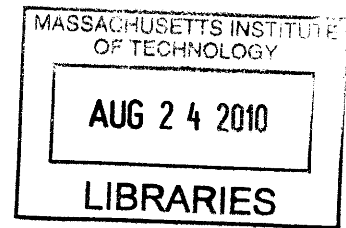
Mobi: Automatic Customization of the Mobile Web


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 2 4 2010
LIBRARIES

by

Richard W. Chan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

ARCHIVES

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May, 2010

Certified by ....
Robert C. Miller
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Mobi: Automatic Customization of the Mobile Web

by

Richard W. Chan

Submitted to the Department of Electrical Engineering and Computer Science
on May, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Mobi is a system that automatically rewrites web pages into formats optimized for web browsing on mobile devices. The system estimates heuristically which parts of a web page's content users are most likely interested in, using previously recorded history of user actions on the Web. By doing so, Mobi is able to remove the unnecessary portions of the web page and rewrite it in a format more fitting for the mobile device's form factor. Mobi accomplishes this by having a proxy server inject a client-side script onto web pages. The script examines previously recorded user actions related to the web page it is injected onto and rewrites the web page accordingly. The design and implementation of the system are described, as well as an evaluation over a variety of web pages.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor

Acknowledgments

I would like to thank Rob Miller for his advice and support throughout the project. This thesis could not have been completed without him, and I am extremely grateful for having him as my research advisor.

I would also like to thank all the members of the User Interface Design group for their feedbacks over the designs of Mobi, and especially Chen-Hsiang Yu and Igor Kopylov, for contributing many valuable ideas and advices.

Special thanks go to my friends at MIT, who make hunting for free food and sleeping late at night (or not at all) feel like the norm.

Lastly, I would like to thank my family for always being so supportive and encouraging throughout the past few years at MIT. I cannot imagine how life would have been without them.

Contents

1 Introduction **13**

2 Related Work **17**

2.1 End-User Customization 17

2.2 PageTailor 18

2.3 Project Joey 20

2.4 Highlight 20

2.5 Creo and Adeo 22

2.6 Mobile Transcoders 22

3 Design **25**

3.1 Early Ideas 25

3.2 Design Prototypes 27

3.2.1 Enlarge in place 27

3.2.2 Copy to top 30

3.2.3 Deleting irrelevant content 31

3.3 Design Decision 32

3.3.1 Mode change between customized view and full page view . . 33

3.3.2 Dynamic interfaces 35

4 Implementation **37**

4.1 Customizer 37

4.1.1 Removing irrelevant elements from view 38

4.1.2 Labels and captions 40
4.1.3 AJAX and Dynamic changes 41
4.1.4 Mode change to full page view 43
4.2 Action Recorder 43
4.2.1 Heuristics for recognizing relevant contents 43
4.2.2 Applying action history data 46
4.3 Proxy Server and Script Injection 46
4.4 Implementation on Mobile Browsers 47
4.4.1 Compatibility Issues 47
4.4.2 Zooming functionalities 47

5 Evaluation **49**
5.1 Results 50
5.1.1 *aa.com* 50
5.1.2 *google.com* 51
5.1.3 *wikipedia.org* 53
5.1.4 *webmail.mit.edu* 54
5.1.5 *facebook.com* 56
5.1.6 *expedia.com* 58
5.2 Summary 59
5.3 Discussions 59
5.3.1 Limitations 59

6 Conclusion **63**
6.1 Contributions 63

7 Future Work **65**
7.1 Further developments 65
7.2 User control 66
7.3 Browser capabilities 66
7.4 User behavior data 66

List of Figures

1-1 Comparison between rendering a web page (http://aa.com) on a desktop versus on a mobile device. 14

1-2 High-level overview of Mobi's architecture. 16

2-1 Platypus's toolbar on Firefox. 18

2-2 Platypus's Editing mode. 18

2-3 PageTailor's toolbar on a PDA. 19

2-4 An example of a web page after being customized with PageTailor. . 19

2-5 An example of how Highlight records and creates a mobile version of a website while user is interacting with it. 21

2-6 Menu for accessing and playing back a recording on Adeo. 23

3-1 Prototypes for the enlarge-in-place customization, labelled by their ratios. 29

3-2 Prototype for the copy-to-top customization. 30

3-3 Prototype that customizes the page by hiding everything except for the relevant content. 31

3-4 Customized version of aa.com created by the Mobi system. 33

3-5 Example showing how user interactions may continue uninterrupted between mode changes. 34

4-1 Illustration of the implementation for hiding irrelevant elements by traversing through the document's DOM tree. 39

4-2 Illustration showing which ancestor nodes must have their CSS style changes undone when two relevant nodes overlap as the result of the style changes. . . . 40
4-3 Illustration showing the area under which the Customizer looks for label elements for a text box. . . . 41

5-1 Evaluation of Mobi running on *aa.com*, showing the elements marked as relevant and the corresponding customized version using this data. 50
5-2 The search interface on *google.com*, with elements determined to be relevant marked in green. . . . 51
5-3 *google.com*'s interface after applying Mobi's customizations. . . . 52
5-4 The search and article interface on *wikipedia.org*, with elements determined to be relevant marked in green. . . . 53
5-5 *wikipedia.org*'s interface after applying Mobi's customizations. . . . 54
5-6 The interface on *webmail.mit.edu* for viewing and composing emails, with elements determined to be relevant marked in green. . . . 55
5-7 *webmail.mit.edu*'s interface after applying Mobi's customizations. . . 55
5-8 Evaluation of Mobi running on *facebook.com*, showing the elements marked as relevant and the corresponding customized version using this data. . . . 57
5-9 Evaluation of Mobi running on *expedia.com*, showing the elements marked as relevant and the corresponding customized version using this data. . . . 58

List of Tables

4.1 Different cases of changes on the DOM tree. 42

5.1 Summary of evaluating Mobi over different web sites. 60

Chapter 1

Introduction

Browsing the Web on mobile devices has become very common in recent years. Thanks to advances in technology, smart phones today enjoy faster network speed and processing power, allowing for increasingly sophisticated web browsers to run. However, the mobile web browsing experience remains highly impeded due to their limited screen size and difficult input methods. Web applications designed for the desktop browsers are therefore often unsuitable for mobile devices.

To alleviate this problem, some websites are beginning to provide mobile versions of their applications. However, the wide variety of functionalities and form factors in different mobile devices makes it impossible for developers to create designs satisfactory for all mobile users. Because of that, many web sites designed their mobile versions for the least functional devices and are therefore significantly less usable or functional than their desktop versions. Alternatively, if developers wish to make use of the full powers of various devices, a different design will have to be created and maintained for each device: the high cost of which makes it infeasible for most web sites.

This thesis describes Mobi, a system that automatically reformats web pages into versions optimized for mobile browsing. Mobi heuristically look for parts of web pages that the user is most likely interested in and removes everything other than the relevant contents from view, providing a stripped down version of the page. The user may also switch back to the original version of the page if the algorithm removes

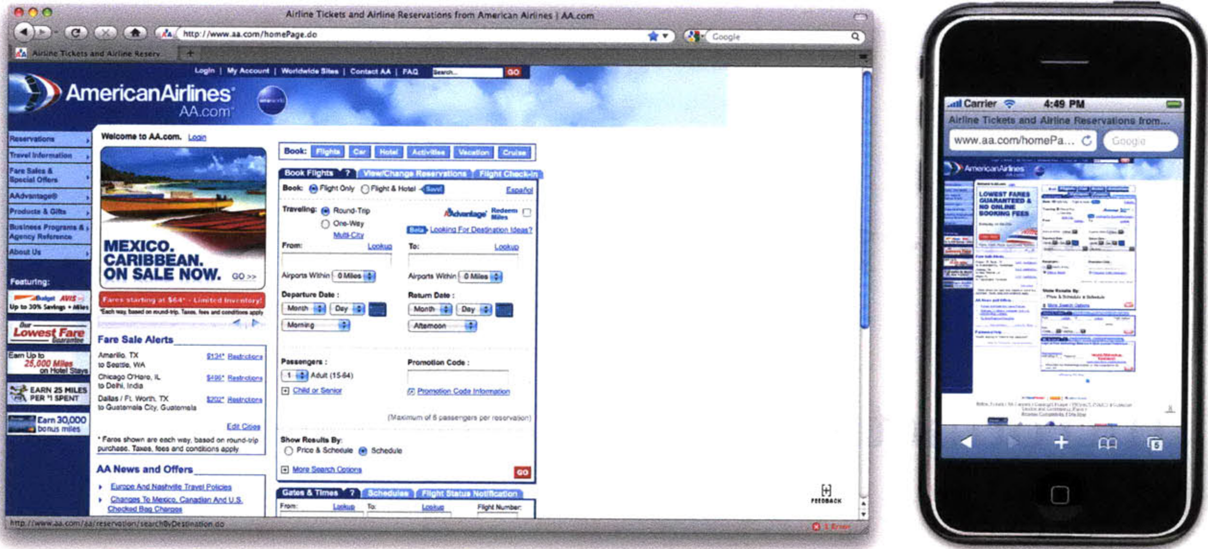


Figure 1-1: Comparison between rendering a web page (http://aa.com) on a desktop versus on a mobile device.

elements the user wishes to interact with.

By removing irrelevant contents from the web page, Mobi increases the user's efficiency in finding, viewing and interacting with the contents he is interested in. While some browsers today may have advanced zooming and panning capabilities, it is still very difficult to interact with a web page that needs to be rescaled to at most 1/3 its original size for it to fit in a mobile device's viewport. For example, a textbox originally 20 pixels by 100 pixels in size will be 6 pixels by 33 pixels on the screen. In order to click on the textbox, the user will have to look for something makes up for less than 0.2% of the screen (assuming dimension of an iPhone browser's viewport) and zoom in on it until it is big enough on the screen to be clicked on. By reducing the amount of contents on the screen, Mobi makes it significantly easier for the user to access contents he is interested in.

To maximize learnability, Mobi keeps the layout and styling of the mobile version as consistent to the original page as possible. This allows users who are familiar with the desktop interface already to make use of a familiar interface rather than having to learn using a completely different design, as is often the case for web sites that provide mobile versions.

Mobi is implemented by having a *Customizer* script that is injected onto all web pages the user accesses. In order to do so, a proxy server is set up so all requests to the web routed through the server will have the script injected. As such, the script is written in JavaScript and may run on any mobile browser that supports JavaScript and proxy connections.

