Automated Fixing of Web UI Tests via Iterative Element Matching

Yuanzhang Lin*
Beihang University
Beijing, China
linyz2020@gmail.com

Guoyao Wen
Huawei Technologies Co., Ltd.
Shenzhen, China
wenguoyao@huawei.com

Xiang Gao†
Beihang University
Beijing, China
xiang_gao@buaa.edu.cn

***Abstract*—Web UI test cases are used for the automatic testing of web applications. When a web application is updated, these UI tests should also be updated for regression testing of the new version of web application. With the rapid evolution, updating UI tests is a tedious and time-consuming task. To solve these problems, automatically repairing web UI tests has gained increasing attention recently. To repair web UI tests, the most important step is to match the UI elements before and after the web page update. Existing work matches UI elements according to visual information, attributes value, or Document Object Model (DOM) structures. However, they either achieve low element matching accuracy or only work on simple UI tests. To solve these problems, we proposed UITESTFIX, an approach based on a novel iterative matching algorithm for improving the accuracy of matching UI elements. UITESTFIX is designed based on two main insights: (1) beyond attribute and DOM structures, the relations between different elements can also guide the matching process, and (2) the matching results of previous iterations could guide the matching of the current iteration. Our evaluation of publicly available datasets and two industrial apps shows that UITESTFIX outperforms four existing approaches by achieving more accurate element matching and producing more correct fixes.**

***Index Terms*—Iterative Element Matching, Web Testing, Test Repair**

I. INTRODUCTION

Web apps are characterized by rapid updates. Existing UI test cases are often used as regression tests to ensure the update does not affect original program behaviors. However, during app evolution, developers may modify attributes in web elements, causing the corresponding UI test cases to be broken on the new versions. To make existing test cases executable on the updated version, testers need to manually (1) execute new test cases, (2) locate the bug, and (3) fix it, e.g., modify the element locator of test cases. Updating broken UI tests manually is time-consuming and labor-intensive. In recent years, several techniques have been proposed in academia and industry to automatically fix broken UI test cases caused by app evolution [1], [2], [3], [4], [5].

To automatically fix broken web UI tests, one of the most important steps is to repair its outdated element locator. Existing approaches fix element locators by matching the UI elements before and after update using either visual information [2], [6], attributes [7] (e.g., identifier name), or Document Object Model (DOM) structures [8], [3]. More specifically, those approaches (1) collect the visual image, attributes, and DOM structure of each element before and after the web page update, (2) compare the similarity of each pair of elements using similarity metrics, (3) if the similarity of two elements is greater than a threshold, they are identified as matched elements. UI test repair tools can then fix the broken locators of UI tests according to element matching results.

However, existing approaches still have some limitations. First, their accuracy in matching UI elements is still quite low. For instance, as shown in our evaluation (Section VI), the state-of-the-art UI element matching tool SFTM achieves 68.8% accuracy, and the UI repair tool VISTA only achieves 46.7% accuracy. The low element matching accuracy will then significantly decrease the effectiveness of automated UI test repair techniques. Second, prior work mainly studied the maintenance of UI tests on open-source web apps with test cases written or translated (migrate test cases from other Capture–Replay tools to Selenium) by researchers. The manually written or translated test cases are usually short and simple.

Moreover, designing a robust framework that can repair complex industrial UI tests is challenging. For instance, VISTA [2], a state-of-the-art repair framework, failed to repair our provided industrial tests correctly because (1) industrial web page tests have many control-flow branches, but VISTA only supports tests with a single control flow, and (2) industrial tests usually adopt Page Object design pattern for improving test maintenance [9], but VISTA parses and extracts statements using regular expressions, (3) industrial web tests may invoke additional APIs on top of Selenium APIs to expand its functionality but VISTA only supports specific Selenium APIs.

To solve the above problem, we propose a general framework UITESTFIX, for automatically fixing the UI tests of practical and large-scale web applications. The core technique of UITESTFIX is a novel iterative element-matching approach that fully utilizes the DOM structure information. The main insight is that the relative positions of elements and the relations with neighboring elements may help to improve matching accuracy. To capture such information, we propose *path similarity* and *region similarity* to measure the similarity

*This work was primarily undertaken by the author during his internship at Huawei while studying at the Southern University of Science and Technology.
†Corresponding author.

of elements according to their relations. In an iterative process, UITestFix first matches the elements that are more likely to be matched with high confidence, and then current matching results could guide the element matching in the following iterations. More specifically, the workflow of UITestFix is as follows: (1) it first simplifies the DOM structure by grouping related elements of web pages; (2) calculates the similarity according to id, attribute, DOM structure and etc.; (3) relies on the *iterative matching* algorithm to continuously recalculate the similarity and rematch the elements in multiple iterations to improve the matching accuracy. Based on our matching algorithm, we implemented the repair framework for automatically fixing broken web UI tests. UITestFix is a robust repair framework that includes a dynamic repair component that can parse and repair the UI tests of all kinds of web applications.

We evaluate UITestFix on a set of open source applications and industrial applications. In terms of element matching, UITestFix correctly matched 2,141 (81.9%) elements, which increased the correct rate by 16.1% compared with the best result of existing tool SFTM (68.8%). Moreover, UITestFix and SFTM take similar time to generate matching results (0.97s and 0.87s), which we believe is acceptable. To compare the repairability of UITestFix with existing work, we integrate the four element matching baseline approaches (WATER [7], VISTA [2], WebEvo [6], and SFTM [8]) into our repair framework (note that the existing tools do not support the web applications used in our evaluations). Evaluation results show that UITestFix can fix 113 (68%) test cases, much higher than the 73 test cases (44%) of the best existing tools. Our results indicate that UITestFix can match more complex element changes and has a higher matching accuracy rate and test case repair rate.

Contributions Our contributions are summarised as follows:

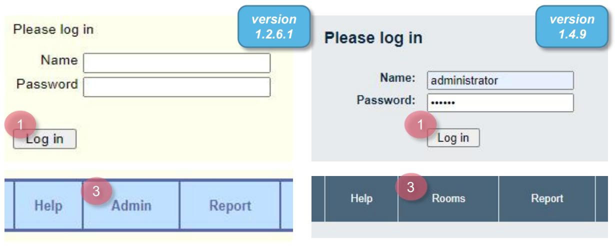
- We proposed a novel element matching algorithm that can match UI elements more accurately by matching elements in multiple iterations.
- We proposed *path similarity* and *region similarity* to measure the similarity of elements according to their relations.
- We design and implement a UI test repair framework, UITestFix, and evaluate it by comparing it against four state-of-the-art baselines. Evaluation results show that UITestFix outperforms existing techniques.
- To facilitate future research on web UI element matching and repair, we relabeled and augmented an open-source UI element matching dataset, which contains 1,620 labelled UI elements. Our dataset and results are available at https://anonymous.4open.science/r/Web-UI-Dataset-9645.

II. Motivating Example

In this section, we use an example of an open-source app MRBS in the test cases dataset [4] to explain the shortcomings of existing approaches [7], [2], [6], [8], and how UITestFix fixes the broken test by matching elements.

