UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

REGINA LETÍCIA SANTOS FELIPE

OTIMIZANDO TESTES DE GUI COM ITERATIVE DEEPENING URL-BASED SEARCH

CAMPINA GRANDE - PB

2024

REGINA LETÍCIA SANTOS FELIPE

OTIMIZANDO TESTES DE GUI COM ITERATIVE DEEPENING URL-BASED SEARCH

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Everton L. G. Alves

CAMPINA GRANDE - PB

2024

REGINA LETÍCIA SANTOS FELIPE

OTIMIZANDO TESTES DE GUI COM ITERATIVE DEEPENING URL-BASED SEARCH

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Everton L. G. Alves

Orientador – UASC/CEEI/UFCG

Patrícia Duarte de Lima Machado

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 16 de Maio de 2024

CAMPINA GRANDE - PB

RESUMO

Testes automáticos em aplicações web são amplamente adotados devido à sua eficiência e relação custo-benefício. Quando realizados pela GUI, esses testes podem simular cenários de uso, com o objetivo de expor falhas visíveis. O scriptless testing, uma abordagem que gera e executa casos de teste automaticamente, pode adotar uma exploração sistemática da GUI. Entretanto, essa abordagem pode demandar um tempo significativo em aplicações web, principalmente numa execução com repetição em visitas de estados. Neste estudo, apresentamos o algoritmo Iterative Deepening URL-Based Search (IDUBS), que se associado ao uso de testes de GUI, resulta em uma exploração otimizada, utilizada para a redução da redundância em visita de estados. Avaliamos a eficácia do algoritmo em um estudo empírico com quatro sistemas web de código aberto, analisando sua contribuição para otimização do tempo de execução e redução da redundância em visita de estados. O IDUBS reduziu o tempo de execução em 39,03% e a redundância de casos de teste em 36,01%. Em suma, o algoritmo IDUBS apresenta uma solução promissora para otimizar a execução de testes em aplicações web, oferecendo benefícios significativos em termos de eficiência e redução de redundâncias.

OPTIMIZING GUI TESTING WITH ITERATIVE DEEPENING URL-BASED SEARCH

ABSTRACT

Automatic testing in web applications is widely adopted because of its efficiency and cost-effectiveness. GUI-based tests can simulate usage scenarios to uncover visible faults. Scriptless testing, which automatically generates and executes test cases, can systematically explore the GUI. However, this approach may require significant time in web applications, particularly during repeated visits to states. In this study, we introduce the Iterative Deepening URL-Based Search algorithm that, when combined with GUI testing, optimizes exploration by reducing redundancy in state visitation. Our empirical study evaluated the algorithm's effectiveness with four open-source web systems and found that IDUBS reduced runtime by 39.03% and test case redundancy by 36.01%. Overall, the IDUBS algorithm offers a promising solution for optimizing test execution in web applications while significantly reducing redundancy.

Otimizando Testes de GUI com Iterative Deepening URL-Based Search

Regina Letícia Santos Felipe
Universidade Federal de Campina Grande
Campina Grande Brasil
regina.felipe@ccc.ufcg.edu.br

Everton L. G. Alves
Universidade Federal de Campina Grande
Campina Grande Brasil
everton@computacao.ufcg.edu.br

Thiago Santos de Moura
Universidade Federal de Campina Grande
Campina Grande Brasil
thiago.moura@copin.ufcg.edu.br

RESUMO

Testes automáticos em aplicações web são amplamente adotados devido à sua eficiência e relação custo-benefício. Quando realizados pela GUI, esses testes podem simular cenários de uso, com o objetivo de expor falhas visíveis. O scriptless testing, uma abordagem que gera e executa casos de teste automaticamente, pode adotar uma exploração sistemática da GUI. Entretanto, essa abordagem pode demandar um tempo significativo em aplicações web, principalmente numa execução com repetição em visitas de estados. Neste estudo, apresentamos o algoritmo Iterative Deepening URL-Based Search (IDUBS), que se associado ao uso de testes de GUI, resulta em uma exploração otimizada, utilizado para a redução da redundância em visita de estados. Avaliamos a eficácia do algoritmo em um estudo empírico com quatro sistemas web de código aberto, analisando sua contribuição para otimização do tempo de execução e redução da redundância em visita de estados. O IDUBS reduziu o tempo de execução em 39,03% e a redundância de casos de teste em 36,01%. Em suma, o algoritmo IDUBS apresenta uma solução promissora para otimizar a execução de testes em aplicações web, oferecendo benefícios significativos em termos de eficiência e redução de redundâncias.

KEYWORDS

Testes de GUI, Scriptless Testing, Algoritmos de busca, Iterative Deepening Search

1 INTRODUÇÃO

As aplicações web têm ganhado popularidade, tornando-se cada vez mais complexas para proporcionar uma experiência enriquecedora ao usuário [23]. São um campo dinâmico frequentemente moldado pelas demandas em constante mudança dos clientes, que buscam software de alta qualidade em prazos curtos. Tal contexto destaca a necessidade crítica de garantir estabilidade e confiabilidade em aplicações web [16].

Mesmo os testes manuais sendo essenciais, eles frequentemente são considerados uma atividade custosa e propensa a erros. Portanto, os testadores têm migrado para estratégias automatizadas para testar aplicações web, especialmente em ambientes industriais [10]. As estratégias automatizadas de teste oferecem uma forma eficaz e repetível de testar software [13].

As análises de software podem ser conduzidas por meio de testes de GUI (*Graphical User Interface*), os quais avaliam o sistema pela interação com a perspectiva do usuário, realizando ações como cliques, rolagens e pressionamentos de teclas em elementos acionáveis. Nos testes de GUI, eventos são empregados para explorar diferentes estados do AUT (*Application Under Test*), validar funcionalidades específicas ou detectar faltas através de falhas visíveis apresentadas pelo sistema [5].

O teste de GUI automatizado envolve ferramentas capazes de automatizar uma ou mais fases do processo de teste, como a elaboração de scripts de testes, a execução desses scripts, a definição e avaliação de oráculos, ou a análise dos resultados obtidos [25]. A automatização pode occorrer de duas formas: manualmente (*Scripted Testing*) e automaticamente (*Scriptless Testing*). No *Scripted Testing* os testadores criam manualmente scripts para cada sequência de teste, com ou sem o auxílio de ferramentas de captura e reprodução [18]. Esses scripts são então executados utilizando frameworks de teste como o Selenium[1] ou Cypress[2]. Por outro lado, os *Scriptless Testing* envolvem o uso de ferramentas automáticas que geram e executam sequências de teste com base nos elementos da GUI[1].

