Iterative Deepening URL-Based Search: Enhancing GUI Testing for Web Applications

Thiago Santos de Moura
Federal University of Campina Grande
Campina Grande, Brazil
thiago.moura@copin.ufcg.edu.br

Regina Letícia Santos Felipe
Federal University of Campina Grande
Campina Grande, Brazil
regina.felipe@ccc.ufcg.edu.br

Everton L. G. Alves
Federal University of Campina Grande
Campina Grande, Brazil
everton@computacao.ufcg.edu.br

Pedro Henrique S. C. Gregório
Federal Institute of Paraíba
Esperança, Brazil
pedrosgregorio@gmail.com

Cláudio de Souza Baptista
Federal University of Campina Grande
Campina Grande, Brazil
baptista@computacao.ufcg.edu.br

Hugo Feitosa de Figueirêdo
Federal Institute of Paraíba
Esperança, Brazil
hugo.figueiredo@ifpb.edu.br

ABSTRACT

Automated GUI testing has become prevalent in web applications due to its efficiency in detecting visible failures. In this context, scriptless testing can systematically explore the application GUI. To achieve this, a GUI tree can be employed to generate test cases. Algorithms such as IDS can iteratively discover the GUI tree of an application while generating the test suite. However, the resulting suite in such scenarios is often redundant, leading to long execution times. This paper introduces the IDUBS algorithm, an optimized version of IDS that aims to reduce redundancies in state access by identifying URL changes during system exploration. It utilizes this information to streamline path discovery for automatic GUI testing. By employing IDUBS, repetitive actions can be replaced with direct URL visits, resulting in faster retrieval of previous GUI states in subsequent iterations and consequently reducing test costs for test suite execution while maintaining performance. We evaluated the performance of IDUBS in two empirical studies involving twenty industrial and four open-source web applications, comparing it with the baseline strategy (IDS). Our results showed that IDUBS achieved a general reduction in execution time and test case redundancy by 43.41% and 49.30%, respectively, while maintaining code coverage. Additionally, IDUBS suites detected more faults, demonstrating improved performance.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

GUI testing, web applications, search algorithms, fault localization

1 INTRODUCTION

Web development is a fast-paced field often shaped by the ever-changing demands of clients who seek high-quality software releases in short timeframes. Such a context highlights the critical need to ensure stability and reliability in web applications [18]. While manual testing is essential, it is often considered a costly and error-prone activity. Therefore, testers have shifted towards automated strategies for testing web applications, especially in industrial settings [11]. Automated testing strategies offer an effective and repeatable way to test software [14].

Most web applications incorporate a Graphical User Interface (GUI) for user interaction. Therefore, GUI testing has become a significant testing strategy, as various behaviors are triggered by sequences of user events (e.g., clicks, text inputs, menu choices) resulting from interactions with GUI elements (e.g., buttons, text boxes, dropdown menus) [7]. In GUI testing, events are used to explore different states of the Application Under Test (AUT), validate specific functionalities, and/or detect faults based on visible failures generated by the system [8].

Two approaches can be employed for automated GUI testing in web systems: scripted and scriptless testing [8]. In scripted testing, testers manually create scripts for each test sequence, either with or without the assistance of capture-replay tools [20]. These scripts are subsequently executed using testing frameworks like Cypress[1]. On the other hand, scriptless testing involves the use of automatic tools that generate and execute test sequences based on identified GUI elements [1]. While scriptless testing offers automation benefits, it may not provide as much control and customization as scripted testing, making it more suitable for identifying states with visible failures caused by GUI faults.

Scriptless testing can be employed by using Smart Monkey tools, such as TESTAR [35] and Murphy [2]. In this case, test sequences are randomly generated by applying heuristics to boost fault discovery. Nonetheless, the probabilistic nature of this approach may lead to faults remain undetected, as some actionable GUI elements might not be triggered.

Other strategies for implementing scriptless testing include the adoption of Systematic GUI Testing [39] or Model-based GUI testing [24]. The first involves an exhaustive exploration of all actionable GUI elements, while the latter generates test cases based on application models. Both strategies present important challenges. For Model-based GUI testing, specification or behavioral models are often not available [6]. On the other hand, systematic exploration often presents practical problems related to generation and execution time, and state explosion, where the number of potential test cases grows exponentially depending on the system's complexity [5].

To address such issues, a combined strategy can be implemented. By conducting a systematic exploration of the AUT, a model representing its GUI can be incrementally constructed and tests generated. Through automated interactions with GUI elements, new

[1]https://www.cypress.io/

GUI states are discovered in an iterative and finite process [40]. This process results in a GUI tree that represents the found AUT states accessed through the GUI [15]. This GUI tree enables the use of graph algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) to systematically explore the search space for creating test cases [5, 12]. Previous work has shown the practical advantages of such algorithms for GUI testing [17]. However, they impose practical limitations. While DFS may get trapped in deep branches, BFS may demand excessive memory due to its expansive exploration [31].

Considering the incremental discovery aspect and the unknown-depth search space, the Iterative Deepening Search (IDS) algorithm can be used to systematically explore the GUI tree of states [36]. IDS combines the strengths of DFS and BFS while mitigating their limitations. IDS conducts multiple iterations of DFS with increasing depth limits until a goal is reached [32]. In this context, a goal could be the discovery of a state with visible failure or exhaustive exploration. Each iteration starts from the root node, ensuring a comprehensive and systematic traversal of the search space.

IDS is the preferred uninformed search when the search space is large and the depth of the solution is not known [31], which aligns with the challenges of systematic GUI testing [5, 17]. However, by potentially revisiting nodes at each iteration, IDS may end up generating redundant and costly test suites. In large graphs with deep paths to the solution, the time taken to revisit nodes can become significant, affecting the overall performance of the algorithm [22, 23]. Moreover, a single version of an AUT could have multiple faults that manifest as visible failures in different states (nodes). This requires multiple executions of IDS or work with multi-goals in pathfinding [9, 13].

Cytestion is a tool designed for automated GUI testing in web applications [27]. It applies a version of the IDS algorithm with multi-goals to enable systematic exploration of the AUT and searches for visible faults (e.g., GUI failure messages, request status issues, and browser console errors). Cytestion, while building the GUI tree of states, generates and executes the tests.

Due to limitations related to the use of IDS, Cytestion test suites are often redundant and time-consuming. Suppose a system with an initial state with ten actionable elements. By exploring each element with IDS, Cytestion discovers ten new states. If each of these new states also includes ten new actionable elements, a test suite that systematically explores the application would include at least 100 test cases. These 100 test cases will visit the first state 100 times to return to the previous state and execute the new actions. These repetitive actions end up increasing the suite's execution time with no testing gains. In this example, we considered three iterations of IDS in pages with only ten actionable elements. Real-world web applications often encounter hundreds of actionable elements per page and very deep branches that can exponentially increase this redundant access.