Figure 1 shows the screenshots of the MRBS app for versions 1.2.6.1 and 1.4.9, the simplified HTML code, and

```
① driver.findElement(By.xpath("/html/body/form/input[3]")).click();
② Thread.sleep(2000);
③ driver.findElement(By.xpath("//a[text()='Admin']")).click();
   ....
   Assert.assertEquals(driver.findElement(By.xpath("//a[@href='admi
   n.php']")).getText(), "Back to Admin");
```



(a) MRBS Web apps of version 1.2.6.1 and 1.4.9, and part of the broken DeleteNegativeAreaTest.

```
<tr>
  <td class="banner" bgcolor="#C0E0FF"
      align="CENTER">
        <a href="admin.php?day=28;month=01">
            Admin</a>
   </td>... </tr>
```

(b) The simplified HTML code of the banner in version 1.2.6.1.

```
<tr>
 <td>
    <a href="admin.php?day=28;month=01;
        area=888;room=409">Rooms</a>
 </td>...</tr>
```

(c) The simplified HTML code of banner in version 1.4.9.

Fig. 1: Motivating example of MRBS web page.

one simplified test. The broken test (*DeleteNegativeAreaTest*) was manually translated to the Selenium test by the authors of prior work [4]. It contains three events: ① *click* the "Log in" button, ② *sleep*, and ③ *click* the "Admin" button. In Figure 1b, we use e_o^{Admin} to denote the old element with text *Admin* and e_o^{ban} to represent the old element with tag *tr*, which is a banner. Similarly, in Figure 1c, we use e_n^{Rooms} to represent the element with text *Rooms* and e_n^{ban} to represent the element with tag *tr*. When MRBS evolved from 1.2.6.1 to 1.4.9, its view and attributes of elements changed drastically. For instance, the text of e_o^{Admin} changed from "*Admin*" to "*Rooms*", the background color of e_o^{Admin} was changed from blue to dark blue, which caused the visual change, the attribute of e_o^{Admin} was extended with "`area=888;room=409`", and all the attributes of the ancestors of e_o^{Admin} (i.e., e_o^{ban}) were removed. The changes cause *DeleteNegativeAreaTest* to fail in the new version. From the motivation example, we can obtain the following observations:

Existing approaches usually match elements by matching either their attributes [7], visual information [2], or both [6]. Due to the changes in the attributes and visual information, the similarity between e_o^{Admin} and e_n^{Rooms} is low. Algorithms based on text information [7], visual information [2], or both [6] fail to match e_o^{Admin} with e_n^{Rooms}. The SFTM algorithm [8], which matches attributes of the parent nodes and child nodes for similarity propagation, also fails to match e_o^{Admin} with e_n^{Rooms} because (1) the similarity score based on attribute matching is quite low due to the changes in attributes, and (2) the parent node of e_o^{Admin} do not share common attribute tokens with the parent nodes of e_n^{Rooms}.

By investigating this example, we have the following observation: *an element matching could depend on its neighboring elements that have been matched.* In this example, there are multiple elements located in the banner e_o^{ban}, including the "Help" button, "Admin" button, "Report" button. Through the word "admin.php" also appears elsewhere on the page, the elements around e_n^{Admin}, e.g., the "Help" button of the old and new version which can be easily matched, can be used to distinguish e_n^{Admin} with the "admin.php" elements located somewhere else. It indicates that element matching could also be guided by the matching results of other elements.

From the observations, we could measure the similarity of elements based on the following ideas.

- **I1: Elements with closer matched ancestors are more similar.** For any element e_o in the old version and any node e_n in the new version, if they have the matched ancestor (e.g., banner), we consider e_o and e_n to be structurally similar. If the matched ancestor is closer to e_o and e_n (e.g., the child element of the banner), e_o and e_n are more likely to match due to higher structural similarity.
- **I2: Ancestors with more common leaf elements are more similar.** For two ancestors (non-leaf elements), if most of their leaf elements match, they are likely to be matched. For example, if most of the buttons in the banner before and after the version updates match, these banners tend to be matched.
- **I3: Matching elements by multiple iterations.** Different from existing approaches that rely on pre-defined similarity metrics in one single step, we could perform an iterative approach for matching. In each iteration, a matching result is calculated. The next iteration will recalculate the similarity based on the matching result and also update the matching result. During the iterations, the higher-confident matching results could help to increase the similarity of the lower-confident matched elements through multiple iterations of matching.

How does UITESTFIX *match these elements?* In the first iteration, UITESTFIX has matched elements with high initial similarities (e.g., the element with text *Help*). In the second iteration, the structural similarities of e_o^{ban} and e_n^{ban} are increased because the leaf nodes in the region represented by e_o^{ban}, i.e., the element with text "Help", are matched in the first iteration. Then, in the third iteration, because the e_o^{Admin} is contained in e_o^{ban} and e_n^{Rooms} is contained in e_n^{ban}, and e_o^{ban} are matched with e_n^{ban}, the structural similarity between e_o^{Admin} and e_n^{Rooms} are increased. After the three iterations, the similarity between e_o^{Admin} and e_n^{Rooms} is larger than the threshold θ, so they can be matched. This example shows the advantages of UITESTFIX: (1) when attributes change significantly, using the matching results to calculate the structural similarity can enhance the accuracy of the matching; (2) in the process of multiple iterations, the elements in the previous iteration could guide the matching of the remaining elements. In some cases, the wrongly matched elements can even be corrected during the iterative process (not shown in this example).

With the matching results, UITESTFIX will then repair the broken UI test by changing the locators of event $\textcircled{1}$ and event $\textcircled{3}$. Specifically, UITESTFIX updates the locator of the "Log in" button from "/html/body/form/input[3]" to "//*[@id="logon_submit"]/input", and the locator of "Admin" button from "//a[text()='Admin']" to "//a[text()='Rooms']".

III. BACKGROUND

We denote a web element as e. When the web application is updated, test cases may fail due to the inability to locate element e in the updated version. The automated UI test repair tool fixes the test case by relocating the element e through the element matching algorithm. Specifically, for an element e_o in the old version, the element matching algorithm needs to find the corresponding element e_n in the new version page, and the match result can be denoted as $\{e_o \mapsto e_n\}$. If there is no matching result for e_o, it will be considered a deleted element. Similarly, when there is no match result for e_n, it will be considered as an added element.

a) ***Existing element match method:*** The element matching algorithm has been proposed for many years. The element matching algorithm can be mainly divided into two categories, element-based, and tree-based approaches. For the element-based approach, the matching algorithm takes the old version element e_o and the element list E_n of the new version as input and outputs the matching result $\{e_o \mapsto e_n\}\texttt{where}\ e_n \in E_n$. Existing approaches check the similarity of elements by measuring the similarity of elements' attributes, e.g., WATER [7], elements' vision information, e.g., VISTA [2], and the combination of attributes and vision, e.g., WEBEVO [6]. While the DOM tree-based algorithm takes the old version Dom tree $\mathcal{T}_o$ and the new version DOM tree $\mathcal{T}_n$ as inputs, and outputs the matching result $\mathcal{M}$, which includes all matching relationships $\{e_o \mapsto e_n | e_o \in \mathcal{T}_o \wedge e_n \in \mathcal{T}_n\}$. The tree-based method can determine the final matching result by considering the relationship between elements. The SFTM is the tree-based matching method, which has shown good matching results.

b) ***Existing repair framework:*** Existing repair frameworks can be divided into offline mode and online mode. In the offline mode, such as WATER [7], information is collected during the execution and repairs are made after the execution. In online mode, such as VISTA [2], information is collected, and repairs are made at the runtime. Compared with offline mode, online mode can fix multiple breakages in a single run. To support online repair, VISTA utilizes Java Parser and regular expressions to parse the test cases into a list of ⟨locator, action, parameters⟩.Then it executes each action according

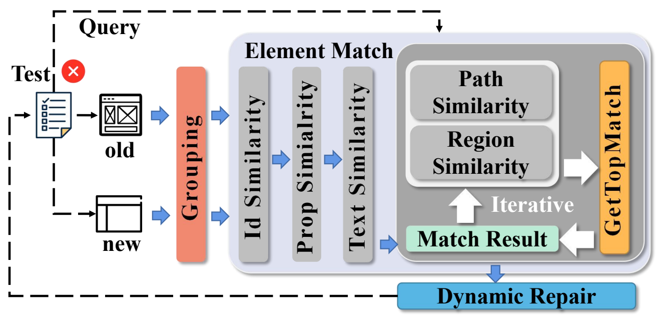


Fig. 2: The workflow of UITESTFIX.

to the sequences of the list. When an element of the new version is failed to be localized, VISTA automatically fixes the locator and executes the broken action again. However, due to complex test scenarios and free coding styles, VISTA may not be able to parse test cases using regular expressions, hence causing repair failures. For example, test cases written by the page-object design pattern [9] are difficult to parse by using regular expressions.

