

Aceleração de preenchimento da tabela Q em ferramenta de automação de teste de dispositivos Android baseada em Aprendizado por Reforço

Kellem Sales
iComp Tech
UFAM
Manaus, Brasil
kellem-sales@ufam.edu.br

Isaac Siqueira Lopes
iComp Tech
UFAM
Manaus, Brasil
isaac.iartes@gmail.com

Renato Lopes de Carvalho
iComp Tech
UFAM
Manaus, Brasil
renatolopes.2278@gmail.com

Marcele Silva das Chagas
iComp Tech
UFAM
Manaus, Brasil
marcelesilvacn@gmail.com

Eliane Collins
Instituto de Desenvolvimento Tecnológico
INDT
Manaus, Brasil
eliane.collins@indt.org.br

José Reginaldo H. Carvalho
iComp Tech
UFAM
Manaus, Brasil
reginaldo@icomp.ufam.edu.br

Abstract—Este artigo apresenta um estudo sobre aceleração do preenchimento da tabela Q através da paralelização da ferramenta em estado da arte DroidbotX e da utilização de múltiplas instâncias, visando avaliar o seu desempenho. No contexto do mercado de aplicações móveis, a qualidade do software é um aspecto crítico que afeta diretamente a satisfação do cliente e a reputação da empresa. Nesse cenário, o teste de software é essencial para evitar que falhas graves ocorram nas mãos dos usuários finais, tornando o processo de teste um fator fundamental de sucesso, pois evitar gargalos é importante. As pesquisas com geração automática de casos de teste vêm ganhando destaque, principalmente para evitar o gargalo na fase de criação de casos de teste, que dependendo da complexidade da aplicação, precisa de esforço. Tecnologias de IA como o Aprendizado de Máquina por Reforço têm sido pesquisadas e estão apresentando potencial de sucesso nessa tarefa. Assim, neste estudo, foi feito um experimento comparativo com a ferramenta de geração de casos de teste que foi executada em quatro emuladores simultaneamente, alimentando a mesma tabela Q, e comparados os resultados com a execução em apenas um emulador. Ao longo de um período de duas horas, em dez aplicações Android, extraímos a taxa de cobertura de código para avaliar o desempenho dos diferentes cenários. Os resultados revelaram uma tendência de aumento na taxa de cobertura de código em algumas métricas ao utilizar as quatro instâncias de emuladores ao invés de uma. Espera-se que esse resultados possam contribuir para novas pesquisas e soluções na área.

Index Terms—Teste de software, Qualidade de software, Aprendizado de máquina por reforço, Validação do Sistema

I. INTRODUÇÃO

O desenvolvimento de software é uma área em constante evolução, impulsionada pela crescente demanda do mercado por aplicações de alta qualidade e confiabilidade [15]. Nesse contexto, a garantia da qualidade de software desempenha um papel crucial, buscando identificar e corrigir possíveis defeitos nos sistemas desenvolvidos [18]. A realização de testes é uma

das principais estratégias para atingir esse objetivo, permitindo a validação e verificação do comportamento das aplicações sob diversas condições [4].

Para O'Reilly [14], o teste de software para dispositivos Android é uma tarefa complexa e desafiadora. Os dispositivos Android são altamente personalizáveis e possuem uma ampla variedade de configurações e configurações. Isso torna difícil testar todos os possíveis cenários de uso e identificar todos os possíveis defeitos. Além disso, o mercado de aplicativos Android está em constante evolução, com novos lançamentos de tecnologias embarcadas. Isso significa que os profissionais de qualidade precisam estar constantemente atualizados sobre os novos recursos e funcionalidades dos aplicativos, para que falhas não ocorram nas mãos do usuário final.

Nesse sentido, a geração automática de teste, que é uma abordagem que usa ferramentas e técnicas para gerar automaticamente testes de software, pode ajudar a melhorar a eficiência e a precisão dos testes, possibilitando que os profissionais da área se concentrem em tarefas mais complexas [3]. Pesquisas como a de Wang et al. [21], mostram que o Aprendizado por Reforço (RL), técnica de aprendizado de máquina, pode ser um aliado na geração automática de testes ao aprender o comportamento de uma aplicação e gerar testes que são mais propensos a encontrar falhas.

