Fastbot: A Multi-Agent Model-Based Test Generation System

Beijing Bytedance Network Technology Co., Ltd.

Tianqin Cai
Product RD & Infrastructure
Bytedance Network Technology
Beijing, China
caitianqin@bytedance.com

Zhao Zhang
Product RD & Infrastructure
Bytedance Network Technology
Beijing, China
zhangzhao.a@bytedance.com

Ping Yang
Product RD & Infrastructure
Bytedance Network Technology
Beijing, China
yangping.cser@bytedance.com

ABSTRACT

Model-based test (MBT) generation techniques for automated GUI testing are of great value for app testing. Existing GUI model-based testing tools may fall into cyclic operations and run out of resources, when applied to apps with industrial complexity and scalability. In this work, we present a multi-agent GUI MBT system named Fastbot. Fastbot performs model construction on the server end. It applies multi-agent collaboration mechanism to speed up the model construction procedure. The proposed approach was applied on more than 20 applications from Bytedance with more than 1500 million monthly active users. Higher code coverage in less testing time is achieved with comparison of three other automated testing tools including Droidbot, Humanoid and Android Monkey.

CCS CONCEPTS

• **Software and its engineering** → Software testing and debugging; Model-based software engineering

KEYWORDS

Model-based GUI testing, dynamic DAG exploration, multi-agent collaboration, automatic testing, traversal algorithm

1 Introduction

Nowadays, functionality of apps is getting increasingly complex. The app with most robustness will have better competiveness in the app stores. Therefore, automating test generation (ATG) for apps becomes an important research direction.

AST '20, October 7–8, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7957-1/20/05.
https://doi.org/10.1145/3387903.3389308

GUI-based ATG techniques have been widely studied, such as model-based testing, search-based testing, coverage-guided fuzzing and symbolic execution. Unlike the ATG methods based on code analysis which require full control flow graph generated from source code, testing based on GUI has the ability to build app model based on the UI information. The state-of-the-art tools includes SAPIENZ [1], which applies random fuzzing, systematic and search-based exploration; DYNODROID [2], which views the app as sequences of events and generates relevant, intelligent inputs; other popular tools include DroidMate [3], PUMA and Android Monkey by Google.

Particularly, model based testing (MBT) in ATG obtains great advantage for better reusability. Droidbot [4] is one of the cut-edge GUI-based MBT tools, which performs on-the-fly model-based testing with graph exploration algorithms. Humanoid [5] brought by Peking University brought up the new strategy of guiding exploration event by user-behaviors. Mostly, model-based GUI tests are based on 2 artifacts: The Finite State Machine (FSM) which describes all possible testing paths, and the operational profiles which describe the transition from one state to another.

However, there are two disadvantages for the traditional model-based methods, when applied to apps with industrial complexity and scalability. Firstly, it is easy to fall into cyclic operations in similar pages using the "GUI changed" as the exploration strategy for the state judgement. By applying DFS and BFS algorithm in exploration process, these tools including DROIDBOT may keep trapping in same scenarios which severely limits the activity coverage. While for Humanoid, the accessibility of user-behavior data becomes largest obstacle. Secondly, the rapidly expanding model will take up the memory of the mobile devices.

To deal with the above-mentioned problems, we proposed Fastbot, a Model-based Automatic Test Generation system, where we achieved multi-client collaboration model construction mechanism, and applied algorithms based on UCB algorithm [6] and reinforcement learning to achieve better exploration capability.

2 Design and Implementation

2.1 Fastbot Workflow

As mentioned above, the memory and calculation capability of mobile devices have become the main limitation for model-based GUI testing. By applying distributed computing system, Fastbot moves the model-related computationally expensive part onto server end and only keeps UI information collection and action injection job on client end. The workflow is shown in Figure1.

