI testing tools. To answer RQ3, we investigate how much BADGE benefits from Thompson Sampling, the MCTSTAR algorithm, and the hierarchical MAB, with experimental studies to quantify our findings.

A. Evaluation Setup

App selection. To evaluate the effectiveness of BADGE, we use 21 popular industrial apps from a widely used industrial app benchmark collected by previous work [32], as shown in Table I. We ask the authors of the previous work [32] to provide the latest versions of app packages that are still compatible with the official Android x64 emulator [33]. We obtain 23 app packages, from which we remove two apps, *Mirrorzoomexposure* and *Flipboard*. Specifically, Mirrorzoomexposure crashes immediately after starting on the emulator, and we were unable to log in to Flipboard on the emulator. As shown in Table I, most apps have more than 100 million installations, indicating their high popularity. In addition, the apps fall into 17 diverse categories, suggesting the representativeness of these apps when being used to evaluate the effectiveness of automated UI testing approaches.

Baseline selection. To evaluate BADGE, we select four state-of-the-art automated UI testing tools and two state-of-the-practice tool to compare with.

For state-of-the-art tools, we select Stoat [16] and its enhancement by Toller [33] (denoted as *Stoatx*), Ape [17], and Q-testing [19]. Stoat and Ape are two model-based UI testing tools. Specifically, Stoat builds a stochastic model of the AUT with fixed UI abstraction rules and leverages Gibbs sampling to mutate the origin model. Ape, on the other hand, adjusts its UI abstraction granularity during testing based on dynamic analysis. Q-testing leverages curiosity-driven reinforcement learning to estimate the exploration values of UI events and guides the test generation toward under-explored functionalities.

In addition to state-of-the-art tools, we also include two state-of-the-practice tools, Monkey [21] and Fastbot [45]. Monkey [21] is a purely randomized tool for Android test input generation (from Google) that generates pseudo-random streams of UI events to the AUT, without considering the AUT's UI status. Due to its simplicity and effectiveness, Monkey is one of the most widely used tools in industrial settings, and it serves as the default baseline for evaluating automated UI testing [46], [16], [32], [17], [19], [24]. Fastbot [45], developed by ByteDance Inc., improves Ape with an exploration strategy based on reinforcement learning and crash-oriented fuzzing strategies [47].

Test platform. We conduct all experiments on the official Android x64 emulator running Android 6.0 on a server with four AMD EPYC 7H12 CPUs. Each emulator is allocated with 4 dedicated CPU cores, 2 GB RAM, and 2 GB internal storage. Hardware graphics acceleration is also enabled using two Nvidia GeForce RTX 3090 graphics cards to ensure the responsiveness of the emulator. We manually write auto-login scripts for apps marked with * in Table I. Each of these scripts is executed only once before the corresponding app starts to be tested in each test run. To alleviate the flakiness of these auto-login scripts, we manually check the collected traces afterward and rerun the experiments with failed login attempts.

Coverage and crash collection. We collect the activity and Java method coverage as code coverage achieved by each test run. To collect activity coverage, we use Android Dumpsys [48]. To collect Java method coverage, we use the MiniTrace [49] tool from Ape. By modifying DalvikVM/ART, our experiments do not require app instrumentation, avoiding unexpected issues from modifying industrial apps. Note that we exclude methods that are already covered after setting up the test environment before the test generation tool under consideration starts to work in each test run for a fair comparison. Following previous work [33], [17], we run each tool on each app three times, with each run lasting for one hour. We calculate the average code coverage achieved across the three runs. As for crashes, we consider only those originating from the AUT's bytecode. We use code locations in stack traces to identify unique crashes, and ignore slight differences using the approach of fuzzy stack hashing [50]. We obtain stack traces by monitoring Android Logcat [51] messages.

B. RQ1. Effectiveness of Code Coverage

Code coverage is one of the most widely used metrics to assess the effectiveness of automated UI testing tools. Given the same resources, higher code coverage indicates better effectiveness of automated UI testing tools.