Com base nesse contexto, o presente artigo tem por objetivo apresentar uma experiência prática e comparativa, para avaliar a possibilidade de acelerar o preenchimento da tabela Q, técnica de RL, na ferramenta DroidBotX, usada para gerar casos de teste em aplicativos Android usando Q-Learning, através da utilização de múltiplas instâncias paralelizadas preenchendo a mesma tabela Q. Este cenário será comparado à execução do próprio DroidbotX com uma instância e do droidbot, outra ferramenta em estado da arte que não utiliza

aprendizado por reforço e é model-based.

Espera-se por fim que os resultados desse estudo comparativo possam contribuir com as pesquisas de geração automática de testes que utilizam RL através da tabela Q.

Para isto, este artigo está organizado da seguinte forma: a Seção 2, apresenta a fundamentação teórica sobre a geração de casos de teste, o Aprendizado por Reforço (RL), a tabela Q e o algoritmo Q-learning, além de detalhar o framework "DroidbotX" e outras ferramentas utilizadas.

Na Seção 3, detalhamos a solução proposta, incluindo a formulação do problema, a proposta de solução e a implementação do experimento. A Seção 4 descreve o experimento realizado com aplicações Android, cenários de teste e métricas. Os resultados são discutidos na Seção 5, comparando com o cenário de um único emulador. As limitações e ameaças à validade do experimento são abordadas na Seção 6. Por fim, a Seção 7 apresenta as conclusões, destacando as contribuições e possíveis trabalhos futuros para contribuição na qualidade do teste de software.

II. FUNDAMENTAÇÃO

Nesta seção, serão apresentados os conceitos fundamentais relacionados à geração de casos de teste, ao Aprendizado por Reforço, à tabela Q e ao algoritmo Q-learning. Além disso, será discutida a utilização do framework "DroidbotX" e outras ferramentas empregadas no experimento.

A. Geração de teste

Segundo Fernandes e Abreu [8], a geração de testes desempenha um papel crucial na garantia da qualidade do software, permitindo a identificação de defeitos e vulnerabilidades antes que os produtos sejam disponibilizados aos usuários finais. Gupta e Mohapatra [10] afirmam que, tradicionalmente, a criação de casos de teste tem sido um processo manual, demandando esforços por parte dos engenheiros de teste. No entanto, com o advento das tecnologias de Inteligência Artificial (IA) e Aprendizado de Máquina (AM), a geração automática de testes tem emergido como um campo promissor de pesquisa, visando agilizar e aprimorar significativamente esse processo crítico.

Nas abordagens tradicionais, a geração de testes é frequentemente um processo manual que envolve a elaboração de cenários de teste, casos de uso e scripts de teste [13]. Essa abordagem, embora eficaz em muitos cenários, pode ser limitada em sua escalabilidade e na capacidade de cobrir todas as possíveis situações de teste, especialmente em sistemas complexos que demandam esforços repetitivos [20].

A pesquisa em geração automática de casos de teste busca superar as limitações das abordagens manuais, utilizando técnicas de IA para criar automaticamente casos de teste eficazes e abrangentes. Essa abordagem visa não apenas reduzir o esforço manual, mas também melhorar a cobertura de testes, identificando potenciais falhas que poderiam passar despercebidas [8].

Diversas técnicas têm sido exploradas nesse contexto, incluindo a geração de testes baseada em modelos, onde modelos

formais da aplicação são usados para gerar automaticamente casos de teste, e o uso de técnicas de busca heurística para explorar diferentes caminhos de execução do software [10].

Outra técnica baseia-se em algoritmos genéticos que trabalham iterativamente, a fim de otimizar uma população de soluções candidatas, selecionando as mais bem-sucedidas de cada geração e as combinando para criar novas soluções candidatas. Este processo é repetido até que uma solução satisfatória seja encontrada [9].