In this work, we present the Iterative Deepening URL-Based Search (IDUBS), an optimized and tailored for web version of the IDS algorithm that considers URL changes to shorten paths. By incorporating URL information at GUI tree nodes, IDUBS can identify new starting points and initiate test cases by directly accessing the new associated URL with a specific node. This allows for initiating test cases from various points within the GUI tree instead of starting from the initial URL and navigating through the GUI to reach a specific state. We also introduce a new version of the Cytestion tool that employs the IDUBS algorithm.

We conducted two empirical studies comparing IDUBS with IDS in twenty industrial applications and four open source web applications. In these studies, we compared both strategies (IDS and IDUBS) based on a set of metrics: test case execution time, number of revisited states during execution, test suite coverage, and number of faults detected. Our findings demonstrate that IDUBS performs better than IDS in GUI testing, reducing test execution time and minimizing state redundancy. It also maintains test coverage and improves fault detection.

This work has the following contributions:

- A novel algorithm, IDUBS, designed for reducing test case redundancy and improving time execution in systematic GUI testing;
- An implementation of IDUBS within a dedicated tool for automated systematic GUI testing;
- Two empirical studies with a diverse set of industrial and open-source web applications to compare the performance and effectiveness of IDS and IDUBS.

The remainder of this paper is organized as follows. In Section 2, we present important concepts to base our work. Section 3 presents the IDUBS algorithm. In sections 4 and 5, we discuss the empirical studies and possible threats to validity, respectively. Section 6 discusses the related work. Finally, in Section 7, we present our conclusions and discuss future work.

2 BACKGROUND

2.1 GUI Testing and Framework

The goal of GUI testing is to test a system through its GUI elements and properties [25]. It involves executing user interactions such as clicks, scrolls, and keystrokes on actionable GUI elements, such as buttons and input fields, in various states of the AUT [7]. GUI testing can be performed manually or with automated tools, which can automate tasks such as creating and executing test sequences, defining and evaluating oracles, and analyzing test results [29].

Cypress is a modern testing framework for creating and executing GUI web tests [26]. It interacts with browsers through actions such as clicking buttons and navigate pages, offering simplicity, speed, reliability, and a streamlined API for direct DOM interaction [19]. Cypress also simplifies setup by bundling all necessary components into a single download and benefits from an active community contributing to its evolution.

2.2 Cytestion

The Cytestion tool [27] is an open-source Cypress-based tool that creates GUI test suites using a scriptless and progressive approach. It employs a multi-goal version of the IDS algorithm (Section 2.3) to generate initial test cases, discovering actionable elements in the system's initial state. New tests are created iteratively by exploring each actionable element, with each test case executing all prior actions from the initial state. The goal is to detect visible faults, such as GUI failures, request status issues, and browser console errors. Cytestion generates and runs tests during state exploration,

producing artifacts such as regression testing suites, summaries of detected faults, and replay videos of faulty executions.

2.3 Iterative Deepening Search

The Iterative Deepening Search (IDS), also known as Iterative Deepening Depth-First Search (IDDFS), efficiently traverses graph-based search spaces by gradually increasing the depth limit with each iteration. This iterative approach combines the memory efficiency of DFS with the completeness of BFS. It starts with a depth limit of zero and increases it with each iteration until it finds a node of the goal or exhausts the search space [31]. At every depth limit, IDS performs a DFS with a limit, exploring nodes up to the specified depth. If a goal node is not found within the current depth limit, IDS increases the limit and performs another DFS iteration.

In its traditional form, IDS focuses on locating a single goal node within the search space. However, in many real-world applications, there may be multiple goal nodes that need to be reached [10]. Integrating a multi-goal strategy into IDS extends its applicability, enabling it to efficiently navigate towards multiple objectives within the search space. This adaptation preserves the core principles of completeness while enhancing the algorithm's flexibility and scalability in addressing complex search scenarios.

Listing 1 presents the IDS algorithm with the multi-goals strategy. It begins by initializing an empty list called `goalNodes` to store the goal nodes found during the search (line 1). The main function `IDS` takes the root node of the graph and the goal as inputs (line 2). It iterates over increasing depths from zero to infinity (line 3). At each depth, it calls the `DFS` function to explore nodes up to that depth and receives a boolean value indicating whether at least one new node was found, which potentially allows further exploration (line 4).

The `DFS` function recursively explores nodes in the graph up to a specified depth. If the depth is zero, it checks if the current node is a goal node. If so, the node is added to the `goalNodes` list and returns `true` to evaluate possible children in the next iteration (lines 13-16). If the depth is greater than zero, the function explores all child nodes of the current node recursively, each time decreasing the depth by one (lines 20). The variable `anyRemaining` serves as a flag indicating whether any new nodes were found during the loop of child nodes at this level of depth (lines 18). When it remains `false`, it indicates that no new nodes were found in any branch, signaling an end to the search (lines 5-6).

2.3.1 Running Example. To illustrate the IDS algorithm with multi-goals, consider a GUI exploration of the open-source *petclinic* application. Represented as a tree that will be progressively constructed (Figure 1), each node corresponds to a unique GUI state, and edges show transitions between states. The goal is to identify all failure states (nodes C and H), but initially, we do not know which states will lead to failures.

We begin with a depth limit of zero, enabling the finding of the root node A. As A is not a goal node, i.e., it does not include a visible failure, we progress to the next depth level by indicating the `true` boolean value (line 16). At depth one, we start again with the root node A and recursively explore its children B and C (line 20). We come across node C, which is identified as a goal node due to a visible failure. We add node C to the list of goal nodes and continue exploring the graph.

Algorithm 1 The IDS with Multi-Goals Algorithm

```
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
```

At depth two, we pass through nodes A, B, C again and then make a recursive DFS call for the children of B. Since no failure is found but a child was found, another remaining value is returned. Moving to a depth three, we encounter nodes A, B, C, D, E once more and proceed to call the DFS on their respective children E and D. As F and G are not goal nodes, we continue to depth four where node H is discovered as a goal node and included in `goalNodes`. Finally, one more deep iteration is done, and all DFS calls return `false` as the deeper nodes do not have children therefore concluding IDS's execution and returning the goal nodes C and H.

Based on the presented execution, the test suite generated by IDS has the following test sequences: (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) $A \rightarrow B \rightarrow E \rightarrow G \rightarrow I$. There is a clear redundancy in the number of accessed states, especially the initial state A, which is visited in nine test cases. The complete test suite generated using IDS for the petclinic application can be found in our repository[2].