The method for assessing relevance of the contents on a web page uses knowledge about users' actions in the past. For instance, if the user always only uses a certain textbox on a web page, the textbox will be considered highly relevant. In order to collect this data, Mobi also has an *Action Recorder* that resides in the user's desktop browser. The recorder is implemented as a browser extension in Firefox, created using Chickenfoot [7], itself an extension that enables end-user programming. The Action Recorder collects data over all clicks and interactions on web pages, as well as data about how long certain elements had been within the user's viewport. Data collected by the Action Recorder are pushed to the proxy server, allowing access from the Customizer script.

By using action history as the basis of the algorithm, Mobi allows for mobile versions of web pages to be generated automatically without explicit instructions from the user or developer of the web pages. Unlike systems that provide end-user mobile customizations of web pages such as Highlight [13], Mobi does not require the user to explicitly define and configure mobile versions for each web application. Every web page visited by the user previously on the desktop will have a corresponding mobile version automatically in Mobi. Furthermore, for web pages that haven't been visited before, if the user has accessed another web page within the same web site that has a similar structure, Mobi will apply the algorithm using data from that page.

The remainder of this thesis covers the design and implementation of Mobi. Chapter 2 describes related work in the area of customizations of the mobile web. Chapter 3 discusses the design of the mobile customizations. Chapter 4 covers the details of the implementation. The results of running and evaluating the system over a variety of different web sites is given in Chapter 5. Lastly, Chapter 6 concludes the findings in this thesis and discusses future works on and related to Mobi.

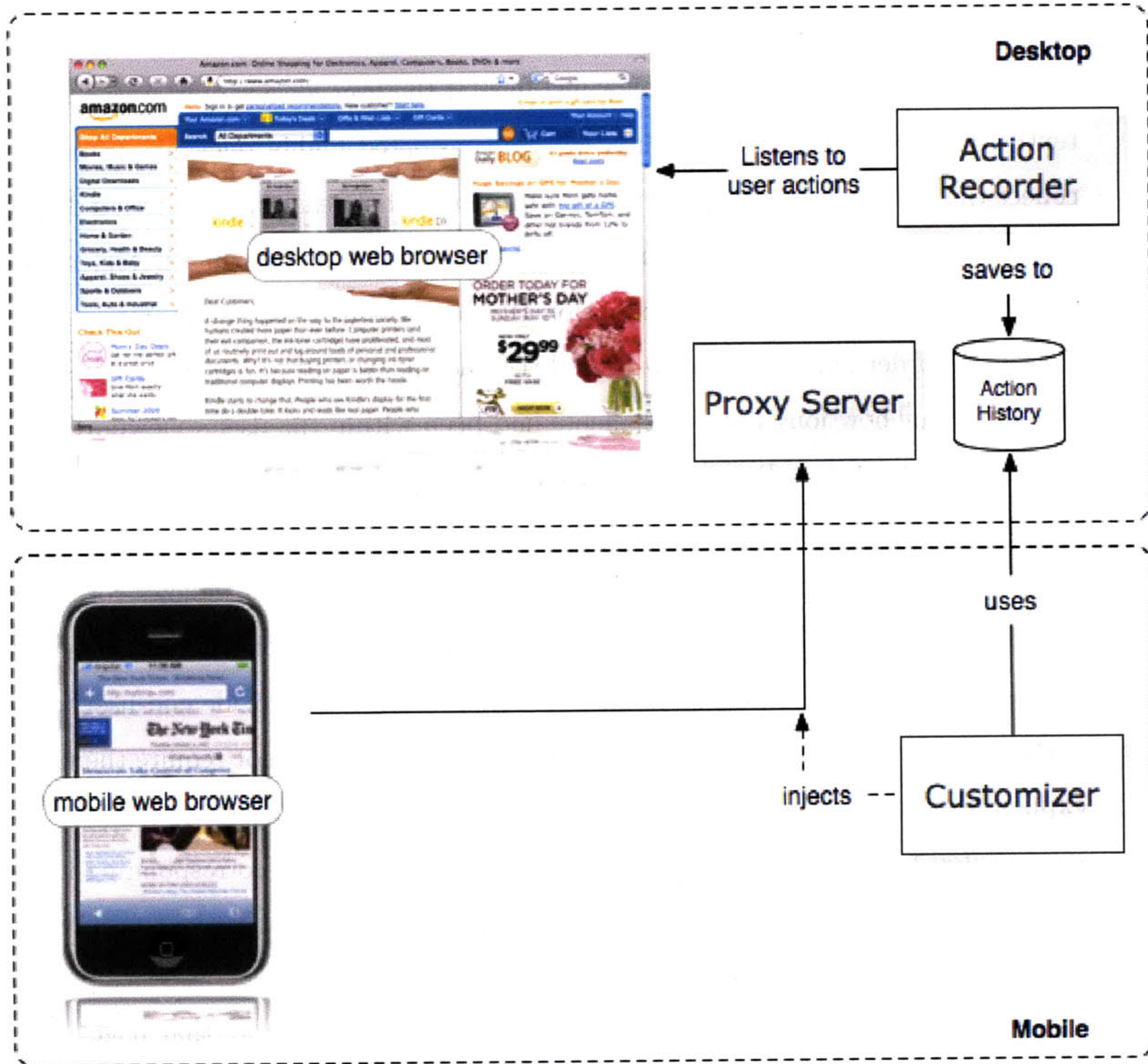


Figure 1-2: High-level overview of Mobi's architecture.

Chapter 2

Related Work

There are several projects in the domain of providing customizations of the web to mobile as well as non-mobile devices. The chapter discusses these projects and how they relate to the motivation and development of Mobi.

2.1 End-User Customization

End-User Customization is a technique for allowing end users of various applications to modify the interface they see. Several projects currently exist for enabling end-user customizations of the web on the desktop environment, such as Greasemonkey and Chickenfoot.

Both Greasemonkey [2] and Chickenfoot [7] [11] are toolkits that empower users to write scripts that change the appearance of web pages in the user's browser. JavaScript code may be written to match certain URL patterns and runs after the web page is loaded. To be more specific, Chickenfoot provides a sidebar in Firefox that allows users to create *triggers* in JavaScript code and configures which pages they should be applied on.

In essence, this type of system gives users complete control over the appearance and function of the interface of the web pages they see. However, they also require the users to be knowledgeable about JavaScript and HTML to program these customizations. The Platypus [3] extension seeks to provide a What You See is What

Figure 2-1: Platypus's toolbar on Firefox.

and Mozilla's technologies supported user mediation in a transparent and useful way.

Platypus is a Firefox extension which lets you modify a Web page from your browser -- "What You See Is What You Get" -- and then save those changes as a **Greasemonkey** script so that they'll be repeated the next time you visit the page. Editing pages to suit your needs is dandy -- but making those changes "permanent" is the real payoff.

Some of the things you can do with **Platypus** include:

Figure 2-2: Platypus's Editing mode.

You Get (WYSIWYG) graphical interface to Greasemonkey for allowing users to create customizations visually to make it easier for non-programmers to create end-user customizations. It provides functions such as erasing certain elements in the page by clicking on a button in the toolbar it adds to the browser, and selecting the element on the page to be erased. Functions such as moving and changing sizes of elements are done similarly.

While these approaches succeed to various degrees in providing end-user customizations on desktop browsers, they are not easily applicable to the mobile environment. The text-based development environment employed in Chickenfoot and Greasemonkey are significantly more difficult to use on a mobile phone, due to the small screens and less convenient typing methods offered by the small built-in or virtual keyboard. Similarly Platypus's graphical approach is also difficult due to the need to directly manipulate elements on the page in a small screen.

2.2 PageTailor

PageTailor [6] is a prototype end-user customization system that allows users to create customizations of web pages directly on the mobile device. As an extension to the Firefox-based Minimo mobile web browser, PageTailor adds a toolbar interface similar to the one provided by Platypus, offering functions for removing, rescaling, and moving elements on the screen. Similar to Platypus, the user may directly manipulate

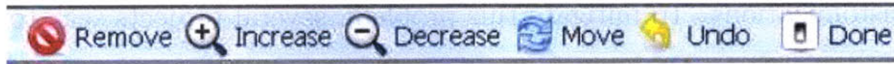


Figure 2-3: PageTailor's toolbar on a PDA.

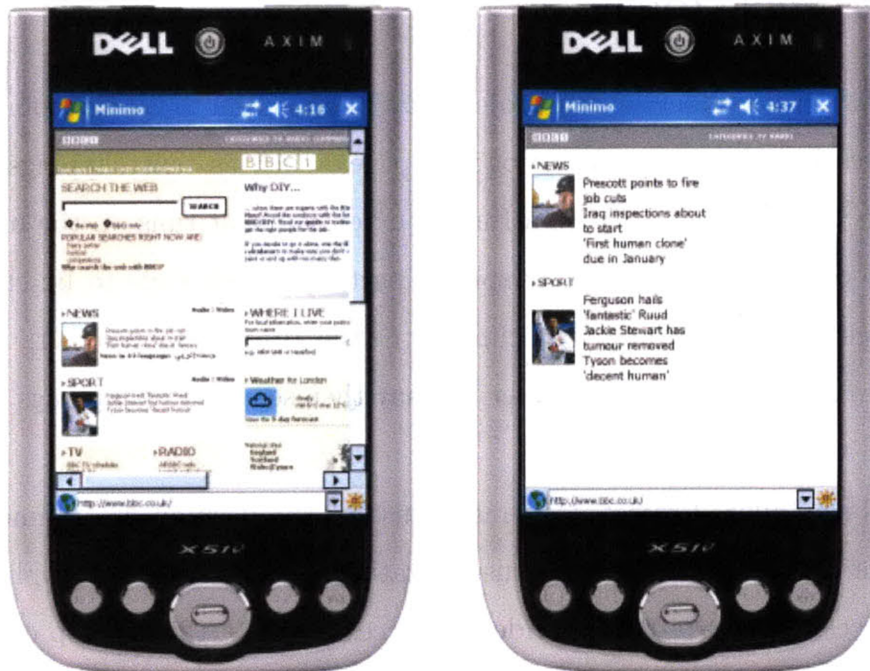


Figure 2-4: An example of a web page after being customized with PageTailor.

certain elements on the page and PageTailor will translate the user's customizations into operations for modifying the Document Object Model (DOM) of the web page.

Due to the need for the user to directly specify and create customizations, users only benefit from PageTailor in their subsequent revisits to the web page. In order to not have the user repeatedly edit the page every time he visits a web page, PageTailor stores customizations into persistent storage on the mobile devices and reapplies the customizations when the user visits the same page again.

The editor toolbar interface provided by PageTailor, while intuitive to use, is still fairly inefficient due to the form factor and input methods of a mobile device. Even with the ability to reapply customizations created previously, PageTailor requires the user to essentially design customizations for every page that they haven't previously visited on the mobile device. Changes on web pages also interfere with its ability to

reapply customizations. To mitigate this problem, several projects seek to make use of the desktop environment in creating customizations.