Table II presents the code coverage effectiveness of automated testing tools. As shown in the table, BADGE achieves the highest activity coverage among all tools except for Stoat. It should be noted that Stoat can reach higher activity coverage

TABLE II
OVERVIEW OF TOOL RESULTS

ID	BADGE			Monkey			Stoat			Stoatx			Q-testing			FastBot			Ape		
	#Met.	**#Act.**	**#C**	**#Met.**	**#Act.**	**#C**	**#Met.**	**#Act.**	**#C**	**#Met.**	**#Act.**	**#C**	**#Met.**	**#Act.**	**#C**	**#Met.**	**#Act.**	**#C**	**#Met.**	**#Act.**	**#C**
A_1	8884	8.0	2	5719	5.7	0	3650	12.7	1	5288	8.3	1	5901	7.0	0	7438	6.7	2	7109	6.7	2
A_2	21561	5.7	3	17592	3.7	0	12275	8.3	1	18542	10.3	2	14641	4.0	0	14682	4.0	2	19466	5.0	2
A_3	40176	6.3	0	26493	4.3	0	10904	15.3	1	21602	6.0	2	33891	4.7	0	35200	6.7	1	34075	6.0	0
A_4	15529	12.3	1	12654	10.3	1	5728	17.7	0	13503	16.3	0	-	-	-	14239	12.7	0	14199	12.7	0
A_5	19512	29.3	1	11319	19.0	1	7971	19.3	1	13873	34.7	2	12369	20.7	0	17720	26.3	0	17892	28.7	2
A_6	4751	4.0	0	3401	3.0	1	1986	4.7	0	2505	3.3	0	2272	1.0	0	4486	3.7	0	3257	1.7	0
A_7	15563	25.7	3	14376	19.7	1	10012	22.7	0	14282	20.3	1	13813	21.3	0	15717	18.3	0	16115	25.3	0
A_8	11138	9.3	0	10385	9.7	0	7046	10.3	1	10162	9.7	0	-	-	-	9162	7.7	0	9252	6.7	0
A_9	10726	15.0	6	7387	14.7	0	3693	12.7	1	8077	15.0	1	4333	10.7	1	7863	11.7	2	10235	17.0	1
A_{10}	7113	22.7	0	6222	15.7	2	4220	22.7	1	5427	17.7	1	5336	22.3	0	5742	16.7	1	6686	22.7	1
A_{11}	10125	3.0	0	6309	2.0	1	6184	10.0	1	9082	6.3	1	7911	2.7	0	8625	3.0	1	8459	3.0	0
A_{12}	2023	1.0	0	230	1.0	0	328	1.3	0	258	1.3	0	1136	1.0	0	1752	1.0	0	699	1.0	0
A_{13}	41677	42.3	2	32382	24.3	4	14360	22.7	1	21106	16.3	0	34202	33.0	1	36538	33.7	5	36666	33.0	2
A_{14}	8881	15.3	1	5684	8.0	1	4351	10.3	0	8280	14.3	2	7356	12.7	0	8618	13.3	1	9039	13.3	0
A_{15}	20432	15.3	2	11720	6.7	0	8622	14.7	0	17112	15.0	0	12920	8.7	0	16072	12.3	1	16524	13.3	0
A_{16}	26901	54.3	5	17414	32.7	1	10052	17.7	1	22023	48.3	2	22792	56.7	1	22053	50.0	2	25304	58.0	1
A_{17}	19751	13.3	0	18912	11.3	0	5218	16.3	0	19803	19.3	2	18095	11.0	0	15441	9.7	0	19447	11.7	0
A_{18}	25647	36.0	0	11053	13.3	0	8669	23.7	2	20686	32.7	2	21403	26.0	0	21950	28.0	0	24141	38.0	0
A_{19}	29722	68.3	3	24527	51.0	6	10218	5.7	0	24559	50.3	1	23699	49.3	3	27754	58.7	7	20722	38.7	3
A_{20}	31200	8.3	1	19406	5.3	0	11112	14.7	0	20018	11.0	0	-	-	-	25350	5.0	0	25865	8.0	0
A_{21}	54975	7.7	1	32952	5.0	1	26541	9.0	1	26959	11.3	1	32346	3.3	0	36157	5.7	1	36384	5.7	0
Tot.	**20299**	**19.2**	**31**	**14102**	**12.7**	**20**	**8245**	**13.9**	**13**	**14435**	**17.5**	**21**	**15245**	**16.5**	**6**	**16789**	**15.9**	**26**	**17216**	**17.0**	**14**

*Notes: Best tool for each app is colored yellow in every row. The row with **Tot.** in the first column is the average of each tool's coverage and the count sum of the crashes triggered on every app. Hyphens (-) denote apps that trigger tool defects and are removed from average calculation. The removal of these apps does not weaken the related tool since its coverage on the removed apps is lower than its average coverage on the other apps.*

likely due to using an internal null-intent fuzzer that directly starts activities using empty intents.