3 ITERATIVE DEEPENING URL-BASED SEARCH

We present the Iterative Deepening URL-Based Search (IDUBS) algorithm, which differs from IDS by retaining minimal information from prior nodes, creating a fresh starting point for depth searches. This approach removes redundancies when revisiting initial nodes in subsequent iterations. IDUBS is adaptable to various web contexts, such as testing, crawling, and data mining. Its effectiveness, simplicity, and scalability make it a versatile solution.

[2] https://gitlab.com/lsi-ufcg/cytestion/opt-study/execute-study-idubs

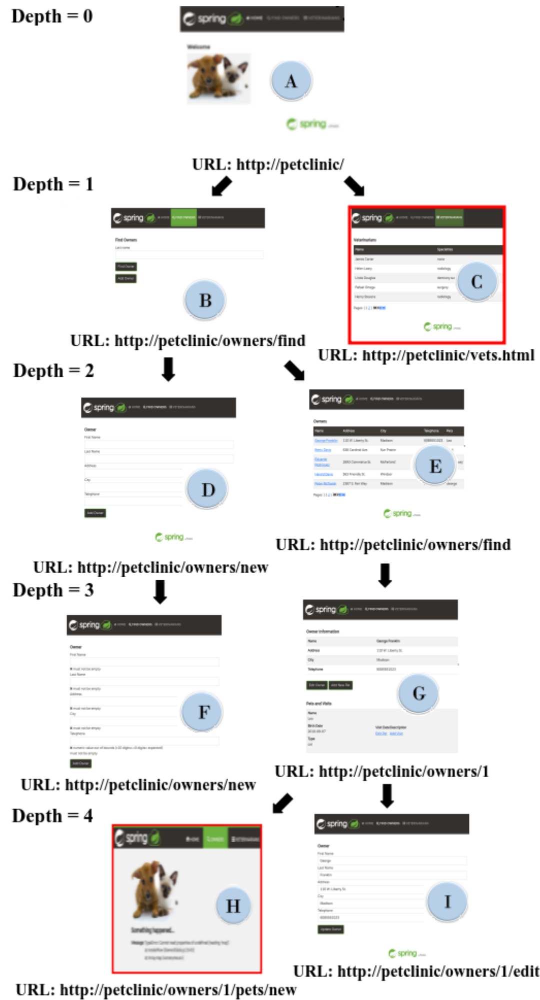

Figure 1: Example of the GUI tree of an AUT.

Our goal is to optimize GUI test execution with IDUBS, reducing redundancy in accessed GUI states and cutting execution time. The algorithm integrates each graph node representing a GUI state with an associated URL. When a new URL is discovered, its node becomes a new root. During exploration, the algorithm can access previously obtained states by directly accessing this new root and continuing the search. This approach is effective for web systems, as direct URL visits provide efficient access to specific states of the AUT and align with contemporary web practices [34, 38].

Listing 2 presents the IDUBS algorithm with support for multiple goal states*. The algorithm initializes two empty lists: `roots`, which tracks root nodes, and `goalNodes`, which stores discovered goal nodes (lines 1-2). In the main function, `IDUBS`, the root node and the goal state are received as arguments (line 3). It begins by setting the initial depth to zero and assigning the level value to the root node, marking its position. After that, the root is added to the `roots` list (lines 4-6). During each iteration of depth, it utilizes all nodes in this list and calls the DFS function while passing the parameters `node`, `goal`, and $|depth - node.level|$ (line 9). This subtraction ensures that the search will adhere to the depth limit even when a deeper root node is used.

Algorithm 2 The IDUBS with Multi-Goals Algorithm

```
 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
```

The `DFS` function recursively explores nodes in the tree up to a specified depth. If the depth is zero, indicating the end of exploration for this branch, it checks if the current node is a *goal node* (line 21). If so, the node is added to the `goalNodes` list (line 22). Subsequently, it returns `true` to evaluate possible children in the next iteration (line 24). If the depth is greater than zero, the function explores all child nodes of the current node to check if it is present in the `roots` list (lines 27-28). This presence indicates that this child has been found and considered a starting point in previous iterations, being used as a new root and having its own separated flow. It occurs due to a difference in the node URL and the child URL being found

*Like IDS, IDUBS can be adapted to single-goal search, returning the first found goal.

(lines 30-32). This change indicates that the node can be directly accessed in the next iteration.

It is important to note that new roots are found in deeper levels of the tree. To handle this and ensure that all flows respect the depth limit, it is crucial to save the level of each child node by adding its parent node's level plus one (line 29). Subsequently, a DFS call is made for the child, which returns a boolean value to `anyRemaining` (line 33). When it remains `false`, this indicates that no new nodes were found in this branch, signaling that the node can be removed from the root list (lines 10-12). Eventually, when no new nodes are found in any branch, this list will become empty. This culminates with the end of the search and results in returning `goalNodes`.

To properly explore the GUI and reveal faults, it is important to adhere to the test case execution order proposed by IDUBS. Faults can manifest in two scenarios: when initially accessing a faulty state or when directly accessing a previously visited state. The latter can be achieved through direct URL access, which may cover different parts of the code. This is particularly evident in systems developed with modern web frameworks which enable server-side rendering and efficient data binding [16]. Code parts may only be accessed through direct URL accesses, due to the way such frameworks handle routing, state management, and data binding. Direct URL access may involve more server-side processing and additional code execution to render the desired GUI state.

3.1 Running Example

To demonstrate the execution of the IDUBS algorithm with multiple goals, we reuse the example from Section 2.3.1 and Figure 1. Our objective is to identify all GUI states where failures occur. We start with a depth limit of zero, including only the `root` in the `roots` list. Since A is not a goal node, we proceed to the next depth level by returning `true` (line 24). At depth one, we perform a DFS through root node A, visiting its children B and C. Both are added to `roots` as they have different URLs from A. Node C is identified as a goal node due to a visible failure and is added to the list of goal nodes.

At depth two, there are nodes A, B, and C as roots (line 8), so three different DFS calls are made. Since both B and C are found at level one, their passed depth is decremented to one. The DFS for node A finds all children inside the `roots`, while the C's DFS finds no child. Both have `false` in *anyRemaining* and are removed from the `roots` list. The DFS of B finds the nodes D and E. None of them are included in the `roots` list. However, D presents a different URL (Figure 1-Depth 2) which leads to its inclusion in the roots list. Both of these nodes are not goal nodes, prompting us to move on to the next iteration.