2.3 Project Joey

Mozilla Labs' Project Joey [4] is a project that allows users to create mobile versions of web pages by selecting portions of web pages, such as text clippings, pictures, videos, etc. Project Joey includes a Firefox extension that runs on the user's desktop Firefox browser. The user may specify parts of a web page to be clipped and upload specification regarding that clipping to the Project Joey server. When the user browses to the Project Joey server, he may then see a list of clippings previously created and view those clippings.

Unlike PageTailor, Project Joey makes use of the desktop environment as the development platform for creating customizations, which are then applied on the mobile devices. This approach provides the accessibility of the desktop environment to the editing process of creating end-user customizations.

2.4 Highlight

IBM's Highlight [13] project is a *programming by demonstration* system that allows users to create customizations for their mobile devices on the desktop. Highlight consists of an extension on the user's desktop Firefox browser and a proxy server that the mobile device access the web behind.

To specify how a web application should be customized using Highlight, the user can open the Highlight designer on the Firefox browser and continue interacting with the web site normally. Highlight keeps track of the users interactions in what they call *traces* of the user actions. Highlight then attempts to generate a mobile customized version of the application that supports the recorded user interactions. The user may also manually modify the customized version generated by Highlight directly in their designer interface.

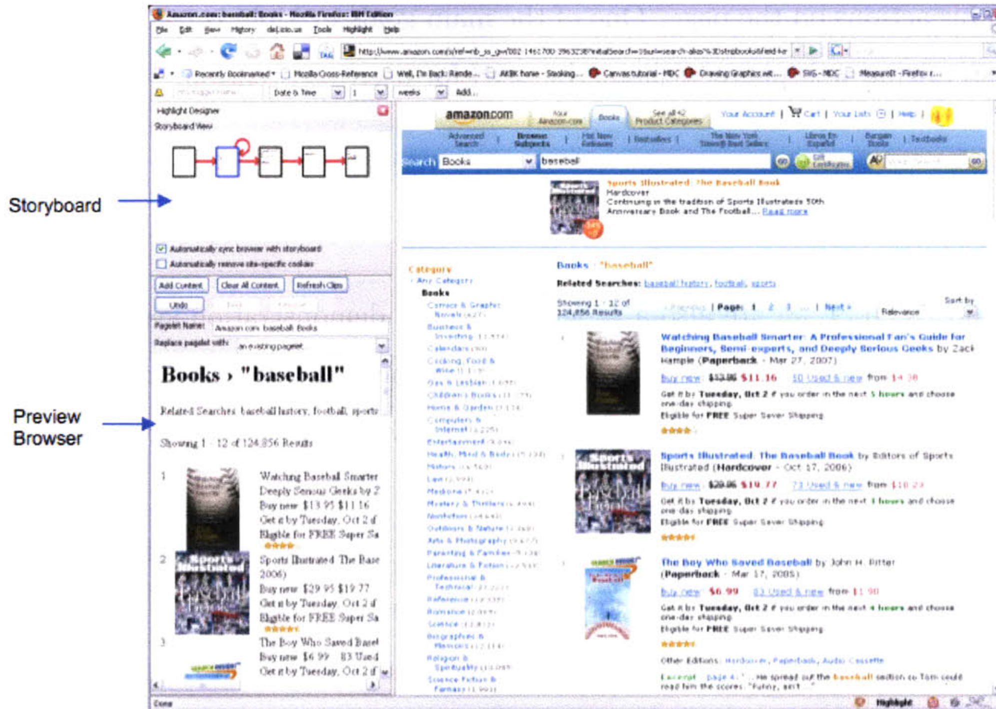


Figure 2-5: An example of how Highlight records and creates a mobile version of a website while user is interacting with it.

To view the customized mobile versions on a mobile device, the user configures their mobile browser to access the web via a proxy server set up by Highlight. The proxy server itself contains a full-fledged Firefox browser. As the user visits different web pages, the in-proxy Firefox browser loads these pages. If customizations had been previously configured for these pages, these customizations are applied onto the web page rendered in this browser. The server then forwards this mobile-customized version of the web page to the mobile browser.

Using this approach, Highlight allows users to easily create customizations of web pages mainly by demonstrating how the web pages should be interacted with. Manual modification of the customizations are also relatively easy due to the desktop environment.

However, Highlight still requires the user to explicitly define customization for each web page. In other words, for each web page that the user wishes to be customized when they visit it on a mobile device, he has to open up Highlight and asks it to

record his actions on the page. One of the main goals in Mobi is to provide a method for customizing web pages without the need for user intervention.

2.5 Creo and Adeo

Creo and Adeo [9] [8] are also a programming by demonstration system, but with a focus different from that of Highlight. Instead of focusing on customizing the interface of a web page, Creo and Adeo seek to provide a way to automate repetitive tasks on mobile devices.

Creo is a plug-in on the desktop Internet Explorer browser. The user starts by opening Creo and starting its recorder when he wishes to record a certain task. By monitoring user actions in the browser, Creo saves them into macros that may be replayed in the browser. To make these tasks generalizable Creo also looks at the user inputs and attempts to generalize them into their associated categories using Miro (a data detector that uses MIT's ConceptNet and Stanford's TAP databases).

After saving these macros in Creo, the user may then open the Adeo program on their mobile device, which will connect him to the desktop running Creo. When asked to run a certain macro, Creo will play the macro on the desktop and return the results to the mobile device.

While Adeo and Creo have a somewhat different focus from the goal of Mobi, they produce interesting findings on the type of automations that may be generated for the mobile web using a programming by demonstration system.

2.6 Mobile Transcoders

Mobile transcoders share a similar goal with Mobi in that they are designed to automatically convert web pages into version fitting for a mobile device. However, these applications focus mainly on making web sites function on low-end devices. In other words, transcoders generally work by removing everything from a web page that requires advanced browser functionalities.

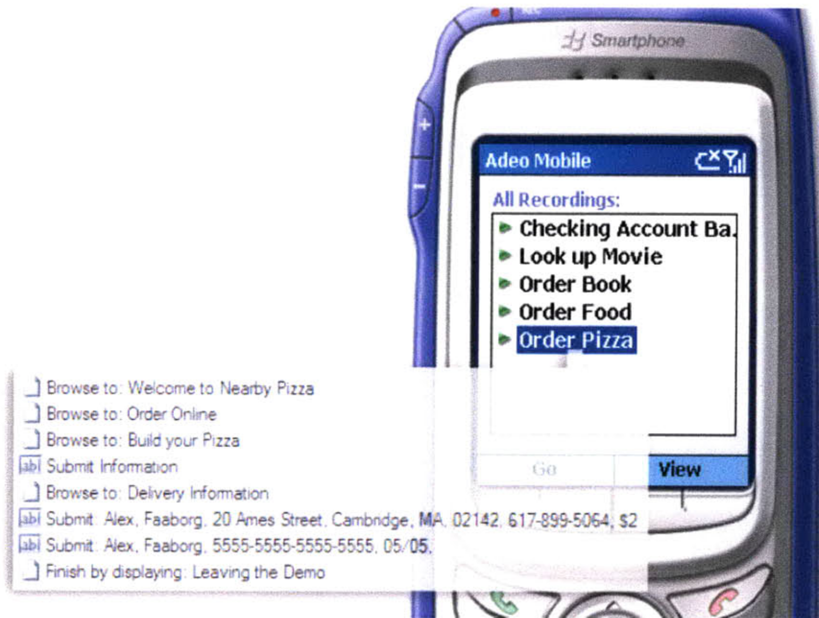


Figure 2-6: Menu for accessing and playing back a recording on Adeo.

Google Mobilizer, for example, is a transcoder that is built into Google search results that enable low-end smart phones to view web pages even if the web pages do not provide mobile versions compatible to these phones. It works mainly by removing many potentially unsupported CSS styles, as well as all the JavaScript code embedded on web pages.

Unlike mobile transcoders, instead of focusing on compatibility, the design of Mobi focuses on maximizing efficiency and usability of web pages on mobile devices.

Chapter 3

Design

Mobi is designed to require as little user intervention as possible. In other words, after setting up the Recorder on the desktop and the proxy connection on the mobile browser, no further configuration will be necessary. As such, the interface is designed to be as learnable as possible. Furthermore, minimalism is highly important due to the small screen on a mobile device. Overlays on the mobile browser included by Mobi must therefore take up as little screen real estate as possible.

This chapter describes several early ideas about the type of transformations to be applied, as well as several prototypes and design decisions of the implemented version. The following discussions assume that Mobi is able to evaluate and assess the relevance of various contents on a web page using the data from the Action Recorder. Section 4.2.1 describes the algorithm for assessing content relevance.

3.1 Early Ideas

The design starts off with several early ideas regarding what kind of transformation techniques should be applied so as to:

1. Maximize efficiency in accessing relevant contents. For example, the need for zooming and panning to look for contents should be reduced as much as possible.

2. Retain a sufficiently similar look to the original page. This way, user who had

used the desktop version in the past can adapt to using the customized version without having to relearn an unfamiliar interface.

Zoom and pan

The first idea considers zooming and panning to contents that are most likely interesting to the user. This reduces the need for the user to find the content and therefore improves the efficiency in accessing that content. However, it is likely for a page to have contents that are relevant to the user in different locations of the page. In other words, if the user wishes to access contents that are not heuristically assumed to be the most interesting, he will not be able to enjoy any improvement.

Copy to top

This idea considers the possibility of copying the most relevant contents to the top of the page. The user may then prioritize checking contents at the top of the page and only look at the rest of the page if he cannot find what he is interested in. While this offers higher efficiency since the user will only have to look at the top of the page most of the time, there will now be multiple copies of the same contents on the page and may potentially confuse the users. Furthermore, having multiple copies of the same elements in the page will likely interfere with the functions of the web page.

Deleting irrelevant content

This idea is perhaps the most aggressive technique. By deleting all content other than the most relevant contents in the page, this produces the least amount of contents in the page and makes finding contents in this view the easiest and most efficient. However, this also makes it impossible to access items that were not considered relevant by Mobi's heuristic algorithm.

Enlarge in place

Enlarging elements proportional to their assessed relevance ratings is another method for making relevant items more visible to the user. This method has the benefit of potentially retaining the same layout of the original page.

Snap-on panning

To minimize the amount of changes to the look of the page, we considered the technique of modifying the panning speed to make it easier to pan to relevant items. To be more specific, the browser may slow down panning speed when the viewport is close to relevant contents and increase panning speed when the viewport is close to relevant contents. This way, it becomes much easier to "snap" onto relevant items in the page. This method has the benefit of being able to not modify the styles and layouts of the page at all.