O *Monkey Testing* é uma técnica frequente no *scriptless testing*, envolvendo a geração e execução aleatória de casos de teste. É possível aplicar heurísticas para aumentar a probabilidade de detecção de falhas [11]. No entanto, devido à sua natureza probabilística e às limitações nas ações executadas, pode ocorrer que certos elementos da GUI nunca sejam acionados [24]. Em contraste, o *Model-based Testing* e o *Systematic GUI Testing* são técnicas robustas na detecção de falhas visíveis. Essas técnicas exploram sistematicamente a GUI, examinando e acionando widgets e eventos disponíveis em cada estado da GUI durante a execução, simulando ações do usuário para identificar falhas visíveis [3, 6, 26]. No entanto, um desafio significativo associado a esse tipo de teste reside em seu elevado tempo de execução e à explosão de estados, onde o número de casos de teste potenciais cresce exponencialmente dependendo da complexidade do sistema [3], especialmente em aplicações web industriais. Isso pode resultar em várias horas entre a exploração, geração e execução dos testes.

O método de busca *Iterative Deepening Search* (IDS) é uma abordagem eficaz para a exploração sistemática da árvore de estados de uma GUI em cenários nos quais a profundidade do espaço de busca não é previamente conhecido [30]. O IDS realiza múltiplas iterações do *Depth-First Search* (DFS), aumentando progressivamente os limites de profundidade até alcançar um objetivo [28]. Esse objetivo pode consistir na identificação de um estado com falha visível ou na completude da busca exaustiva. A cada iteração, a busca é iniciada a

[1] https://selenium.dev/
[2] https://www.cypress.io/

partir do nó raiz, garantindo uma cobertura completa e estruturada do espaço de busca.

Dessa forma, o IDS é a estratégia preferida em contextos nos quais há um grande espaço de busca e a profundidade da solução é desconhecida [27], o que é especialmente relevante para os desafios enfrentados nos testes sistemáticos de GUI [3, 15]. No entanto, ao revisitar nós em cada iteração, o IDS pode gerar conjuntos de testes redundantes e onerosos. Em grafos extensos com caminhos longos até a solução, o tempo necessário para revisitar nós pode se tornar significativo, impactando o desempenho global do algoritmo [20, 21]. Além disso, uma única aplicação pode ter múltiplas falhas que se manifestam em diferentes estados, exigindo múltiplas execuções do IDS ou o trabalho com múltiplos objetivos na busca de caminhos [7, 12].

Cytestion é uma ferramenta projetada para testes de GUI automatizados em aplicações web [24]. Ela utiliza uma versão do algoritmo IDS com múltiplos objetivos para permitir a exploração sistemática da AUT. Enquanto constrói a árvore de estados da interface gráfica, Cytestion gera e executa os testes, visando identificar falhas visíveis, como mensagens de erro na GUI, status de requisições inadequados e erros apresentados no console do navegador. Além disso, durante a exploração da AUT, Cytestion fornece artefatos como conjuntos de testes de regressão, resumos detalhados de falhas e vídeos ilustrativos demonstrando problemas identificados.

As limitações associadas ao uso do IDS resultam frequentemente em conjuntos de testes redundantes e demorados no Cytestion. Por exemplo, ao explorar um sistema com um estado inicial contendo dez elementos acionáveis, o Cytestion pode descobrir dez novos estados para cada elemento explorado. Se cada um desses novos estados também incluísse dez novos elementos acionáveis, isso poderia resultar em um conjunto de testes com pelo menos 100 casos, com ações repetidas no estado inicial, o que aumenta o tempo de execução sem trazer benefícios significativos em termos de teste. Essa questão é agravada em aplicações web que possuem muitos elementos e caminhos complexos, onde a redundância se torna ainda mais evidente.

Para abordar essa questão, apresentamos o *Iterative Deepening URL-Based Search* (IDUBS), uma versão otimizada e adaptada do IDS para web, que incorpora informações de URL para reduzir caminhos redundantes. Ao utilizar informações de URL nos nós da árvore de GUIs, o IDUBS identifica novos pontos de partida e inicia casos de teste diretamente desses estados obtidos a partir da nova URL encontrada, ao invés de começar da URL inicial e navegar pelos estados. Por fim, apresentamos uma nova versão do Cytestion que emprega o algoritmo IDUBS.

Realizamos uma análise comparativa das execuções da ferramenta Cytestion com a implementação do IDS e IDUBS. Este estudo foi feito em quatro aplicações de código aberto. O objetivo foi avaliar a eficácia da otimização proporcionada através da aplicação do IDUBS comparando os tempos de execução dos casos de teste e o número de estados revisitados durante a execução. Nossas descobertas mostram que o IDUBS tem um desempenho melhor que o IDS em testes de GUI, reduzindo o tempo de execução e a redundância de estados.

As contribuições desta pesquisa compreendem:

- O algoritmo Iterative Deepening URL- Based Search (IDUBS) que lida com a redundância no algoritmo IDS em testes de GUI
- A implementação de IDUBS dentro de uma ferramenta dedicada para testes sistemáticos e automatizados de GUI;
- Um estudo empírico que compara o desempenho e a eficácia do IDUBS em comparação ao IDS.

O restante deste trabalho está estruturado da seguinte maneira: A Seção 2 apresenta os tópicos essenciais adotados em nossa pesquisa, seguida pela Seção 3, que apresenta o algoritmo e a implementação do IDUBS. A Seção 4 dedica-se à discussão dos estudos de avaliação, a Seção 5 apresenta as ameaças à validade e a Seção 6 aborda trabalhos relacionados. Por fim, na Seção 7, apresentamos nossas conclusões e delineamos possíveis direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Testes de GUI

No teste de GUI o sistema é avaliado através dos seus componentes e atributos visuais [22]. Isso implica na execução de sequências de interações do usuário, tais como cliques, rolagens e pressionamentos de teclas, nos elementos visíveis e interativos da interface, como botões e campos de entrada, em diferentes contextos da AUT [4]. A realização do teste de GUI pode ser feita manualmente por um avaliador ou através de uma ferramenta automatizada. Essas ferramentas automatizadas podem ser empregadas para diversas atividades de teste, como a criação e execução de sequências de testes, a definição e verificação de critérios de aceitação, e a análise dos resultados obtidos [25].

2.2 Frameworks

Os frameworks de automação de teste possibilitam a execução de scripts de GUI para a web [9]. Selenium é um framework de teste de código aberto para aplicações web que oferece suporte a várias linguagens de programação e permite interações com o navegador e páginas da web por meio de ações como clicar em botões e navegar por páginas. Testadores frequentemente mencionam desafios práticos com o Selenium, como problemas de infraestrutura, relatórios de resultados e limitações ao lidar com sistemas robustos e dinâmicos [19]. Cypress é um framework de teste moderno que permite interações com o navegador e páginas web através de ações como clicar em botões e navegar por páginas. Ele promete simplicidade, velocidade, confiabilidade e uma API simplificada para interação direta com o DOM [17]. O Cypress simplifica o processo de configuração ao agrupar todos os componentes necessários em um único download. Além disso, o Cypress tem uma comunidade muito ativa que contribui para a evolução do framework.