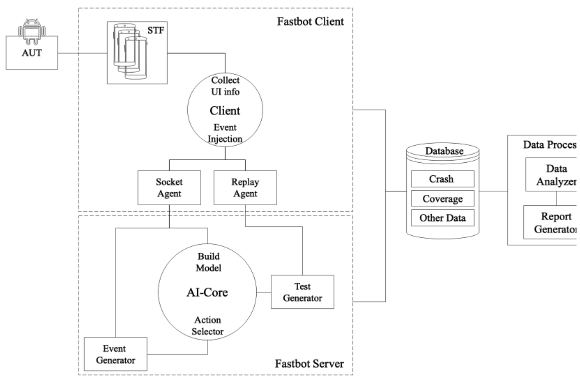

Figure 1: Fastbot System Workflow

The Fastbot system supports multiple app testing tasks working simultaneously without affecting each other. For every single task, the App Under Test (AUT) and user-defined configurations will be deployed on multiple mobile devices. On each device, a Fastbot client will take charge of UI recognition and event injection work. Obtained UI info of current state will be sent to server through the socket agents. Correspondingly, on server end there are agents analyzing the received UI info, formatting this info into a so-called state as input of the algorithms in AI-core. Each agent will select next event based on their input state, assigned algorithms, and the model info, meanwhile collaborate to construct a static model stored in server memory. Selected events will be transferred back to client side for execution, then new UI-info will be captured and sent back to server again. In this process, data of crash info, code coverage and effective paths will be collected for data analyzing, case replay and test generation.

2.2 Fastbot Model Description

By defining state as abstraction of UI info on current page and actions as the events to take, a directed acyclic graph is constructed from the event trace of clients with state as graph nodes and actions as edges. The model in Fastbot is based on this DAG. The left part in Figure 2 shows a brief example of our model. The arrow dashed lines represent actions that directs and connects the states shown in circles. With multi-agent collaborating, our composite model is shown in the right part of Figure 2, where each color represents the traversal path of a unique agent.

Defining states with fine granularity is a challenging job. Without any abstraction on GUI info, the amount of states will explode sharply resulting in OOM problem on server for the reason of unlimited Feed Pages and so on. Through our work, the state abstraction function defined by Activity name, Action Type and widgets distribution obtained from flattened GUI tree structure is observed to have best performance.

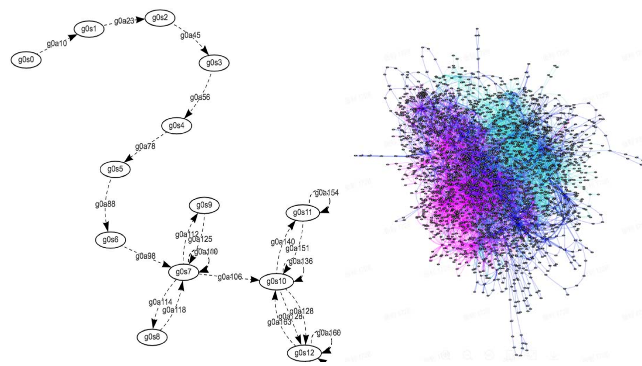

Figure 2: DAG Model

2.3 Algorithms in AI core

Traditional traversal algorithms based on DFS and BFS have their limitation on a dynamic DAG model built on dynamic app rather than native app, where duplicated path is not promised to lead to constant destination, for the reason of the disturb from forever-changing contents in Feed Pages. Here, we present the following better fitting algorithms in this scenario.

Aiming to cover more actions in every state, we define the priority of a state as the total value of actions in this state, where value of action is determined by action type and action visited tag. States with higher priority are of more value to be visited again.

For greedy algorithm that always distributes actions with max target priority, the actions leading to Feed Page state that cannot be reached again will trap the agent in infinite loops.