3.2 Design Prototypes

Of the early ideas considered above, three were chosen for testing and prototyping. In order to better assess the feasibility of these techniques, several prototypes were made prior to implementation. The computer prototypes discussed below were created by modifying screenshots of websites, adding basic simulated interactivity, and viewing these modified images on a mobile device. The sections below also include discussions of potential implementation challenges discovered during the prototyping process.

3.2.1 Enlarge in place

Prototypes for the enlarge-in-place technique are shown in Figure 3-1. The prototypes use *aa.com* as the web page to be customized, and assume that the most relevant contents are the forms on the right used for searching for tickets and checking flight statuses. As mentioned before, the enlarge-in-place technique increases efficiency for

accessing the relevant contents by making them more visible than the other contents on the page.

Several prototypes are created with different ratios in order to assess what ratio is best for the enlarge-in-place technique. We also considered the possibility of letting the user change the ratio by including a slider on the page that allows the user to select a ratio dynamically.

In the process of creating these prototypes, we discovered challenges regarding how to best apply this technique. As seen from the screenshots in 3-1, it is difficult to maintain the look of the page while enlarging certain elements in the page. Elements that are not enlarged, for instance, may be pushed aside by the enlarged items, in ways much different from how the original design expected. Furthermore, transformations via current versions of HTML and CSS do not yet allow for enlarging and rescaling of elements in a web page. Implementation of this technique will therefore likely require writing a very customized mobile browser with these functions.

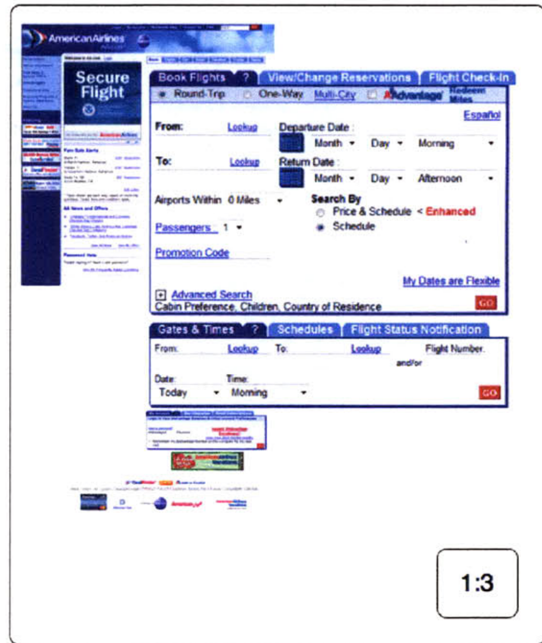


Figure 3-1: Prototypes for the enlarge-in-place customization, labelled by their ratios.

3.2.2 Copy to top

An example of the prototypes made for the copy-to-top technique is shown in Figure 3-2. By placing the relevant contents (the two input forms) in the top of the page, we may configure the browser to zoom in onto the top of the page when the page loads, giving user easy access to the most relevant contents on the page. If the user wishes to access the rest of the page, he may simply pan to the bottom.

Figure 3-2: Prototype for the copy-to-top customization.

As mentioned before, this has the negative effect of having multiple copies of the

same contents on the page. It is uncertain as to what would happen if the user is to interact first with the contents at the top, and then interact with the other copy on the screen. For instance, in Figure 3-2, the user may fill in the *From* field in the form on the top of the page and then continue to fill in the *To* field in the original copy.

One potential mitigation is to have the contents on the top of the page be a mirror image of the original content. For instance, we may watch for all events on the mirror image in the page and apply the same actions onto the original copies. However, having duplicates of the same content on a user interface is not ideal.

Another complication comes from the low learnability of this interface. Users who are not aware of what this prototype is supposed to do have trouble figuring out what it is doing. The contents on the top of the page look like they are a part of the original interface and can potentially confuse users.

3.2.3 Deleting irrelevant content

Figure 3-3: Prototype that customizes the page by hiding everything except for the relevant content.

The last prototype was made for the customization that removes everything on a web page other than the relevant contents. This technique, while simple, is effective in that it gives users access only to the relevant contents on the page, making it very efficient to find and access those contents.

Unfortunately, this relies greatly on the ability of the algorithm for determining which elements in the page the user is looking for. If the user wishes to access something that is not included in the customized view, this interface will not be usable.

3.3 Design Decision

Through the prototyping process, we are able to determine the strengths and weaknesses of the various customization techniques mentioned above. As a result, we decide to create Mobi using a combination of the techniques mentioned in sections 3.2.2 and 3.2.3.

Instead of simply deleting all the irrelevant contents from a web page, Mobi will allow the users to switch between the customized view and the full, original version. In other words, this is similar to the copy-to-top technique in that it will push the relevant contents to the top of the page, making them easily accessible as soon as the page load. Instead of panning to the bottom to access the rest of the contents, however, the user will instead switch to the full page mode by clicking a button on a toolbar Mobi overlays on the page. This, in effect, eliminates the issue of having multiple duplicates of the same contents on the screen. Furthermore, unlike the original idea of simply deleting all the irrelevant content from the page, the users are still able to access the rest of the page, giving this approach the advantages of both of these techniques.

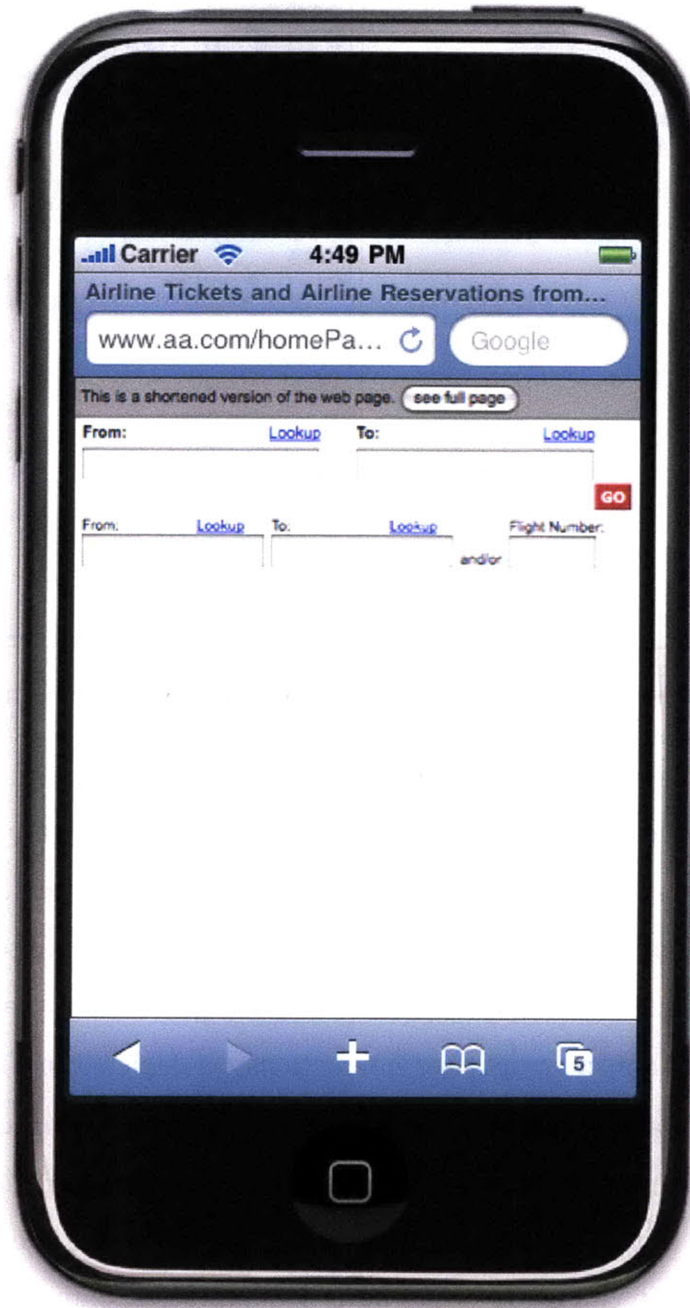


Figure 3-4: Customized version of aa.com created by the Mobi system.

3.3.1 Mode change between customized view and full page view

Mobi is designed so that interactions with a web page will continue seamlessly under both the customized mode and the full page mode. For example, if the user fills in

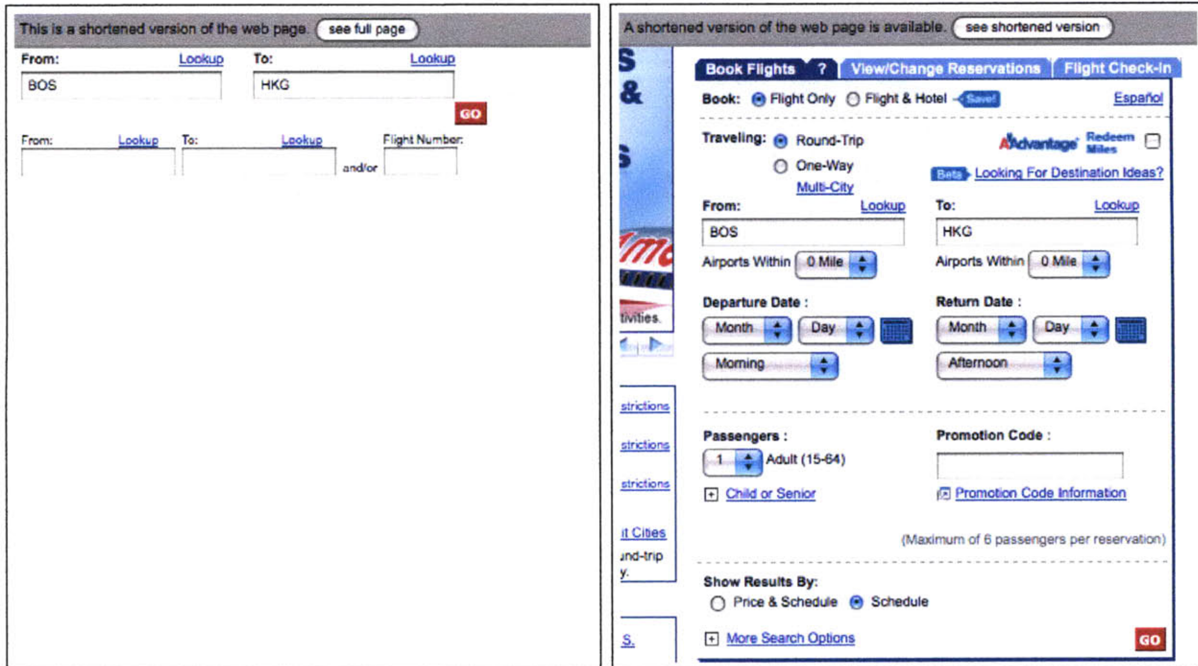


Figure 3-5: Example showing how user interactions may continue uninterrupted between mode changes.

the *From* and *To* fields in the customized view depicted in Figure 3-4 and realizes that there are fields in the original page they wish to access (such as the Departure Date fields), they may switch to the full page view without having to fill in the *From* and *To* fields again. In other words, the mode change between the customized and full page mode will not interrupt user interactions, as depicted in Figure 3-5.