*c) **Definition of successful repair and successful match**:* To test a certain functionality, developers usually use assertion to determine if the execution result of the test case is correct. Therefore, a correctly fixed test case should meet two criteria: (1) the test execution can cover the original functionality, and (2) the test case assertion passes. In terms of successful element matching, due to element nesting, there may be more than one element implementing a certain function on a web page. Different algorithms' strategies may produce different and correct matching results, i.e., match one element to different elements that represent the same function. As these elements implement the same functionality, we consider the matching results of the elements to be a many-to-many relationship.

IV. METHODOLOGY

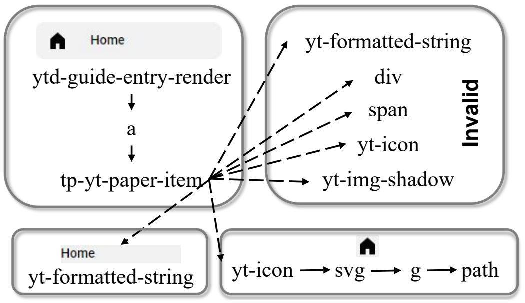
Figure 2 shows the overall workflow of UITESTFIX. UITESTFIX contains several components: grouping, similarity metrics, iterative matching and dynamic repair. When the element cannot be found by the original locator causing a crash, UITESTFIX takes the current web page and the corresponding web page in the old version as input, and then obtains the patch via grouping, calculating similarity and iterative matching. After replacing the broken locator with the correct one, the test can continue its execution where subsequently broken locators are fixed until the end of execution.

A. Grouping

To automatically fix broken UI test cases, the first step is to match the elements from old and new versions. However, if we use the original DOM as input directly, the following problems will arise: (1) the number of elements on web pages is usually quite high, which increases the complexity of element matching; (2) there are a lot of invisible elements, which may also affect the matching accuracy of visible elements; (3)

```
1.<ytd-guide-entry-renderer class="..." active
  ="">
2.  <a id="endpoint" class="yt-simple-endpoint">
3.    <tp-yt-paper-item class="style-scope">
4.      <yt-icon class="guide-icon">
5.        <svg viewBox="0 0 24 24">
6.          <g class="style-scope yt-icon">
7.            <path d="M4,3Z"></path></g></svg>
      </yt-icon>
8.      <yt-formated-string>Home</yt-formated-
  string>
9.      <yt-img-shadow hidden=""></yt-img-shadow
  >
10.     <yt-icon class="guide-entry-badge"></yt-
  icon>
11.     <span class="guide-entry-count"></span>
12.     <div id="newness-dot"></div></a> </ytd
  -...>
```

(a) A simplified HTML code of Youtube's Home button



(b) A gourping process for the YouTube's Home button.

Fig. 3: DOM Structure of YouTube's Home Buttons in 2023.

unable to distinguish which elements represent regions and which elements represent functionality (e.g., button).

For instance, Figure 3b shows the DOM structure of YouTube's home button in 2023 and its simplified HTML code (Figure 3a). This simple button contains 12 elements as labeled in Figure 3a. By deeply analyzing this HTML code snippet and its DOM structure, we observed this button is composed of three main components (1) button region (elements 1–3), (2) button icon (elements 4–7), and (3) button text (element 8). From the user's viewpoint, elements 9–12 are not visible, e.g., the size of element 10 is 0, and the test case could not execute the event in these elements. However, the invisible elements and duplicated elements cause the number of elements on the HTML page to exceed the actual number of visible components, which affects the matching results. To solve this problem, we filter invisible and merge duplicated elements into a group using the steps below:

1) Filter invisible elements: If an element has (1) width or height less than 5 (too small to be visible) or (2) `isDisplayed=false`.

2) Merge elements: If an element has only one visible child element, the child element will be merged into its parent element.

3) Merge properties: After merging, the parent element will have all properties of the merged child element. If the parent element has neither id nor text, it will use the id/text of the child element.

After grouping, we rebuild a new tree using the remaining elements. In Figure 3b, elements 1, 4, and 8 are eventually saved, where elements 4 and 8 are saved as children of element 1.

B. Similarity Metrics

In this subsection, we define the similarity metrics used to compute the similarity between the old and new elements.

Given two DOM trees $\mathcal{T}_o$ and $\mathcal{T}_n$, representing the old and new versions of web pages, we denote the element list of $\mathcal{T}_o$ as $\mathcal{E}_o$=$\{e_o^1, e_o^2, \dots\}$, and element list of $\mathcal{T}_n$ as $\mathcal{E}_n = \{e_n^1, e_n^2, \dots\}$. The attribute of a node e is denoted as $e.\texttt{id}$, $e.\texttt{class}$, $e.\texttt{text}$, etc. If e_o^i and e_n^j is determined as match, $e_o^i \mapsto e_n^j$ will be saved to matching list $\mathcal{M}$.

1) Id similarity. The most straightforward similarity metric is the exact matching of $n.\texttt{id}$. Since the id attribute of elements is usually unique in web pages, we define elements e_o^i and e_n^j match if (1) $e_o^i.\texttt{id}$ is exactly the same as $e_n^j.\texttt{id}$, and (2) $e_o^i.\texttt{id}$ is a unique id among $\mathcal{E}^o$, and $e_n^j.\texttt{id}$ is a unique id among $\mathcal{E}^n$.

Besides id similarity, we also used two existing similarity metrics:

2) Property similarity (S_{prop}). TF-IDF (Term Frequency–Inverse Document Frequency) is a widely used text similarity metric [10]. As some words in properties appear in many elements, these repeated words will not be able to distinguish elements (e.g., div). Therefore, TF-IDF can effectively use words with important meanings to calculate similarity. Compared to SFTM [8], which uses IDF, we use TF-IDF to incorporate the term frequency (TF). Using TF to normalize the final similarity helps to combine the property similarity with other similarities, and avoiding that the property similarity being large enough to cover other similarities. Moreover, SFTM [8] considers XPath, tag, and attribute as properties. According to our observation, text is more stable than XPath during page evolution. Hence, we use text, tag, and attribute to calculate S_{prop} (i.e., we replace XPath with text).

3) Text similarity (S_{text}). To calculate the similarity of text properties, we use Levenshtein distance [11], which performs well in existing tool WEBEVO [6] when texts have small changes. If the texts of elements are empty, text similarity will be left as 0. Otherwise, the text similarity is calculated as follows:

$$S_{text}(e_o^i, e_n^j) = edit\_distance(e_o^i.text, e_n^j.text)$$

Apart from the above three metrics, we propose two new metrics: *region similarity* and *path similarity*. These two metrics allow us to better utilize structural information for matching elements.