2.3 Cytestion

Cytestion é a ferramenta de testes *scriptless* selecionada para nossos estudos. Trata-se de uma ferramenta de código aberto que utiliza o Cypress. Ela adota uma abordagem progressiva e sem script para criar conjuntos de testes de GUI. Ao executar uma versão com múltiplos objetivos do algoritmo IDS (detalhes na Seção 2.4), o Cytestion começa a gerando um caso de teste inicial para descobrir elementos acionáveis dentro do estado inicial do sistema. Em seguida, novos

testes são gerados de forma iterativa ao explorar sistematicamente cada elemento acionável. Cada caso de teste executa todas as ações anteriores, começando pelo estado inicial. O objetivo dos conjuntos gerados é identificar falhas visíveis, como falhas de GUI, problemas de status de requisições e erros no console do navegador. A ferramenta não apenas gera e executa testes durante a exploração do estado, mas também produz um conjunto de artefatos, incluindo conjuntos para teste de regressão, um resumo das falhas detectadas e vídeos de replay que mostram as execuções com falhas.

2.4 Iterative Deepening Search

O *Iterative Deepening Search* (IDS), também conhecido como *Iterative Deepening Depth-First Search* (IDDFS), percorre espaços de busca baseados em grafos aumentando gradualmente o limite de profundidade a cada iteração. Essa abordagem iterativa combina a eficiência de memória do DFS com a completude do BFS. Ele começa com um limite de profundidade zero e aumenta esse limite a cada iteração até encontrar um nó objetivo ou esgotar o espaço de busca [27]. Em cada limite de profundidade, o IDS realiza um DFS com limite, explorando nós até a profundidade especificada. Se um nó objetivo não for encontrado dentro do limite de profundidade atual, o IDS aumenta o limite e realiza outra iteração do DFS.

Na sua forma original, o IDS foca em localizar um único nó objetivo dentro do espaço de busca. No entanto, em muitas aplicações do mundo real, podem haver múltiplos nós objetivos que precisam ser alcançados [8]. Integrar uma estratégia de múltiplos objetivos no IDS amplia sua aplicabilidade, permitindo que ele navegue eficientemente em direção a múltiplos objetivos dentro do espaço de busca. Essa adaptação preserva os princípios fundamentais de completude ao mesmo tempo em que aprimora a flexibilidade e escalabilidade do algoritmo ao lidar com cenários de busca complexos.

O Algoritmo 1 apresenta o IDS com a estratégia de múltiplos objetivos. Ele começa inicializando uma lista vazia chamada `goalNodes` para armazenar os nós objetivos encontrados durante a busca (linha 1). A função principal `IDS` recebe o nó raiz do grafo e o objetivo como entradas (linha 2). Ela itera sobre profundidades crescentes de zero até o infinito (linha 3). Em cada profundidade, ela chama a função `DFS` para explorar nós até essa profundidade e recebe um valor booleano indicando se pelo menos um novo nó foi encontrado, o que potencialmente permite uma exploração adicional (linha 4).

A função `DFS` explora recursivamente nós no grafo até uma profundidade especificada. Se a profundidade for zero, ela verifica se o nó atual é um nó objetivo. Se sim, o nó é adicionado à lista `goalNodes` e retorna `true` para avaliar possíveis filhos na próxima iteração (linhas 13-16). Se a profundidade for maior que zero, a função explora todos os nós filhos do nó atual recursivamente, diminuindo a profundidade a cada vez (linhas 20). A variável `anyRemaining` serve para indicar se novos nós foram encontrados durante o loop dos nós filhos nesse nível de profundidade (linhas 18). Quando permanece `false`, ela indica que nenhum novo nó foi encontrado em nenhum ramo, sinalizando o fim da busca (linhas 5-6).

2.4.1 Exemplo de Execução. Para ilustrar a execução do algoritmo IDS com múltiplos objetivos, considere uma exploração da GUI do projeto de código aberto *petclinic*. Ela é representada como uma estrutura em forma de árvore que será progressivamente construída

Algoritmo 1 Algoritmo IDS com múltiplos objetivos

```
1:  goalNodes ← []
2:  function IDS(root, goal)
3:      for depth from 0 to ∞ do
4:          remaining ← DFS(root, goal, depth)
5:          if not remaining then
6:              return goalNodes
7:          end if
8:      end for
9:  end function
10:
11: function DFS(node, goal, depth)
12:     if depth = 0 then
13:         if node is a goal then
14:             goalNodes.add(node)
15:         end if
16:         return TRUE
17:     else if depth > 0 then
18:         anyRemaining ← FALSE
19:         for all child of node.children do
20:             anyRemaining ← DFS(child, goal, depth − 1)
21:         end for
22:         return anyRemaining
23:     end if
24: end function
```

(Figura 1). Cada nó na árvore corresponde a um estado único da GUI, e as arestas representam transições entre estados acionados. O objetivo é identificar todos os estados da GUI onde ocorrem falhas (nós C e H). No entanto, no início da execução, não temos conhecimento prévio de quais estados levam a falhas.

Começamos com um limite de profundidade zero, permitindo encontrar o nó raiz A. Como A não é um nó objetivo, ou seja, não inclui uma falha visível, avançamos para o próximo nível de profundidade indicando o valor booleano `true` (linha 16). Na profundidade um, começamos novamente com o nó raiz A e exploramos recursivamente seus filhos B e C (linha 20). Encontramos o nó C, que é identificado como um nó objetivo devido a uma falha visível. Adicionamos o nó C à lista de nós objetivos e continuamos explorando o grafo.

Na profundidade dois, passamos pelos nós A, B, C novamente e então fazemos uma chamada recursiva de DFS para os filhos de B. Como nenhuma falha é encontrada mas dois filhos foram encontrados, outro valor de `remaining` é retornado. Indo para a profundidade três, encontramos novamente os nós A, B, C, D, E e prosseguimos para chamar o DFS em seus respectivos filhos E e D. Como F e G não são nós objetivos, continuamos até a profundidade quatro, onde o nó H é descoberto como um nó objetivo e incluído em `goalNodes`. Finalmente, mais uma iteração profunda é feita, e todas as chamadas de DFS retornam `false` já que os nós mais profundos não têm filhos, concluindo assim a execução do IDS e retornando os nós objetivos C e H.