This method of allowing users switch between modes, however, causes a different problem. When the user realizes they wish to look for something that may be in the original page but not in the customized view, clicking on the mode-switch button causes a very abrupt change to what he is looking at. Contents on the page are repositioned and resized very differently under these two modes. Because of that, user may find it difficult to find the elements he was interacting with previously and continue his work.

To mitigate this problem, we tried different techniques to make it more obvious to see the connection between contents in the customized view and their counterparts in the full page view. The first design attempts to do this by creating *ghost images* of the contents (semitransparent images of these contents) in the customized view

and showing an animation of these ghost images moving from their locations in the customized view to their corresponding locations in the full page view.

We eventually decided on a simpler design for the transition due to the ghost images approach being too distracting and ineffective on a small screen that requires panning and zooming for the animations to be visible. Instead, elements that are in the customized view are simply *highlighted* when Mobi switches to the full page mode. More specifically, these elements have background colors that change gradually from yellow to their original background colors. This type of effect is common in AJAX applications and is offered by various web interface toolkits such as *script.aculo.us*, used for drawing attention to dynamic changes in a page.

3.3.2 Dynamic interfaces

Mobi is also designed to support dynamic interfaces such as AJAX (asynchronous JavaScript and XML) applications. This type of applications often add or remove elements from a web page dynamically using JavaScript. On *facebook.com*, for example, within a profile page, switching between different tabs involve loading contents asynchronously and adding them onto the web page dynamically.

When faced with these applications, Mobi will maintain the customized view even when elements are added or removed dynamically. When elements that are considered relevant to the user are added to the page, these elements will appear dynamically as expected. Conversely, when elements considered irrelevant are inserted to the page, Mobi will hide them from view and only show them when the user switches to full page mode.

Chapter 4

Implementation

This chapter discusses the implementation of the Mobi system, as well as the algorithms involved in generating the customizations described in Chapter 3. As illustrated in Figure 1-2, the system consists of mainly the *Customizer*, a script written in JavaScript that is injected into the mobile browser as it visits web pages through the proxy server in Mobi; the *Action Recorder*, an extension on Firefox created using JavaScript with the Chickenfoot system; and the *Proxy Server* script injection components, consisting of a server running Squid [5] and an Apache web server, whose purpose is to inject the Customizer as well as provide access to action history data for the Customizer script.

4.1 Customizer

The Customizer script is responsible for applying the customizations as illustrated in Chapter 3 onto web pages. The Customizer is implemented in JavaScript and mainly works by manipulating the Document Object Model (DOM) [17] and Cascading Style Sheets (CSS) [16] of web pages. The discussion in the following sections assume being able to detect what the most relevant elements are on a web page using a heuristic algorithm that will be described in more detail in Section 4.2.1.

4.1.1 Removing irrelevant elements from view

The script starts by looking at the most relevant elements on the web page. The script then attempts to modify the page so that only the most relevant elements remain in view. One challenge in its implementation is to somehow accomplish this without overly modifying the structure of the page. The page must remain functional after the customizations are applied. Scripts in the page, for instance, may modify and make use of the DOM tree of the page at any time. Because of this, we must attempt to customize the page without having to add, remove or move nodes in the DOM tree.

To accomplish this, the Customizer traverses through the DOM tree: nodes that are considered highly relevant by the relevance algorithm and the descendants of these nodes will not be modified; nodes that are not considered relevant will be removed from view by making them invisible via changing their CSS styles; lastly, ancestors of relevant nodes must also remain visible (the relevant nodes that are descendants of these nodes will be hidden otherwise).

In order to make this work efficiently on mobile devices, the algorithm is optimized as follows: The script starts by getting the list of highly relevant nodes, and marks these nodes and their ancestors. It then parses the tree from the top. The algorithm only continues looking at children of a node if the node is an ancestor node of a relevant element. Nodes that are not relevant or ancestors of relevant nodes is set to be hidden and their descendants are also hidden as a result. Note that the algorithm also does not need to look at descendants of a relevant node because those should be remained untouched. Figure 4-1 illustrates this algorithm, where the nodes with dashed borders represent the nodes with CSS styles modified so they are hidden from view; the gray nodes represent elements that are hidden as a result; and the blue node with bolded border represents a highly relevant node in the page. In effect, this optimization allows the algorithm to only consider nodes that are immediate children of nodes marked as ancestors of relevant nodes instead of having to parse the entire tree.

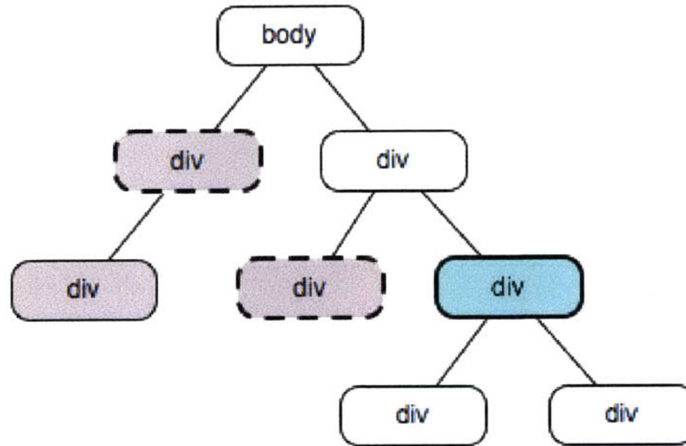


Figure 4-1: Illustration of the implementation for hiding irrelevant elements by traversing through the document's DOM tree.

Reducing white space

Under the algorithm described above, styles of the ancestors of relevant nodes are not modified. Under this approach however, lots of white space, and, hence, valuable screen real estate, are wasted due to the spacing and padding in the CSS styling of the ancestor nodes that are originally intended in the original page to separate contents. Much of this white space is no longer necessary since much of those contents are now hidden from view.

We therefore designed the Customizer to remove these white spaces mainly by looking at the CSS styling of these ancestor nodes and selectively removing padding and margin when possible.

This, however, must be done carefully. It is not uncommon for web pages to have designs where these CSS styles are not only used to add spacing between elements, but also to tweak how elements are positioned. Therefore, it is possible that, after removing these styling properties, elements may overlap with one another. To avoid this problem, the Customizer checks, after applying these customizations, whether elements from different subtrees overlap with one another. If elements are found to overlap with one another, the Customizer must undo the CSS style changes on the ancestors of these elements. To be more specific, for pairs of elements that overlap and are not ancestor of one another, CSS changes on all ancestors of the two ele-

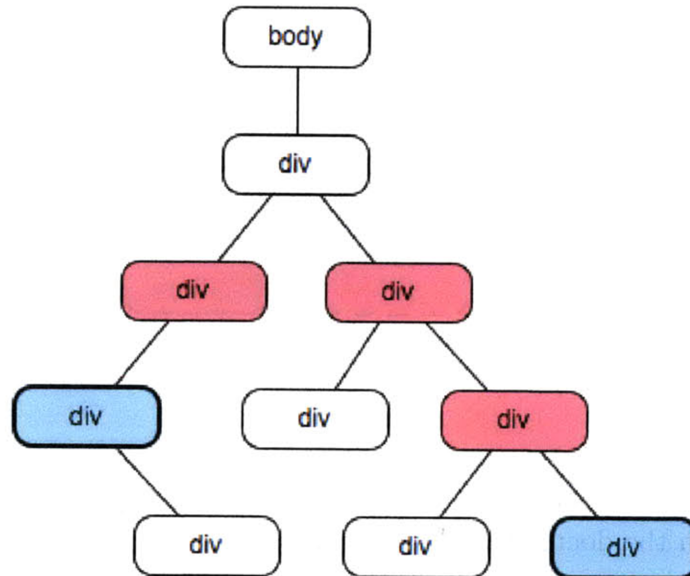


Figure 4-2: Illustration showing which ancestor nodes must have their CSS style changes undone when two relevant nodes overlap as the result of the style changes.

ments (but not common ancestors between the two) are undone. Figure 4-2 provides an illustration of what happened when two nodes (marked with blue background) overlap. The nodes with red backgrounds represent ancestor nodes that have their style changes undone.

4.1.2 Labels and captions

When Mobi determines that elements such as text boxes and images are highly relevant and must be included in the customized view, Customizer also makes sure that their corresponding labels or captions are included in the view as well. In the example of *aa.com* shown in Figure 3-4, for instance, labels for the *From:* and *To:* fields are also included in the customized view.

This is done mainly by having the Customizer look for elements close to images and input fields that are marked as relevant, and include those in the customized view as well. More specifically, the Customizer looks at each element that is marked as relevant. It traverses the subtree under the ancestor several levels above the marked element, looking for elements that are enclosed by the rectangle centered at the marked element, with twice the size of the marked element. Figure 4-3 illustrates how this

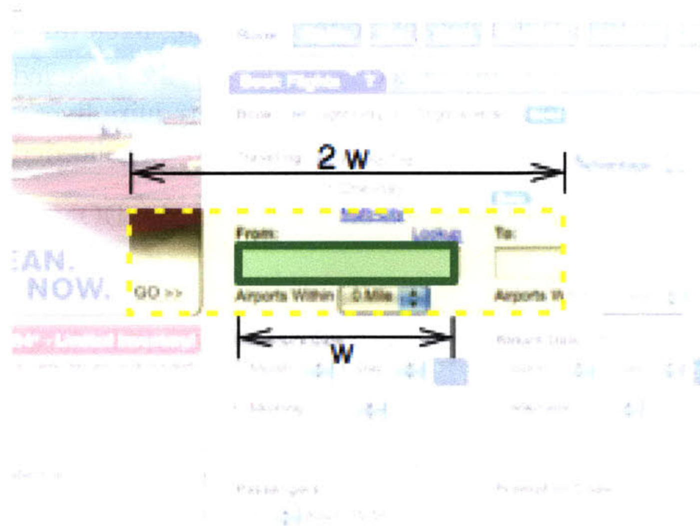


Figure 4-3: Illustration showing the area under which the Customizer looks for label elements for a text box.

works: the text box marked by the green box is the element marked as relevant, while the rectangle with yellow dotted border represents the area under which the algorithm looks for elements that are potentially labels for the input field. Note that, in the case of images, instead of double the size of the marked element, a constant value is added to width and height.