A busca aleatória é também é outra técnica para encontrar soluções para problemas complexos. Ela envolve gerar aleatoriamente uma série de soluções candidatas e avaliar cada solução de acordo com algum critério. A melhor solução encontrada é então mantida. Este método é frequentemente usado em conjunto com outras técnicas, como os algoritmos genéticos, para melhorar a eficiência da busca por soluções [17].

Zhang e Jiao [12] afirmam que nos últimos anos, técnicas de IA, como o Aprendizado de Máquina por Reforço (RL), têm sido aplicadas em pesquisas na geração automática de testes. O RL permite que um agente de teste aprenda a interagir com o ambiente, neste caso o software, de maneira a maximizar uma recompensa e caso de uma ação correta, no cenário de software pode ser geralmente definida em termos de cobertura de código ou detecção de falhas.

B. Aprendizado por reforço, Tabela Q e algoritmo Q-learning.

O Aprendizado por Reforço (Reinforcement Learning - RL) é uma técnica de aprendizado de máquina que treina um agente para aprender a tomar decisões em um ambiente incerto por meio do recebimento de reforço positivo ou negativo para cada ação tomada ao longo do tempo. De acordo com Brown e Lee [5], o RL é amplamente utilizado em diversas áreas, incluindo a geração de casos de teste.

Conforme descrito no livro Aprendizado por Reforço: uma introdução [19], a tabela Q, também conhecida como função Q, é uma estrutura de dados que armazena valores que representam a qualidade das ações em diferentes estados de um ambiente. Esses valores são atualizados iterativamente com base nas recompensas obtidas pelo agente durante a exploração do ambiente, permitindo ao agente aprender a tomar decisões inteligentes para maximizar recompensas ao longo do tempo. O algoritmo Q-Learning é a técnica que utiliza essa tabela Q para treinar o agente.

A tabela Q é atualizada para refletir a expectativa de recompensa futura que o agente pode obter se tomar essa ação naquele estado. A partir de um processo de exploração e exploração, onde o agente escolhe ações, observa recompensas e atualiza os valores Q de acordo com uma fórmula específica. Com o tempo, o Q-Learning permite que o agente aprenda a tomar decisões sequenciais em ambientes complexos, tornando-se uma ferramenta essencial em diversas aplicações, desde jogos de computador até automação e robótica [?].

Existem pesquisas que apontam o potencial do RL para gerar casos de teste para aplicativos Android, como o trabalho

de Deng et al. [7], que propuseram uma abordagem baseada em RL para gerar casos de teste para aplicativos Android.

Em outro estudo [6] foi utilizada a técnica de aprendizado por reforço profundo, com o algoritmo Deep Q-Network para gerar casos de teste, trazendo resultados promissores sendo mais efetiva, nesse caso, para navegação e em executar operações nas aplicações, gerando testes úteis e efetivos que exercitam variações de entrada de dados que facilitam a cobertura de funcionalidades de aplicações móveis e com maior possibilidades de identificar a presença de erros ou falhas.

C. DroidbotX: Framework de automação de testes para Android

Em 2021, Yasin et al [22] apresentaram o DroidBotX, um framework de automação de geração de testes desenvolvido para aplicativos Android. Ele oferece uma abordagem permitindo que os desenvolvedores testem seus aplicativos de forma mais eficiente. O DroidBotX fornece recursos como simulação de interações do usuário, captura de tela e monitoramento de cobertura de código

Seu precursor foi o DroidBot, que segundo Yang [11] é um gerador de entrada de teste (inputs), que utiliza model-based, guiado pela interface do usuário. Ele é capaz de interagir com um aplicativo Android em praticamente qualquer dispositivo. A principal técnica por trás do DroidBot é a geração de entradas de teste guiadas pela interface do usuário com base em um modelo de transição de estado gerado em tempo real, permitindo que os usuários integrem suas próprias estratégias ou algoritmos.

Portanto, tanto o DroidbotX quanto o droidbot, conseguem gerar dados de entrada (input) sobre cada elemento clicável encontrado em uma Activity da aplicação que está sendo testada. Cada teste criado e executado e as Activities por onde percorre, consistem nas saídas (outputs) da ferramenta, que disponibiliza o logcat da manipulação da aplicação e um mapa de interação, que contém todas as Activities percorridas durante a utilização da ferramenta [11] [22].