In addition, the number of activities covered by all tools is relatively low, caused by two main factors. First, many activities are deprecated during the app evolution. This finding has been raised by the authors of WCTester [15], who have studied the activity coverage achieved by their automated UI testing tool and have found that around 40% not-covered activities simply cannot be accessed by users interacting with AUT UIs. Second, accessing some activities requires a specific type of user account, e.g., an administrator account (which is impractical to acquire) is needed to access management features.

In terms of method coverage, BADGE substantially outperforms all the other tools with 18%-146% relative improvement. In particular, BADGE improves the method coverage on almost all apps over the other tools. To examine the significance of the improvement on method coverage, we apply Tukey's tests [52] on the coverage for every tool on each app. We find that out of the 18 apps that BADGE performs the best, BADGE is the only tool in the best group in 12 of the 18 apps. Additionally, for the 3 apps that BADGE is not the best, BADGE is still among the top group on 2 of the 3 apps, demonstrating the significant improvement achieved by BADGE on almost all of the apps.

Note that most tools achieve similar activity coverage, while the method coverage achieved by different tools varies substantially. This result confirms our insight that the enormous UI elements in mobile apps raise challenges for automated UI testing to handle. Ape achieves the second highest method coverage, standing out from the baselines. While Toller [33] points out that Ape benefits from the efficiency of UI hierarchy capturing by exploiting private Android accessibility APIs, compared with Stoatx, the state abstraction refinement strategy in Ape can still help alleviate the complexity of industrial apps.

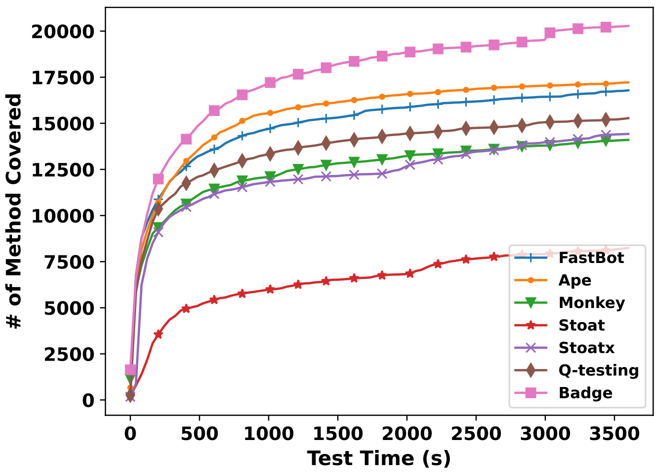

Fig. 3. Progressive code coverage of different tools.

BADGE uses UIAutomator to dump UI hierarchies and execute UI events instead of using private accessibility APIs, consistent with other state-of-the-art tools except for Ape. Even with a bit lower infrastructure efficiency, BADGE still substantially outperforms Ape with 18% method coverage improvement. Compared with tools with similar infrastructure efficiency, BADGE achieves >20% relative method coverage improvement, with the greatest being 146%.

Figure 3 presents the progressive code coverage achieved

TABLE III
CRASH CATEGORIZATION OF DIFFERENT TOOLS

Exception Type	Ba.	Ape	Fa.	Mo.	St.	Stx.	Qt.
NullPointer	7	5	10	8	4	8	4
Runtime	5	3	3	7	11	10	1
IllegalState	6	2	3	2	0	3	0
IllegalArgument	1	2	3	1	0	1	0
IndexOutOfBounds	8	1	4	0	0	0	0
ArrayIOOB	1	0	1	0	0	0	0
Assertion	1	0	0	0	0	0	0
ClassCast	1	1	0	1	0	0	0
NumberFormat	1	0	0	0	0	1	0
Others	0	0	2	1	0	0	1
Sum	**31**	**14**	**26**	**20**	**15**	**22**	**6**

Note:Due to space limit, in the first column of the table, we remove 'Exception' from the end of each exception type and shorten 'ArrayIndexOutOfBounds' to 'ArrayIOOB'.

by different tools. We find that at the beginning, BADGE starts with a lower method coverage since it has no knowledge of the AUT. However, with the test time advancing, BADGE quickly gains knowledge of the AUT from trials and surpasses all the other tools. In addition, the method coverage achieved by BADGE does not converge within one hour and grows steadily, showing its promising potential and effectiveness. From the experimental results, we conclude that BADGE substantially outperforms existing state-of-the-art/practice tools, with the potential to be further improved given better infrastructure support.