4) Path similarity. Based on the first idea **I1**, if two elements exist in the same region, they should have a higher similarity.

All ancestor nodes of n form a path from the root node to n. Given two nodes n_o^i and n_n^j, the closer the common ancestor in the path they have, the higher similarity they share. This is because a closer common ancestor indicates they are located in the same smaller region. The common ancestor of the two elements is determined according to the current matching results stored in $\mathcal{M}$. Note that the path similarity changes along with the change of $\mathcal{M}$ in the iteration phases, which will be explained in Section IV-C.

Formally, given two elements e_o^i and e_n^j, their *path similarity* S_p is defined as the length from the root node element to the deepest matching element, normalized with the total path length. Formally,

$$S_p(e_o^i, e_n^j, \mathcal{M}) = min(\frac{path(M_d(e_o^i))}{path(e_o^i) - 1}, \frac{path(M_d(e_n^j))}{path(e_n^j) - 1})$$

where $path(e)$ is the length of the path from the root node element to e, and $M_d(e_o^i) \mapsto M_d(e_n^j) \in \mathcal{M}$ is the deepest matched element in the paths of e_o^i and e_n^j, respectively.

5) Region similarity. For non-leaf node elements, we can use the second idea **I2** to improve its correct matching rate. The non-leaf node element can be considered as a region, and the similarity of two regions can be measured by the similarity of elements located in those two regions. Hence, we propose region similarity S_r, which calculates the similarity of regions (non-leaf node elements) using the matching result of leaf node elements in two regions, normalized by the total number of leaf node elements. Given two non-leaf node elements e_o^i and e_n^j, their region similarity is defined as:

$$S_r(e_o^i, e_n^j, \mathcal{M}) = \frac{|\{(e_1, e_2) | e_1 \in C(e_o^i) \wedge e_2 \in C(e_n^j) \wedge e_1 \mapsto e_2 \in \mathcal{M}\}|}{max(|C(e_o^i)|, |C(e_n^j)|)}$$

where $C(e)$ is the list of child leaf nodes of e. Similar to the path similarity, the region similarity also changes along with the change of $\mathcal{M}$ in the iteration phases.

C. Iterative Matching

Based on similarity metrics, we then present our iterative approach for optimizing the matching results. Algorithm 1 shows our iterative matching algorithm. The algorithm takes as inputs the element lists of the old and new version web pages and outputs the matching result. Firstly, our tool initializes the similarity for each pair of elements according to id, attribute and text similarity (lines 3-12). Specifically, if the ids of two nodes are exactly the same, we are confident of adding them to *sure* matching list $\mathcal{M}$, which will not be modified in the later stage. Otherwise, their initial similarity will be calculated based on attribute and text similarity (line 10).

The iterative matching algorithm mainly updates the similarity, and then updates the matching results iteractively. In t iterations (lines 13-31), the similarity of each pair is continuously updated according to path similarity at line 18. For non-leaf node elements, their similarity is also updated using region similarity (lines 19-21). As mentioned before, path and region similarity values are determined by current matching results (i.e., $\mathcal{M}_w$). In each iteration, after updating the similarities, all candidate matching pairs whose similarity is greater or equal to the threshold θ are selected and saved in

Algorithm 1: Iterative matching.

```
Input: $E_o$, $E_n$: element lists of old and new web page
       $t$: maximum number of iterations.
Output: $\mathcal{M}$: the matching results
1  $\mathcal{M}$={ };
2  InitialS=EmptyMap;
   /* initial similarity. */
3  for $e_1$ in $E_o$ do
4      for $e_2$ in $E_n$ do
5          if $e_1$.id == $e_2$.id then
6              $\mathcal{M} = \mathcal{M} \cup \{e_1 \mapsto e_2\}$;
7              $E_o = E_o - e_1$;   $E_n = E_n - e_2$;
8              break
9          end
10         InitialS[($e_1$, $e_2$)] = $\alpha * S_{prop}(e_1, e_2)$+ $S_{text}(e_1, e_2)$;
11     end
12 end
13 $\mathcal{M}_w$=$\mathcal{M}$;
   /* iterative match. */
14 for $i = 1$ to $t$ do
15     $S$=EmptyMatrix;
       /* Update similarity. */
16     for $e_1$ in $E_o$ do
17         for $e_2$ in $E_n$ do
18             S[($e_1$, $e_2$)] = InitialS[($e_1$, $e_2$)]+$\beta * S_p(e_1, e_2, \mathcal{M}_w)$;
19             if $IsNonLeaf(e_1) \wedge IsNonLeaf(e_2)$ then
20                 S[($e_1$, $e_2$)] += $\gamma * S_r(e_1, e_2, \mathcal{M}_w)$;
21             end
22         end
23     end
24     $\mathcal{M}_w = \mathcal{M}$;
       /* Update match result. */
25     $E \leftarrow \{(e_i, e_j, S[(e_i, e_j)]) \mid S[(e_i, e_j)] \geq \theta\}$;
26     for $e \in E$ do
27         $\{e_x \mapsto e_y\}$ = getMatchedPair($e$);
28         $\mathcal{M}_w = \{e_x \mapsto e_y\} \cup \mathcal{M}_w$;
29     end
30 end
31 return $\mathcal{M}_w$;
```

E at line 25. We choose to only consider one-to-one matching and use a greedy method to match them. In each iteration, we select the matching pair $\{e_x \mapsto e_y\}$ with the highest similarity from E at line 25 and add it to the matching result $\mathcal{M}$ at line 28. At the end of the loop, the matching results for this iteration are generated. The iteration terminates when the number of iterations reaches limit t or when the matching results will no longer change.

Note that the initial outputs of the matching algorithm are matching between element groups (refer to Section IV-A). To get the real element matching, the concrete elements are selected from the group according to the following rules: (1) if there is an element in the group with the same tag as the old version element, then that element is selected, (2) otherwise, the outermost element is selected.

D. Dynamic repair

Based on the generated matching results between the old and new versions, UITESTFIX then produces patches for fixing the broken UI tests. Specifically, for each test, UITESTFIX only works when UI actions are triggered. For each UI action performed on element e_o that fails to be executed, UITESTFIX will search for the matched element e_n by querying the matching result. If UITESTFIX finds e_o's matched element e_n, it will then executes the action on e_n. Otherwise, UITESTFIX terminates the test execution because the element may have been deleted or not exist on this page anymore. The repaired test cases are then validated by executing it again on the new version of the web page. Finally, UITESTFIX produces the repaired test cases.

V. IMPLEMENTATION

To handle complex industrial apps and dynamically repair test cases, we implement UITESTFIX, which mainly consists of three modules: instrumentation, element matching, and repair.

Instrumentation. The instrumentation module utilizes aspect-oriented programming (AOP) [12] instrumentation, adapted from VISTA [2], to instrument Selenium tests. The AOP instrumentation dynamically instruments the invocations of *WebDriver.findElement*, *WebElement.findElement*, *WebElement.click*, and other similar methods used for element locating and performing actions on elements. The inserted code records the trace of actions and corresponding UI status before/after each action, allowing for a detailed analysis of the actions performed.