No contexto do DroidBotX, conforme figura abaixo, o algoritmo Q-learning é utilizado para aprender a melhor sequência de eventos (ações) a serem executados em um aplicativo Android (estado), com o objetivo de maximizar a eficiência dos testes. Durante a fase de exploração do aplicativo, o DroidBotX gera aleatoriamente eventos de toque e entrada, coleta as recompensas obtidas e atualiza os valores na função Q com base nessas recompensas [11].

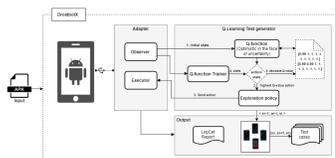


Fig. 1. Funcionamento DroidbotX [22].

III. ABORDAGEM PROPOSTA

Nesta seção, detalharemos a solução proposta, incluindo a formulação do problema, a proposta de solução e a descrição da implementação do experimento. Serão enfatizadas as instâncias de emuladores e o processo de preenchimento da tabela Q utilizando múltiplas instâncias.

A. Formulação do Problema

O problema abordado neste artigo é a otimização do processo de geração de casos de teste para aplicativos Android utilizando técnicas de Aprendizado por Reforço. O objetivo é aumentar a eficiência da geração automática de casos de teste, elevando a taxa de cobertura de teste alcançada pelos casos gerados.

B. Proposta de Solução

A solução proposta consiste em utilizar múltiplas instâncias de emuladores de dispositivos Android em conjunto com o framework DroidbotX, visando aprimorar a eficiência da geração de casos de teste. Ao empregar o algoritmo Q-learning, as instâncias de emuladores executam simultaneamente diferentes trajetórias de teste, permitindo a exploração de várias sequências de eventos e cenários de interação com o aplicativo em teste. A figura abaixo ilustra a proposta.

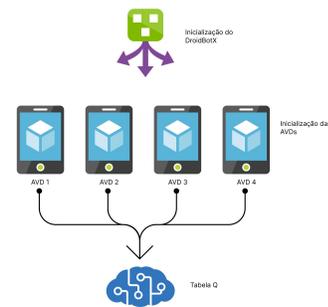


Fig. 2. Proposta de Solução.

Com o intuito de otimizar o preenchimento da tabela Q, cada instância de emulador interage com o aplicativo de forma independente, atualizando suas respectivas tabelas Q. Ao longo do processo de aprendizado, espera-se que as instâncias compartilhem informações, colaborando para uma aprendizagem mais rápida e abrangente. Essa abordagem busca explorar de forma mais eficiente o espaço de estados do aplicativo, aumentando a probabilidade de alcançar estados não visitados e, conseqüentemente, melhorando a cobertura de código.

C. Implementação do Experimento

Para a implementação deste experimento, foi selecionado um conjunto de dez aplicações Android instrumentadas, obtidas através do estudo [6] representando uma variedade de cenários e características. Cada execução do experimento teve a duração de duas horas, sendo repetida quatro vezes para obter um valor médio de taxa de cobertura. A ferramenta Java Code Coverage - JaCoCo [2] foi utilizada para medir a cobertura

de código alcançada pelos casos de teste gerados em cada execução.

Para a execução dessas rotinas, foi criado um script cuja descrição está na Figura 5, onde começa definindo os caminhos para os diretórios de aplicativos, saída e resultados de cobertura local. Em seguida, é definida uma lista de portas de emuladores disponíveis. O algoritmo itera sobre cada APK encontrado no diretório `caminho_apps` e, para cada APK, itera sobre cada emulador disponível. Dentro desse loop, para cada APK, é reiniciada a conexão com os emuladores via Android Debug Bridge (ADB) [1], reiniciados emuladores, instalada a APK instrumentada no emulador com permissões necessárias, para que então possa executar o DroidBotX no emulador por duas horas, fazendo a cópia dos arquivos de cobertura do emulador para o diretório `diretorio_resultados_local`. Esse processo é repetido para cada APK encontrado e emulador disponível, permitindo que o DroidBotX teste os aplicativos em vários emuladores para obter informações sobre a cobertura de testes.

```

1 função principal()
2 caminho_apps = "caminho/para/apps"
3 caminho_saida = "caminho/para/output/Multi"
4 caminho_resultado_local = "caminho/para/resultats_cov/all_coverage"
5
6 emuladores = {5554, 5556, 5558, 5560}
7
8 para cada apk faça
9
10     para cada emulador faça
11
12         matar_e_iniciar_adb()
13         iniciar_emuladores()
14         instalar_apk_emuladores_com_permissoesapk_emulador()
15         executar_droidbotx_emuladores_por_2h()
16         copiar_arquivo_cobertura_emulador(diretorio_resultados_local)
17
18     fim para
19
20 fim função

