Automated Repair of Mobile Friendly Problems in Web Pages

Sonal Mahajan
University of Southern California, USA

Negarsadat Abolhassani
University of Southern California, USA

Phil McMinn
University of Sheffield, UK

William G. J. Halfond
University of Southern California, USA

ABSTRACT

Mobile devices have become a primary means of accessing the Internet. Unfortunately, many websites are not designed to be mobile friendly. This results in problems such as unreadable text, cluttered navigation, and content overflowing a device's viewport; all of which can lead to a frustrating and poor user experience. Existing techniques are limited in helping developers repair these mobile friendly problems. To address this limitation of prior work, we designed a novel automated approach for repairing mobile friendly problems in web pages. Our empirical evaluation showed that our approach was able to successfully resolve mobile friendly problems in 95% of the evaluation subjects. In a user study, participants preferred our repaired versions of the subjects and also considered the repaired pages to be more readable than the originals.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**;

KEYWORDS

Mobile Friendly Problems, automated repair, web apps

ACM Reference Format:
Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G. J. Halfond. 2018. Automated Repair of Mobile Friendly Problems in Web Pages. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering (ICSE '18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180262

1 INTRODUCTION

Mobile devices have become one of the most common means of accessing the Internet. In fact, recent studies show that for a significant portion of web users, a mobile device is their primary means of accessing the Internet and interacting with other web-based services, such as online shopping, news, and communication [12, 18, 19, 40]. Unfortunately, many websites are not designed to gracefully handle users who are accessing their pages through a non-traditional sized device, such as a smartphone or tablet. These problematic sites may exhibit a range of usability issues, such as unreadable text, cluttered navigation, or content that overflows the device's viewport and forces the user to pan and zoom the page in order to access content. Such usability issues are collectively referred as *mobile friendly problems* [4, 15] and lead to a frustrating and poor user experience.

Despite the importance of mobile friendly problems, they are highly prevalent in modern websites — in a recent study over 75% of users reported problems in accessing websites from their mobile devices [19]. Over one third of users also said that they abandon mobile unfriendly websites and find other websites that work better on mobile devices. This underscores the importance for developers in ensuring the mobile friendliness of the web pages they design and maintain. Adding to this motivation is the fact that, as of April 2015, Google has incorporated mobile-friendliness as part of its ranking criteria when returning search results to mobile devices [17]. This means that unless a website is deemed to be mobile friendly, it is less likely to be highly ranked in the results returned to users.

Making websites mobile friendly is challenging even for a well motivated developer. These challenges arise from the difficulties in detecting and repairing mobile friendly problems. To detect these problems, developers must be able to verify a web page's appearance on many different types and sizes of mobile devices. Since the scale of testing required for this is generally quite large, developers often use mobile testing services, such as BrowserStack [6] and SauceLabs [39], to determine if there are problems in their sites. However, even with this information it is difficult for developers to improve or repair their pages. The reason for this is that the appearance of web pages is controlled by complex interactions between the HTML elements and CSS style properties that define a web page. This means that to fix a mobile friendly problem, developers must typically adjust dozens of elements and properties while at the same time ensuring that these adjustments do not impact other parts of the page. For example, a seemingly simple solution, such as increasing the font size of text or the margins of clickable elements, can result in a distorted user interface that is unlikely to be acceptable to end users or developers.

Existing approaches are limited in helping developers to detect and repair mobile friendly problems. For example, the Mobile Friendly Test Tools produced by Google [15] and Bing [4], only focus on the detection of mobile friendly problems in a web page. While these tools may provide hints or suggestions as to how to repair the pages, the task of performing the repair is still a manual effort. Developers may also use frameworks, such as Bootstrap and Foundation, to help create pages that will be mobile friendly. However, the use of frameworks cannot guarantee the absence of mobile-friendly problems [1]. Some commercial websites attempt to automate this process (e.g., [5, 11, 29]), but are generally targeted for hobbyist pages as they require the transformed website to use

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180262

one of their preset templates. This leaves developers with a lack of automated support for repairing mobile friendly problems.

To address this problem we designed an approach to automatically generate CSS patches that can improve the mobile friendliness of a web page. To do this our approach builds graph-based models of the layout of a web page. It then uses constraints encoded by these graphs to find patches that can improve mobile friendliness while minimizing layout disruption. To efficiently identify the best patch, our approach leverages unique aspects of the problem domain to quantify metrics related to layout distortion and parallelize the computation of the solution. We implemented our approach in a prototype tool, *M*Fix, and evaluated its effectiveness on the home pages of 38 of the Alexa Top 50 most visited websites. The results showed that our approach could effectively increase the mobile friendliness ratings of a page, typically by 33%, while minimizing layout distortion. Our approach was also fast, needing less than 5 minutes, on average, to generate the CSS patch. We also evaluated our results with a user study, in which participants overwhelmingly preferred the repaired version of the website for use on mobile devices, and also considered the repaired page to be more readable than the original. Overall, these results are very positive and indicate that our approach can help developers to improve the mobile friendliness of their web pages.

The contributions of our paper are as follows:

(1) A technique for automatically generating CSS based patches to improve the mobile friendliness of a web page
(2) An empirical study on popular websites that shows the approach's effectiveness at improving mobile friendliness scores while maintaining the original pages' layout.
(3) A user study that shows that pages patched by our approach are preferred for mobile usage and are rated as more readable.

Our paper is organized as follows. In Section 2, we provide background information about typical mobile friendly problems and solutions. Our approach is presented in Section 3. We evaluated our approach on popular websites and conducted a user study of the results, which are described in Section 4. We discuss related work in Section 5, and conclude in Section 6.

2 BACKGROUND

In this section we discuss a variety of mobile friendly problems and current ways of addressing them in order to build a mobile friendly website.

2.1 Types of Mobile Friendly Problems

Widely used mobile testing tools provided by Google [15] and Bing [4] report mobile friendly problems in five areas:

1. Font sizing: Font sizes optimized for viewing a web page on a desktop are often too small to be legible on a mobile device, forcing users to zoom in to read the text, and then out again to navigate around the page.

2. Tap target spacing: "Tap targets" are elements on a web page, such as a hyperlinks, buttons, or input boxes, that a user can tap or touch to perform actions, such as navigate to another page or fill and submit a form. If tap targets are located close to each other on a mobile screen, it can become difficult for a user to physically select the desired element without hitting a neighboring element accidentally. Targets may also be too small, requiring users to zoom into the page in order to tap them on their device.

3. Content sizing: When a web page extends beyond the width of a device's viewport, the user is required to scroll horizontally or zoom out to access content. Horizontal scrolling is particularly considered problematic since users are typically used to scrolling vertically but not horizontally [20]. This can lead to important content being missed by users. Therefore attention to content sizing is particularly important on mobile devices, where a smaller screen means that space is limited, and the browser may not be resizable to fit the page.

4. Viewport configuration: Using the "`meta viewport`" HTML tag allows browsers to scale web pages based on the size of a user's device. Web pages that do not specify or correctly use the tag may have content sizing issues, as the browser may simply scale or clip the content without adjusting for the layout of the page.