Com base na execução apresentada, a suíte de testes gerada pelo IDS possui as seguintes sequências de testes: (1) A; (2) $A \rightarrow B$; (3) $A \rightarrow C$; (4) $A \rightarrow B \rightarrow D$; (5) $A \rightarrow B \rightarrow E$; (6) $A \rightarrow B \rightarrow D \rightarrow F$; (7) $A \rightarrow B \rightarrow E \rightarrow G$; (8) $A \rightarrow B \rightarrow E \rightarrow G \rightarrow H$; (9) $A \rightarrow B \rightarrow E \rightarrow G \rightarrow I$. Há uma

clara redundância no número de estados acessados, especialmente no estado inicial *A*, que é visitado em nove casos de teste. A suíte de testes completa gerada usando IDS para a aplicação petclinic pode ser encontrada em nosso repositório[3].

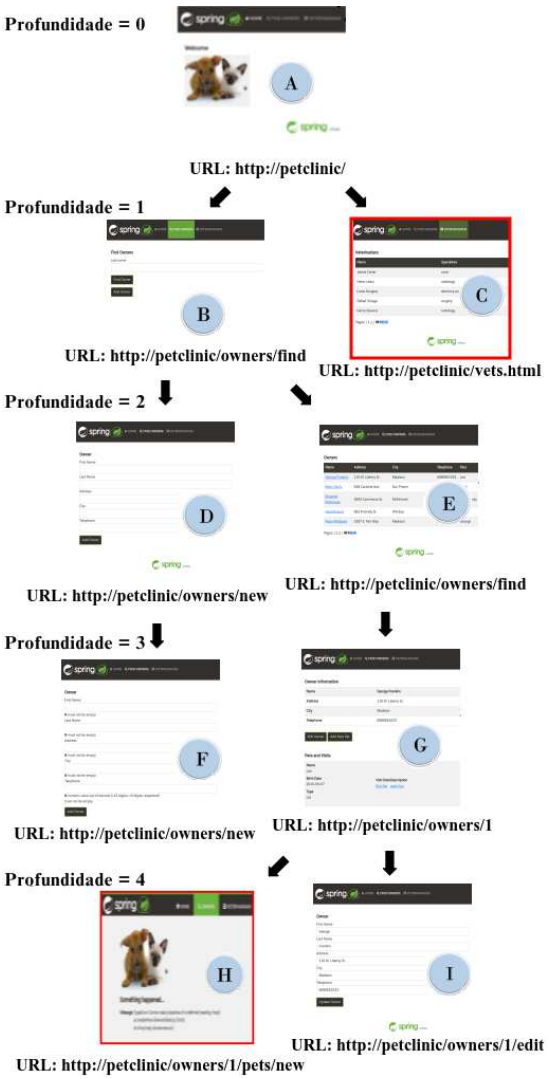

Figura 1: Exemplo da árvore de GUI na AUT.

3 ITERATIVE DEEPENING URL-BASED SEARCH

Nesta seção, vamos apresentar o algoritmo Iterative Deepening URL-Based Search (IDUBS), que se diferencia do IDS ao manter o mínimo de informações dos nós anteriores para iniciar uma nova DFS a partir de um novo ponto de partida. Essa abordagem elimina redundâncias ao retornar ao nó inicial durante iterações posteriores.

Nosso objetivo é otimizar a execução de casos de teste em testes de GUI usando o IDUBS, pois ele pode ajudar a evitar a redundância de estados de GUI acessados e, consequentemente, reduzir o tempo de execução. O algoritmo proposto associa cada nó do gráfico que representa um estado de GUI a uma URL correspondente. Ao descobrir uma nova URL, o novo nó se torna um ponto de partida. Durante a exploração, o algoritmo pode acessar o estado previamente obtido na iteração anterior diretamente por meio desse novo ponto de partida e continuar a busca nesse ramo. Essa abordagem é viável em sistemas web, pois as visitas diretas às URLs oferecem acesso eficiente a estados específicos do AUT e estão alinhadas com as práticas web modernas [29, 32].

O Algoritmo 2 apresenta o IDUBS com suporte para múltiplos estados objetivos*. Ele começa inicializando duas listas vazias: `roots`, que armazena os nós raiz, e `goalNodes`, que armazena os nós objetivos encontrados (linhas 1-2). Na função principal, `IDUBS`, são passados como argumentos o nó raiz e o estado objetivo (linha 3). O algoritmo começa definindo a profundidade inicial `depth` como zero e marcando o nó raiz com o valor de nível correspondente. Em seguida, o nó raiz é adicionado à lista `roots` (linhas 4-6). Durante cada iteração de profundidade, o algoritmo utiliza todos os nós na lista `roots` e chama a função DFS enquanto passa os parâmetros de nó, objetivo e profundidade: `node`, `goal`, e $|depth - node.level|$ (linha 9). Essa subtração garante que a busca respeite o limite de profundidade mesmo quando um nó raiz mais profundo é usado.

A função `DFS` explora de forma recursiva os nós na árvore até a profundidade especificada. Quando a profundidade é zero, indicando o fim da exploração para um ramo, ela verifica se o nó atual é um nó objetivo (linha 21). Se for, o nó é adicionado à lista `goalNodes` (linha 22). Em seguida, retorna `true` para avaliar possíveis filhos na próxima iteração (linha 24). Se a profundidade for maior que zero, a função explora todos os nós filhos do nó atual para verificar se ele está presente na lista `roots` (linhas 27-28). A presença na lista indica que este filho já foi encontrado em iterações anteriores e, portanto, tem seu próprio fluxo separado. Isso ocorre quando há uma diferença entre a URL do nó e a URL do filho (linhas 30-32), permitindo que o nó seja acessado diretamente na próxima iteração.

É importante observar que novos pontos de partida são encontrados em níveis mais profundos da árvore. Para garantir que todos os fluxos respeitem o limite de profundidade, é crucial salvar o nível de cada nó filho adicionando o nível do nó pai somado a um (linha 29). Em seguida, uma chamada de DFS é feita para o filho, que retorna um valor booleano para `anyRemaining` (linha 33). Quando esse valor permanece `false`, indica que nenhum novo nó foi encontrado neste ramo, permitindo remover o nó da lista `roots` (linhas 10-12). Eventualmente, quando nenhum novo nó é encontrado em nenhum ramo, a lista se torna vazia, encerrando a busca e resultando no retorno de `goalNodes`.

3.1 Exemplo de Execução

Para demonstrar a execução do algoritmo IDUBS com múltiplos objetivos, reutilizamos o exemplo apresentado na Seção 2.4.1 e na Figura 1. Nosso objetivo continua sendo identificar todos os estados de GUI onde ocorrem falhas. Começamos com um limite de profundidade zero, incluindo apenas o `root` na lista `roots`. Como *A* não é um nó objetivo, avançamos para o próximo nível de profundidade

[3]https://gitlab.com/lsi-ufcg/cytestion/opt-study/execute-study-idubs/-/tree/main/results/petclinic/IDS

*Assim como o IDS, o IDUBS pode ser adaptado para buscar apenas um objetivo, retornando o primeiro estado encontrado.