Our first algorithm is based on one-step UCB equation (upper confidence bound) [6] to balance exploration and exploitation. In cases where state still has unvisited actions, the action priority simply equals previously defined action value. Otherwise, the priority of action is denoted as UCB value calculated from Equation 1 with corresponding illustration in Figure 3.

$$UCB\ value\ of\ visited\ Action = \frac{P_t}{V_t} + C\sqrt{\frac{\ln V_c}{V_t}},\ C{:}\ Constant \quad (1)$$

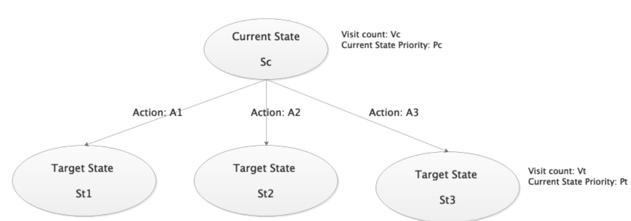

Figure 3: Definition of One-Step UCB

One of the drawbacks of the above-mentioned strategy is that it only calculates the UCB value within one step, while states hiding several steps behind may have higher value to be visited. The n-step UCB algorithm is brought up as an optimization to overcome this drawback. Instead of the using priority of action's target state in UCB equation, we use accumulation of all the target priority multiplied by discount factor γ in the following n steps as Pt in Equation 1. The n-step UCB value of Action is given in Equation 2. However, this algorithm requires traversal in the following n steps and demands an exponential time complexity. A gradually-increasing step number n fits best in this scenario.

$$n - step\ UCB\ value\ of\ Action = \frac{\sum_i^n(\gamma^i \sum P_{ti})}{V_t} + C\sqrt{\frac{\ln V_c}{V_t}} \qquad (2)$$

The algorithm is described in the following pseudocode:

Algorithm 1: Action selection with UCB on DAG model

```
While test not finished:
    Receive GUI-Tree from clients
    Current state ← f(GUITree)
    Update model with current state and previous movement
For actions in current state:
        Calculate one-step or n-step UCB value of actions
Select action based on UCB value
Mark selected action as visited, update action value and state priority
Send Action back to clients for execution
```

Still, the information beyond n steps is not fully utilized for the current decision, since backpropagation is not applicable on DAG. Another algorithm named MTree is designed inspired by Monte-Carolo Search Tree Algorithm applied in AlphaGo [7], where we use tree structure in addition of DAG as our model. Each tree node represents one state, and the target states leading by the actions in tree node are added as child nodes. The actions leading to a previous visited state is added as step child to prevent cycle. An activity list is stored in every tree node suggesting the following reachable activities. When new activities are discovered, a backpropagation process from current node to the root will be applied, updating the activity list in child nodes. An example of MTree structure is shown in Figure 4. Each color represents a unique activity. Starting from State 0 as root node, the tree structure expands downward during the process of exploration. Action 5 shown in dashed line leads back to the visited Root Node and forms a cycle; thus, we add State 5 as step child node. Step children won't involve in the backpropagation or action selection process. The use of them is to navigate back when all the following children are saturated. Correspondingly, in the action selection part, exploration for unvisited actions is still the main choice; for the visited actions, we have the UCB equation for actions leading to child nodes shown in Equation 3, where Vc and Vp means visit count of child node and parent node.

Algorithm 2: MTree Algorithm

```
    …
    Current state ← f(GUITree)
    Build tree node N from current state
    If N not in MTree:
        Add N as previous state node's child
        Update reachable activity list from current node to root
    Else:
        Add N as step child
    For unvisited actions and actions pointing to child nodes:
        Calculate UCB value of target node by action
    If no children and no unvisited actions:
        Select action from step children
    Else:
        Select action according to UCB value
    Send Action back to clients for execution
```