5. Flash usage: Flash content is not rendered by most mobile browsers. This makes content based on Flash, such as animations and navigation, inaccessible.

In our approach, detailed in Section 3, we focus on addressing the first three of these problems. We regard the Flash usage as out of scope for our approach, since it requires a major content change in the page; while the viewport configuration problem is trivial to address, as it only requires insertion of a missing "`meta viewport`" tag into the page's HTML head.

2.2 Current Methods of Addressing Mobile Friendly Problems

There are a number of ways in which a website can be adjusted to become more mobile friendly. In the early days of mobile web browsing, a common approach was to simply build an alternative mobile version of an existing desktop website. Such websites were typically hosted at a separate URL and delivered to a user when the web server detected the use of a mobile device. However, the cost and effort of building such a separate mobile website was high. To address this problem, commercial services, such as bMobilized [5] and Mobify [29], can automatically create a mobile website from a desktop version using a series of pre-designed templates. A drawback of these templated websites, however, is that they fail to capture the distinct design details of the original desktop version, making them look identical to every other organization using the service. Broadly speaking, although having a separate mobile website could address mobile friendly concerns, it introduces a heavy maintenance debt on the organization in ensuring that the mobile website renders and behaves consistently and as reliably as its regular desktop version, thereby doubling the cost of an organization's online presence. Furthermore, having a separate mobile-only site would not help improve search-engine rankings of the organization's main website, since the two versions reside at different URLs.

To avoid developing and maintaining separate mobile and desktop versions of a website, an organization may employ *responsive design* techniques. This kind of design makes use of CSS media queries to dynamically adjust the layout of a page to the screen size on which it will be displayed. The advantage of this technique

over mobile dedicated websites is that the URL of the website remains the same. However, converting an existing website into a fully responsive website is an extremely labor intensive task, and is better suited for websites that are being built from scratch. As such, repairing an existing website may be a more cost effective solution than completely redeveloping the site. Furthermore, although a responsive design is likely to allow for a good mobile user experience, it does not necessarily preclude the possibility of mobile friendly problems, since additional styles may be used or certain provided styles may be incorrectly overridden [1].

Our approach introduces a novel technique for handling mobile friendly problems by adjusting specific CSS properties in the page and producing a *repair patch*. The repair patch uses CSS media queries to ensure that the modified CSS is only used for mobile viewing – that is, it does not affect the website when viewed on a desktop.

3 APPROACH

The goal of our approach is to automatically generate a patch that can be applied to the CSS of a web page to improve its mobile friendliness. Our technique addresses the three specific problem types introduced in Section 2, namely font sizing, tap target spacing, and content sizing for the viewport – factors used by Google to rate the mobile friendliness of a page.

There is usually a straightforward fix for these problems – simply increase the font size used in the page and the margins of the elements within it. The result, however, is one that would likely be unacceptable to an end-user: such changes tend to significantly disrupt the layout of a page and require the user to perform excessive panning and scrolling. The challenge in generating a successful repair, therefore, involves balancing two objectives – addressing a page's mobile friendliness problems, while also ensuring an aesthetically pleasing and usable layout.

With this in mind, the goal of our technique is to generate a solution that is as faithful as possible to the page's original layout. This requires fixing mobile friendliness problems while maintaining, where possible, the relative proportions and positioning of elements that are related to one another on the page (for example, links in the navigation bar, and the proportions of fonts for headings and body text in the main content pane).

Our approach for generating a CSS patch can be roughly broken down into three distinct phases, *segmentation*, *localization*, and *repair*. These are shown in Figure 1. The input to our approach is the URL of a page under test (PUT). Typically, this would be a page that has been identified as failing a mobile friendly test (e.g., Google's [15] or Bing's [4]), but it may also be a page for which a developer would like to simply improve mobile friendliness. The segmentation phase identifies elements that form natural visual groupings on the page – referred to as *segments*. The localization phase then identifies the mobile friendly problems in the page, and relates these to the HTML elements and CSS properties in each segment. The last phase – repair – seeks to adjust the proportional sizing of elements within segments, along with the relative positions of each segment and the elements within them in order to generate a suitable patch. We now explain each of these three phases in more detail.

3.1 Phase 1: Segmentation

The first phase analyzes the structure of the page to identify *segments* – sets of HTML elements whose properties should be adjusted together to maintain the visual consistency of the repaired web page. An example of a segment is a series of text-based links in a menu bar where if the font size of any link in the segment is too small, then all of the links should be adjusted by the same amount to maintain the links' visual consistency. The reason our approach uses segments is that through manual experimentation with pages that contained mobile friendly problems, we found that once we identified the optimal fix value for an element, to maintain visual consistency, the same value would also need to be applied to closely related elements (i.e., those in the element's segment). This insight motivated the use of segments, and it allowed the approach to treat many HTML elements as an equivalence class, which also reduced the complexity of the patch generation process.

To identify the segments in a page, the approach analyzes the Document Object Model (DOM) tree of the PUT. In informal experiments, we evaluated several well-known page segmentation analyses, such as VIPS [7], Block-o-matic [38], and correlation clustering [8]. We chose to use an automated clustering-based partitioning algorithm proposed by Romero et al. [33], as its segmentation results more readily conformed to our definition of a segment. We summarize the algorithm here for completeness. The approach starts by assigning each leaf element of the DOM tree to its own segment. Then, to cluster the elements, the approach iterates over the segments and uses a cost function to determine when it can merge adjacent segments. The cost function is based on the number of hops in the DOM tree between the lowest common ancestors of the two segments under consideration. If the number of hops is below a threshold based on the average depth of leaves in the DOM tree, then the approach will cluster the adjacent segments. The value of this threshold is determined empirically. The approach continues to iterate over the segments until no further merges are possible (i.e., the segment set has reached a fixed point). The output is a set of segments, *Segs*, where each segment contains a set of XPath IDs denoting the HTML elements that have been grouped together in the segment.

Figure 2a shows a simplified version of the segments that were identified for one of the web pages, Wiley, used in our evaluation. The red overlay rectangles show the visible elements that were grouped together as segments. These include the header content, a left-aligned navigation menu, the content pane, and the page's footer.

3.2 Phase 2: Localization

The second phase identifies the parts of the PUT that must be targeted to address its mobile friendly problems. The second phase consists of two steps. In the first step, the approach analyzes the PUT to identify which segments contain mobile friendly problems. Then, based on the structure and problem types identified for each segment, the second step identifies the CSS properties that will most likely need to be adjusted to resolve each problem. The output of the localization phase is a mapping of the potentially problematic segments to these properties.