At depth three, nodes B and D serve as roots, leading to two different DFS calls. Node D was found at level two, so we use a depth of one (line 9). The DFS for node B identifies its child node E, which was not included in the list of `roots`. Another DFS call for node E results in finding its child node G. Since G is not included in the `roots` list and has a different URL, it is added to `roots` list. Simultaneously, during the DFS of D, a child node F is found that is not included in the root nodes. None of these discovered nodes are goal nodes, then we just move to the next iteration.

At depth four, we have the nodes B, D, and G as roots, so three different DFS calls are made. The DFS of B goes to E and does not find any child nodes not included in the `roots`, therefore being removed. The DFS of D goes to F and does not find any child nodes at all, also being removed. The DFS of G finds child nodes H and I with different URLs. They are pointed as roots and H is found as a goal node. One more iteration is performed with roots G, H, and I which are then all removed from the root, finalizing the search result by returning the goal nodes C and H.

Based on the given execution, the test suite generated by IDUBS has the following test sequences: (1) A; (2) $A \to B$; (3) $A \to C$; (4) $B \to D$; (5) $B \to E$; (6) $B \to E \to G$; (7) $D \to F$; (8) $G \to H$; (9) $G \to I$. Comparing to the one presented in Section 2.3, we can see that the new suite is composed of smaller test cases with fewer state repetitions and uses direct access to nodes. The complete test suite generated using IDUBS for the petclinic application is available in our repository[4].

4 EVALUATION STUDIES

In this section, we present empirical studies evaluating IDUBS for GUI testing. We compared IDUBS with a baseline strategy (IDS) on four aspects: test case execution time, revisited states, test suite coverage, and detected faults. Our investigation was guided by two research questions:

- RQ_1: Can IDUBS effectively reduce GUI testing costs?
- RQ_2: Does IDUBS maintain test suite performance?

RQ_1 examines the redundancy of GUI state visits in IDUBS tests and its impact on test case execution time, which affects costs. RQ_2 compares the performance of generated suites in terms of code coverage and fault detection against IDS.

We conducted two empirical studies to address these questions. The first examined a diverse set of industrial projects, while the second focused on open-source projects. Both studies used the Cytestion tool for generating GUI test suites, using the same configuration. The original Cytestion version employs IDS for test case generation. We extended the Cytestion infrastructure by implementing IDUBS, creating a new version (Cytestion IDUBS). This new version is available in our repository [5]. With Cytestion IDUBS, we compared the performance of the IDS and IDUBS algorithms across different projects. Each algorithm was executed separately, as they do not incorporate aleatory aspects. The generated suites systematically and exhaustively explored the AUTs.

4.1 Metrics and Configuration

We established four metrics to address our research questions. For RQ_1, we use *execution time for each test case* and *frequency of visited states in a test suite.* As our goal is to minimize testing costs, we assess this aspect considering test execution time and test suite redundancy. Faster execution and fewer visited states signify a more efficient and less repetitive test suite.

A cost-effective test suite should maintain its testing efficacy. For RQ_2, we evaluate performance using code coverage and the number of visible failures detected. Code coverage is measured with an official Cypress dependency[6] that quantifies frontend code elements. This dependency uses Istanbul[7] to instrument the source code, enabling Cypress to analyze it during execution.

[4] https://gitlab.com/lsi-ufcg/cytestion/opt-study/execute-study-idubs
[5] https://gitlab.com/lsi-ufcg/cytestion/cytestion/-/tags/2.0
[6] https://github.com/cypress-io/code-coverage
[7] https://istanbul.js.org/

We perform different statistical tests to support our conclusions based on the collected data [4]. We used the Wilcoxon rank sum test to compare the top 5 most visited states. For execution times, we used the Mann-Whitney U test, which assesses differences in continuous measurements. To compare coverage rates, we applied the Wilcoxon signed-rank test, suitable for paired data and non-normal distributions.

In our Cytestion setup, we need to configure a generic oracle to assess the identified states. The default configuration includes checking for: (i) failure messages in the browser console; (ii) HTTP status codes in the 400 or 500 families following server requests; or (iii) default error messages in the GUI such as "Error" and "Exception". However, due to its generic nature, this approach may result in false positives and required additional manual analysis to confirm the presence of actual faults.

Despite their deterministic nature, the algorithms may produce varying numbers of test cases due to different exploration strategies. To compare directly, we map the corresponding tests of the generated suites. Moreover, to mitigate execution time outliers caused by external factors like network latency changes, we applied the Winsorization transformation [4], which limits extreme values to reduce the impact of spurious outliers.

Our empirical studies executed on a desktop with an Intel Core i7 10700KF processor, 32GB of RAM DDR4 3200MHz, an Nvidia GTX 1060 6GB GDDR5 video card and a SATA SSD 1TB 500Mpbs/s.

4.2 A Study with Industrial Applications

In our first study, we examined twenty industrial React-based applications from a partner company, each developed by different teams. The applications handle specific fiscal and cost management tasks for companies. Table 1 shows the size (KLOC), and the number of test cases generated and executed by Cytestion with IDS and IDUBS. For confidentiality reasons, the applications are labeled A1 - A20. It is important to highlight that all projects are in production, having been tested by both their development teams and the company QA team. Any discovered faults were reviewed and, if confirmed, registered as bugs.

4.2.1 Results and Discussion. Figure 2 presents the frequency of visits of the top-5 most visited states of each generated test suite using IDS and IDUBS. The initial state is the most accessed GUI state across all projects. With IDS, every test case starts at the root, therefore, each test case visits the initial state. Except for the A12 project, IDUBS effectively reduced revisits to the initial state. This reduction was anticipated as home pages typically serve as starting points with access to various features of the system and often lead to new URLs being accessed in subsequent iterations of IDUBS.

When investigating the A12 executions, we found that the generated test cases did not reach new URLs due to the project's unique characteristic: the URLs simply do not exist. This project has few features, all accessed under the same URL, unlike other applications. In the other 19 projects, IDUBS showed a noticeable decrease in repetitions in the 2nd through 5th states. This was anticipated, as industrial applications often have many intermediate states that must be reached to access deeper functionality. Consequently, these states are repeatedly accessed by IDS, while IDUBS partially avoids them.

Application	KLOC	# of IDS Tests	# of IDUBS Tests
A1	68.5	346	340
A2	82	463	447
A3	52.8	229	231
A4	77.6	443	450
A5	306.8	1756	1780
A6	178.5	794	847
A7	65.9	101	97
A8	75	363	366
A9	37	251	248
A10	228.9	1283	1179
A11	78.1	800	802
A12	32.4	90	90
A13	109.1	420	407
A14	43.7	262	270
A15	62	174	171
A16	73	410	362
A17	58.5	112	116
A18	41.4	357	361
A19	42.1	191	165
A20	397.9	444	444

Table 1: Industrial apps: KLOC, IDS, and IDUBS test counts.