```

Fig. 3. Algoritmo de execução do teste.

Após a rodada de execuções, os dados do arquivo criado `coverage.ec` foram analisados através da utilização do JaCoCo. Cada aplicação foi recompilada no Android Studio e após a configuração do JaCoCo, foram obtidos os relatórios de cobertura de código de cada uma das quatro execuções para obtenção da média.

Pretende-se observar se o uso de múltiplas instâncias de emuladores possibilita a aceleração do aprendizado e melhoria de desempenho da ferramenta de geração de casos de teste em relação à execução com somente um emulador. Espera-se que essa abordagem permita explorar cenários mais diversos, o que pode resultar em um aumento na cobertura de código alcançada.

A cobertura de código, segundo Pressman [16], é uma técnica de teste de software que mede o quão bem um conjunto de testes cobre o código-fonte de um programa. É uma ferramenta valiosa para identificar possíveis caminhos de execução que não são cobertos pelos testes, o que pode levar a erros. Sendo assim utilizada para medir a eficácia dos testes e garantir que eles estão cobrindo todo o código [16].

IV. EXPERIMENTO

Será apresentado o experimento realizado, com informações sobre as aplicações Android utilizadas, os cenários de teste e a configuração das instâncias de emuladores. Detalhes sobre a coleta de dados e as métricas utilizadas também serão abordados nesta seção.

A. Aplicações Android e Cenários de Teste

Selecionamos 10 aplicativos que foram retirados do repositório de código aberto F-droid descritos na Tabela I e cujos dados foram retirados do estudo [6]. Esses aplicativos foram selecionados por terem uma diversidade de elementos GUI e interação como Editar Textos, botões, botões de imagem, links, caixas de seleção, botões de opção, giradores, seletores de data, opções menu, menu pop-up e seleção de lista, e são de código aberto. Todos os aplicativos foram instrumentados por o bytecode utilizando a ferramenta JaCoCo. Para cada ferramenta comparada foi realizado um ciclo de teste de duas horas.

TABLE I
INFORMAÇÕES DAS APLICAÇÕES SELECIONADAS

Id	App and Version	Instruc- tions	Branches	Lines	Meth- ods
App1	Add Loan v.1.0.1	9,376	494	1,974	327
App2	Dailypill v.1.0	2,006	150	655	137
App3	Exceer v.0.2.3	4,957	360	1,024	201
App4	Farmer diary v.1.72	8,159	868	2,036	260
App5	Just do v.2.1	13,382	198	853	94
App6	Money track v.2.1.3	36,297	907	5,275	1,026
App7	Open workout v.1.3.1	23,533	1,496	9,475	1,776
App8	Periodical v.1.64	28,715	667	3,253	362
App9	Todo list v.1.1	757	34	275	55
App10	Tricky v.1.6.2	35,302	2,414	14,485	2,914

B. Configuração das Instâncias de Emuladores

Optou-se para execução o uso de emuladores virtuais, sendo assim, utilizamos quatro instâncias idênticas de emuladores de dispositivos Android, todos configurados com a mesma especificação técnica. Os emuladores foram criados com a versão 11 do Android R, devido a maior compatibilidade com as aplicações selecionadas, sendo definida ainda a API Level 30 e Pixel 2 no Android Studio, cuja versão foi a Flamingo de 2022.2.1 disponível em Google [1].