Algoritmo 2 Algoritmo IDUBS com múltiplos objetivos

```
1:  goalNodes ← []
2:  roots ← []
3:  function IDUBS(root, goal)
4:      depth ← 0
5:      root.level ← 0
6:      roots.add(root)
7:      while roots is not empty do
8:          for all node in roots do
9:              remaining ← DFS(root, goal, depth − node.level)
10:             if not remaining then
11:                 roots.remove(node)
12:             end if
13:         end for
14:         depth ← depth + 1
15:     end while
16:     return goalNodes
17: end function
18:
19: function DFS(node, goal, depth)
20:     if depth = 0 then
21:         if node is a goal then
22:             goalNodes.add(node)
23:         end if
24:         return TRUE
25:     else if depth > 0 then
26:         anyRemaining ← FALSE
27:         for all child of node.children do
28:             if child not in roots then
29:                 child.level ← node.level + 1
30:                 if child.url ≠ node.url then
31:                     roots.add(child)
32:                 end if
33:                 anyRemaining ← DFS(child, goal, depth − 1)
34:             end if
35:         end for
36:         return anyRemaining
37:     end if
38: end function
```

retornando o valor booleano `true` (linha 24). Na profundidade um, novamente realizamos uma DFS passando pelo nó raiz *A*. Visitamos seus filhos *B* e *C* já que estes não estão incluídos em `roots` e têm URLs diferentes de *A*, ambos são adicionados a `roots`. Além disso, o nó *C* é identificado como um nó objetivo devido a uma falha visível, então é adicionado à lista de nós objetivos.

Na profundidade dois, existem os nós *A*, *B* e *C* como raízes (linha 8), então três chamadas DFS diferentes são feitas. Como tanto *B* quanto *C* são encontrados no nível um, sua profundidade passada é decrementada para um. A DFS para o nó *A* encontra todos os filhos dentro de `roots`, enquanto a DFS de *C* não encontra nenhum filho. Ambos têm `false` em *anyRemaining* e são removidos da lista `roots`. A DFS de *B* encontra os nós *D* e *E*. Nenhum deles está incluído na lista `roots`. No entanto, *D* apresenta um URL diferente (Figura 1-Profundidade 2) o que leva à sua inclusão na lista de raízes. Ambos esses nós não são nós objetivos, nos levando a passar para a próxima iteração.

Na profundidade três, os nós *B* e *D* servem como raízes, levando a duas chamadas DFS diferentes. Como o nó *D* foi encontrado no nível dois, usamos uma profundidade de um (linha 9). A DFS para o nó *B* identifica seu nó filho *E*, que não estava incluído na lista de `roots`. Outra chamada DFS para o nó *E* resulta na descoberta de seu nó filho *G*. Como *G* não está incluído na lista `roots` e tem um URL diferente, ele é adicionado à lista `roots`. Simultaneamente, durante a DFS de *D*, é encontrado um nó filho *F* que não está incluído nos nós raiz. Nenhum desses nós descobertos é um nó objetivo, então apenas passamos para a próxima iteração.

Na profundidade quatro, temos os nós *B*, *D* e *G* como raízes, então três chamadas DFS diferentes são feitas. A DFS de *B* vai para *E* e não encontra nenhum nó filho que não esteja incluído nas `roots`, portanto é removido. A DFS de *D* vai para *F* e não encontra nenhum nó filho, também sendo removido. A DFS de *G* encontra os nós filhos *H* e *I* com URLs diferentes. Eles são apontados como raízes e *H* é encontrado como um nó objetivo. Uma iteração adicional é realizada com raízes *G*, *H* e *I*, que então são todos removidos das raízes, finalizando o resultado da busca ao retornar os nós objetivos *C* e *H*.

Com base na execução fornecida, a suíte de testes gerada pelo IDUBS possui as seguintes sequências de teste: (1) A; (2) $A \rightarrow B$; (3) $A \rightarrow C$; (4) $B \rightarrow D$; (5) $B \rightarrow E$; (6) $B \rightarrow E \rightarrow G$; (7) $D \rightarrow F$; (8) $G \rightarrow H$; (9) $G \rightarrow I$. Comparando com a apresentada na Seção 2.4, podemos ver que a nova suíte é composta por casos de teste menores com menos repetições de estado que usam acessos diretos aos nós. A suíte de testes completa gerada usando IDUBS para a aplicação petclinic está disponível em nosso repositório[5].

4 ESTUDO DE AVALIAÇÃO

Nesta seção, apresentamos o estudo empírico realizados para avaliar o IDUBS no contexto de testes de GUI. Para isso, comparamos execuções da ferramenta Cytestion com IDUBS e com IDS, focando em dois aspectos principais: tempo de execução dos casos de teste e número de estados revisitados. Para orientar nossa investigação, estabelecemos duas questões de pesquisa:

- RQ_1: O IDUBS pode reduzir efetivamente o tempo dos testes de GUI?
- RQ_2: O IDUBS consegue reduzir a redundância na visita de estados?

RQ_1 diz respeito ao tempo de execução dos casos de teste, um fator crítico ao considerar os custos em aplicações web. Enquanto isso, RQ_2 explora a redução de visitas redundantes de estados de GUI durante a execução do IDUBS

Para abordar essas questões, realizamos um estudo empírico em projetos de código aberto. No estudo utilizamos a ferramenta Cytestion para testes automatizados de GUI em aplicações web com a mesma configuração. A versão original do Cytestion emprega o IDS para geração de casos de teste. Reutilizamos a infraestrutura do Cytestion e implementamos o IDUBS, criando uma nova versão da ferramenta (Cytestion IDUBS). A nova versão está disponível em

[5]https://gitlab.com/lsi-ufcg/cytestion/opt-study/execute-study-idubs/-/tree/main/results/petclinic/IDUBS

nosso repositório[6]. Com essa nova ferramenta, pudemos comparar o desempenho dos algoritmos (IDS e IDUBS) para diferentes projetos por meio de duas execuções, uma para cada algoritmo. As aplicações foram exploradas sistematicamente e de forma exaustiva.

4.1 Métricas e Configurações

Duas métricas foram estabelecidas para ajudar a abordar nossas questões de pesquisa. Para RQ_1, utilizamos o *tempo de execução para cada caso de teste* e para RQ_2, avaliamos a *frequência de estados visitados em um conjunto de testes*. Como nosso objetivo é minimizar os custos de testes, um tempo de execução mais rápido e menos estados visitados indicam um conjunto de teste mais eficiente e menos repetitivo.

Na configuração da ferramenta definimos um oráculo genérico para avaliar os estados identificados. A configuração padrão do nosso oráculo inclui a verificação de: (i) mensagens de falha no console do navegador; (ii) códigos de status HTTP nas famílias 400 ou 500 após requisições ao servidor; ou (iii) mensagens de erro padrão na GUI, como “Error” e “Exception”. No entanto, devido à sua natureza genérica, essa abordagem pode resultar em falsos positivos e exigir análises manuais adicionais para confirmar a presença de uma falha real.