In total, IDS accessed 41,710 states, while IDUBS accessed 20,853 states, achieving a 50% reduction in access for industrial projects. This indicates that IDUBS significantly reduced redundancy compared to IDS. The Wilcoxon rank sum tests on the top 5 most visited states revealed significant differences for all systems ($p < 0.05$), except A12, with large Cohen's d values (1.022 to 1.690), indicating a statistical difference between IDS and IDUBS.

Figure 3 shows the execution time of each test case (x axis) for the IDS and IDUBS suites in each project. The blue lines refer to IDS tests, while the green lines refer to IDUBS tests. It is important to highlight that we used in this analysis only the tests found in both suites, in the same order. Each blue point has a corresponding green point, and the execution time is measured in seconds (y-axis).

Our analysis shows that execution times vary across projects, with IDUBS consistently performing faster. IDS fluctuates between 21.8 seconds and 6.2 seconds, while IDUBS ranges from 18.2 to 4.7 seconds. IDUBS reduced the total execution time by 43.60% in the industry setting. Although both suites initially had similar execution times, IDUBS improved over time by discovering and utilizing new URLs. The exception is A12, where IDUBS did not reduce the execution times. This conclusion was supported by the Mann-Whitney U test where we found significant p-values $< 10^{-15}$ and large Vargha-Delaney effect sizes ($d > 1.5$) for all systems, excepting A12, indicating that IDUBS generally outperforms IDS.

It is possible to observe a gap effect in the executions. As test cases start accessing more complex functionalities that involve intricate database queries, it leads to slow server responses and results in execution peaks. This situation was observed in both executions, but IDUBS consistently showed lower values compared

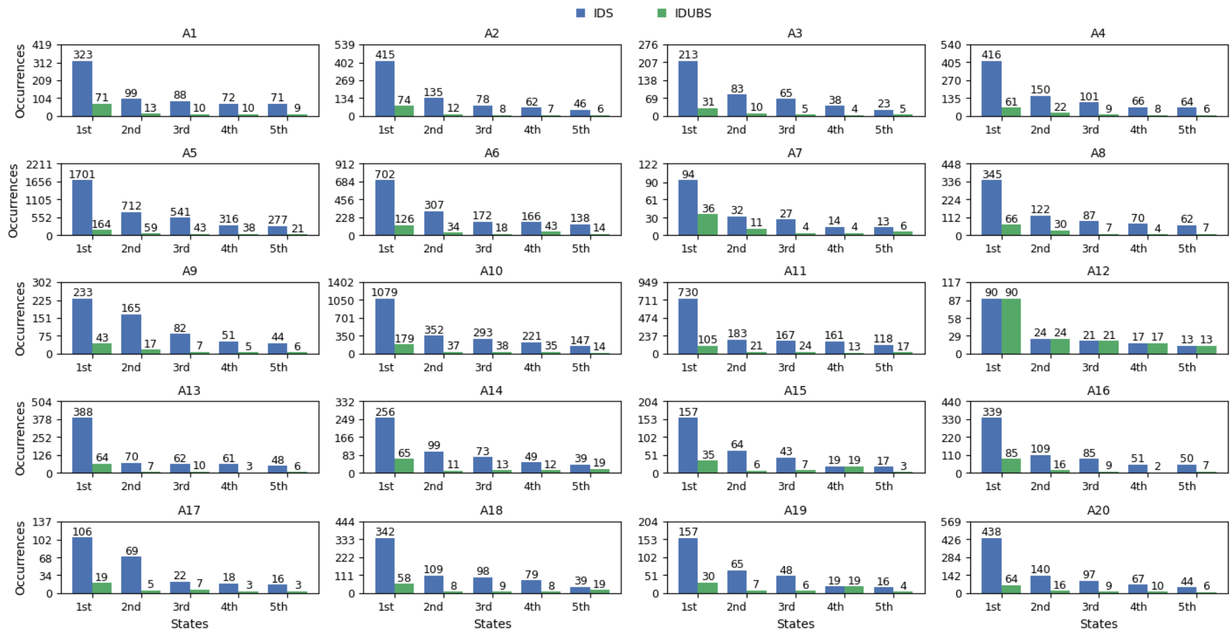

Figure 2: Number of access occurrences in most accessed states.

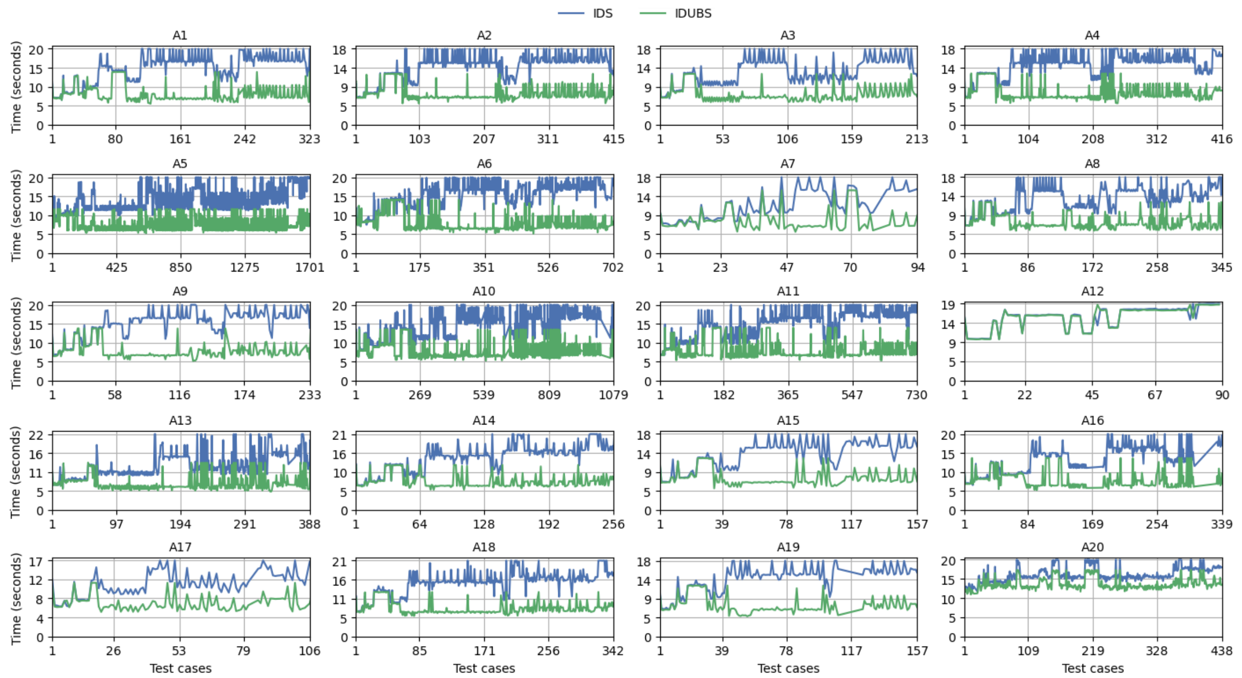

Figure 3: Test case execution times for IDS and IDUBS.

to IDS due to its ability to avoid revisits and shorten paths. These findings help us answer RQ_1 by providing evidence of cost reduction in both state access redundancy and execution time, thus affirming that IDUBS can effectively reduce costs.