Executed in the old version. After instrumenting the tests, UITESTFIX would execute the tests on the instrumented program to record its action traces. When a test is executed on the old version, for an action a, our instrumentation records the action id represented by $\langle l, m \rangle$, where l is the line number where action a is triggered (e.g., the line number of *element.click()*), and m represents the method name that triggers a. Meanwhile, before triggering a, our instrumentation also records the web page state s, a tree with element information (visibility, size, attributes, text, tag and XPath). Specifically, after executing a test with n actions on the old version, UITESTFIX generates a set of action traces in the form of $\{(\langle l_1, m_1 \rangle,\ s_1),\ \ldots,\ (\langle l_n, m_n \rangle,\ s_n)\}$.

Executed in the new version. Using the recorded action traces, the repair module fixes broken actions as follows:

1) *Check if the action needs to be fixed.* Executes the test action and uses try-catch to catch *NoSuchElementException* or *InvalidElementStateException*, which are two exceptions related to the element exception. If no exception is caught (i.e., normal test execution), the subsequent repair steps will not be performed. Otherwise, UITESTFIX uses the following steps to fix the broken events.
2) *Find the execution information of the event to be fixed in the old version.* When an exception is caught, the repair module first collects the line number and the method's name that triggers the broken action a_n. Then, it queries the action traces to determine the matched action a_o of the old version and the state s_o. Meanwhile, it also saves the new version's web page state s_n when encountering the broken action.
3) *Match element.* Use the approach presented in Section IV to match s_o and s_n.
4) *Repair the action online.* UITESTFIX further executes the crashed action on the matched element to achieve the

original purpose of crashed action. After the repaired action is executed, UITestFix saves the repair information as the generated patch.

After completing the execution of the test case, UITestFix will output a patched test case.

VI. Evaluation

We evaluate the effectiveness of UITestFix in web UI tests repair and answer the following research questions:

RQ1: How effective is UITestFix in UI element matching compared with existing methods?

RQ2: How efficient are different matching methods?

RQ3: How effective is UITestFix in test case repair compared with existing methods?

Selected baselines: We compared UITestFix with four baselines, including two repair tools (WATER [7], VISTA [2]), and two UI element matching algorithms (WebEvo [6], and SFTM [8]). We exclude the model-based tool [4] because the code in the GitHub repository fails to compile due to missing dependencies. WATER is a UI test repair tool that uses the similarity of DOM properties of elements for matching. VISTA is a repair tool that uses image processing and computer vision techniques by comparing visual information in two versions of the apps. WebEvo computes both text similarity and image similarity of the elements in two versions of the apps to match elements. The recently proposed algorithm, SFTM, assigns attribute scores to all potential matches between nodes in two versions and adjusts the scores using the parent-child relationships of nodes. Similar to VISTA [2], UITestFix performs online repair, which fixes broken tests during execution. On the other hand, WATER [7] uses an offline mode, which collects information during test execution and repairs after the execution. Therefore, we use the re-implemented version in VISTA [2] to evaluate WATER. For all other tools, we use their open-source implementations and parameters. The parameters of UITestFix in Algorithm 1 include: maximum iterations t=5, attribute similarity weight α=0.7, path similarity weight β=1.0, region similarity weight γ=2 and similarity threshold θ=1.2. All these parameters are tuned via grid searches for the best values.

UI Match Dataset: To evaluate the performance of different UI element matching algorithms, we use an extended WebEvo dataset [6] (EWD). The WebEvo dataset includes 13 web applications in total. We excluded the web pages *ClassDojo* and *Home Depot* because they could not be loaded correctly. To expand our dataset, we select the Top 15 web apps on Semrush [13] (which presents a ranked list of most visited websites) and remove web pages that overlap with the WebEvo dataset [6]. We use *Wayback Machine* [14] to obtain the historical web pages. In total, our expanded dataset contains 44 web pages of 22 web apps (with 11 apps from WebEvo and 11 extended apps).

Moreover, to show the ability of different element matching algorithms on large-scale industrial applications, we also include two industrial apps. To maintain anonymity, we omit the application and company name in the paper review process.

TABLE I: Statistics of our UI match dataset

Dataset	Add	Delete	Update	Total
EWD	301	218	1,101	1,620
IND	170	101	631	902
Total	471	319	1,732	2,522

We select two apps FabricInsight and CampusInsight, two big data analysis platforms, which are two core products of Huawei. To ensure sufficient changes are collected for our study of web UI elements evolution, we select web pages which contain sufficient changes. Specifically, we select web pages with more than 20 changed elements. In the end, we collected 13 web pages from two industrial apps. The types of selected web pages are diverse, which include home pages, pages for configuring system settings, data pages, and pages with charts.

Similar to prior work [6], we also consider three types of element changes: *Update*, *Add*, and *Delete*. Table I shows the statistics of our UI element matching dataset. In the UI element matching dataset, we try to label all the elements on the pages as much as possible. *The labeling process does not only label the change types, but also annotate the matching pairs between the old and new versions, which are then served as the ground truth of element matching.* However, some elements, such as content elements, are difficult to label. As a result, we made the decision to exclude them from the labeling process. Since VISTA only supports matching the innermost elements, to make a fair comparison of the performance of different algorithms, we only consider leaf nodes in our experiment. Totally, we labelled 2,522 changed elements.

UI Test Dataset: Since the WebEvo dataset does not have open-source test cases, we choose to evaluate the effectiveness of UITestFix using open-source and industrial test datasets. For the open-source tests, we use an existing test dataset [4]. Specifically, we include all web apps in prior work [4], except for the `Password Manager` app because the provided link for this app is no longer available for download. For the original test case dataset, we mainly made two modifications, (1) remove test cases without assertion statements or repeated test cases, as the assertions are used as the correctness specification for UITestFix, and (2) insert necessary UI actions (e.g., login) so that test cases can be executed independently. To avoid the problem caused by missing data (such as username does not exist or password is incorrect), we first insert some SQL statements to prepare the data for test execution (e.g., create user and save username/password information into the database). The selected two industrial apps FabricInsight and CampusInsight have 5,839 test cases in total. Conducting experiment on all the tests is too time-consuming, since executing these tests in parallel will take two or three days in industrial environment. Hence, we randomly select 250 test cases (the number is aligned with the existing work [4]) to execute. We select the test cases that must be repaired, regardless of the number of pages used by these tests. We

TABLE II: The original and modified versions of the selected web pages

Label	Application	Original Version	Modified Version
I	AddressBook	4.0	6.1
II	Claroline	1.10.7	1.11.5
III	Collabtive	0.65	1.0
IV	MantisBT	1.1.8	1.2.0
V	MRBS	1.2.6.1	1.4.9

then exclude 46 tests as they require configuring a specific test environment. Table II shows the original and the modified versions of the selected web pages. The open-source dataset, following settings from the provided GitHub repository in the prior work [4], involves changes across multiple versions. Note that fixing tests across multi-version changes may not be realistic. Hence, we use single-version changes for the industrial dataset, which is practical in an actual industrial environment.

During the classification of the matching results (RQ1) and the generated fixes (RQ3), two authors of the paper independently analyzed them by inspecting the matched UI elements to check if the matching was correct and if it led to any newly thrown exception. Then, they met and discussed resolving any disagreements.

All experiments for the open-source apps are run on a computer with an Intel Core i5 processor (2.3GHz) and 24 GB RAM, whereas the experiments for the industrial apps are conducted on a computer with an Intel Core processor (3.20GHz) and 32GB RAM.

A. *[RQ1] Effectiveness on Matching Elements*

Table III summarizes the evaluation results on web element matching. For each algorithm, the table lists the correct number and accuracy of different types of changes in the extended WebEvo dataset (EWD) and industrial dataset (IND) . The "Total" row summarizes the total number of corrects and the overall accuracy of different algorithms. Overall, UITESTFIX achieves the best result on all types of changes (Add, Delete and Update) for both open-source and industrial datasets.