4.1.3 AJAX and Dynamic changes

To ensure that the customized version works for dynamic interfaces, the Customizer listens for changes in the DOM. When a change occurs, the Customizer ensures that the elements that are affected appear or disappear according to their relevancy. In other words, after each change to the DOM, the customized version should look as if customizations have been reapplied to the page with the new DOM structure.

However, it is infeasible for the Customizer to reapply the entire script every time a change occurs to the page, due to the potentially very high frequency of dynamic changes in advanced interfaces. A typical AJAX call and interface change, (as is the case with the AJAX search interface on *google.com*) for instance, may cause more than 50 changes to the DOM within less than a second. Because of this, the Customizer

Type of parent	Inserting relevant node	Inserting irrelevant node	Removing node
relevant node	no fixes needed	no fixes needed	no fixes needed
ancestor of relevant node	no fixes needed	require fixes	no fixes needed
neither	require fixes	no fixes needed	no fixes needed

Table 4.1: Different cases of changes on the DOM tree.

must be designed so that, after these dynamic changes occur, it should take as little computations as possible for the customized interface to quickly adapt and fix itself.

To achieve this, the Customizer installs a listener on the DOM tree that gets notified whenever a node is inserted or changed. The script must then apply fixes depending on whether the node inserted is considered relevant, as well as the type of parent node this node is inserted into to. An overview of different cases of these events is shown in Table 4.1.

The main types of events that cause problems are ones where irrelevant nodes are inserted into parent nodes that are ancestors of relevant nodes; and when relevant nodes are inserted onto parent nodes that are hidden. In the first case, the irrelevant nodes being added to parent nodes that are ancestors of relevant nodes will cause them to be visible (since the ancestor nodes are visible). In the second case, relevant nodes that are inserted onto parent nodes that are irrelevant and therefore hidden will cause those relevant nodes to be hidden as well.

In order to fix these problems, upon changes to the DOM, the Customizer script looks at the node that is added. If the node is assessed to be irrelevant, and if the parent is marked as an ancestor node, we hide the node by modifying its CSS styles. On the other hand, if the node is assessed as relevant to the page, we must check to make sure the parent is not hidden. This is done by tracing through its ancestors. If the first marked ancestor encountered is a relevant node, we can be sure that the parent is visible (since it is a descendant of a relevant node) and no modification is necessary. However, if the first marked ancestor is marked as an ancestor of a relevant node, this indicates that the parent is hidden and all ancestors of the added node are modified to have their styles reset to be visible.

4.1.4 Mode change to full page view

As discussed in the design, the user must be able to switch between the full page view and the customized view dynamically. To achieve this, upon clicking the mode switch button, the Customizer traverses the tree. For each element that had their CSS styles changed, we make sure to have their original styles saved in their JavaScript object prior to the changes. By doing this, switching mode simply requires resetting the styles of the elements back to the ones previously saved.

4.2 Action Recorder

The Action Recorder is the component responsible for monitoring user actions on the desktop and using these action histories to evaluate relevance ratings of the elements inside web pages.

The Action Recorder is written as a Firefox extension built using Chickenfoot. The Action Recorder is basically a script that runs every time the user visits a web page. The script installs listeners that monitor clicks and inputs on a web page, as well as a *View Tracker* that keeps track of what contents the user is looking at on a web page. Using these data, the Recorder is able to heuristically compute scores for different elements on a web page representing an estimate of how relevant they are to the user. These scores are saved into a database and are therefore accessible by the Customizer.

4.2.1 Heuristics for recognizing relevant contents

The algorithm Mobi uses for assessing and estimating how relevant contents are on a web page depends on the assumptions that:

- Elements that user interacts with frequently are highly relevant to the function of the web page;
- Elements with readable contents that the user spends a long time looking at are also highly relevant to the web page's function.

Detecting what the user interacts with directly is fairly easy. The Action Recorder sets up listeners that listen to clicks over the web page and records the targets of the clicks (whenever the targets are input fields or links). Similar listeners may be set up for other user actions.

Determining what the user is looking at, however, is very challenging. The most accurate way to do this is clearly with eye-tracking apparatus similar to the ones described in Jakob Nielsen's research on using eye-tracking technology for usability studies [14]. However, Mobi is developed for use by normal end-users and we will therefore attempt to use some simple heuristics to simulate this type of functions by monitoring user's actions in the browser.

The Action Recorder includes a View Tracker that keeps track of the contents within the browser's viewport. One basic assumption it makes is that the user must be looking at contents within the viewport. By doing so, it can increment the relevance scores for elements that contain readable contents and are *mostly* within the viewport of the browser. To determine whether elements are mostly in the viewport, we use a basic calculation: (where C represents the area of the content element, and V represents the area of the viewport)

$$\max\left(\frac{C \cap V}{C}, \frac{C \cap V}{V}\right)$$

This basically computes a value representing how much of a certain element is inside the viewport, and, in the case that the content is very large, computes how much of the viewport is used for displaying the element. The View Tracker increments the score for the element if this value is above a threshold of 50%. The scores of these elements are incremented periodically (every 2 seconds) to track how long they have been in the view.

Additionally, to increase accuracy of the View Tracker, instead of simply incrementing the scores periodically, the Tracker only does so when the web page has *focus*. This way the Tracker will not fire when the browser is not actually in the screen. Furthermore, to make sure the Tracker doesn't fire unnecessarily, when the

user interacted with the page (e.g. clicked on a button or entered something into a text field), which indicates that the user is probably not reading the page, the Tracker will delay firing again for a period of time. Similarly, scrolling the page quickly will also cause a delay to the Tracker.

To order for the Customizer to be able to figure out what the most relevant contents are in a page, the Action Recorder keeps track of an integral *score* for each relevant element. The View Tracker increases the scores of the contents in view by 1 every time it runs, while interactions such as clicking increases the scores by 4 per click. In other words, each click on an element is regarded as roughly equivalent to having the element in view for 8 seconds.

Originally, the View Tracker is implemented to only consider elements that contain *readable contents*: e.g. elements with lots of rendered text or ones with large images. However, this proves inadequate as much of the web today is structured so text and contents are divided into many elements. For instance, an HTML table is a common method for presenting contents, but is not recognizable under this method because each cell in a table is unlikely to contain a lot of text.

We therefore added another method for identifying content elements: using the number of repeated similar elements in a node. The *google.com* search results page, for example, contains a list of search results, each having very similar styles and look. The View Tracker should, therefore, attempt to also detect these lists as content elements and consider them relevant if the user spends a long time looking at the lists.

To do this, we notice that similar elements often share the same CSS class names and are of the same types of nodes (e.g. DIV, TR, P). Therefore, the View Tracker will consider elements that have children of similar style classes and node types content elements and update their scores accordingly.

To implement this efficiently, we have the script traverse the entire DOM once after the web page finishes loading. For each node, the script keeps a count of the types of classes seen in its top two levels of descendants (children and grandchildren). The node is considered a content node if a significant fraction of its children or

grandchildren share the same class. The View Tracker then periodically checks if the content elements on the page are inside the viewport and updates the scores accordingly. Note that, to cope with pages with dynamic contents, the script also periodically traverses the DOM to look for new content nodes.

4.2.2 Applying action history data

The scores representing the relevance of elements on webpages are stored in a database, identified by the URL of the web page and the XPath [18] identifying the elements. In order to make the system robust and generalizable, the XPaths are constructed to be as general as possible. For instance, whenever possible, CSS class names and ID names will be used instead of HTML tag names. This decreases the dependence of the XPaths on the exact DOM structure of the web page and allows history data from one page to be useable on different pages as well.

As mentioned previously, the Customizer script requires using history data in creating customizations specific for the web page it is running on. To do so, the script always attempts to look for data for URLs that matches the web page's location most accurately. More specifically, if exact URL matches cannot be found, the script will use data from a URL that shares the longest prefix string with the page's location.

4.3 Proxy Server and Script Injection

The proxy server uses Squid, a caching proxy that supports all common protocols such as HTTP and HTTPS [5]. It is configured to inject a SCRIPT tag onto all web pages. The SCRIPT tag links to a JavaScript script that bootstraps the Customizer script as well as requests history data from the database corresponding to the URL of the web page.

Note that one small challenge in this approach is that, because the database resides on a different domain from that of the current web page, the script may not make direct AJAX calls to the server. Instead, it creates another SCRIPT element that loads the data from the database in JSON format and fires a callback upon

completion, starting the Customizer script and providing it with the necessary history data.

4.4 Implementation on Mobile Browsers

4.4.1 Compatibility Issues

While mobile browsers such as the iPhone's Safari browser and the Android browser both support most of the features in modern web browsers such as JavaScript, various quirks and challenges were discovered during implementation.

Both browsers lack support for XPath evaluation (the Android browser has an interface for accessing XPath evaluation functions, but are not yet implemented), which is necessary since the history data identifies elements on a web page by XPaths. As a workaround, Mobi uses Google AJAXSLT [12], an old AJAX library made in 2005 when XPath is not yet commonly supported. This library provides a method for doing XPath evaluation, although it is clearly less efficient than the native functions provided on desktop browsers.

Another issue is the absence of the *getBoundingClientRect* [10] method on iPhone's Safari. The method is needed by functions such as finding labels and captions as described in Section 4.1.2 and finding overlaps between elements of different subtrees as described in Section 4.1.1. While a workaround exists (traversing through all the parents of an element and summing up their position offsets from their respective parents), again, this creates another performance hit in comparison to having a native method for computing an element's location and dimension.

4.4.2 Zooming functionalities

Another implementation issue comes from the lack of a way to dynamically zoom in/out or change the scale of a web page. While both the iPhone and Android browsers allow the users to manually zoom in and out via clicking a button or using gestures, neither of which provides another programmable interface to do this from

client-side JavaScript.

Part of the designs of Mobi's customizations include being able to zoom in so that the customized view fits the screen on page load and zoom out automatically when the user switches to full page mode. In order to prototype this idea, Read4Me, an experimental multimodal web browser built on top of the Android browser, is modified to include a JavaScript API for zooming. Mobi's functionalities, integrated with the zooming functions are successfully developed and tested on the Read4Me browser.

Chapter 5

Evaluation

This chapter provides an evaluation of the Mobi system as well as discussions over the findings from the evaluation. Mobi is tested and evaluated over a variety of different web sites. Various tasks are performed and recorded on the desktop browser on these web sites and the mobile customized version created by Mobi are evaluated formatively.

Functions of the interfaces in the websites evaluated may roughly be categorized as follows:

- Readable contents: pages whose main purpose is to provide contents. Article pages on news sites such as *nytimes.com* are examples of this type of interface. These pages often contain mostly static readable contents rather than interactive elements.