Foi utilizado um computador com processador intel i7 de 13a geração, 64GB de RAM e uma placa de vídeo NVIDIA RTX 3070-TI, e com o sistema operacional Ubuntu 22.04

C. Coleta de Dados e Métricas de Avaliação

O experimento foi baseado nos estudos da área, como o [6], e consistiu em executar a ferramenta de teste por duas horas e obter o arquivo `coverage.ec` a cada 10min de execução, para que ao final das duas horas fosse possível gerar o relatório de cobertura de código via JaCoCo. Esse ciclo de testes foi repetido quatro vezes por cada aplicação para confirmar a tendência.

As métricas de avaliação da taxa de cobertura de teste, foram as fornecidas pelos relatório do JaCoCo [2], sendo elas as métricas Instruction Coverage, que indica a quantidade de código-fonte que foi exercitada pelos testes, Branch Coverage, que mede a porcentagem de ramos condicionais, Line Coverage, que mede a porcentagem de linhas de código que foram executadas pelos testes e Method Coverage, que mede a porcentagem de métodos (funções) do código que foram executados pelos testes.

Após a conclusão do experimento, os resultados foram tabulados e analisados comparativamente com a execução em um único emulador, para avaliar o desempenho e a eficácia da abordagem de múltiplas instâncias de emuladores.

V. RESULTADOS

Nesta seção, apresentaremos os resultados obtidos a partir do experimento realizado com a utilização do DroidbotX Multi (DBX-MULT) em comparação com o Droidbot (DB) e DroidbotX com uma instância. Os dados coletados das métricas de Instruction Coverage, Branch Coverage, Line Coverage e Method Coverage para cada aplicativo estão apresentados nas tabelas abaixo:

TABLE II
MÉTRICAS DE COBERTURA DE INSTRUÇÃO (INSTRUCTION COVERAGE)

APPS	DB	DBX	DBX-MULT
App1	0,14	0,21	0,69
App2	0,75	0,76	0,78
App3	0,42	0,54	0,52
App4	0,34	0,03	0,04
App5	0,46	0,46	0,10
App6	0,33	0,36	0,35
App7	0,25	0,14	0,36
App8	0,56	0,53	0,57
App9	0,34	0,45	0,54
App10	0,17	0,24	0,43

TABLE III
MÉTRICAS DE COBERTURA DE RAMIFICAÇÃO (BRANCH COVERAGE)

APPS	DB	DBX	DBX-MULT
App1	0,05	0,09	0,47
App2	0,42	0,45	0,53
App3	0,30	0,42	0,41
App4	0,23	0,01	0,01
App5	0,39	0,41	0,01
App6	0,15	0,20	0,21
App7	0,12	0,06	0,19
App8	0,49	0,40	0,50
App9	0,14	0,29	0,42
App10	0,08	0,12	0,26

TABLE IV
MÉTRICAS DE COBERTURA DE LINHA (LINE COVERAGE)

APPS	DB	DBX	DBX-MULT
App1	0,14	0,22	0,68
App2	0,78	0,79	0,81
App3	0,46	0,52	0,56
App4	0,35	0,03	0,04
App5	0,52	0,52	0,09
App6	0,28	0,33	0,35
App7	0,27	0,14	0,37
App8	0,59	0,51	0,61
App9	0,30	0,46	0,58
App10	0,19	0,25	0,44

A análise dos resultados revelou que a abordagem DBX-MULT, ao empregar múltiplas instâncias de emuladores de dispositivos Android, apresentou potencial para superar o desempenho da abordagem DBX com um único emulador em algumas métricas de cobertura de código.

TABLE V
MÉTRICAS DE COBERTURA DE MÉTODO (METHOD COVERAGE)

APPS	DB	DBX	DBX-MULT
App1	0,2	0,31	0,79
App2	0,82	0,83	0,83
App3	0,61	0,71	0,69
App4	0,45	0,06	0,07
App5	0,53	0,68	0,35
App6	0,33	0,38	0,44
App7	0,33	0,16	0,40
App8	0,53	0,44	0,56
App9	0,45	0,59	0,70
App10	0,23	0,30	0,52

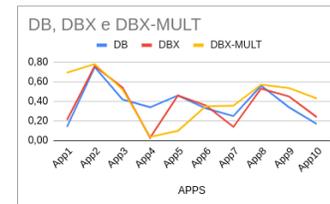


Fig. 4. Instruction Coverage.