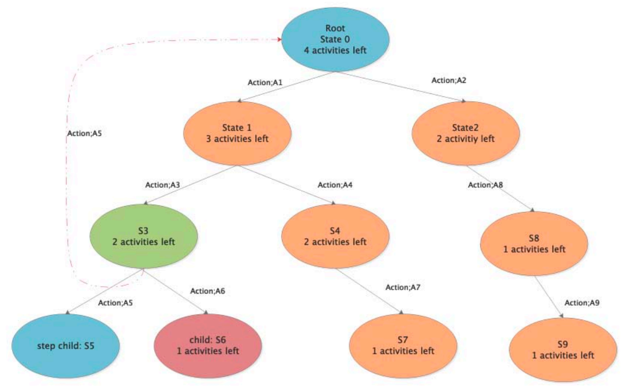

Figure 4: An Example of MTree

$$UCB\ value\ of\ child = \frac{f(Activities\ under\ child\ node)}{V_c} + C\sqrt{\frac{\ln V_P}{V_c}} \quad (3)$$

Q-Learning algorithm capturing action selection feature on DAG model also brought smart navigation. In this algorithm, Q value is calculated for every state-action pair with forward actions gets positive reward based on a state difference function, while actions leading to visited states will get negative rewards based on visit count. Thus, agent learns to avoid infinite loop and discover new states. N-step Q-learning with UCB is applied for optimization.

Algorithm 3: Q-learning with State-Diff Reward Function

```
    …
    Update model based on current state and previous movement
    Reward ← f(Current State, Previous State)
    Q(S,A) ← (1 − α)Q(S,A) + α[Reward + γQ(S′,A′)]
    For actions in current state:
        Get UCB of action based on Q(current State, action)
    Select action based on UCB value
    …
```

3 Results

Fastbot has already been integrated into Bytedance testing framework, serving more than 20 applications as the main stability and compatibility testing tool. Over 300 daily-build tasks are scheduled, applied with various customized configuration to match the demand from every product line. Around 5,000 crashes are being exposed by Fastbot every day, among which more than 100 crashes are newly discovered. The crash info is reported to our bug system and assigned to related developers for investigation. With this tool, the culprit patch is assumed to be found and fixed before app released.

The following experimental evaluation includes code coverage and activity coverage data on one of our app named Toutiao. Table 1 shows one-hour and three-hour testing data on a single device. Comparing with the popular traversal testing tools including Monkey, Droidbot and Humanoid, Fastbot achieved a much higher coverage performance.

Metrics	Droidbot	Monkey	Humanoid	Fastbot
Activity Coverage/1h	4.31%	——	4.33%	16.14%
Code Coverage/1h	8.33%	9.79%	8.73%	19.00%
Activity Coverage/3h	6.10%	——	6.34%	18.07%
Code Coverage/3h	11.33%	14.70%	12.05%	23.00%

Table 1: Coverage Comparison

Moreover, Fastbot's multi-device collaboration in client/server pattern dramatically enhances the exploration capability. Figure 5 shows the activity coverage performance of Fastbot working on 1, 3, 20 and 50 devices. Obviously, multiple devices collaboration brings a higher coverage rate and faster exploartion speed. Up to 100 devices collaborating on a single model for 50 hours are supported by our distributed system on server end, resulting in a 47.88% activity coverage, without triggering OOM problem or slowing down the action decision speed. As Table2 shows, three clients collaboration by Fastbot achieved a 84% activity coverage enchancement compared to single device test, while for Droidbot and Humanoid, the enhancement from one device to three devices is only 20.41% and 24.97%, and for Monkey is 45.79%.

Code Coverage/1h	Droidbot	Monkey	Humanoid	Fastbot
One Device	8.33%	9.79%	8.73%	19.00%
Three Devices	10.03%	14.28%	10.91%	35.00%