A1: BADGE substantially outperforms state-of-the-art/practice tools in terms of code coverage.

C. RQ2. Effectiveness of Crash Revelation

In addition to code coverage, the crash-revealing capability is another widely used metric for assessing the effectiveness of automated UI testing. We specifically care about the number of unique crashes that a tool can trigger through UI interactions. A notable case to consider is Stoat, which has an internal null-intent fuzzer that can trigger trivial crashes [17]. These crashes are reproducible only on emulators and will not be exposed to users. To exclude these trivial crashes caused by non-UI testing, following the practice of Ape [17], we run a null-intent fuzzer for one hour for each app and filter out the crashes triggered by the fuzzer.

We categorize the stack traces of the collected crashes by the exception types. Table III presents the number of unique crashes triggered by each tool. As shown by the table, BADGE detects the highest number of unique crashes by pure UI testing. Fastbot finds the second highest number of unique crashes, achieved through integrating the technique of crash-oriented fuzzing proposed by previous work [47] and the technique of system-event fuzzing such as screen rotation and changing network conditions. Note that Stoat also benefits from the technique of system-event fuzzing. In addition, we find that the technique of generating proper text inputs helps trigger certain crashes. For example, Ape triggers an IllegalArgumentException on the AccuWeather app with a string containing '%'. The finding is consistent with previous work [14], [47].

Our evaluation focuses on characterizing the benefits solely

TABLE IV
COVERAGE RESULTS OF INDIVIDUAL ALGORITHMS

ID	T_1	T_2	T_3	T_4	T_5	T_6	BADGE
A_1	7963	7295	7890	7160	7542	3068	8884
A_2	15993	18540	19833	16264	18936	1732	21561
A_3	34865	28168	30810	33083	28113	31959	40176
A_4	13444	14783	14540	13240	14150	2238	15529
A_5	14981	16833	18330	16349	17518	7767	19512
A_6	2786	4254	4309	2543	4516	3251	4751
A_7	13713	14650	15323	15008	15318	6699	15563
A_8	8066	10458	10806	6981	10621	6516	11138
A_9	5345	9880	9010	9594	9553	2005	10726
A_{10}	4866	5469	5452	6179	5466	5908	7113
A_{11}	5409	9332	8612	9843	9065	8612	10125
A_{12}	1884	1694	1671	524	1669	1992	2023
A_{13}	38243	37420	38294	42496	39476	22305	41677
A_{14}	8427	8263	8308	8708	8612	4641	8881
A_{15}	16742	14482	17075	19558	12513	4803	20432
A_{16}	5291	24970	22600	22700	24400	16285	26901
A_{17}	16370	19328	19587	18418	19602	3690	19751
A_{18}	8155	22092	21980	26867	20436	4724	25647
A_{19}	25480	24496	27049	28254	25450	17815	29722
A_{20}	4766	16978	21081	19575	18571	10520	31200
A_{21}	36438	33984	27896	37259	42295	23823	54975
Avg.	**13773**	**16351**	**16688**	**17172**	**16849**	**9064**	**20299**

Note: In the table header, EXPLORE, GREEDY, MCRW, MYOPIA, NMAB, *and* GROUP *are denoted by T_1 to T_6, respectively. The technique achieving the best result for each app is colored yellow in every row.* ***Avg.*** *is the average method coverage on all apps.*

from the exploration-effectiveness improvement by BADGE. In addition, the preceding techniques complement the design of BADGE. Consequently, BADGE can further improve the crash-revealing capability by integrating these techniques, which can be our future work.

A2: BADGE achieves state-of-the-art crash-revealing capability compared with state-of-the-art/practice tools and benefits from its superior code-coverage capability. Its crash-revealing capability can be further enhanced with proper text inputs and system events along with finer state abstraction.

D. RQ3. Benefits Brought by BADGE's Individual Algorithms

To evaluate the benefits brought by individual algorithms that instantiate BADGE, we conduct experimental studies by instantiating BADGE with different baseline algorithms for prioritizing UI events, estimating exploration values of UI events, and formulating the MAB (i.e., whether to exploit the exploration diversity of UI groups).