Nosso estudo foi realizados em um desktop com um processador Intel Core i7 10700KF, 32GB de RAM DDR4 3200MHz, uma placa de vídeo Nvidia GTX 1060 6GB GDDR5 e um SSD SATA 1TB 500Mpbs/s.

4.2 Projetos Selecionados

Para realizar nosso estudo selecionamos quatro aplicações web de código aberto[7]: i) *school-educational*, um site HTML5 que implementa funcionalidades comuns encontradas em aplicações escolares; ii) *petclinic*, uma aplicação SpringBoot para gerenciar o registro de proprietários de animais de estimação e agendar visitas ao veterinário; iii) *learn-educational*, um site responsivo que apresenta portfólios de cursos educacionais online; e iv) *bistro-restaurant*, um site desenvolvido com HTML, JavaScript e CSS para exibir portfólios de restaurantes.

A tabela 1 fornece detalhes sobre os projetos analisados, incluindo o tamanho em milhares de linhas de código (KLOC) e o número de casos de teste gerados e executados pelo Cytestion. A coluna "Teste IDS"indica a quantidade de testes criados pela ferramenta usando a abordagem IDS, enquanto a coluna "Testes IDUBS"mostra os testes criados com a metodologia IDUBS. A coluna "Testes usados"representa os testes selecionados para análise após a exclusão de valores atípicos e a aplicação de *Winsorization*[2]. Essa abordagem limita os valores extremos para reduzir o efeito de possíveis outliers.

Apesar da sua simplicidade aparente, esses sistemas oferecem recursos de navegação que abrangem uma ampla gama de estados potenciais da interface gráfica do usuário (GUI), fornecendo informações relevantes e facilitando operações de registro que podem levar a falhas visíveis durante a execução dos testes. A eficácia desses sistemas é destacada pelo número significativo de casos de teste gerados e utilizados para análise posterior.

[6]https://gitlab.com/lsi-ufcg/cytestion
[7]https://gitlab.com/lsi-ufcg/cytestion/gui-testing-study/applications

Projeto	KLOC	Testes Usados	Testes IDS	Testes IDUBS
school-educational	30,2	231	231	231
petclinic	25,7	49	50	50
learn-educational	19	219	225	225
bistro-restaurant	33,4	207	212	212

Tabela 1: Informações sobre os projetos.

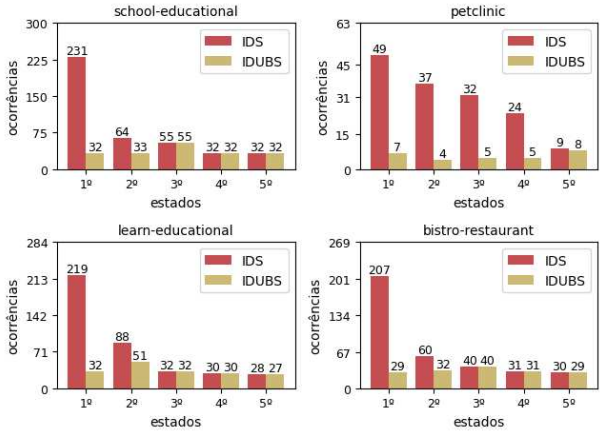

Figura 2: Frequência de acessos em estados.

4.3 Resultados e Discussão

A Figura 2 apresenta a frequência de visitas dos cinco estados mais visitados de cada conjunto de testes gerado usando o IDS e seu valor correspondente com o IDUBS. O estado inicial é o estado de GUI mais acessado em todos os quatro projetos. Com o IDS, cada caso de teste começa na raiz, portanto, cada caso de teste visita o estado inicial. O IDUBS reduziu efetivamente essas revisitas, diminuindo a redundância em pelo menos 85% em todos os projetos ao considerar os estados iniciais. Essa redução era esperada, pois as páginas iniciais geralmente servem como pontos de partida com acesso a vários recursos do sistema e frequentemente levam a novos URLs sendo acessados nas iterações subsequentes do IDUBS.

Ao considerarmos o 3º, 4º e 5º estados mais acessados, notamos menos variação na repetição. Com exceção do projeto *petclinic*, todos os outros projetos tiveram um número igual de acessos nesses três estados usando ambos os algoritmos. Isso ocorreu porque esses estados oferecem numerosas ações que não alteram o URL, levando todos os casos de teste a revisitá-los nas iterações subsequentes. Finalmente, o IDS acessou um total de 2191 estados enquanto o IDUBS acessou 1402 estados. Portanto, o IDUBS resultou em redundância com uma taxa de acesso de $\frac{2191-1402}{2191} \times 100\% \approx 36.01\%$.

A Figura 3 apresenta o tempo de execução de cada caso de teste (eixo x) para os conjuntos do IDS e do IDUBS em cada projeto. As linhas vermelhas representam o tempo de execução de cada caso de teste do conjunto IDS, enquanto as linhas amarelas apresentam a execução do conjunto IDUBS. É importante destacar que cada ponto vermelho tem seu correspondente amarelo e que o tempo de execução é medido em segundos (eixo y).

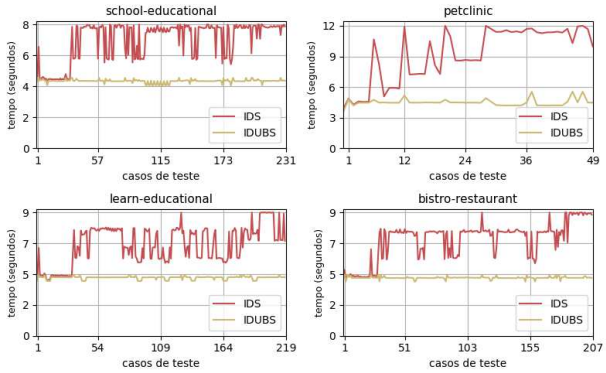

Figura 3: Tempos de execução dos casos de teste por algoritmo.

Nossa análise revela uma diminuição consistente nos tempos de execução para todos os quatro projetos. No *school-educational*, os testes do IDUBS levaram entre 4,3 e 4,4 segundos, em comparação com os testes do IDS que variaram de 4,4 a 9,4 segundos, resultando em uma redução de até 4,9 segundos no tempo de execução. Da mesma forma, no *petclinic*, os testes do IDUBS variaram entre 4,1 e 4,6 segundos, mostrando uma redução significativa em relação aos testes do IDS, que variaram de 4,1 a 11,7 segundos. A maior queda no tempo de execução foi observada nos projetos *learn-educational* e *bistro-restaurant*, ambos mostrando reduções de até 5 segundos.