- Search: a feature provided by many web sites for locating contents. The search engine functions on *google.com* is one example of this feature. The interface for looking for tickets on *aa.com* may also be considered a search interface. This type of interface often contains a form of one or more input fields for specifying the parameters of the search and a page listing the results of the search.

- Function-based: interfaces that serve certain specific functions and often have side effects. For instance, the interface for composing an email, submitting a comment or making a bid on an auction site all fall under this category.

Various web sites are qualitatively evaluated based on how successful tasks of the categories above continue to function on the customized version generated by Mobi. Web sites that generated particularly interesting findings are discussed further in the following section. A list of all the web sites and their corresponding results are summarized in Section 5.2.

5.1 Results

The results of an end-to-end evaluation of the Mobi system is shown in the following sections. To perform this evaluation, we first performed various tasks on these web sites, allowing the Action Recorder to capture our actions on the page and assess which elements on the page are most relevant to the user. The Customizer then uses this information to apply customizations onto the page.

5.1.1 *aa.com*

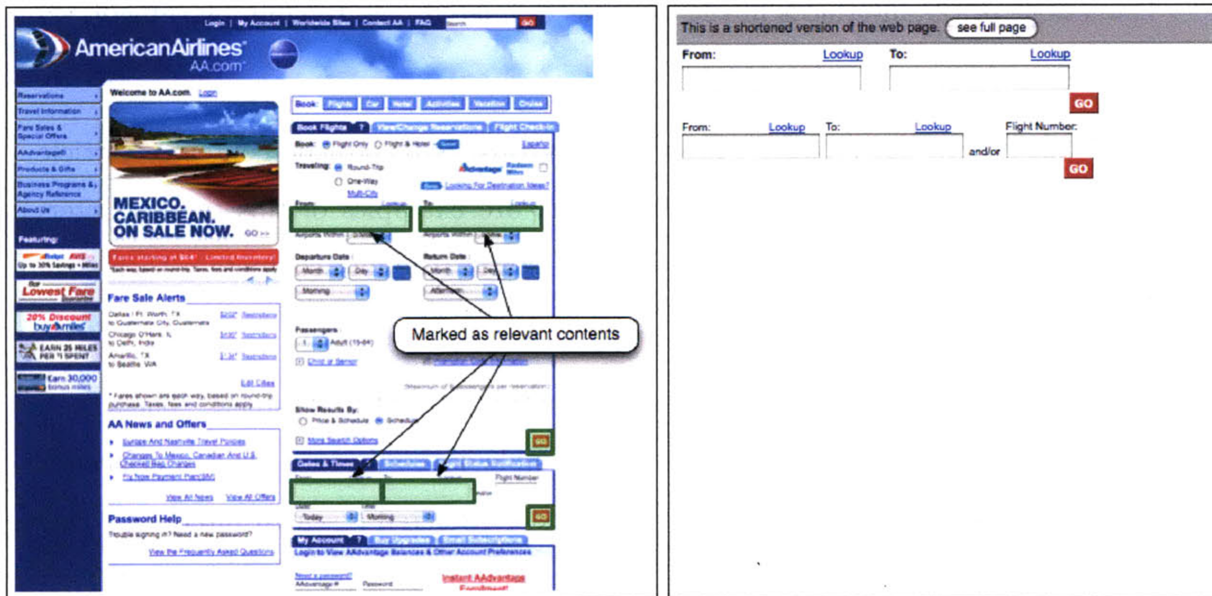


Figure 5-1: Evaluation of Mobi running on *aa.com*, showing the elements marked as relevant and the corresponding customized version using this data.

The home page of *aa.com* was used as the basis of the prototypes in Mobi's design. We first perform tasks such as searching for flight tickets and checking flight statuses.

The implemented Mobi system is able to correctly detect the input fields necessary for doing these tasks as relevant contents and generate result similar to the original design as described in Section 3.3, as seen in Figure 5-1.

The search interface for flight tickets and flight statuses work in the same ways as they do in the original interface, including the dynamic auto-completion function for the location input fields, which continues to work properly in the customized view. Submitting the form takes the user to the results page showing a list of tickets they may purchase, as it did on the desktop interface.

The customization offered by Mobi successfully removes everything in the page other than the fields the user needs to complete their task of finding tickets or checking flight statuses, making it a significantly more efficient interface than the original desktop version. If the user wishes to access the rest of the page, the mode switch button on the top allows him to easily switch back to the original view. The transition flashes the fields he sees in the customized view, allowing him to continue interacting with those elements.

5.1.2 *google.com*

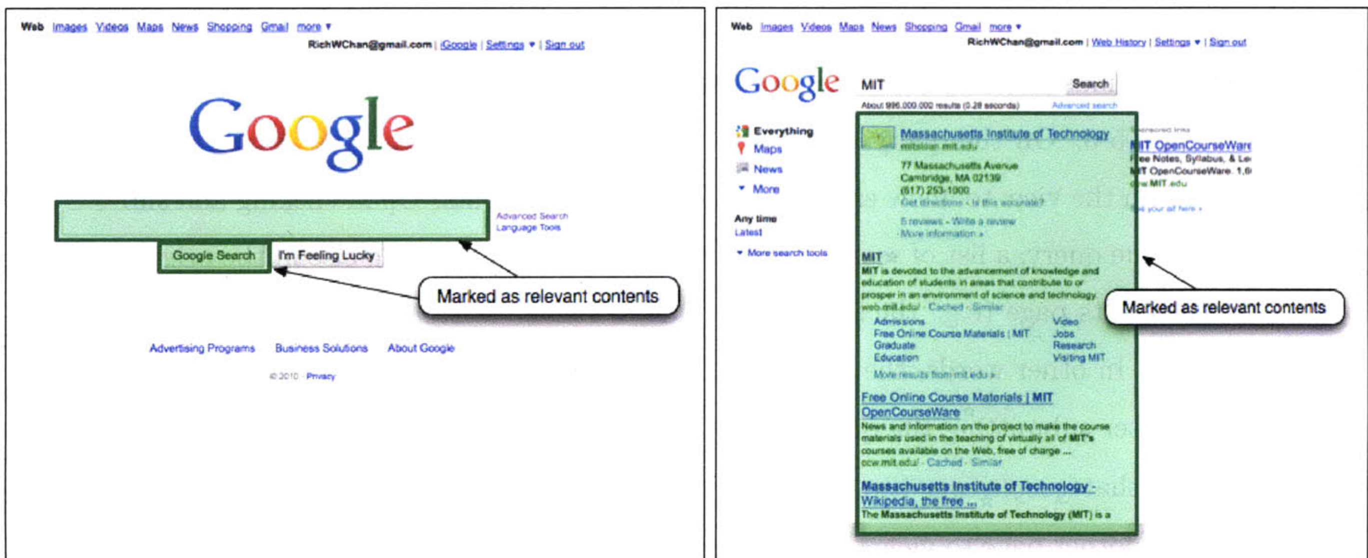


Figure 5-2: The search interface on *google.com*, with elements determined to be relevant marked in green.

The search engine *google.com* is a good representative of common search interfaces, and is also one of the most popular websites on the internet. The interface of the search engine mainly consists of the input form in the home page and the results page that appears after submitting a query, as seen in Figure 5-2.

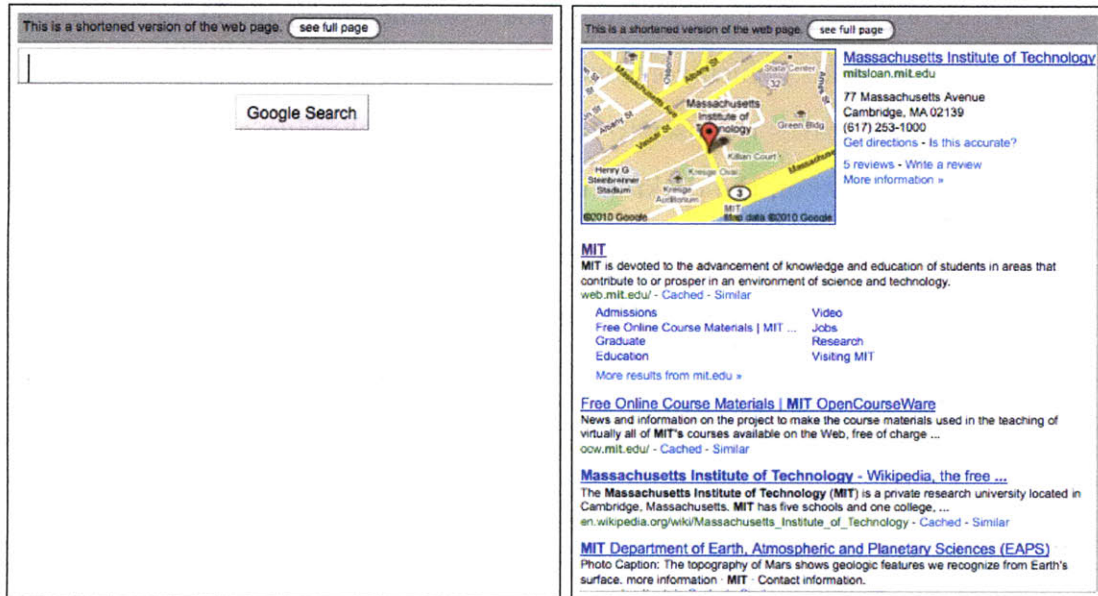


Figure 5-3: *google.com*'s interface after applying Mobi's customizations.

After demonstrating the tasks of inputting search queries and viewing search results in the results page, Mobi was able to correctly detect the input field for the search query, the button for searching, and the list of search results as relevant contents. The customizations made by Mobi also runs successfully on *google.com*'s interface, as seen in Figure 5-3. On the query page, only the query input box and the search button remains in the view to allow efficient access to these elements. Upon clicking and submitting the query, a list of search results is shown while everything else in the original search results page (e.g. the sidebar and ads) considered irrelevant is removed from the view. In other words, the user is able to immediately view and access the search results after submitting the query.

Note that *google.com* uses AJAX in their retrieval of search results. Instead of loading another page to view the search results, in order to reduce latency, *google.com* queries its server via JavaScript and displays the results asynchronously onto a DIV element that is added dynamically onto the page. In effect, *google.com* shows an ex-

ample of Mobi working correctly under dynamic interfaces, as intended in the original design described in Section 3.3.2.