For UI-event prioritization, we use two popular prioritization algorithms as baselines, namely EXPLORE [39] and GREEDY [31]. EXPLORE always prioritizes unexplored UI events if there exist unexplored ones; otherwise, EXPLORE behaves the same as BADGE. GREEDY adopts an $\epsilon-$greedy prioritization algorithm [31]. With probability ϵ, GREEDY prioritizes unexplored UI events to interact with. With probability $1 - \epsilon$, GREEDY behaves the same as BADGE. For GREEDY, we set $\epsilon = 1$ at the beginning and slowly decay it to $\epsilon = 0.5$ by the end of testing.

For exploration value estimation, we use two algorithms as baselines, namely MYOPIA and MCRW. MYOPIA determines the exploration values of UI events based on changes in the UI screens after triggering the UI events (similar reward

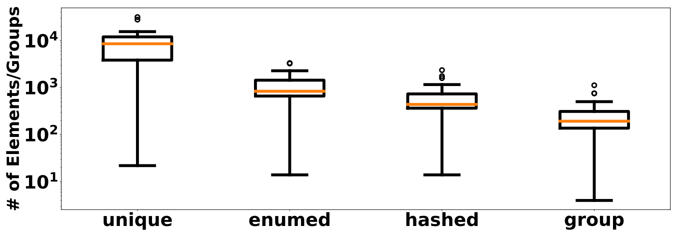

Fig. 4. Distributions of the number of different UI elements/groups encountered by BADGE's explorations in each app with different abstractions, where **unique** counts unique elements based on their types along with text-attribute values and position-attribute values; **enumed** is the same as **unique** except ignoring elements' text-attribute values; **hashed** is the same as **enumed** except ignoring elements' position-attribute values; **group** counts unique groups based on their group IDs. The Y-axis is on a logarithmic scale.

generation mechanisms are used by previous work [53], [22]). In particular, MYOPIA considers one-step exploration rooted from a UI event. MCRW replaces the MCTSTAR algorithm with standard Q-learning [31] and updates Q values (i.e., the exploration values for UI events) with Bellman Equation [31].

For MAB formulation, we use two algorithms as baselines, namely NMAB and GROUP. NMAB does not estimate the exploration diversity of UI groups and instead estimates the exploration values and exploration diversity of UI events, ignoring the hierarchical relationships of UI elements/events. GROUP estimates and exploits exploration values and exploration diversity of UI groups instead of individual UI events. When selecting UI events, GROUP randomly selects a UI event from the UI group with the highest priority. GROUP can be seen as BADGE instantiated with an algorithm of coarse UI-event abstraction that treats UI events with the same group identifier as equivalent ones.

Table IV shows the results of code coverage. **Impact of using different prioritization algorithms**. EXPLORE is substantially worse than BADGE since EXPLORE considers only the exploration values of UI elements when there exist unexplored UI events. As shown in Figure 4, about 10,000 unique UI elements (belonging to 300 UI groups on average) on average are encountered during testing. Without considering the exploration diversity of UI events, EXPLORE fails to prioritize UI events to achieve good code coverage. GREEDY outperforms EXPLORE by considering both exploration diversity and exploration values of UI events in $1-\epsilon$ time of testing, but the algorithm's coverage is still worse than BADGE, which balances exploration diversity and exploration values of UI events. The comparison among EXPLORE, GREEDY, and BADGE demonstrates the benefits of prioritizing UI events by jointly considering their exploration values and exploration diversity.

Impact of using different algorithms for exploration value estimation. From Table IV, MCRW is worse than MYOPIA in terms of code coverage. Since the exploration values of UI events evolve during testing, standard Q-learning cannot quickly adapt to the changes and fails to achieve satisfactory effectiveness. MYOPIA is comparative to BADGE on some of the apps but worse on the others. On complex industrial apps, differences between S_{e_1} and S_{e_2} for two events e_1 and e_2 may not be revealed until several steps later, and these differences cannot be recognized by MYOPIA since it considers only one-step exploration rooted from a UI event. The MCTSTAR algorithm considers the changes of exploration values during testing and a longer horizon of future exploration rooted from a UI event, outperforming the two baseline algorithms. The comparison among MCRW, MYOPIA, and BADGE demonstrates the benefits of estimating exploration values of UI events by considering a longer horizon of future exploration and the changes of exploration values during the testing.