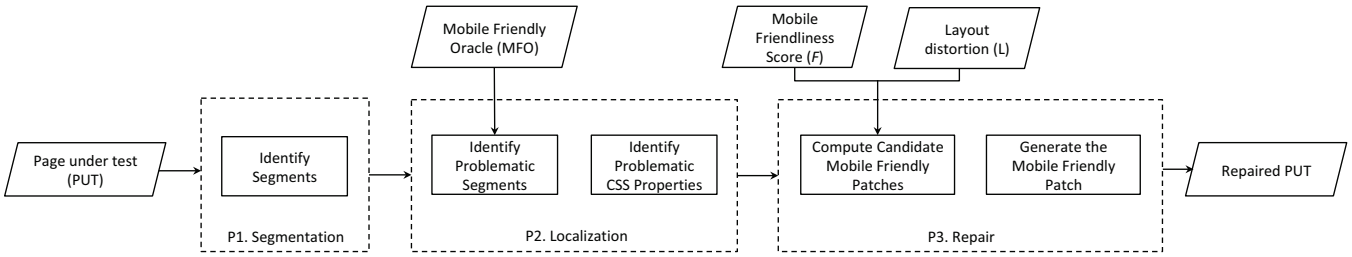

Figure 1: Overview of the approach.

(a) PUT with segments highlighted **(b) PUT with distortions highlighted** **(c) Repaired PUT**

Figure 2: Example for demonstrating our approach. Thick red rectangles highlight segments identified by our technique, while dashed red ovals indicate distortions caused by undesirable adjustments to the page's layout.

3.2.1 Identifying Problematic Segments. In the first step of the localization phase, the approach identifies mobile friendly problem types in the PUT and the subset of segments that will likely need to be adjusted to address them.

In our approach, mobile friendly problems in the PUT are detected by an Mobile Friendly Oracle (MFO). An MFO is a function that takes a web page as input and returns a list of mobile friendly problem types it contains. The MFO can identify the presence of mobile friendly problems but cannot identify the faulty HTML elements and CSS properties responsible for the observed problems. In the implementation of our approach, we use the Google Mobile-Friendly Test Tool (GMFT) as our approach's MFO. However, any detector or testing tool may also be used as an MFO. The basic requirement for an MFO is that it can accurately report whether there are *any* types of mobile friendly problems present in the page. Ideally, the MFO should also detail *what* types of problems are present, along with a mapping of each problem to the corresponding HTML elements. However, these are not strict requirements: our approach can correctly function with the assumption that all segments have all problem types. Though this over-approximation can increase the amount of time needed to compute the best solution in the second phase.

Since we leverage the GMFT in our implementation, we discuss how the output of this particular tool is used by our approach. We expect that other MFOs, such as Bing, could be adapted in a similar way. Given a PUT, the GMFT returns, for each problem type it detects, a reference to the HTML elements that contain that problem. However, through experimentation with the GMFT, we learned that the list of HTML elements it supplies is generally incomplete. Therefore, given a reported problem type, our approach applies a conservative filtering to the segments to identify which ones may be problematic with respect to that problem type. For example, if the GMFT reports that there is a problem with font sizing in the PUT, then our technique identifies any segment that contains a visible text element as potentially problematic. As mentioned above, this over-approximation may increase the time needed to compute the best solution, but does not introduce unsoundness into our approach.

The output of this step is a set of tuples of the form $\langle s, T \rangle$ where $s \in Segs$ is a potentially problematic segment and T is the set of

problem types associated with s (i.e., in the domain of {*tap_targets*, *font_size*, *content_size*}). Referring back to the example in Figure 2a, GMFT identified S3 as having two problem types, the tap targets were too close and the font size was too small, so the approach would generate a tuple for S3 where T includes these two problem types.

3.2.2 Identifying Problematic CSS Properties. After identifying the subset of problematic segments, the approach needs to identify the CSS properties that may need to be adjusted in each segment to make the page mobile friendly. The general intuition of this step is that each of a segment's identified problem types generally map to a set of CSS properties within the segment. However, this step is complicated by the fact that HTML elements may not explicitly define a CSS property (i.e., they may inherit a style from a parent element) and that our approach adjusts CSS properties at the segment level instead of the individual element level.

To address these issues, we introduce the concept of a Property Dependence Graph (PDG), which for a given segment and problem type, models the relevant style relationships among its HTML elements based on CSS inheritance and style dependencies. Formally, we define a PDG as a directed graph of the form $\langle E, R, M \rangle$. Here $e \in E$ is a node in the graph that corresponds to an HTML element in the PUT that has an explicitly defined CSS property, $p \in P$, where P is the set of CSS properties relevant for a problem type (e.g., `font-size` for font sizing problems, `margin` for tap target issues, etc.). $R \subseteq E \times E$ is a set of directed edges, such that for each pair of elements $\langle e_1, e_2 \rangle \in R$, there exists a dependency relationship between e_1 and e_2. M is a function $M : R \mapsto 2^C$ that maps each edge to a set of tuples of the form $C : \langle p, \varphi \rangle$, where $p \in P$ and φ is a ratio between the values of p for e_1 and e_2. This function is used in the following repair phase (Section 3.3) to ensure that style changes made to a segment remain consistent across pairs of elements in a dependency relationship.

Our approach defines a variant of PDG for each of the three problem types: the Font PDG (FPDG), the Content Size PDG (CPDG), and the Tap Target PDG (TPDG). Each of these three graphs has a specific set of relevant CSS properties (P), a dependency relationship, and a mapping function (M). Due to space constraints, we only present the formal definition of the FPDG, as the other two graphs are defined in a similar manner.

The FPDG is constructed for any segment for which a font sizing problem type has been identified. For this problem type, the most relevant CSS property is clearly `font-size`, but the `line-height`, `width`, and `height` properties of certain elements may also need to be adjusted if font sizes are changed. Therefore P = {`font-size`, `line-height`, `width`, `height`}. A dependency relationship exists between any $e_1, e_2 \in E$, if and only if e_1 is an ancestor of e_2 in the DOM tree and e_2 has an explicitly defined CSS property, $p \in P$, i.e., the value of the property is not inherited from e_1. The general intuition of using this dependency relationship is that only nodes that explicitly define a relevant property may need to be adjusted and the remainder of the nodes in between e_1, e_2 will simply inherit the style from e_1. The ratio, φ, associated with each edge is the value of p defined for e_1 divided by the value of p defined for e_2. To illustrate consider two HTML elements in S3 of Figure 2a. The first, e_1, is a ⟨`div`⟩ tag wrapping all of the elements in S3 with `font-size` = `13px` and the second, e_2, is the ⟨`h2`⟩ element containing the text "Resources" with `font-size = 18px`. A dependency relationship exists from e_1 to e_2 with p as `font-size` and the ratio φ = 0.72.

The output of this final step is the set, I, of tuples where each tuple is of the form $\langle s, g, a \rangle$ where s identifies the segment to which the tuple corresponds, g identifies a corresponding PDG, and a is an adjustment factor for the PDG that is initially set to 1. The adjustment factor is used in the repair phase and serves as a multiplier to the ratios defined for the edges of each PDG. A tuple is added to I for each problem type that was identified as applicable to a segment. Referring back to the example in Figure 2a, the approach would generate two tuples for S3, one containing an FPDG and the other containing an TPDG.