O IDUBS demonstrou tempos de execução estáveis em todos os casos de teste em diversos projetos. A abordagem de acessar o URL toda vez que ele muda ajudou a manter tempos de execução consistentes. Se o URL muda após cada ação, cada nova iteração sempre conterá uma visita à nova URL e uma ação. O IDS, por outro lado, teve que acessar continuamente a página inicial, realizar uma série de ações no AUT e esperar pelas requisições para a API terminarem. Como a interação com elementos acionáveis do AUT naturalmente exige um tempo de resposta variável influenciado pelas respostas de eficiência das solicitações da API, essa variabilidade impacta diretamente nos tempos de execução. Em contraste, o acesso direto do IDUBS em URLs requer menos ações para realizar testes. Essas descobertas nos ajudam a responder as questões RQ_1 e RQ_2 fornecendo evidências de redução de custos tanto no acesso aos estados quanto no tempo de execução, afirmando assim que o IDUBS pode reduzir custos de forma eficaz.

Para garantir que a performance da ferramenta não foi negativamente impactada analisamos a saída das duas execuções do conjunto de testes, cada uma identificou nove estados com falhas visíveis. Investigamos manualmente os estados e classificamos todas as falhas como falsos positivos. Elas envolviam elementos acionáveis vinculados a sites externos com solicitações falhas. O Cytestion lida com o limite de exploração para evitar explorar estados que não pertencem ao AUT. No entanto, os casos de teste que tentam acessar esses estados ainda são avaliados pelo oráculo. Apesar da geração não continuar nesse ramo, falhas ainda podem ser encontradas neste site externo. Essa situação pode ser vista como uma falha na implementação da ferramenta Cytestion, mas foi importante para demonstrar a equivalência na detecção de falhas pelos dois algoritmos.

5 AMEAÇAS À VALIDADE

Os resultados obtidos em nossos estudos são específicos para os projetos analisados, embora sua aplicabilidade possa permitir generalizações em contextos similares. Encontramos algumas limitações para confirmar que o estudo manteve a eficiência devido à falta de informações sobre cobertura e à ausência de falhas reais detectadas.

Nossas conclusões são fundamentadas na utilização do IDS e do IDUBS dentro da ferramenta Cytestion. Ambas as implementações foram meticulosamente validadas pelos autores por meio de uma série de cenários de teste. Vale ressaltar que os princípios fundamentais desses algoritmos podem ser aplicados de forma independente, sem depender de uma ferramenta específica. Isso indica que as vantagens do IDUBS não se limitam a uma única implementação, pois outras ferramentas ou implementações podem igualmente se beneficiar de suas características.

É importante mencionar que fatores externos, como as condições de rede ou mudanças no ambiente da aplicação web, podem introduzir variabilidade nos resultados e afetar o desempenho dos algoritmos. Para mitigar esse risco, realizamos esforços para executar os conjuntos de testes em um ambiente consistente, utilizando uma máquina dedicada e realizando execuções próximas. Além disso, adotamos a técnica de *Winsorization* para lidar com possíveis valores discrepantes. Os resultados consistentes obtidos em diferentes projetos demonstram a capacidade do IDUBS de se manter resiliente diante de fatores externos.

6 TRABALHOS RELACIONADOS

O algoritmo IDS tem sido estudado no contexto de aplicações web. Weise et al. [31] conduziram um estudo onde mostra que a busca não informada realizada pelo IDS foi considerada ineficiente devido a custos excessivos e limitações do algoritmo em comparação com outros métodos para localizar funcionalidades complexas em serviços web.

Moura et al. [24] discutem as desvantagens dos testes manuais e a demanda por soluções melhores. Eles apresentam Cytestion, uma abordagem e ferramenta que utiliza uma versão do algoritmo IDS para gerar automaticamente uma árvore de GUI enquanto cria e executa o conjunto de testes. Os resultados demonstram a eficácia do Cytestion identificando faltas reais por meio de falhas visíveis. Apesar dos bons resultados, os autores discutem problemas como alto uso de memória e tempo significativo de execução nos conjuntos de testes gerados.

Jiang et al. [15] enfatizam a importância dos testes de GUI em aplicativos Android na detecção de erros. Eles comparam a busca aleatória e a busca sistemática com algoritmos BFS e DFS usando 33 aplicativos reais para estudar seus efeitos na taxa de detecção de falhas e na cobertura de código. Suas descobertas indicam que tanto a busca aleatória quanto a busca sistemática são igualmente eficazes, como também a equivalência de estado tem um impacto significativo na taxa de detecção de falhas e cobertura.

Wen [32] introduz uma nova abordagem para testar aplicações web, denominada Teste Automatizado Dirigido por URL (URL-DAT). Esse método combina o uso de URLs conhecidos previamente com

testes orientados por dados para automatizar a execução dos testes. No entanto, ele não utiliza algoritmos de busca, pois a navegação dentro da aplicação não é o foco principal.

Hu et al. [14] sugerem que os testes automatizados podem melhorar a eficiência dos testes de software, usando ferramentas de automação de teste como Selenium e QTP para aprimorar a precisão dos casos de teste. Sendo assim, os autores representam o fluxo de trabalho do projeto de software como um grafo dirigido e atravessam com o algoritmo DFS para gerar caminhos de teste, visando aumentar a manutenibilidade e reutilização dos testes. Por fim, a conclusão apresenta resultados promissores em testes industriais, como em um projeto de gerenciamento clínico de pesquisa científica.

Os estudos mencionados abrangem uma ampla gama de tópicos, como o uso do IDS, DFS e BFS em testes de GUI, acesso direto por URL nos testes e algoritmos que aprimoram o IDS. O trabalho de Moura et al. foi o único que investigou o uso do IDS em testes de GUI. Nosso trabalho é distinto ao propor uma maneira eficaz de reduzir os custos relacionados aos testes de GUI com o IDUBS, preservando a eficácia dos testes.

7 CONCLUSÕES

Neste trabalho, apresentamos o algoritmo *Iterative Deepening URL-Based Search* (IDUBS) como uma solução para lidar com a redundância no algoritmo IDS em testes de GUI. O IDUBS utiliza informações sobre URLs de estado para localizar novos pontos de partida durante a execução do teste, reduzindo assim a redundância e otimizando o tempo de execução.

Avaliamos o desempenho do IDUBS por meio de um estudo empírico em aplicações de código aberto. Comparamos o IDUBS com o IDS utilizando a ferramenta Cytestion. Os resultados demonstram que o IDUBS foi capaz de reduzir custos e superou o IDS. Concretamente, o IDUBS reduziu o tempo de execução em 39,03% e a redundância de casos de teste em 36,01%. Além disso, o conjunto de testes do IDUBS detectou todas as falhas identificadas pelo IDS.