Impact of using different algorithms for MAB formulation. As shown in Table IV, GROUP achieves the worst method coverage. As shown in Figure 4, the number of UI groups is less than 10% of the total number of unique elements. UI elements/events in the same UI group can trigger very different behaviors. GROUP treats different UI events in the same group as being equivalent, losing great opportunities to reveal new code coverage. NMAB does not estimate the exploration diversity of UI groups and achieves a worse estimation of exploration values and exploration diversity compared with BADGE. Consequently, NMAB can spend more time on UI events with low exploration diversity or exploration values. BADGE estimates the exploration diversity of UI groups with the hierarchical MAB model, achieving higher code coverage than the baseline algorithms.

A3: The Thomspon-sampling-based algorithm for UI-event prioritization, MCTSAR-based algorithm for exploration value estimation, and the hierarchical MAB construction algorithm all bring high benefits to BADGE in terms of achieving code-coverage effectiveness.

VI. THREATS TO VALIDITY

The main external threats to the validity include the extent to which the subject apps and tools selected for our evaluation are representative of real-world practice. To mitigate the impact of the bias introduced by app selection, we use highly popular industrial apps widely used by related work. To mitigate the impact of the bias introduced by tool selection, we choose the latest and widely used state-of-the-art/practice automated UI testing tools for comparison.

The threats to internal validity are instrumentation effects that can bias our results, including faults in our implementation of BADGE and randomness. Faults in the BADGE implementation may affect the effectiveness of BADGE. To reduce these threats, all the authors carefully test and validate the BADGE implementation on the Autoscout24 app to assure the normal behavior of the BADGE implementation. To reduce the randomness of experiments, we run all the tools under comparison on each app three times following previous work to mitigate the issue.

VII. Related Work

Automated mobile UI testing has been a hot research topic [21], [14], [16], [5], [6], [7], [8], [9], [10], [11], [12], [13], [17], [18], [19], [24] as well as a popular industry practice [14], [15], [54]. Existing tools fall into four main categories. (1) Some tools [21], [14], [8], [9], [18] generate test inputs randomly and/or applying evolutionary algorithms upon these test inputs. For example, Monkey [21] generates pseudo-random inputs such as clicks and drags to stress the AUT. VTest [54] generates random inputs based on only app screenshots to facilitate cross-platform UI testing. (2) Some other tools [11], [12], [13] conduct systematic exploration. A3E [11] systematically explores the AUT with a depth-first search, and EvoDroid [12] uses evolutionary algorithms to perform a step-wise search for test inputs to reach deep into the code under test. (3) Model-based tools and their variants [5], [6], [16], [10], [17] use a *UI transition model* to determine the current test progress and target those not-yet-explored functionalities. Stoat [16] first constructs a UI transition model of the AUT and uses Gibbs sampling [55] to generate test inputs toward unexplored parts of the model. (4) Machine-learning-based tools [22], [23], [19], [24] adopt deep learning or reinforcement learning techniques to guide the exploration of the AUT. Q-testing [19] adopts curiosity-driven reinforcement learning toward less frequently explored UI screens, and ARES [24] adopts deep reinforcement learning to effectively generate long event sequences. BADGE is closely related to model-based or reinforcement-learning-based approaches.

Model-based approaches [5], [6], [16], [10], [17] build models with dynamic or static strategies to describe the AUT's behaviors and then derive test cases from the models. Model-based UI testing approaches usually adopt UI event abstraction techniques [16], [17] to alleviate the explosion of the large space of UI events as well as state space. To some extent, the current implementation of BADGE builds a UI model at the activity level. However, BADGE focuses on a problem (characterizing the exploration diversity of UI events) different from typical model-based approaches, and we believe that BADGE can be combined with a finer model of the AUT to further improve the test effectiveness.

Reinforcement-learning-based approaches [22], [19], [24] adopt reinforcement learning (RL) algorithms to guide test generation. BADGE benefits from RL algorithms to estimate the exploration values of UI events. Existing RL-based approaches focus on exploration values with novel RL algorithms. Q-testing [19] uses curiosity-driven reward and ARES [24] uses deep learning to improve the generalizability of automated UI testing. On one hand, BADGE may also benefit from similar designs. On the other hand, BADGE characterizes the exploration diversity of UI events, which can help improve existing RL-based approaches.