3.3 Phase 3: Repair

The goal of the third phase is to compute a repair for the PUT. The best repair has to balance two objectives. The first objective is to identify the set of changes – a *patch* – that will most improve the PUT's mobile friendliness. The second objective is to identify the set of changes that does not significantly change the layout of the PUT.

3.3.1 Metrics. A key insight for our approach is that both of the aforementioned objectives – mobile friendliness and layout distortion – can be quantified. For the first objective, it is typical for mobile friendly test tools to assign a numeric score to a page, where this score represents the page's mobile friendliness. For example, the Google PageSpeed Insights Tool (PSIT) assigns pages a score in the range of 0 to 100, with 100 being a perfectly mobile friendly page. By treating this score as a function, F, that operates on a page, it is possible to establish an ordering on solutions and use that ordering to identify a best solution among a group of solutions. The second objective can also be quantified as a function, L, that compares the amount of change between the layout of a page containing a candidate patch versus the layout of the original page. The amount of change in a layout can be determined by building models that express the relative visual positioning among and within the segments of a page. We refer to these models as the *Segment Model (SM)* and *Intra-Segment Model (ISM)*, respectively. Given these two models, our approach uses graph comparison techniques to quantify the difference between the models for the original page and a page with an applied candidate solution.

We now provide a more formal definition of the SM and ISM. A Segment Model (SM) is defined as a directed complete graph where the nodes are the segments identified in the first phase (Section 3.1) and the edge labels represent layout relationships between segments. To determine the edge labels, the approach first computes the Minimum Bounding Rectangles (MBRs) of each segment. This done by finding the maximum and minimum X and Y coordinates of all of the elements included in the segment, which can be found by querying the DOM of the page. Based on the coordinates of each pair of MBRs, the approach determines which of the following relationships apply: (1) intersection, (2) containment, or (3) directional (i.e., above, below, left, right). Each edge in an SM is labeled in this manner. Referring to Figure 2a, one of the relationships identified would be that S1 is above S3 and S4. An ISM is the same, but is built

for each segment and the nodes are the HTML elements within the segment.

To quantify the layout differences between the original page and a transformed page to which a candidate patch has been applied, the approach computes two metrics. The first metric is at the segment level. The approach sums the size of the symmetric difference between each edge's labels in the SM of the original page and the SM of the transformed page. Recall that both models are complete graphs, so a counterpart for each edge exists in the other model. To illustrate, consider the examples shown in Figures 2a and 2b. The change to the page has caused segments S3 and S4 to overlap. This change in the relationship between the two segments would be counted as a difference between the two SMs and increase the amount of layout difference. The second metric is similar to the first but compares the ISM for each segment in the original and transformed page. The one difference in the computation of the metric is that the symmetric difference is only computed for the intersection relationship. The intuition behind this difference in counting is that we consider movement of elements within a segment, except for intersection, to be an acceptable change to accommodate the goal of increasing mobile friendliness. Referring back to the example shown in Figure 2b, nine intra-segment intersections are counted among the elements in segment S4 as shown by dashed red ovals. The difference sums calculated at the segment and intra-segment level are returned as the amount of layout difference.

3.3.2 Computing Candidate Mobile Friendly Patches. To identify the best CSS patch, the approach must find new values for the potentially problematic properties, identified in the first phase, that make the PUT mobile friendly while also maintaining its layout. To state this more formally, given I, the approach must identify a set of new values for each of the adjustment factors (i.e., a) in each tuple of I so that the value of F is 100 (i.e., the maximum mobile friendliness score) and the value of L is zero (i.e., there are no layout differences).

A direct computation of this solution faces two challenges. The first of these challenges is that an optimal solution that satisfies both of the above conditions may not exist. This can happen due to constraints in the layout of the PUT. The second challenge is that, even if such a solution were to exist, it exists in a solution space that grows exponentially based on the number of elements and properties that must be considered. Since many of the CSS properties have a large range of potential values, a direct computation of the solution would be too expensive to be practical. Both of these challenges motivate the use of an approximation algorithm to identify a repair. Therefore, the approach must find a set of values that minimizes the layout score while maximizing the mobile friendliness score.

The design of our approximation algorithm takes into account several unique aspects of the problem domain to generate a high quality patch in a reasonable amount of time. The first of these aspects is that, through manual experimentation, we learned that good or optimal solutions typically involve a large number of small changes to many segments. This motivates targeting a solution space comprised of candidate solutions that differ from the original page in many places but by only small amounts. The second of these aspects is that computing the values of the L and F functions is expensive. The reason for this is that F requires accessing an API on the web and L requires rendering the page and computing layout information for the two versions of the PUT. This motivates us to avoid algorithms that require sequential processing of L and F (e.g., simulated annealing or genetic algorithms).

To incorporate these insights, the approximation algorithm first generates a set of size n of candidate patches. To generate each candidate patch, the approach creates a copy of I, called I', then iterates over each tuple in I' and with probability x, randomly perturbs the value of the adjustment factor (i.e., a) using a process we describe in more detail in the next paragraph. Then I' is converted into a patch, R, using the process described in the next section (Section 3.3.3), and added to the set of candidate patches. This process is repeated until the approach has generated n candidate patches. The approach then computes, in parallel, the values of F and L for a version of the PUT with an applied candidate patch. (Our implementation uses Amazon Web Services (AWS) to parallelize this computation.) The objective score for the candidate patch is then computed as a weighted sum of F and L. The candidate patch with the maximum score, i.e., with the highest value of F and the lowest value of L, is selected as the final solution, R_{max}. Figure 2c shows R_{max} applied to the example page.

Our approach perturbs adjustment factors in such a way as to take advantage of our insight that the optimal solutions differ from the original page in many places but by only small amounts. To represent this insight, we based our perturbation on a Gaussian distribution around the original value in a property. Through experimentation, we found that it was most effective to have the mean (μ) and standard deviation (σ) values used for the Gaussian distribution vary based on the specific mobile friendly problem type being addressed. For each problem type, the goal was to identify a μ and σ that provided a large enough range to allow sufficient diversity in the generation of candidate patches. For identifying μ values, we found through experimentation that μ set at the values suggested by the GMFT [14] was not effective in generating candidate patches that could improve the mobile friendliness of the PUT. Therefore we added an amendment factor to the values suggested by the GMFT to allow the approach to select a value considered mobile friendly with a high probability. The specific amendment factors we found the most effective were: +14 for font size, -20 for content sizing, and 0 for tap target sizing problems. For example, if the GMFT suggested value for font size problems was 16px, we set μ at 30px. For each problem type, we then identified a σ value. The specific values we determined to be most effective were: σ = 16 for content size problems, σ = 5 for font size problems, and σ = 2 for tap target spacing problems.