Na figura 6, vemos que o DBX-MULT apresentou resultado inferior em 2 apps e chegou a alcançar uma cobertura de até 0,79%, enquanto o DBX obteve valores, variando de 0,03% a 0,76%, e o DB obteve cobertura de 0,14% a 0,75%.

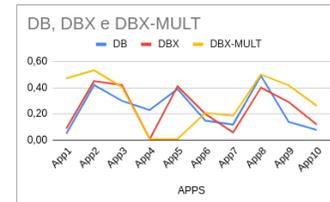


Fig. 5. Branch Coverage.

Na métrica de Branch Coverage, figura 7, o DBX-MULT atingiu até 0,53% de cobertura, enquanto o DBX variou entre 0,01% e 0,50% e o DB obteve cobertura de 0,05% a 0,49%.

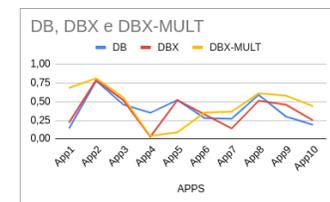


Fig. 6. Line Coverage.

A análise da cobertura de linhas de código (Line Coverage) mostrada na figura8 também indica o favorecimento do DBX-MULT, com cobertura variando de 0,09% a 0,81%, enquanto o DBX alcançou cobertura entre 0,14% e 0,79%, e o DB obteve cobertura de 0,22% a 0,78%.

Quanto à métrica de Method Coverage o DBX-MULT apresentou uma cobertura variando de 0,07% a 0,70%, em

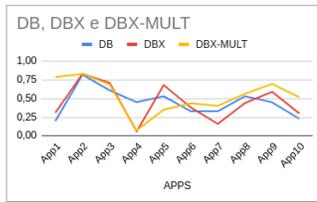


Fig. 7. Method Coverage.

contraste com o DBX, que obteve cobertura entre 0,2% e 0,83%, e o DB, que obteve cobertura de 0,2% a 0,82%, ficando abaixo portanto da abordagem com um emulador e do droidbot.

Esses resultados sugerem que o uso de múltiplas instâncias de emuladores pode proporcionar uma melhoria na cobertura de código e no alcance de estados relevantes das aplicações Android durante a geração de casos de teste, ficando abaixo na métrica de Method Coverage. Essa análise comparativa destaca a relevância da abordagem DBX-MULT como uma contribuição para a pesquisa na área, observando-se que a oscilação de desempenho necessita de mais exploração para melhor compreensão.

VI. AMEAÇAS À VALIDADE DO EXPERIMENTO

Identifica-se como ameaça externa o número de aplicações utilizadas no experimento. Para mitigar essa ameaça foram escolhidos 10 aplicativos Android com diferentes características de complexidade, comportamento e interface de usuário.

Sobre as ameaças à validade interna, notou-se a configuração dos computadores e dos aparelhos que para amenizar esse fator, foram apagados e refeitos a cada novo experimento para não comprometer os resultados.

Por fim, quanto à ameaça à validade de construção, foi definida a métrica de cobertura de código para avaliar o desempenho das ferramentas de geração de teste utilizadas no estudo, em consonância com o indicado na literatura técnica da área.

VII. CONCLUSÃO E TRABALHOS FUTUROS

O presente trabalho analisou a abordagem DroidbotX Multi (DBX-MULT), que ao invés de utilizar uma única instância de emulador a cada execução, utilizou múltiplas instâncias de emuladores de dispositivos Android, com o objetivo de verificar se seria possível acelerar o preenchimento da tabela Q nesta ferramenta de geração de casos de teste baseada em Aprendizado por Reforço.