MABs and their applications. In recent years, MAB algorithms have attracted a lot of attention in various applications [56], [56], [57], [58], [59] due to their stellar performance with relatively limited feedback to learn a good policy. In particular, MAB has been used for software engineering tasks including test case prioritization [45], coverage-guided fuzzing [60], and UI testing [53]. In the task of UI testing, Degott et al. [53] leverage an MAB algorithm to infer valid event types for UI elements with UI change feedback. Different from their work, BADGE learns the exploration diversity and exploration values of UI events, and derives event types for UI elements directly from the Android system by using UIAutomator.

VIII. Conclusion

In this paper, we have pinpointed and formulated UI elements' exploration diversity, which substantially affects test effectiveness of automated UI testing. We have proposed the BADGE approach to prioritize UI events taking into consideration their exploration values and exploration diversity. We have designed a novel hierarchical multi-armed bandit model that jointly characterizes the exploration values and exploration diversity to maximize the achieved code coverage. Our extensive evaluation on 21 widely used industrial apps demonstrates that BADGE substantially outperforms state-of-the-art/practice automated UI testing tools with 18%-146% relative method coverage improvement and finding 1.19-5.20x unique crashes. Our evaluation also pinpoints the potential of improving automated UI testing effectiveness with techniques of finer UI abstraction, system event fuzzing, and proper text inputs. We plan to design and integrate these techniques in BADGE as future work.

Acknowledgments

Tao Xie is also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education China. This work was partially supported by National Natural Science Foundation of China under Grant No. 62161146003, and the Tencent Foundation/XPLORER PRIZE.

References

[1] IDC and Gartner. (2019) Share of Android OS of Global Smartphone Shipments from 1st Quarter 2011 to 2nd Quarter 2018. [Online]. Available: https://www.statista.com/statistics/236027/global-smartphone-os-market-share-of-android/
[2] Statista. (2022) Number of smartphone subscriptions worldwide. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/
[3] I. Intelligence. (2022) US Time Spent with Connected Devices 2022. [Online]. Available: https://www.insiderintelligence.com/content/us-time-spent-with-connected-devices-2022
[4] 42matters. (2022) Data on App Update Frequency. [Online]. Available: https://42matters.com/google-play-aso-with-app-update-frequency-statistics
[5] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *OOPSLA*, 2013.
[6] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *MobiSys*, 2014.
[7] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *FASE*, 2013.
[8] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *ESEC/FSE*, 2013.
[9] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android apps with intent-filter tag," in *MoMM*, 2013.