3.3.3 Generating the Mobile Friendly Patch. Given a set I, the approach generates a repair patch, R, and modifies the PUT so that R will be applied at runtime. The general form of R is a set of CSS style declarations that apply to the HTML elements of each segment in I. To generate R, the approach iterates over all tuples in I. For each tuple, the approach iterates over each node of its PDG, starting with the root node, and computes a new value that will be assigned to the CSS property represented by the node. The new value for a node is computed by multiplying the new value assigned to its predecessor by the ratio, φ, defined on the edge with the predecessor. Once new

property values have been computed for all nodes in the PDG, the approach generates a set of fixes, where each fix is represented as a tuple $\langle i, p, v \rangle$, where i is the XPath for each node in the PDG that had a property change, p is the changed CSS property, and v is the newly computed value. These tuples are made into CSS style declarations by converting i into a CSS selector and then adding the declarations of p and v within the selector. All of the generated CSS style declarations are then wrapped in a CSS media query that will cause it to be loaded when accessed by a mobile device. In practice we found that the size range specified in our patch's media query is applicable to a wide range of mobile devices. However, to allow developers to generate patches for specific device sizes, we provide configurable size parameters in the media query.

Referring back to the example, the ratio (φ) between e_1 (⟨`div`⟩ containing all elements in S3) and e_2 (⟨`h2`⟩ containing text "Resources") is 0.72. Consider a tuple ⟨`S3, font-size, 2`⟩ in I. Thus, a value v of 26px is calculated for the predecessor node e_1 based on the adjustment factor 2. Accordingly v = 26px * 1/0.72 = 36px is calculated for e_2. Thus, the approach generates two fix tuples: ⟨`div, font-size, 26px`⟩ and ⟨`h2, font-size, 36px`⟩.

4 EVALUATION

To evaluate our approach, we designed experiments to determine its effectiveness, running time, and the visual appeal of its solutions. The specific research questions we considered were:

RQ1: How effective is our approach in repairing mobile friendly problems in web pages?

RQ2: How long does it take for our approach to generate patches for the mobile friendly problems in web pages?

RQ3: How does our approach impact the visual appeal of web pages after applying the suggested CSS repair patches?

4.1 Implementation

We implemented our approach in Java as a prototype tool named MFix [21]. For identifying the mobile friendly problems in a web page, we used the Google Mobile-Friendly Test Tool (GMFT) [15] and Google PageSpeed Insights Tool (PSIT) [16] APIs. We also used the PSIT for obtaining the mobile friendliness score (labeled as "usability" in the PSIT report). For identifying segments in a web page and building the SM and ISM, we first built the DOM tree by rendering the page in an emulated mobile Chrome browser v60.0 and extracting rendering information, such as element MBRs and XPath, using Javascript and Selenium WebDriver. The segmentation threshold value determined by the average depth of leaves in a DOM tree was capped at four to avoid the situation where all of the visible elements in a page were wrapped in one large segment. This constant value was determined empirically, and was implemented as a configurable parameter in MFix. We used `jStyleParser` for identifying explicitly defined CSS properties for HTML elements in a page for building the PDG. We parallelized the evaluation of candidate solutions using a cloud of 100 Amazon EC2 t2.xlarge instances pre-installed with Ubuntu 16.04.

Table 1: Subjects

ID	URL	Category	Rank	#HTML
1	http://aamc.org	Health	23	598
2	https://arxiv.org	Science	21	381
3	http://us.battle.net	Kids and teens	2	615
4	https://bitcointalk.org	Science	25	1302
5	http://blizzard.com	Kids and teens	33	313
6	https://boardgamegeek.com	Games	31	4474
7	https://bulbagarden.net	Kids and teens	26	151
8	http://coinmarketcap.com	Science	8	1964
9	http://correios.com.br/para-voce	Society	14	769
10	http://dict.cc	Reference	20	633
11	https://www.discogs.com	Arts	26	5738
12	http://drudgereport.com	News	23	779
13	http://www.finalfantasyxiv.com	Games	37	61
14	http://www.flashscore.com	Sports	16	6621
15	https://www.fragrantica.com	Health	35	1091
16	http://forum.gsmhosting.com/vbb	Home	39	2618
17	http://www.intellicast.com	Science	38	1393
18	https://www.irctc.co.in	Regional	34	1031
19	https://www.irs.gov	Home	14	569
20	https://www.leo.org	Reference	31	990
21	http://letour.fr	Sports	3	1260
22	http://lolcounter.com	Kids and teens	30	1257
23	http://www.mmo-champion.com	Games	29	1903
24	http://myway.com	Computers	42	135
25	https://www.ncbi.nlm.nih.gov	Science	2	833
26	http://www.nexusmods.com	Games	28	2108
27	http://nvidia.com	Games	20	719
28	http://rotoworld.com	Sports	41	2523
29	http://sigmaaldrich.com	Science	37	141
30	http://us.soccerway.com	Sports	30	2708
31	http://www.square-enix.com	Games	30	198
32	https://travel.state.gov	Home	26	440
33	http://www.weather.gov	Science	18	1101
34	http://www.bom.gov.au	Kids and teens	48	685
35	http://www.wiley.com	Shopping	14	460
36	http://onlinelibrary.wiley.com	Business	33	824
37	https://www.wowprogress.com	Games	46	2828
38	https://xkcd.com	Arts	48	121

4.2 Subjects

For our experiments we used 38 real-world subjects collected from the top 50 most visited websites across all seventeen categories tracked by Alexa [3]. The subjects are listed in Table 1. The columns "Category" and "Rank" refer to the source Alexa category and rank of the subject within that category, respectively. The column "#HTML" refers to the total number of HTML elements in a subject, which we counted by parsing the subject's DOM for node type "element". This value gives an approximation for the size and complexity of the subject.

We used Alexa as the source of our subjects as the websites represent popular widely used sites and a mix of different layouts. From the 651 unique URLs that were identified across the 17 categories, we excluded the websites that passed the GMFT or had adult content. Each of the remaining 38 subjects was downloaded using the Scrapbook-X Firefox plugin, which downloads an HTML page and its supporting files, such as images, CSS, and Javascript. We then removed the portions of the subject pages that made active internet connections, such as for advertisements, to enable running of the subjects in an offline mode.

4.3 Experiment One

To address RQ1 and RQ2, we ran MFix ten times on each of the 38 subjects to mitigate the non-determinism inherent in the approximation algorithm used to find a repair solution.

For RQ1, we considered two metrics to gauge the effectiveness of our approach. For the first metric, we used the GMFT to measure how many of the subjects were considered mobile friendly after the

patch was applied. For the second metric, we compared the before and after scores for mobile friendliness and layout distortion for each subject. For comparing mobile friendliness score, we selected, for each subject over the ten runs, the repair that represented a median score. For layout distortion, we selected, for each subject over the ten runs, the best and worst repair, in terms of layout distortion, that passed the mobile friendly test. Essentially, for each subject, these were the two patched pages that passed the mobile friendly test and had the lowest (best) and highest (worst) amount of distortion. For the subjects that did not pass the mobile friendly test, we considered the patched pages with the highest mobile friendly scores to be the "passing" pages.

For RQ2, we measured the average total running time of MFix for each of the ten runs for each of the subjects, and also measured the time spent in the different stages of the approach.