Figure 4 shows that both IDS and IDUBS achieve similar coverage levels for frontend code lines across all systems (Wilcoxon signed-rank test p-value of 0.1004). Despite using shorter test cases, IDUBS produces test suites with coverage nearly equivalent to IDS. This suggests both algorithms offer comparable coverage efficacy. Although the coverage levels range from 33.03% to 55.88%, it is

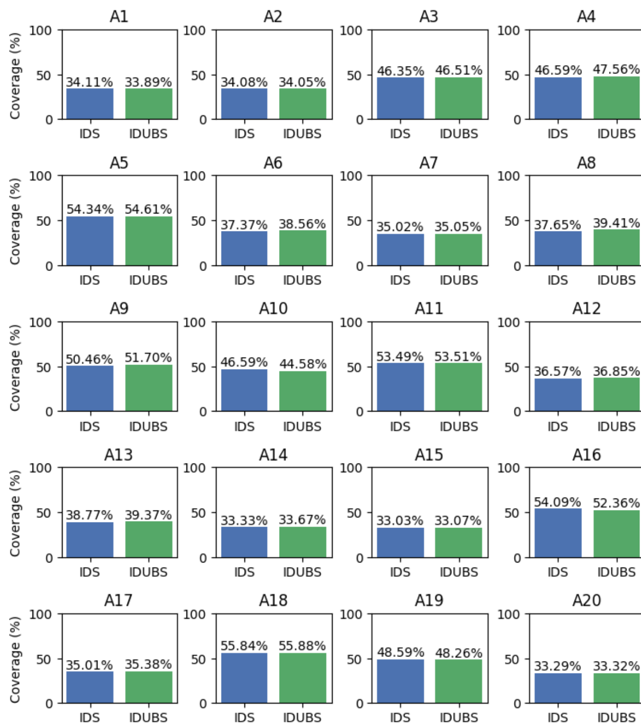

Figure 4: Frontend code coverage results.

noteworthy that these suites were automatically generated. Additionally, IDS has proven effective in detecting visible GUI faults in real-world scenarios [27], making it a valuable option.

We analyzed the visible failures detected by both suites. The IDS suites identified 48 faulty states, which we manually inspected. These faults correspond to six actual issues related to various types of bugs, including button-triggered processes displaying the error message "An unexpected error occurred" and page crashes when the edit button is clicked.

The IDUBS suites identified 317 states with visible failures. Moreover, all found IDS failed states were also detected by the IDUBS. We carefully analyzed each failure and discovered that a fault in one of the horizontal components of the applications was exposed only when the page was reloaded or accessed directly via the URL. Consequently, this fault appeared on all pages using this component exclusively when running the IDUBS suite. In total, seven faults were registered, six found by both suites (IDS and IDUBS), and one detected only by the IDUBS suite. The faults were presented to the QA team and managers, who provided positive feedback. They noted that these issues had been overlooked by the company's quality process and could impact the user experience.

The findings discussed here demonstrate the benefits of using IDUBS in industrial settings. The generated suites provide similar coverage while detecting new faults and significantly reduce the costs associated with test execution, including time and redundancy.

4.3 A Study with Open Source Applications

In our second study, four open-source web applications were selected as objects: i) *school educational*, an HTML5 website that implements common functionalities found in school applications; ii) *petclinic*, a SpringBoot application to manage pet owners' registration and scheduling veterinarian visits; iii) *learn educational*, a responsive website that showcases online educational course portfolios; and iv) *bistro restaurant*, a website developed with HTML, JavaScript, and CSS to display restaurant portfolios. They are available in our repository[8]. Since our first study (Section 4.2) dealt with React-based projects, here we selected projects that do not use any modern web framework. This decision is motivated by our objective to ascertain the continued relevance of our findings across a wider spectrum of applications.

Project	KLOC	# of IDS Tests	# of IDUBS Tests
school educational	30.2	231	231
petclinic	25.7	50	50
learn educational	19	225	225
bistro restaurant	33.4	212	212

Table 2: Open projects: KLOC, IDS, and IDUBS counts.

Table 2 provides information on the projects, including their size (KLOC), and the number of test cases generated and executed by Cytestion with IDS and with IDUBS. Despite their simplicity, these systems offer navigation features with a wide range of potential GUI states, display important information, and facilitate registration operations that can result in visible failures. This is evidenced by the number of test cases generated.

4.3.1 Results and Discussion. Figure 5 shows the frequency of visits of the top-5 most visited states of each generated test suite using IDS and IDUBS. Again, the initial state is the most accessed GUI state across all four projects. IDUBS effectively reduced revisits to the initial state, decreasing redundancy by at least 85% across all projects.

When we consider the 3rd, 4th, and 5th most accessed states, we noticed less variation in repetition. With the exception of the *petclinic* project (Wilcoxon rank sum tests, $p - value = 0.01193$, and Cohen's $d = 2.298$), all other projects had a similar number of accesses in these three states using both algorithms. This happened because these states offer numerous actions that do not change the URL, leading all test cases to revisit them in subsequent iterations. Finally, considering only open sources, IDS accessed a total of 2191 states while IDUBS accessed 1402 states. Therefore, IDUBS resulted in an access reduction of 36.01%.