More specifically, UITESTFIX is more accurate than all other evaluated tools, with the overall success matching rate being 84.9%, while the best result of existing tools is only 68.8%. UITESTFIX produces better results than existing tools mainly because our optimization approaches iteratively improve the matching results based on region and path similarity, while other tools produce matching results directly. It is worth noting that tree-based matching algorithms (SFTM, UITESTFIX) have significantly higher accuracy in identifying added and deleted elements than other matching algorithms. For instance, among 170 added elements from the industrial dataset, SFTM successfully matches 133 of them with 78.2% accuracy, while the accuracy of WATER, VISTA, and WEBEVO is 16.5%, 4.7%, and 5.9%, respectively. UITESTFIX even achieves better results, and its accuracy is 83.5%. This is because the tree-based matching algorithm matches elements by referring to parent-child relations, which has a stronger ability to identify added and deleted elements. In terms of deleted elements, UITESTFIX outperforms SFTM (76.2% vs 31.7%) mainly because UITESTFIX can also consider the relations of sibling nodes in the iterative, while SFTM only measure the parent-child relations.

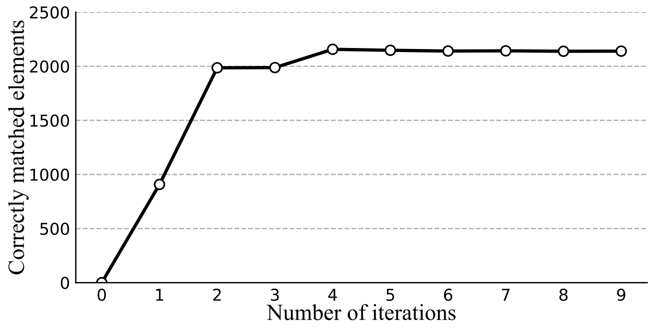


Fig. 4: The accuracy rate varies with the number of iterations.

We further evaluate the effectiveness of the iterative algorithm. Figure 4 presents the number of correctly matched elements during the iterative process. The x-axis shows the number of iterations, while the y-axis displays the count of correctly matched elements. It is evident from the graph that the matching accuracy increases progressively with each iteration. In the first iteration, only 908 elements were correctly matched, while the following three iterations successfully matched 1986, 1988, and 2157 elements. However, after the fourth iteration, there is no significant improvement in the matching accuracy. Therefore, we decided to set the maximum iterations to five. The results indicate that the iterative process significantly enhances the matching accuracy.

B. *[RQ2] Efficiency on Matching Elements*

To compare the efficiency of different algorithms, we calculate the average time taken by each algorithm to generate final matching results. We use the saved web page for matching to avoid inaccuracy caused by the network (e.g., page loading latency).

The "AVG Time" row in Table III shows the average time taken (in seconds) by different algorithms to generate matching results across all web pages. Among all evaluated tools, WATER is the fastest in producing matching results (0.0003s) because it matches elements by computing similarities of DOM properties which is very efficient. WEBEVO and VISTA are slower than WATER since they rely on time-consuming image similarities check. WEBEVO is much faster than VISTA since it uses a fast image similarity algorithm (aHash). UITESTFIX and SFTM take similar time to generate matching results (0.97s and 0.87s). Compared to WATER and WEBEVO, UITESTFIX is slower mainly because it requires multiple iterations for optimizing the matching results. With improved accuracy, we believe it is acceptable to take <1s to generate matching results for each web page.

TABLE III: Accuracy of UI element matching for all evaluated approaches

Modified Type	WATER		VISTA		WEBEVO		SFTM		UITESTFIX	
	EWD	IND	EWD	IND	EWD	IND	EWD	IND	EWD	IND
Add	4 (1.3%)	28 (16.47%)	30 (10.0%)	8 (4.7%)	87 (28.9%)	1 (5.9%)	162 (53.8%)	133 (78.2%)	**170 (56.5%)**	**142 (83.5%)**
Delete	11 (5.0%)	1(1.0%)	49 (22.5%)	5 (5.0%)	44 (20.2%)	0 (0%)	100(45.9%)	32 (31.7%)	**111 (50.9%)**	**77 (76.2%)**
Update	500(45.4%)	344(54.52%)	651(59.1%)	434(68.78%)	803(72.9%)	278(44.1%)	808(73.4%)	500(79.2%)	**1036(94.1%)**	**605(95.88%)**
Total	515(31.8%)	373(41.35%)	730(45.1%)	447(49.6%)	934(57.7%)	279(30.93%)	1070(66.1%)	665(73.3%)	**1317(81.3%)**	**824(91.4%)**
	888 (35.2%)		1177 (46.7%)		1213(48.1%)		1735 (68.8%)		**2141 (84.9%)**	
AVG Time	**0.0003s**		7.7623s		0.0799s		0.9726s		0.8685s	

TABLE IV: UI test repair effectiveness

Application	Total	ΔV	WATER*	VISTA*	WEBEVO*	SFTM*	UITESTFIX
AddressBook	8	30	1	**2**	0	**2**	**2**
Claroline	29	7	0	**24**	0	0	**24**
Collabtive	5	4	**3**	1	2	0	**3**
MantisBt	38	1	18	4	19	6	**24**
MRBS	24	12	0	**4**	0	2	**4**
FabricInsight	46	1	36	33	37	40	**43**
CampusInsight	16	1	6	5	6	11	**13**
Total	166	-	64(39%)	73(44%)	64(39%)	61(37%)	**113(68%)**

C. [RQ3] Effectiveness on Repairing UI Tests

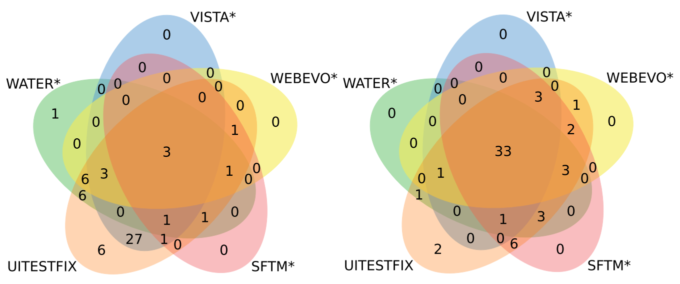
To fairly compare the capabilities of different algorithms in repairing broken UI tests, we integrated the matching algorithm of WATER, VISTA, WEBEVO and SFTM into our repair framework. We named them WATER*, VISTA*, WEBEVO* and SFTM*, respectively.

Successful repair. We consider a test case to be successfully fixed if (1) the test passes after the repair and (2) each action in the test has been correctly executed (we verify this by manually comparing the screenshots of the traversed web pages).

The dataset contains 135 test cases in open-source apps and 204 test cases in industrial apps. After excluding 163 non-crashing test cases (which do not need to be repaired) and 10 test cases that cannot be repaired due to the tested modules being removed, we obtained 62 crashing test cases in open-source apps and 104 crashing test cases in industrial apps (166 test cases in total).

Table IV presents the test case repair results of different algorithms. For each application, the table reports the total number of test cases (Total) and the number of test cases successfully repaired by different algorithms. Column ΔV shows the number of released versions between the original and new versions. As shown in Table IV, UITESTFIX achieves the best results across all applications. In total, UITESTFIX successfully fixes 113 broken test cases with a repair rate of 68%, while the best existing tool (i.e. VISTA*) only fixes 73 of them with a repair rate of 44%. Moreover, we observe that the larger ΔV is, the more changes introduced to the new versions since the original version, the more challenging to fix the broken tests. For instance, for MRBS, the best tool only fix 4 of 24 broken test cases.