4.3.1 Discussion of results. The results for *effectiveness (RQ1)* were that 95% (36 out of 38) of the subjects passed the GMFT after applying MFix's suggested CSS repair patch. This shows that the patches generated by MFix were effective in making the pages pass the mobile friendly test.

Figure 3 shows the results of comparing the before and after median mobile friendliness scores for each subject. For each subject, the dark gray portion shows the score reported by the PSIT for the patched page and the light gray portion shows the score for the original version. The black horizontal line drawn at 80 indicates the value above which the GMFT considers a page to have passed the test and to be mobile friendly. On average, MFix improved the mobile friendliness score of a subject by 33% Overall, these results show that our approach was able to consistently improve a subject's mobile friendliness score.

We also compared the layout distortion score for the best and worst repairs of each subject. On average, the best repair had a layout distortion score 55% lower than the worst repair. These results show that our approach was effective in identifying patches that could reduce the amount of distortion in a solution that was able to pass the mobile friendly test. (For RQ3, we examined, via a user study, if this reduction in distortion translates into a more attractive page.)

We investigated the results to understand why two subjects did not pass the GMFT. The patched version of the first subject, gsmhosting, contained a content sizing problem. The original version of the page did not contain this problem, which indicates that the increased font size introduced by the patch caused content in this page to overflow the viewport width. For the second subject, aamc, MFix was not able to fully resolve its content sizing problem as the required value was extremely large compared to the range explored by the Gaussian perturbation of the adjustment factor. Both of these issues suggest further refinements to our techniques that could be explored in future work, such as making the process iterative and expanding the initial search space.

The *total running time (RQ2)* required by our approach for the different subjects ranged from 2 minutes to 10 minutes, averaging a little less than 5 minutes. As of August 2017, an Amazon EC2 t2.xlarge instance was priced at $0.188 per hour. Thus, with an average time of 5 minutes the cost of running MFix on 100 instances was $1.50 per subject. Figure 4 shows a breakdown of the average

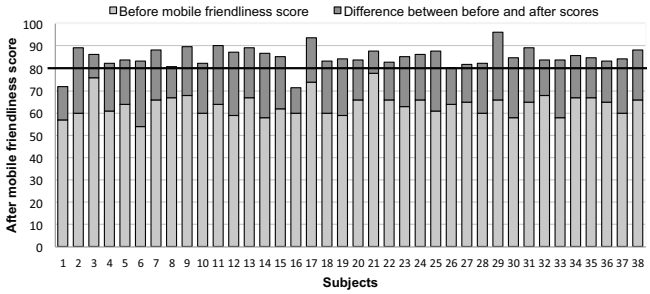

Figure 3: Distribution of the median mobile friendliness score across 10 runs

time for the different stages of the approach. As can be seen from the chart, finding the repair for the mobile friendly problems (phase 3) was the most time consuming, taking up almost 60% of the total time. A major portion of this time was spent in evaluating the candidate solutions by invoking the PSIT API. The remainder of the time was spent in calculating layout distortion, which is dependent on the size of the page. The overhead caused by network delay in communicating with the Amazon cloud instances was negligible. For the API invocation, we implemented a random wait time of 30 to 60 seconds between consecutive calls to avoid retrieving stale or cached results. Identifying problematic segments was the next most time consuming step as it required invoking the GMFT API.

4.4 Experiment Two

To address RQ3, we conducted a user-based survey to evaluate the aesthetics and visual appeal of the repaired page. The main intent of the study was to evaluate the effectiveness of the layout distortion metric, L (Section 3.3), in minimizing layout disruptions and producing attractive pages. The general format of our survey was to ask participants to compare the original and repaired versions of a subset of the subjects. To make the survey length manageable, we divided the 38 subjects into six different surveys, each with six or seven subjects. For each subject, the survey presented, in random order, a screenshot of the original and repaired pages when displayed in a frame of the mobile device. The screenshots were obtained from the output of the GMFT. An example of one such screenshot is shown in Figure 2c. We asked each human subject to (1) select which of the two versions (original or repaired) they would prefer to use on their mobile device; (2) rate the readability of each version of the page on a scale of 1–10, where 1 means low and 10 means high; and (3) rate the attractiveness of the page on a scale of 1–10, where 1 means low and 10 means high. We had

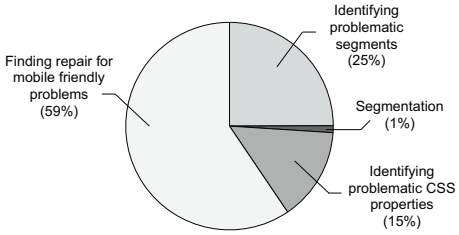

Figure 4: Breakdown of the running time of MFix

two variants of the survey, one that used the best repair as the screenshot of the repaired page and the other one that used the worst repair as the screenshot of the repaired page. Here, the best and worst repairs were as defined in Experiment 1.

We used Amazon Mechanical Turk (AMT) service to conduct the surveys. AMT allows users (requesters) to anonymously post jobs which it then matches them to anonymous users (workers) who are willing to complete those tasks to earn money. To avoid workers who had a track record of haphazardly completing tasks, we only allowed workers that had high approval ratings for their previously completed tasks (over 95%) and had completed more than 5,000 approved tasks to complete our survey. In general, this is considered a fairly selective criteria for participant selection on AMT. For each survey, we had 20 anonymous participants, giving us a total of 240 completed surveys across both variants of the survey. Each participants was paid $0.65 for completing a survey.

4.4.1 Discussion of results. Based on the analysis of the results of the first variant of the survey, we found that the users preferred to use the repaired version in 26 out of 38 subjects, three subjects received equal preference for the original and repaired versions, and only nine subjects received a preference for using the original version. Interestingly, users preferred to use the repaired version even for the two subjects that did not pass the GMFT. For readability, all but four subjects were rated as having an improved readability over the original versions. On average, the readability rating of the repaired pages showed a 17% improvement over original versions (original = 5.97, repaired = 6.98). This result was also confirmed as statistically significant using the Wilcoxon signed-rank test with a p-value = $1.53 \times 10^{-14} < 0.05$. Using the effect size metric based on Vargha-Delaney A measure, readability of the repaired version was observed to be 62% of the time better than the original version. With regards to attractiveness, no statistical significance was observed, implying that $\mathcal{M}$Fix did not deteriorate the aesthetics of the pages in the process of automatically repairing the reported mobile friendly problems. In fact, overall, our repaired versions were rated slightly higher than original versions for attractiveness (avg. original = 6.50, avg. repaired = 6.67 and median original = 6.02, median repaired = 7.12).

We investigated the nine subjects where the repaired version was not preferred by the participants. Based on our analysis, we found two dominant reasons that applied to all of the nine subjects. First, these subjects all had a fixed sized layout, meaning that the section and container elements in the pages were assigned absolute size and location values. This caused a cascading effect with any change introduced in the page, such as increasing font sizes or decreasing width to fit the viewport. The second reason was linked to the first as the pages were text intensive, thereby requiring $\mathcal{M}$Fix to increase font sizes. These results motivate future work in techniques that can better handle these types of pages.