[10] Y. Li, Z. Yang, Y. Guo, and X. Chen, “DroidBot: A lightweight UI-guided test input generator for Android,” in *ICSE-C*, 2017.
[11] T. Azim and I. Neamtiu, “Targeted and depth-first exploration for systematic testing of Android apps,” in *OOPSLA*, 2013.
[12] R. Mahmood, N. Mirzaei, and S. Malek, “EvoDroid: Segmented evolutionary testing of Android apps,” in *FSE*, 2014.
[13] S. Anand, M. Naik, M. J. Harrold, and H. Yang, “Automated concolic testing of smartphone apps,” in *FSE*, 2012.
[14] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for Android applications,” in *ISSTA*, 2016.
[15] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, “Automated test input generation for Android: Towards getting there in an industrial case,” in *ICSE-SEIP*, 2017.
[16] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based GUI testing of Android apps,” in *ESEC/FSE*, 2017.
[17] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, “Practical GUI testing of Android applications via model abstraction and refinement,” in *ICSE*, 2019.
[18] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, “Time-travel testing of Android apps,” in *ICSE*, 2020.
[19] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement learning based curiosity-driven testing of Android applications,” in *ISSTA*, 2020.
[20] T. Cai, Z. Zhang, and P. Yang, “Fastbot: A multi-agent model-based test generation system,” in *AST*, 2020.
[21] Google. (2021) Android Monkey. [Online]. Available: https://developer.android.com/studio/test/monkey
[22] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, “QBE: Qlearning-based exploration of Android applications,” in *ICST*, 2018.
[23] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Humanoid: A deep learning-based approach to automated black-box Android app testing,” in *ASE*, 2019.
[24] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, “Deep reinforcement learning for black-box testing of Android apps,” *TOSEM*, 2022.
[25] R. Real and J. M. Vargas, “The probabilistic basis of jaccard’s index of similarity,” *Systematic biology*, 1996.
[26] S1T2. (2015) Generously apply crap to design. [Online]. Available: https://s1t2.com/blog/step-1-generously-apply-crap-to-design
[27] M. Xie, Z. Xing, S. Feng, C. Chen, L. Zhu, and X. Xu, “Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images,” *arXiv preprint arXiv:2206.10352*, 2022.
[28] L. Clapp, O. Bastani, S. Anand, and A. Aiken, “Minimizing GUI event traces,” in *FSE*, 2016.
[29] J. GUO, S. LI, J.-G. Lou, Z. YANG, and T. LIU, “SARA: Self-replay augmented record and replay for Android in industrial cases,” in *ISSTA*, 2019.
[30] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine learning*, 2002.
[31] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
[32] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, “An empirical study of Android test generation tools in industrial cases,” in *ASE*, 2018.
[33] W. Wang, W. Lam, and T. Xie, “An infrastructure approach to improving effectiveness of Android UI testing tools,” in *ISSTA*, 2021.
[34] W. Wang, W. Yang, T. Xu, and T. Xie, “Vet: identifying and avoiding UI exploration tarpits,” in *ESEC/FSE*, 2021.
[35] Google. (2022) Android Components. [Online]. Available: https://developer.android.com/guide/components/fundamentals
[36] Google. (2021) Android View. [Online]. Available: https://developer.android.com/reference/android/view/View
[37] Google. (2021) Android ViewGroup. [Online]. Available: https://developer.android.com/reference/android/view/ViewGroup
[38] ——. (2022) GUI Events. [Online]. Available: https://developer.android.com/topic/architecture/ui-layer/events
[39] J. Vermorel and M. Mohri, “Multi-armed bandit algorithms and empirical evaluation,” in *ECML*, 2005.
[40] W. R. Thompson, “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples,” *Biometrika*, 1933.
[41] Wikipedia. (2023) Beta distribution. [Online]. Available: https://en.wikipedia.org/wiki/Beta_distribution
[42] Google. (2022) UI Automator. [Online]. Available: https://developer.android.com/training/testing/uiautomator
[43] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of Monte Carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in games*, pp. 1–43, 2012.
[44] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, 2016.
[45] J. A. do Prado Lima and S. R. Vergilio, “A multi-armed bandit approach for test case prioritization in continuous integration environments,” *TSE*, 2020.
[46] S. R. Choudhary, A. Gorla, and A. Orso, “Automated test input generation for Android: Are we there yet?” in *ASE*, 2015.
[47] T. Su, J. Wang, and Z. Su, “Benchmarking automated GUI testing for Android against real-world bugs,” in *ESEC/FSE*, 2021.
[48] Google. (2022) Dumpsys command-line tool. [Online]. Available: https://developer.android.com/studio/command-line/dumpsys
[49] T. Gu. (2021) Minitrace. [Online]. Available: http://gutianxiao.com/ape
[50] D. Molnar, X. C. Li, and D. A. Wagner, “Dynamic test generation to find integer bugs in x86 binary Linux programs,” in *USENIX Security*, 2009.
[51] Google. (2022) Logcat command-line tool. [Online]. Available: https://developer.android.com/studio/command-line/logcat
[52] H. Abdi and L. J. Williams, “Tukey’s honestly significant difference (hsd) test,” *Encyclopedia of research design*, 2010.
[53] C. Degott, N. P. Borges Jr, and A. Zeller, “Learning user interface element interactions,” in *ISSTA*, 2019.
[54] D. Ran, Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, H. Jin, J. Cui, X. Tang, and T. Xie, “Automated visual testing for mobile apps in an industrial setting,” in *ICSE-SEIP*, 2022.
[55] W. R. Gilks, N. G. Best, and K. K. Tan, “Adaptive rejection metropolis sampling within Gibbs sampling,” *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 1995.
[56] A. Biswas, G. Aggarwal, P. Varakantham, and M. Tambe, “Learn to intervene: An adaptive learning policy for restless bandits in application to preventive healthcare,” *arXiv preprint arXiv:2105.07965*, 2021.
[57] D. Bouneffouf, A. Bouzeghoub, and A. L. Gançarski, “Contextual bandits for context-based information retrieval,” in *ICNIP*, 2013.
[58] Q. Zhou, X. Zhang, J. Xu, and B. Liang, “Large-scale bandit approaches for recommender systems,” in *ICNIP*, 2017.
[59] X. Huo and F. Fu, “Risk-aware multi-armed bandit problem with application to portfolio selection,” *Royal Society open science*, 2017.
[60] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, “EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit,” in *USENIX Security*, 2020.