The number of successful repairs generated by the same approach differs across apps. WATER and WEBEVO performed well on MantisBt but did not repair any test cases on Claroline. SFTM performed well on both FabricInsight and CampusInsight but did not work well for the five open-source applications. This is mainly due to the different element change types of different applications. For instance, on the login web page of MantisBt, the button with the text "Enter" is transformed into the input with the value attribute "Enter". VISTA is weak at distinguishing the elements with text, leading to the "Enter" button matched with the "Log out" button, resulting in test cases that cannot be repaired. Even so, the repair rate of UITESTFIX on different applications is the same or higher than all existing algorithms, which shows the high robustness of UITESTFIX.



(a) Open-source test cases (b) Industry test csaes

Fig. 5: Distribution of test cases repaired by different algorithms.

Figure 5 shows a Venn diagram of repaired test cases of WATER*, VISTA*, WEBEVO*, SFTM* and UITESTFIX in the open-source and the industrial test dataset. The green, blue, pink, yellow, and orange represent the number of test cases repaired by WATER*, VISTA*, WEBEVO*, SFTM* and UITESTFIX, respectively. There are 8 test cases in the open-source dataset that can only be fixed by UITESTFIX, while there are 2 test cases in the industrial dataset that can only be fixed by UITESTFIX. On the other hand, there is only one test case from the open-source dataset that cannot be fixed by UITESTFIX but fixed by other tools (WATER). The test case, that UITESTFIX failed to fix, needs to click a series of actions and then click an "add task" button. After the web page update, UITESTFIX failed to match the "add task" button because this button was removed from the original page. In contrast, WATER matches this button with an "add task" button from a different page. These two buttons are matched by WATER just because they simply have the same text. Since the matched "add task" button also completes the function of adding tasks and passes the assertion checking, this test is considered to be repaired successfully by WATER. This situation happens

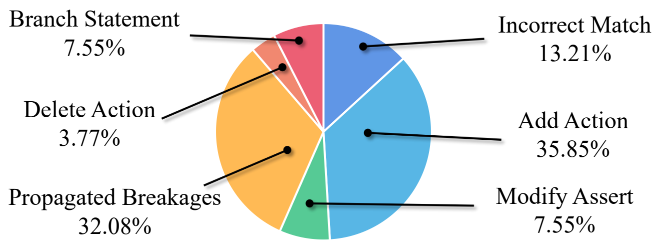


Fig. 6: Reasons behind the repair failures of UITestFix.

relatively rarely (only once in the experiment). We also noticed that some test cases could only be correctly fixed by a specific algorithm other than UITestFix. For example, in the open-source test cases, there are 27 test cases that can only be fixed by VISTA and UITestFix, while in the industrial dataset, there are 6 test cases that can only be fixed by SFTM and UITestFix. This means that although different algorithms have their own advantages, UITestFix has a stronger ability to fix test cases.

To better understand the capability of UITestFix for future improvement, we manually analyze the 53 broken test cases which UITestFix fails to repair. Specifically, for each case, we categorize the reason behind the test failure by comparing the correct repair with the algorithm's repair results. Figure 6 shows the distribution of the six reasons behind the failure of UITestFix.

1) *Add Action.* Add Action denotes that an action needs to be added to repair the test.
2) *Delete Action.* Delete Action denotes that an action needs to be deleted to repair the test.
3) *Modify Assert.* Modify Assert means that the expected value of the assertion needs to be updated in the test.
4) *Branch Statement.* Some test cases use some branch statements to enhance the stability of test cases, such as using if-else statements to handle the empty table data and the case of the table data not being empty. The branch statements executed in the new version may not have been executed in the old version, causing the inability to repair the test case in the new version due to the lack of action information. We label this reason of failure as Branch Statement.
5) *Propagated Breakages* [15]. Propagated Breakages refer to the scenario where the element is pointing to an incorrect element using the original locator without throwing an immediate exception but triggers a subsequent crash.
6) *Incorrect Match.* The test case can be fixed by relocation, but the algorithm matches incorrectly or cannot find the matched element.

As only 7 of them are caused by Incorrect Match (13.21%), and UITestFix can repair 113 (94.16%) test cases requiring locator fixes, and this shows that UITestFix performs well in repairing broken locators. This indicates that for the relocation type of repairs, UITestFix already has a strong capability. Therefore, in the future, we will focus on the remaining types of repair failures.

VII. Discussion

Open dataset for web UI maintenance. Although many techniques have been proposed for UI element matching and repairing broken UI tests in the past decade, we notice that there is still a lack of labelled web UI open dataset. Having a community-driven dataset is an important direction to encourage future research in studying and automating the repair of broken web UI tests. Although we could not share our industrial dataset due to the policy in Huawei, we have manually labelled the UI web page dataset (the expanded WebEvo [6] dataset), and released it as an open dataset as an initiative towards this direction.

Types of repairs in broken UI test cases. As most prior techniques focus on fixing broken locators, this further motivates us to improve over existing techniques by designing UITestFix, a more accurate technique for fixing broken locators in web UI tests. Although as shown in Figure 6, there are still 120 out of 166 test cases can be fixed through relocation, of which UITestFix currently cannot fix 7. However, the remaining 46 test cases involving actions such as addition, deletion, propagation of crashes, and branch statements are worth further investigation. It is worthwhile to design techniques that specialize in repairing broken UI events in the future.

Importance of DOM structure information. Prior techniques in matching UI elements usually use (1) attribute information (e.g., WATER [7]), (2) visual information (i.e., VISTA [2]), and (3) structure information (i.e., SFTM [8]). Our evaluation shows that approaches that use structural information for repairing broken locators are more accurate than those that do not. This indicates the importance of structural information in improving the accuracy of matching UI elements. Although our approach uses textual information (e.g., id) for matching UI elements, relying merely on textual information leads to inaccurate results as textual information is often subject to changes. Compared to using structural information, our results show that VISTA that relies on visual information takes the longest time due to the overhead in matching elements via image processing and computer vision. Meanwhile, matching using visual information also relies heavily on the accuracy of image processing and computer vision, which is less mature than textual matching used in structural and textual information. For example, in Figure 1a, although the general structures of the old and new MRBS pages are similar, VISTA fails to match correctly due to visual changes.

Iterative matching. The key idea behind UITestFix is an iterative approach that performs multiple rounds of matching to increase the similarities of weakly matched elements. As our experiment shows that our matching algorithm leads to improved matching accuracy, and the algorithm can be theoretically applied for any task that requires matching of UI elements, in the future, it is worthwhile to study using an iterative matching approach for tasks beyond the repair of broken tests (i.e., compatibility testing [16]).

Threats to Validity. Internal threats arise primarily from potential errors in labeling and variations in element change definitions. To address this concern, two authors independently checked and confirmed the accuracy of the annotations, ensuring a higher level of reliability. External threats originate from two main sources. Firstly, the flakiness of test cases can lead to unstable repair results. To minimize the impact of flakiness on the algorithm's repairs, we closely monitor the execution process by capturing screenshots of the executed elements. If any issues arise due to external factors, such as network instability, we reprocess the repair procedure accordingly. Another concern is the generalization of experimental results. We conducted evaluations on limited web pages and test cases. However, our datasets consist of both open-source and industrial applications. We extensively tested changed elements across various types of famous pages. These measures contribute to the reliability and credibility of our experimental findings.