Overall, these results indicate that $\mathcal{M}$Fix was very effective in generating repaired pages that (1) users preferred over the original version, (2) considered to be more readable, and (3) that did not suffer in terms of visual aesthetics.

The results for the second variant of the survey underscored the importance of our layout distortion objective and the impact visual distortions can have on end users' perception of a page's attractiveness. The results showed that the users preferred to use the original, non-mobile friendly version, in 22 out of 38 subjects and preferred to use the repaired version for only 16 subjects. Readability showed similar results as the first survey variant. On average, an improvement of 11% in readability was observed for the repaired pages compared to the original versions, and was still found to demonstrate statistical significance (p-value = $7.54 \times 10^{-6} < 0.05$). This is expected as the enlarged font sizes can make the text very readable in the repaired versions despite layout distortions. However, in this survey a statistical significance (p-value = $2.20 \times 10^{-16} < 0.05$) was observed for the attractiveness of the original version being rated higher than the repaired version. On average, the original version was rated 6.82 (median 7.00) and the repaired version was rated 5.64 (median 5.63). In terms of the effect size metric, the repaired version was rated to have a better attractiveness only 38% of the time. These results strongly indicate that the layout distortion objective plays an important role in generating patches that make the pages more attractive to end users.

4.5 Threats to Validity

External Validity: The first potential threat is bias in the selection of participants for the user-based study in Experiment two. To address this threat, we used AMT that provided us with a large pool of anonymous participants. We only specified qualification requirements for the participants in our study (i.e., high numbers of previously completed tasks and high approval ratings, as outlined in Section 4.4) to ensure authentic results. Another potential threat is in the selection of subject web pages for our evaluation. To mitigate any bias, we used the home pages of websites drawn from Alexa's 50 top ranked websites from different categories.

Internal Validity: One potential threat is that the survey used in the user-based study may not render in full resolution when viewed on small screen devices, potentially impacting the results. To mitigate this threat, we asked participants to enter the device they used for answering the survey, so that we could isolate those results. However, only a small minority of our participants did not use a desktop or laptop, and the results from the few who used a mobile phone or tablet to answer the survey did not indicate any anomalous responses. Another potential threat is the use of GMFT and PSIT in our approach to determine mobile friendly problems and mobile friendliness score. However, the PSIT is the only publicly available tool that reports a mobile friendliness score. Bing only offers a web interface for detecting mobile friendly problems, unlike GMFT, which provides an API. Furthermore, GMFT and PSIT are stable tools that are used by Google to rank pages in their own search results.

Construct Validity: A potential threat is that the layout distortion objective used in our approach quantifies the aesthetic value of a page, which is a subjective aspect of a web page. To address this threat, we conducted two user-based studies (i.e., Experiment Two) to qualitatively understand the impact of layout distortion on the visual appeal of a page. A second potential threat is that the numbers supplied by participants in response to the readability and attractiveness ratings that we asked them to provide are also subjective. To mitigate this threat, we used *relative* values given

by the participants for the before and after repair versions for the subjects, as opposed to using their absolute values. That is, although two participants may supply different numbers for the ratings for the same pair of web pages, they will supply higher values for one of the pages if they believe that readability/attractiveness is better for that page. A third potential threat is that we used screenshots of the subject pages in the user-based study as opposed to allowing the users to interact with the pages on mobile devices. We selected this mechanism as we wanted the users to visualize the before and after repair versions of the pages next to each other to allow for an easy comparison. Also, we did not have control over participants' mobile devices and wanted to avoid variations in results that this could cause. A fourth potential threat is that the participants have a bias in selecting the repaired version based on the order in which the original and repaired versions are presented in the survey. To mitigate this threat, we randomized the order of the two versions for each question of the survey. A final potential threat is that the definition of correct repair used in our approach may be different from developer intent. However, this threat is mitigated by the user study results discussed in RQ3, which show that the repairs generated by $\mathcal{M}$Fix were considered visually appealing and preferred by the participants.

5 RELATED WORK

There have been approaches in the literature that attempt to fix presentation issues in a web page, but none of these attempt to repair mobile friendly problems. The XFix [22, 23] technique, for example, repairs *layout Cross Browser Issues (XBIs)* — presentation failures arising from the inconsistencies in the rendering of a website across different browsers. In this domain, the "correct" presentation of the page is available through one of the browsers, the layout of which must be mimicked in another. Mobile friendly problems entail a different approach, in which the presentation of a page must be changed to correct a series of identified issues. There is no correct reference rendering available to the repair process, which must also maintain as much of the original aesthetics of the page as possible. The latter constraint motivates the use of the segment model in this paper's approach (Section 3), which enables certain properties of related HTML elements to be adjusted in proportion to one another. Elsewhere, Cassius [32], proposes a framework based on automated reasoning for debugging and repairing faulty CSS. However, the technique assumes as input the faulty source lines in CSS files, and a set of page layout examples that the technique can use to synthesize repair. These are unavailable in the problem domain of mobile-friendly issues, however, where the only information available is the types of mobile friendly problems that exist in a page. Meanwhile, PhpRepair [37] and PhpSync [31] detect and repair HTML syntax problems in web applications. Also, Wang et al. [45] present a technique that uses static and dynamic analysis to repair web applications by propagating a given presentation fix to the server side source code. However, none of these techniques specifically address mobile friendly problems and would be unable to provide suitable repairs for them.

Other approaches circumvent mobile friendly problems by presenting alternative versions of a desktop website, rather than by issuing repairs. For example, commercial services such as bMobilized [5], WompMobile [46], Mobilifyit [30], Duda [11], and Mobify [29], can convert a given desktop website to a mobile friendly version using pre-designed templates. Although helpful, the solutions are not appropriate in all situations. Firstly, the templates are unlikely to capture the carefully crafted layout and graphics designed for the desktop versions, possibly undermining the branding efforts a company is trying to achieve. Our approach avoids these limitations by maintaining a close similarity to the original version. Second, the output represents a separate mobile friendly website with a new URL, requiring the development team to maintain two websites. In contrast, our approach generates a CSS media query patch that is added to the existing CSS of the original website and that will only be triggered if the page is requested from a device with a smaller screen size. Alternatively, modern browsers, such as Chrome [10], Safari [36], and Firefox [13], provide a "reader" mode intended for easy clutter-free viewing of web pages on mobile devices by presenting its text only and stripping out layout and page styling. The primary purpose of this mode, however, is to allow for easier reading of a page's primary content, rather than to address mobile friendly problems.