Em relação a trabalhos futuros, planejamos: i) expandir nossos estudos empíricos para incluir uma gama mais ampla de projetos de código aberto e incluir projetos industriais; ii) investigar o uso de paralelismo para melhorar o processo de busca aplicado pelo IDUBS, iniciando com raízes simultâneas para reduzir o tempo de execução e aprimorar a eficiência; e iii) avaliar como os desenvolvedores percebem a qualidade dos testes gerados com o IDUBS em termos de legibilidade e manutenção.

AGRADECIMENTOS

Gostaria de expressar meus sinceros agradecimentos às pessoas que tornaram possível a realização deste trabalho. Ao meu orientador Professor Everton L. G. Alves pelos seus insights e conselhos fundamentais para o desenvolvimento deste estudo. Agradeço ao Professor Cláudio de Souza Baptista e ao Professor Hugo Feitosa de Figueirêdo por me terem dado a oportunidade de participar do projeto de iniciação científica no Laboratório de Sistemas de Informação, que foi meu suporte e fonte de desenvolvimento pessoal e profissional durante a graduação. Também agradeço ao meu amigo e orientador Thiago Moura, que me apoiou e tornou possível a realização deste trabalho. Agradeço a todos os colegas que conheci durante a graduação, que se tornaram amigos e ajudaram para que esta conquista se tornasse possível. Sou grata pelo suporte que me deram durante a graduação, especialmente a José Itallo, Eniedson Junior, Lucian Júlio, Gustavo Machado e Pedro Gregório. Por fim, um agradecimento especial à minha família, que sempre me apoiou em toda a jornada, especialmente ao meu marido Rogério Carlos, que foi minha base nos momentos difíceis.

REFERÊNCIAS

[1] Pekka Aho, Teemu Kanstren, Tomi Räty, and Juha Röning. 2014. Automated extraction of GUI models for testing. In *Advances in Computers*. Vol. 95. Elsevier, 49–112.

[2] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[3] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.

[4] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology* 55, 10 (2013), 1679–1694.

[5] Axel Bons, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. 2023. Scripted and scriptless GUI testing for web applications: An industrial case. *Information and Software Technology* 158 (2023), 107172.

[6] Alexandre Canny, Philippe Palanque, and David Navarre. 2020. Model-based testing of GUI applications featuring dynamic Instanciation of widgets. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 95–104.

[7] Dmitry Davidov and Shaul Markovitch. 2002. Multiple-goal search algorithms and their application to Web crawling. In *AAAI/IAAI*. 713–718.

[8] Dmitry Davidov and Shaul Markovitch. 2006. Multiple-goal heuristic search. *Journal of Artificial Intelligence Research* 26 (2006), 417–451.

[9] Boni García, Micael Gallego, Francisco Gortázar, and Mario Munoz-Organero. 2020. A survey of the selenium ecosystem. *Electronics* 9, 7 (2020), 1067.

[10] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Creating GUI testing tools using accessibility technologies. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 243–250.

[11] Birgit Hofer, Bernhard Peischl, and Franz Wotawa. 2009. Gui savvy end-to-end testing with smart monkeys. In *2009 ICSE Workshop on Automation of Software Test*. IEEE, 130–137.

[12] Liu Hongyun, Jiang Xiao, and Ju Hehua. 2013. Multi-goal path planning algorithm for mobile robots in grid space. In *2013 25th Chinese Control and Decision Conference (CCDC)*. IEEE, 2872–2876.

[13] Md Hossain, Hyunsook Do, and Ravi Eda. 2014. Regression testing for web applications using reusable constraint values. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 312–321.

[14] Xiaoming Hu and Yibo Huang. 2021. Research and Application of Software Automated Testing Based on Directed Graph. In *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*. IEEE, 661–664.

[15] Bo Jiang, Yaoyue Zhang, Wing Kwong Chan, and Zhenyu Zhang. 2019. A systematic study on factors impacting gui traversal-based test case generation techniques for android applications. *IEEE Transactions on Reliability* 68, 3 (2019), 913–926.

[16] Imran Akhtar Khan and Roopa Singh. 2012. Quality Assurance And Integration Testing Aspects In Web Based Applications. *ArXiv* abs/1207.3213 (2012). https://doi.org/10.5121/ijcsea.2012.2310

[17] Inessa V Krasnokutska and Oleksandr S Krasnokutskyi. 2024. Implementing E2E tests with Cypress and Page Object Model: evolution of approaches. In *CEUR Workshop Proceedings*. 101–110.

[18] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 272–281.

[19] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. 2023. Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 339–350.

[20] Kai Li Lim, Kah Phooi Seng, Lee Seng Yeong, Li-Minn Ang, and Sue Inn Ch'ng. 2016. Pathfinding for the navigation of visually impaired people. *International Journal of Computational Complexity and Intelligent Algorithms* 1, 1 (2016), 99–114.

[21] Kai Li Lim, Kah Phooi Seng, Lee Seng Yeong, Li-Minn Ang, and Sue Inn Ch'ng. 2015. Uninformed pathfinding: A new approach. *Expert systems with applications* 42, 5 (2015), 2722–2730.

[22] Atif M Memon. 2002. GUI testing: Pitfalls and process. *Computer* 35, 08 (2002), 87–88.

[23] Fatini Mobaraya, Shahid Ali, et al. 2019. Technical Analysis of Selenium and Cypress as functional automation framework for modern web application testing. In *9th International Conference on Computer Science*.

[24] Thiago Santos de Moura, Everton LG Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. 2023. Cytestion: Automated GUI Testing for Web Applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 388–397.

[25] Olivia Rodríguez-Valdés, Tanja EJ Vos, Pekka Aho, and Beatriz Marín. 2021. 30 years of automated GUI testing: a bibliometric analysis. In *Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8–11, 2021, Proceedings 14*. Springer, 473–488.

[26] Dacong Yan Shengqian Yang Atanas Rountev, D Yan, and S Yang. 2013. Systematic testing for resource leaks in android applications.

[27] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Pearson.

[28] Nema Salem, Hala Haneya, Hanin Balbaid, and Manal Asrar. 2024. Exploring the Maze: A Comparative Study of Path Finding Algorithms for PAC-Man Game. (2024).

[29] Arie Van Deursen. 2015. Testing web applications with state objects. *Commun. ACM* 58, 8 (2015), 36–43.

[30] Yan Wang, Jianguo Lu, and Jessica Chen. 2014. Ts-ids algorithm for query selection in the deep web crawling. In *Web Technologies and Applications: 16th Asia-Pacific Web Conference, APWeb 2014, Changsha, China, September 5-7, 2014. Proceedings 16*. Springer, 189–200.

[31] Thomas Weise, Steffen Bleul, Diana Comes, and Kurt Geihs. 2008. Different approaches to semantic web service composition. In *2008 Third International Conference on Internet and Web Applications and Services*. IEEE, 90–96.

[32] Robert B Wen. 2001. URL-driven automated testing. In *Proceedings Second Asia-Pacific Conference on Quality Software*. IEEE, 268–272.