VIII. Related Work

Studies on web UI test maintenance. Several studies investigated the characteristics of UI test maintenance for industrial web apps [17], [9], [18]. Prior studies of an industrial app (i.e., the Learning Content Management System) revealed that ID-based tests required less maintenance effort than the XPath-based ones [17], and recommended using page object pattern for improving test maintenance [9]. Meanwhile, IBM researchers described their experience of using ATA, an approach based on natural-language processing and backtracking exploration to automatically convert manual test steps in one enterprise web app to test cases for improving test automation [18]. We have analyzed the reasons why the UITestFix cannot repair some test cases, and these reasons will guide future improvements in automated program repair.

Automatic UI Test Case Repair. In recent years, automated program repair [19], [20], [21], [22], [23], [24], [25] has been widely explored to fix many kinds of bugs. Meanwhile, to help testers fix broken UI test cases, many techniques have been proposed [7], [1], [26], [27], [28], [2], [29], [4], [3], [30], [31], [5], [32]. Most existing approaches fix broken tests by replacing outdated locators [7], [2], [29], [3], [30], [31]. The number of successful repairs usually depends on the correct matching rate and can only fix crashes caused by locator changes. Several model-based approaches are proposed for fixing broken UI tests for Java apps [33], [34], web apps [5], [4], and Android apps [28], [26], [27]. To repair crashes in broken tests, these approaches usually rely on building a model (e.g., Event Flow Graph) and then picking a new path from the model to replace. However, it is not easy to obtain accurate app models in real-world situations. Our experiment did not compare with any model-based approaches as they either fail to run [4] or are not publicly available [5]. Instead of fixing broken locators, a multi-locator was proposed to strengthen test cases by selecting the best locator to use [35]. Our technique is inspired by the newly proposed SFTM [8] matching algorithm, and our repair framework is adapted from the framework in VISTA [2]. Compared to these approaches, our approach improves the accuracy of matching UI elements, leading to more correct repairs.

Matching UI elements. UI element matching are used for automated maintenance of test cases [29], [6], [7], [2], automated compatibility testing [16], and test reuse [36]. WATER [7] relies on attribute information for matching . Meanwhile, more recent approach approaches [16], [32], [6], [31], [29] use a combination of attributes and visual information to match elements. VISTA [2] is a representative tool that focuses on matching elements using visual information, whereas Yoon et al. leverage word and layout embedding for matching elements [30]. Our iterative matching algorithm is most closely related to the SFTM algorithm [8], which utilizes structural information and attributes for matching. Nevertheless, our experiments show that UITestFix outperforms all existing tools in matching accuracy.

IX. Conclusions

In recent years, the automatic repair of web UI test cases has gained widespread attention. However, existing methods do not fully utilize structural information, resulting in lower accuracy when matching elements with significant changes. In this paper, we proposed UITestFix, an approach based on a novel iterative matching algorithm for fixing broken UI tests. UITestFix can use the existing matching results to update the similarity and adjust it during the iteration process. Therefore, it can use high-similarity elements to help match low-similarity elements. Our evaluation of publicly available and industrial datasets shows that UITestFix outperforms four prior approaches in producing more accurate matching and correct repairs. We analyzed the test cases in which UITestFix failed to repair, and the results showed that UITestFix could repair 94.16% of the test cases that only require relocation.

X. DataSet

Since this is a project of business company, because of the copyright issues, we are not allowed to open source the code for UITestFix. To make our results to be replicated, our dataset and experimental data are available at https://anonymous.4open.science/r/Web-UI-Dataset-9645.

XI. Acknowledgements

We greatly appreciate the suggestions and effort provided by my MS thesis advisor, Shin Hwei Tan. Additionally, Zhuolin Xu's assistance in conducting certain experiments was invaluable. This work partly supported by Huawei (GUI Automation Testing Technology Collaboration Project - Automated Repair of Test Cases for adapting to UI changes), and the National Natural Science Foundation of China (Grant No. 62202026, 62141209).

References

[1] M. Hammoudi, G. Rothermel, and A. Stocco, "Waterfall: An incremental approach for repairing record-replay tests of web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016, p. 751–762. [Online]. Available: https://doi.org/10.1145/2950290.2950294

[2] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 503–514.

[3] S. Brisset, R. Rouvoy, L. Seinturier, and R. Pawlak, "Erratum: Leveraging flexible tree matching to repair broken locators in web automation scripts," *Information and Software Technology*, vol. 144, p. 106754, 2022.

[4] J. Imtiaz, M. Z. Iqbal *et al.*, "An automated model-based approach to repair test suites of evolving web applications," *Journal of Systems and Software*, vol. 171, p. 110841, 2021.

[5] W. Chen, H. Cao, and X. Blanc, "An improving approach for dom-based web test suite repair," in *Web Engineering*. Springer International Publishing, 2021, pp. 372–387.

[6] F. Shao, R. Xu, W. Haque, J. Xu, Y. Zhang, W. Yang, Y. Ye, and X. Xiao, "Webevo: taming web application evolution via detecting semantic structure changes," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 16–28.

[7] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011, pp. 24–29.

[8] S. Brisset, R. Rouvoy, L. Seinturier, and R. Pawlak, "Sftm: Fast matching of web pages using similarity-based flexible tree matching," *Information Systems*, vol. 112, p. 102126, 2023.

[9] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 108–113.

[10] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[11] E. Ristad and P. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th European Conference Object-Oriented Programming (ECOOP'97)*. Springer, 1997, pp. 220–242.

[13] "Semrush," https://www.semrush.com/blog/most-visited-websites/, 2008, accessed: 2022-12-16.

[14] "Wayback machine," https://archive.org/web/, 2014, accessed: 2022-12-20.

[15] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?" in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 180–190.

[16] Y. Ren, Y. Gu, Z. Ma, H. Zhu, and F. Yin, "Cross-device difference detector for mobile application gui compatibility testing," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2022, pp. 253–260.

[17] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Repairing selenium test cases: An industrial case study about web page element localization," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 487–488.

[18] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, and S. Kumar, "Efficient and change-resilient test automation: An industrial case study," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1002–1011.

[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[20] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[21] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

[22] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 187–198.

[23] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[24] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–27, 2021.

[25] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 1–37, 2018.

[26] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "Atom: Automatic maintenance of gui test scripts for evolving mobile applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 161–171.

[27] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-based test script maintenance for android apps," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 215–225.

[28] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "Sitar: Gui test script repair," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 170–186, 2016.

[29] T. Xu, M. Pan, Y. Pei, G. Li, X. Zeng, T. Zhang, Y. Deng, and X. Li, "Guider: Gui structure and vision co-guided test script repair for android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 191–203.

[30] J. Yoon, S. Chung, K. Shin, J. Kim, S. Hong, and S. Yoo, "Repairing fragile gui test cases using word and layout embedding," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 291–301.

[31] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[32] H. Kirinuki, H. Tanno, and K. Natsukawa, "Color: correct locator recommender for broken test scripts using various clues in web application," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 310–320.

[33] A. M. Memon, "Automatically repairing event sequence-based gui test suites for regression testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 2, pp. 1–36, 2008.

[34] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving GUI applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, Lugano, Switzerland, Jul. 2013, pp. 45–55.

[35] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[36] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of gui events for test reuse: Are we there yet?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 177–190. [Online]. Available: https://doi.org/10.1145/3460319.3464827