Comparativamente, os dados da abordagem com várias instâncias sugerem que houve um potencial de aumento de desempenho da abordagem convencional, em algumas aplicações em relação à utilização de um único emulador. Apesar do cenário promissor em alguns casos, é importante notar que a magnitude dessa melhoria varia consideravelmente entre os diferentes aplicativos testados.

Assim, sugere-se que futuros estudos explorem a escalabilidade da abordagem DBX-MULT em cenários de maiores

proporções. Essas investigações podem contribuir para o avanço contínuo da área e inspirar o desenvolvimento de novas soluções que impulsionem a eficiência e qualidade do teste de software em aplicações embarcadas como no caso apps Android.

Ficará como contribuição deste trabalho para a área de pesquisa, os padrões de configuração da solução através deste artigo, os relatórios do experimento e disponibilização desses dados para reprodução no repositório <https://anonymous.4open.science/r/DroidbotX-35AB/README.md> e com isso espera-se que esses achados contribuam para pesquisa e inspirem novas soluções que impulsionem a avaliação de aplicações ambarcadas no Android.

REFERENCES

- [1] Android studio, 2023.
- [2] Jacoco, 2023.
- [3] A. Almukhlifi and P. Sarro. A survey on automatic test case generation for mobile applications. In *2018 IEEE 26th international conference on software analysis, evolution, and reengineering (SANER)*, pages 591–600. IEEE, 2018.
- [4] B. W. Boehm. *Aspiral: uma abordagem incremental evolucionária para o desenvolvimento de software*. *IEEE Computer*, 21(12):61–72, 1988.
- [5] M. Brown and S. Lee. Automated test case generation using reinforcement learning. In *Proceedings of the International Conference on Software Testing*, pages 56–67, 2021.
- [6] Eliane Figueiredo Collins. *DeepRLGUMAT: A Deep Reinforcement Learning-based Approach for GUI Mobile Application Testing*. PhD thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, 2022.
- [7] S. Deng, L. Zhang, and H. Li. Automated test case generation using reinforcement learning. In *Proceedings of the 2019 International Conference on Software Engineering*, pages 56–67, 2019.
- [8] José Fernandes and Rui Abreu. Automated test generation: A literature review. *Journal of Software Testing*, 1(1):1–10, 2018.
- [9] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Publishing Company, 1989.
- [10] Rakesh Gupta and Deb Mohapatra. Automated software testing: A survey. *ACM Computing Surveys (CSUR)*, 48(4):1–41, 2016.
- [11] Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: a lightweight ui-guided test input generator for android. *ICSE (Companion Volume)*, pages 23–26, 2017.
- [12] Hai Ma, Xiaoping Zhang, and Lei Jiao. A survey on automated software testing using reinforcement learning. *IEEE Transactions on Software Engineering*, 43(12):1353–1375, 2017.
- [13] João Oliveira and Rui Abreu. A survey on automated test generation using machine learning. *arXiv preprint arXiv:1904.02103*, 2019.
- [14] O'Reilly. *Android App Testing in Practice*. O'Reilly Media, 2020.
- [15] R. S. Pressman. *Engenharia de software: uma abordagem profissional*. McGraw-Hill Education, 2011.
- [16] Roger S. Pressman. *Software testing: a practical approach*. McGraw Hill Education, 2011.
- [17] Herbert Robbins. Asymptotically efficient solutions of the robbins-monro problem. *The Annals of Mathematical Statistics*, 1951.
- [18] I. Sommerville. *Engenharia de software (9ª ed.)*. Addison-Wesley Professional, 2011.
- [19] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*. MIT press, 2018.
- [20] Vijayan Vasudevan and Bala Ramesh. Automated test generation: A survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(4):1–62, 2016.
- [21] Y. Wang, W. Wu, Y. Liu, and J. Liu. Reinforcement learning-based automatic test case generation for web applications. In *2019 IEEE 33rd international conference on software engineering (ICSE)*, pages 1410–1421. IEEE, 2019.
- [22] H. N. Yasin, S. H. A. Hamid, and R. J. R. Yusof. Droidbotx: Test case generation tool for android applications using q-learning. *Symmetry*, 13(2):310, 2021.