Figure 6 presents the execution time of each test case for IDS and IDUBS per project. Our analysis reveals a consistent decrease in execution times for all four projects. In the *school-educational*, IDUBS tests took between 4.1 and 4.6 seconds, compared to IDS tests which ranged from 4.2 to 8.1 seconds, resulting in up to a 3.4-second reduction in execution time. Additionally, the *petclinic* project experienced the most significant drop, with a reduction of 6 seconds, while the *learn-educational* and *bistro-restaurant* projects saw decreases of 4.6 and 4.7 seconds, respectively. The Mann-Whitney

[8] https://gitlab.com/lsi-ufcg/cytestion/opt-study/applications

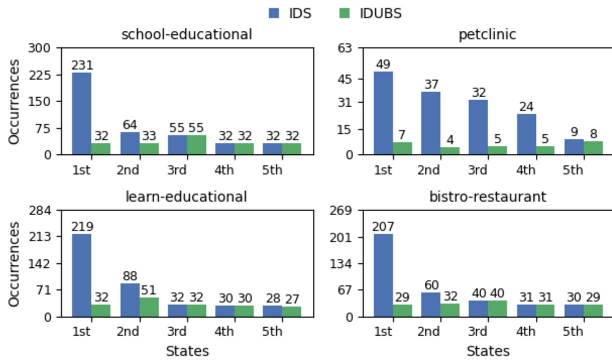

Figure 5: Frequency of accesses in highly accessed states.

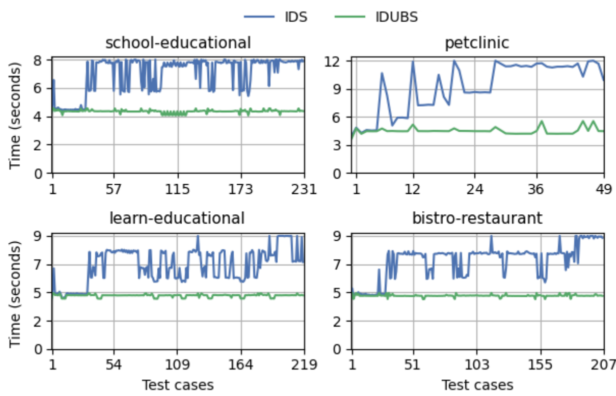

Figure 6: Test case execution times by algorithm.

U test confirms this conclusion by presenting significant differences between the two strategies, with p-values $< 10^{-14}$ and large Vargha-Delaney effect sizes ($d > 2$).

IDUBS has demonstrated stable runtimes in all test cases across various projects. The approach of accessing the URL every time it changes has helped maintain consistent execution times. If the URL changes after each action, every new iteration will always contain a visit to the new URL and one action. IDS, on the other hand, had to continuously access the home page, perform a series of actions in the AUT, and wait for API requests to finish. As the interaction with actionable elements of the AUT naturally demands a variable response time influenced by API request efficiency responses, this variability directly impacts execution times. In contrast, IDUBS direct URL access requires fewer actions to perform tests. These findings help us answer RQ_1 by providing evidence of cost reduction in both state access and execution time, thus affirming that IDUBS can reduce costs effectively.

Regarding performance, we were unable to measure frontend code coverage due to compatibility issues with the Istanbul dependency, which supports only projects using frameworks like React that use JavaScript ES5. Therefore, we focus our analysis on the found faulty states. Each suite identified nine states with visible failures. We manually investigated the states and found that all failures were false positives. They involved actionable elements linked to external websites with failing requests. Cytestion deals with the exploration limit to avoid exploring states that do not belong to the AUT. However, test cases that try to access such states are still evaluated by the generic oracle. Despite generation not continuing in that branch, faults can still be found on this external site. This situation can be viewed as a limitation of the generic oracle implemented by the Cytestion tool. However, for the purpose of our investigation the executions show an equivalence in fault detection of the two algorithms.

Based on the results discussed in Sections 4.2 and 4.3, we can answer RQ_1 and RQ_2 by stating that IDUBS can effectively reduce GUI testing costs (execution time and test redundancy) while maintaining or improving the performance (coverage and new faults), when compared to IDS.

5 THREATS TO VALIDITY

Our results are based on the specific projects examined in our studies. However, we analyzed a set that combined open-source and industrial projects, which we consider to be a reliable sample of web applications. It is important to emphasize the substantial representation of industrial projects in our analysis, enhancing the relevance of our findings to similar industrial contexts.

Computational overhead can be a key aspect when evaluating an algorithm. In our studies, we indirectly analyze this aspect by comparing the execution time of IDS and IDUBS. However, other metrics are yet to be analyzed in the future (e.g., memory usage).

The performance analysis (RQ2) in the study on open-source applications was limited because we were unable to collect coverage information, and no real faults were detected. Nevertheless, we contend that greater significance lies in the evaluation carried out in the industrial study. Considering the diverse sizes and complexities of the industrial objects, we believe they offer robust evidence regarding the stability of IDUBS concerning testing efficacy. Industrial settings adhere to rigorous quality standards and involve various stakeholders, thereby ensuring the reliability and applicability of the results. Additionally, the open-source study further validated IDUBS's ability to reduce GUI costs (RQ1).

Our findings rely on the utilization of IDS and IDUBS within the Cytestion tool. The authors meticulously validated both implementations through a series of testing scenarios. Furthermore, the fundamental principles of these algorithms can be applied autonomously, irrespective of any particular tool. This implies that the found IDUBS advantages go beyond a singular implementation, as other implementations or tools can likewise harness their benefits.

The IDS algorithm highlights its combination of BFS and DFS. While IDS inherently performs a BFS through multiple DFS executions, alternative methods like Bidirectional Search and Heuristic-Enhanced IDS can be used to enhance efficiency [33].

External factors, such as network conditions, or changes in the web application environment, could introduce variability in the results and impact both algorithm performance. To mitigate this risk, we executed the test suites in a controlled environment on a dedicated machine, running each suite only once with minimal delay between them. Moreover, we used the Winsorization transformation to mitigate possible outliers. The consistent results found across different projects indicate IDUBS's resilience to external factors.

6 RELATED WORK

The IDS algorithm has been studied in the context of web applications. Weise et al. [37] conducted a study on the importance of ontology in defining parameter semantics and efficient web service discovery. In their analysis, the uninformed search performed by IDS was found to be inefficient due to excessive costs and algorithm limitations compared to other methods for locating composite semantics in web services.

Our previous work [27] discusses the drawbacks of manual testing and the demand for better solutions. It introduces Cytestion, an approach and tool that utilizes a version of the IDS algorithm to automatically generate a GUI tree while creating and executing the test suite. The results demonstrate Cytestion's efficacy in industrial projects, identifying real faults through visible failures. Despite the good results, the authors discuss problems with the generated suites such as high memory usage and significant execution time.

Jiang et al. [17] emphasize the significance of GUI testing in Android apps and examines the impact of GUI state equivalence choices on error detection. It compares random search and systematic search with BFS and DFS algorithms using 33 real applications to study their effects on fault detection rate and code coverage. Their findings indicate that both random search and systematic search are equally effective, while state equivalence has a significant impact on fault detection rate and coverage.