Techniques from the research community target various parts of the detection and localization process for various types of UI related problems in web apps, such as XBIs [9, 34, 34, 35], presentation failures [24, 25, 25–28, 41], internationalization [2], responsive web page problems [42, 43], and need-to-translate strings [44]. However, the problems detected by these approaches do not overlap with mobile friendly problems and their solutions are too specific to generalize to this broader domain.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced an approach for the automated repair of mobile friendly problems in web pages. Our approach first segments the page into elements forming natural visual groupings. It then builds graph-based models of the segments and layout of the page and uses the constraints represented by these graphs to compute a repair that can improve mobile friendliness while minimizing layout disruption. In the evaluation, we found that our approach was effective in resolving mobile friendly problems for 95% of the subjects and required only an average of five minutes per subject. In a user study, the participants overwhelmingly preferred the repaired version of the website for use on mobile devices and also considered the repaired page to be significantly more readable than the original. Overall, these results are very positive and indicate that our approach can help developers to improve the mobile friendliness of their web pages.

One possible direction of future work is to extend our approach to handle complex transformations, such as converting navigation links to dropdown menus. This would pose new research challenges, such as analysis to identify which lists of hyperlinks could be grouped into a drop down menu, a refactoring to carry out this change, and a method to quantify the change's impact.

ACKNOWLEDGMENTS

This work was supported by U.S. National Science Foundation grant CCF-1528163.

REFERENCES

[1] StackOverflow Search — Mobile Friendly Problems with Bootstrap. Retrieved Aug 2017 from https://stackoverflow.com/search?q=bootstrap+mobile+problem

[2] Abdulmajeed Alameer, Sonal Mahajan, and William G.J. Halfond. 2016. Detecting and Localizing Internationalization Presentation Failures in Web Applications. In *Proceeding of the 9th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*.

[3] Alexa. 2017. Alexa Top 50 Websites by Category. Retrieved Aug 2017 from https://www.alexa.com/topsites/category

[4] Bing. 2017. Bing Mobile Friendly Test Tool. Retrieved Aug 2017 from https://www.bing.com/webmaster/tools/mobile-friendliness

[5] bMobilized. 2017. bMobilized Website. Retrieved Aug 2017 from http://bmobilized.com/

[6] Browserstack. 2017. BrowserStack for Testing Mobile Websites. Retrieved Aug 2017 from https://www.browserstack.com/

[7] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. *VIPS: a Vision-based Page Segmentation Algorithm.* Technical Report.

[8] Deepayan Chakrabarti, Ravi Kumar, and Kunal Punera. 2008. A Graph-theoretic Approach to Webpage Segmentation. In *Proceedings of the 17th International Conference on World Wide Web (WWW '08)*.

[9] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Washington, DC, USA, 171–180.

[10] Chrome. 2017. Chrome Reader Mode. Retrieved Aug 2017 from https://github.com/chromium/dom-distiller

[11] Duda. 2017. Duda Website. Retrieved Aug 2017 from https://www.dudamobile.com/

[12] eMarketer. 2017. Estimates for Digital Users. Retrieved Aug 2017 from https://www.emarketer.com/Article/eMarketer-Releases-Updated-Estimates-US-Digital-Users/1015275

[13] Firefox. 2017. Firefox Reader Mode. Retrieved Aug 2017 from https://support.mozilla.org/en-US/kb/firefox-reader-view-clutter-free-web-pages

[14] Google. 2017. Google Mobile Friendly Problem Types. Retrieved Aug 2017 from https://support.google.com/webmasters/answer/6352293

[15] Google. 2017. Google Mobile Friendly Test Tool. Retrieved Aug 2017 from https://search.google.com/test/mobile-friendly

[16] Google. 2017. Google PageSpeed Insights Tool. Retrieved Aug 2017 from https://developers.google.com/speed/pagespeed/insights/

[17] Google. 2017. Google Search Ranking based on Mobile Friendliness. Retrieved Aug 2017 from https://support.google.com/adsense/answer/6196932?hl=en

[18] Google. 2017. Google Study for Mobile Usage. Retrieved Aug 2017 from https://developers.google.com/search/mobile-sites/

[19] Google. 2018. Consumer Study. Retrieved Feb 2018 from https://www.consumerbarometer.com/en/insights/?countryCode=US

[20] Google. 2018. Content Sizing. Retrieved Feb 2018 from https://developers.google.com/web/fundamentals/design-and-ux/responsive/

[21] Sonal Mahajan. 2017. $\mathcal{M}$Fix Project. Retrieved Aug 2017 from https://github.com/USC-SQL/mfix

[22] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. 2017. Automated Repair of Layout Cross Browser Issues using Search-Based Techniques. In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)*.

[23] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G.J. Halfond. 2017. XFix: Automated Tool for Repair of Layout Cross Browser Issues. In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA) – Tool Track*.

[24] Sonal Mahajan and William G. J. Halfond. 2014. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE) – New Ideas track*.

[25] Sonal Mahajan and William G. J. Halfond. 2015. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[26] Sonal Mahajan and William G. J. Halfond. 2015. WebSee: A Tool for Debugging HTML Presentation Failures. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) – Tool track*.

[27] Sonal Mahajan, Bailan Li, Pooyan Behnamghader, and William G. J. Halfond. 2016. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[28] Sonal Mahajan, Bailan Li, and William G. J. Halfond. 2014. Root Cause Analysis for HTML Presentation Failures Using Search-based Techniques. In *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST)*.

[29] Mobify. 2017. Mobify Website. Retrieved Aug 2017 from https://www.mobify.com/

[30] Mobilifyit. 2017. Mobilifyit Website. Retrieved Aug 2017 from http://www.mobilifyit.com/

[31] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2011. Auto-locating and Fix-propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Washington, DC, USA, 13–22.

[32] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[33] Richard Romero and Adam Berger. 2004. Automatic Partitioning of Web Pages Using Clustering. In *Proceedings of Mobile Human-Computer Interaction - MobileHCI 2004: 6th International Symposium*.

[34] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. 702–711.

[35] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. 1–10.

[36] Safari. 2017. Safari Reader Mode. Retrieved Aug 2017 from https://en.wikipedia.org/wiki/Safari_(web_browser)

[37] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 277–287.

[38] AndrÃĮs Sanoja and StÃĮphane GanÃğarski. 2014. Block-o-Matic: A web page segmentation framework. In *Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS)*.

[39] SauceLabs. 2017. SauceLabs for Testing Mobile Websites. Retrieved Aug 2017 from https://saucelabs.com/

[40] Statcounter. 2017. Mobile Market Share. Retrieved Aug 2017 from http://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-201407-201707

[41] Michael Tamm. 2009. Fighting layout bugs. https://code.google.com/p/fighting-layout-bugs/.

[42] Thomas Walsh, Gregory Kapfhammer, and Phil McMinn. 2017. Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle. In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)*.

[43] Thomas A. Walsh, Phil McMinn, and Gregory M. Kapfhammer. 2015. Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages. In *International Conference on Automated Software Engineering (ASE)*. ACM, 709–714.

[44] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2010. Locating Need-to-Translate Constant Strings in Web Applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*.

[45] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*. ACM, New York, NY, USA, 16:1–16:11.

[46] WompMobile. 2017. WompMobile Website. Retrieved Aug 2017 from http://www.wompmobile.com/