Table 2: Effect of device number

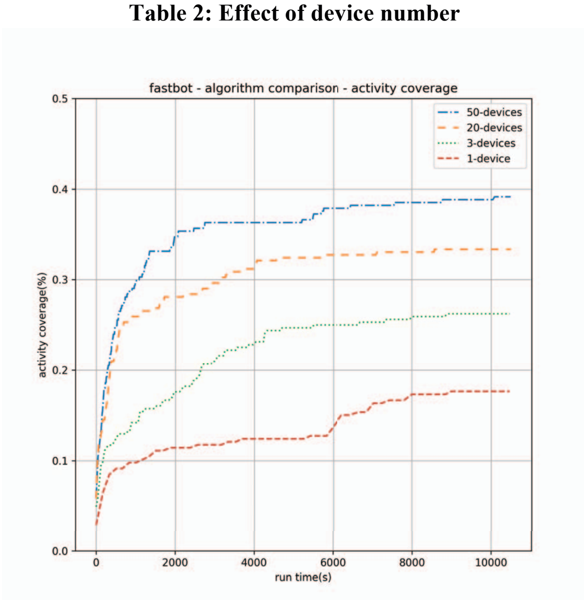

Figure 5: Coverage Comparison on multiple clients

Figure 6 compares the traversing capability by the four above-mentioned algorithms, together with traditional DFS and Random algorithm. Each test is deployed with 20 devices. As shown in the figure, DFS agent stuck soon in the internal loops, and activity coverage stops increasing. Random Agent has better performance in the dynamic DAG circumstances, achieving around 26% coverage. One-step UCB algorithm possesses best exploration speed at the early stage, while the potential to discover activities hidden behind complex path becomes its weakness. On the contrary, coverage rate of n-step UCB agent grows slower for the reason of (n-1) repeated actions it needs to cover before reaching target state, but in later periods it demonstrates better exploration capability for the deeper inception. Performance of DQN agent is unstable. It has upper-limit better than n-step agent, and in the meantime, requires less computation resources. Nevertheless, the performance is severely affected by the uncontrollable action choices in early exploration stage. Among our tests, the MTree Agent exhibits best overall performance.

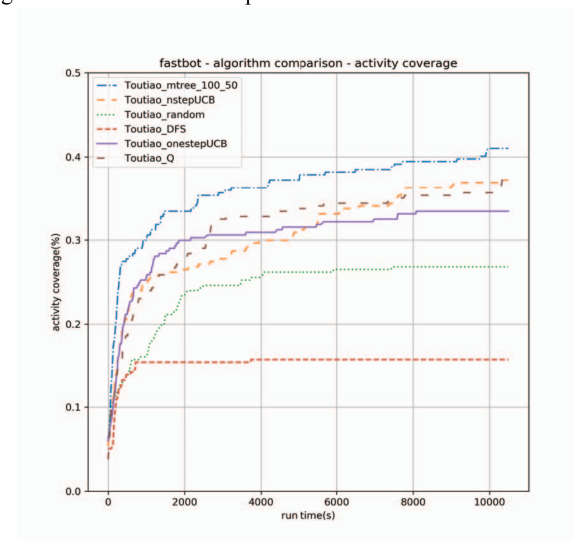

Figure 6: Algorithm Coverage Comparison

REFERENCES

[1] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). Association for Computing Machinery, New York, NY, USA, 94–105.

[2] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). Association for Computing Machinery, New York, NY, USA, 224–234.

[3] K. Jamrozik and A. Zeller, "DroidMate: A Robust and Extensible Test Generator for Android," *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, Austin, TX, 2016, pp. 293-294.

[4] Yuanchun Li, Ziyue Yang, Yao Guo and Xiangqun Chen, "DroidBot: a lightweight UI-Guided test input generator for android," *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, 2017, pp. 23-26.

[5] Li, Yuanchun, et al. "A Deep Learning based Approach to Automated Android App Testing." *arXiv preprint arXiv:1901.02633* (2019).

[6] Carpentier, Alexandra & Lazaric, Alessandro & Ghavamzadeh, Mohammad & Munos, Remi & Auer, Peter. (2011). Upper-Confidence-Bound Algorithms for Active Learning in Multi-Armed Bandits. 6925. 189-203. 10.1007/978-3-642-24412-4_17.

[7] Silver, D., Huang, A., Maddison, C. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529,** 484–489 (2016).