Wen [38] presents a new methodology for testing web-based applications and technologies, the URL-Driven Automated Testing (URL-DAT). This method involves using previously known URLs and data-driven testing to guide data through the automation of test execution, thereby combining them. However, no search algorithm is used since navigation through the AUT is not the goal.

Hu et al. [15] suggest that automated testing can improve software testing efficiency by using test automation tools such as Selenium and QTP to enhance test case accuracy. It involves representing the software project workflow as a directed graph and traversing it with the DFS algorithm to generate test paths, aiming to increase maintainability and reuse of tests. The conclusion presents promising results in industrial tests, such as in a scientific research clinical management project.

Lim et al. [21] introduce Boundary Iterative-Deepening Depth-First Search (BIDDFS), an algorithm that combines the IDS and Dijkstra algorithms to optimize pathfinding by setting node storage limits and following a specific expansion pattern. Through simulation experiments, BIDDFS showed superior performance when performing blind searches in unknown environments, evidencing its potential for real-world pathfinding efficiency improvements. However, it does not share chain information from previous nodes or directly access any node in the graph.

Bons et al. explore scriptless testing using the TESTAR tool [8], which aims to identify visible failures in GUI applications. This tool is widely used in the industry, demonstrating significant results and providing a basis for further research [3, 28, 30]. Additionally, Aho et al. present Murphy [2], a tool that automates GUI testing using intelligent agents triggered by specific GUI states to detect failures in behavior and functionality. While both tools leverage advanced techniques for uncovering potential faults, they have limitations as they employ a random search approach. Therefore, a more systematic approach would provide a comprehensive solution for identifying these faults.

The mentioned studies cover a wide range of topics, such as the use of IDS in GUI testing, DFS and BFS in GUI testing, direct URL access in tests, algorithms that enhance IDS, and alternatives testing tools. Our previous work was the only one that investigated the use of IDS in GUI testing. Our work is distinct in proposing an effective way to reduce the costs related to GUI testing with IDUBS but preserving its testing power.

7 CONCLUDING REMARKS

In this paper, we presented the Iterative Deepening URL-Based Search (IDUBS) algorithm as a solution to address redundancy in IDS algorithm for GUI testing. IDUBS uses information about state URLs to locate new starting points during test execution, thereby reducing redundancy and optimizing execution time. This algorithm is particularly useful in applications where the URL changes reflect the state changes, enabling consistent save points and faster access. However, when dynamic web content changes frequently without changing the URL, the IDUBS basically perform like an IDS.

We evaluated the use of IDUBS through empirical studies on industrial and open-source web applications. We compared IDUBS with the IDS using the Cytestion tool. The results demonstrate that IDUBS was able to reduce costs and outperformed IDS. IDUBS reduced execution time by 43.41% (43.60% for industrial and 39.03% for open source projects) and decreased test case redundancy by 49.30% (50% for industrial and 36.01% for open source projects), making it a beneficial solution while maintaining the same code coverage. Additionally, the IDUBS suite detected all faults detected by IDS and an extra critical one that was disseminated across several states of the industrial systems.

As for future work, we plan to: i) expand our empirical studies by encompassing a wider range of open source projects; ii) investigate the use of parallelism to improve the search process applied by IDUBS, by starting with simultaneous roots to reduce execution time and enhance efficiency; iii) evaluate how developers evaluate the quality of the tests generated with IDUBS on readability and maintainability aspects.

REFERENCES

[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] Pekka Aho, Matias Suarez, Teemu Kanstrén, and Atif M Memon. 2014. Murphy tools: Utilizing extracted gui models for industrial software testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 343–348.

[3] Pekka Aho, Tanja EJ Vos, Sami Ahonen, Tomi Piirainen, Perttu Moilanen, and Fernando Pastor Ricos. 2019. Continuous piloting of an open source test automation tool in an industrial environment. *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)* (2019), 1–4.

[4] 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.

[5] 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*. Association for Computing Machinery, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[6] Alexander Bainczyk, Alexander Schieweck, Bernhard Steffen, and Falk Howar. 2017. Model-based testing without models: the TodoMVC case study. *ModelEd,*

TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday (2017), 125–144.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 204–217.

[13] 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.

[14] 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.

[15] 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.

[16] Taufan Fadhilah Iskandar, Muharman Lubis, Tien Fabrianti Kusumasari, and Arif Ridho Lubis. 2020. Comparison between client-side and server-side rendering in the web development. In *IOP Conference Series: Materials Science and Engineering*, Vol. 801. IOP Publishing, 012136.

[17] 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.

[18] 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

[19] 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.

[20] 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.

[21] Kai Li Lim, Kah Phooi Seng, LS Yeong, SI Ch'ng, and K Ang Li-minn. 2013. The boundary iterative-deepening depth-first search algorithm. In *Second International Conference on Advances in Computer and Information Technology: ACIT 2013*. Institute of Research Engineers and Doctors, LLC, 119–124.

[22] 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.

[23] 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.

[24] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. IEEE, 260–269.

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

[26] 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*.

[27] Thiago Santos de Moura, Everton L. G. 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.

[28] Fernando Pastor Ricós, Pekka Aho, Tanja Vos, Ismael Torres Boigues, Ernesto Calás Blasco, and Héctor Martínez Martínez. 2020. Deploying TESTAR to enable remote testing in an industrial CI pipeline: a case-based evaluation. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I 9*. Springer, 543–557.

[29] 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.

[30] Urko Rueda, Tanja EJ Vos, Francisco Almenar, MO Martınez, and Anna I Esparcia-Alcázar. 2015. TESTAR: from academic prototype towards an industry-ready tool for automated testing at the user interface level. *Actas de las XX Jornadas de Ingenierıa del Software y Bases de Datos (JISBD 2015)* (2015), 236–245.

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

[32] 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).

[33] Shahaf S Shperberg, Steven Danishevski, Ariel Felner, and Nathan R Sturtevant. 2021. Iterative-deepening bidirectional heuristic search with restricted memory. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 31. 331–339.

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

[35] Tanja EJ Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. 2021. TESTAR–scriptless testing through graphical user interface. *Software Testing, Verification and Reliability* 31, 3 (2021), e1771.

[36] 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.

[37] 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.

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

[39] Dacong Yan, Shengqian Yang, and Atanas Rountev. 2013. Systematic testing for resource leaks in Android applications. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 411–420. https://doi.org/10.1109/ISSRE.2013.6698894

[40] Xun Yuan and Atif M Memon. 2010. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology* 52, 5 (2010), 559–575.