5.1.3 *wikipedia.org*

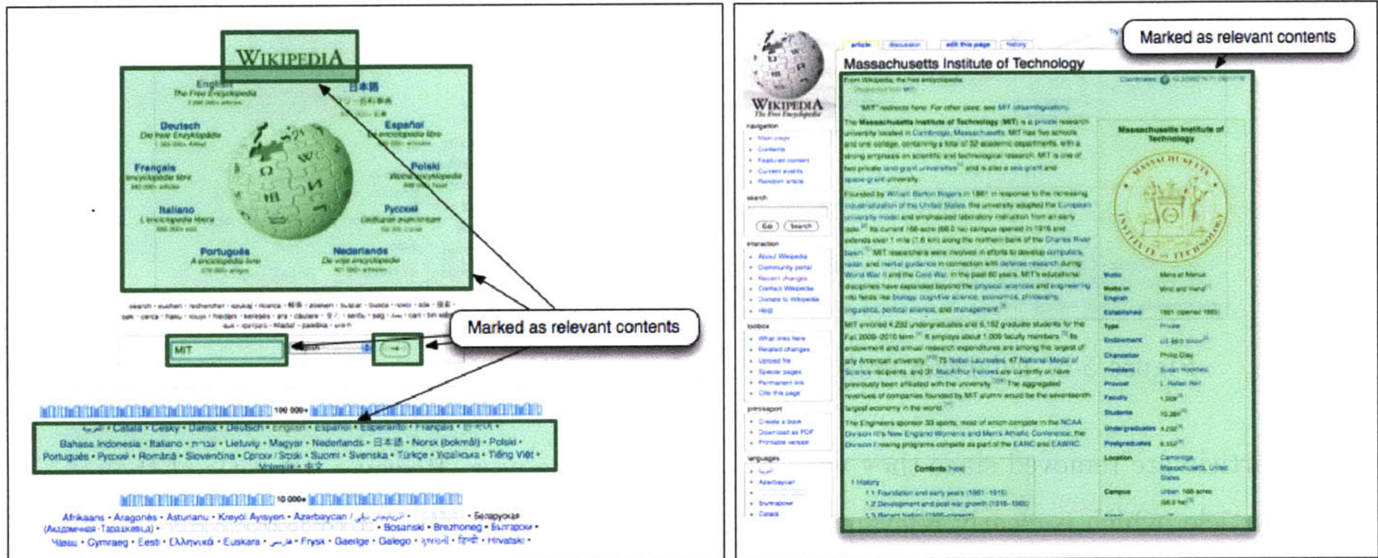


Figure 5-4: The search and article interface on *wikipedia.org*, with elements determined to be relevant marked in green.

wikipedia.org, a web-based collaborative encyclopedia, gives an example of an interface that contains functions for both searching and serving readable contents. As shown in figure 5-4, *wikipedia.org* consists mainly of a search page and a encyclopedia article page corresponding to the entry the user searched for.

The customizations made by Mobi remove various unnecessary decorative elements from the search page, such as the image of a stack of books near the bottom of the page; as well as the many various language options that were never used or looked at. Note that elements such as Wikipedia's icon and the list of common languages remain in the customized view. This is because they were inside the browser's viewport for a significant period of time, causing Mobi to consider them potentially relevant content elements.

After searching for a term, the article page is shown with only the contents of the article. Elements such as the sidebar and options such as discussing and editing the

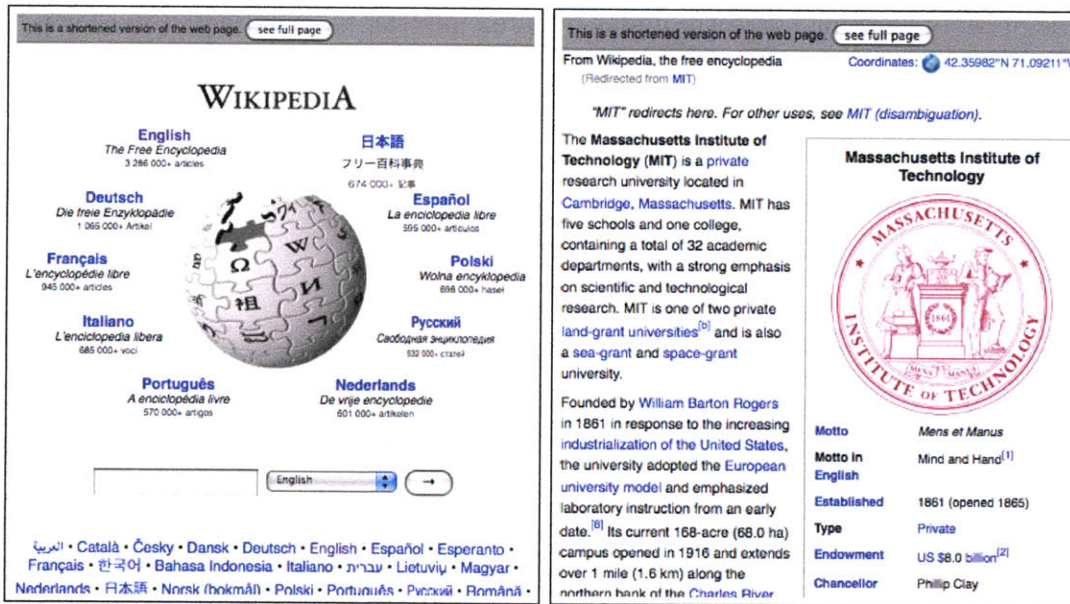


Figure 5-5: *wikipedia.org*'s interface after applying Mobi's customizations.

article are removed since they were never used by the user. However, the title of the article, which is clearly relevant to the function of the page, is also removed.

Note that the customization is not specific to only the articles with recorded history. Mobi is able to apply the same customization as shown in Figure 5-5 to different articles successfully, giving an example of Mobi's ability to apply action history across similar pages, as described by Section 4.2.2.

5.1.4 *webmail.mit.edu*

webmail.mit.edu is a web-based email system with a fairly common interface. For this interface, we tests specifically functions such as composing emails to make sure functions of the original interface is retained in the customized interface Mobi creates.

As seen in Figure 5-7, Mobi retains all the elements necessary for reading the list of emails and composing new emails. In the email composition page, Mobi keeps all the elements within the form, and the functions that existed in the original page continues working as expected in this view.

While the page functions fine, Mobi is keeping more elements in the screen then necessary. Ideally, since there are fields in the email composition page that the user

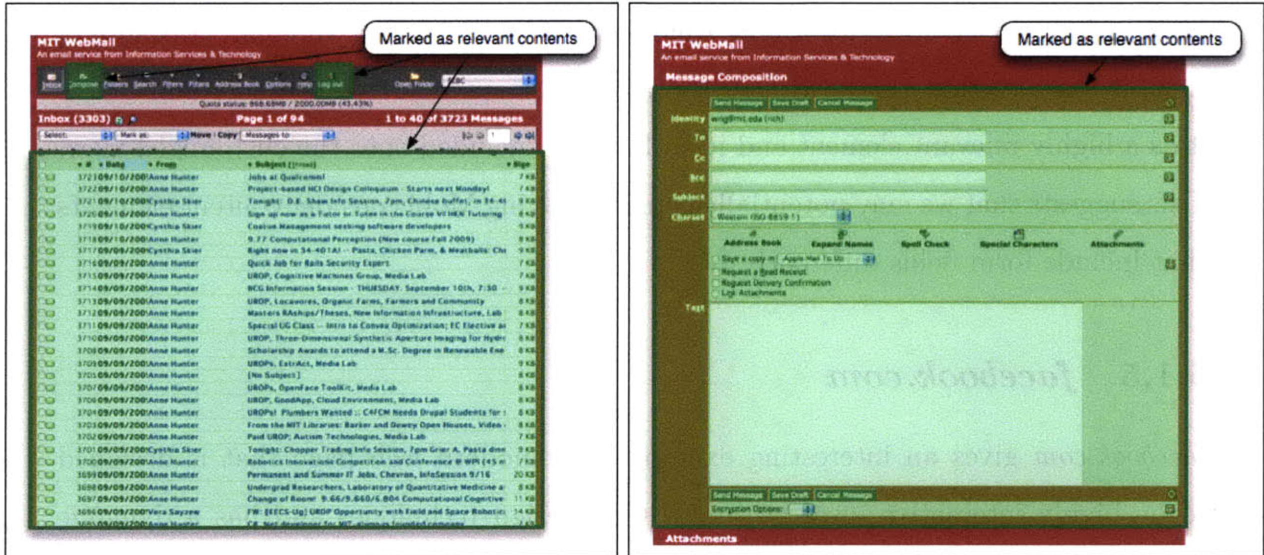


Figure 5-6: The interface on *webmail.mit.edu* for viewing and composing emails, with elements determined to be relevant marked in green.

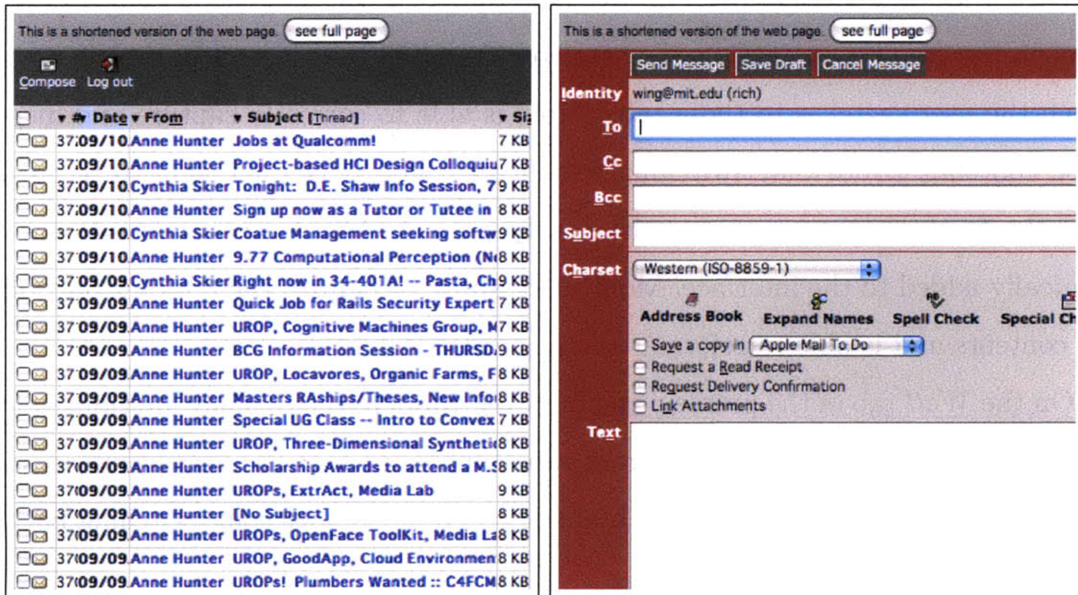


Figure 5-7: *webmail.mit.edu*'s interface after applying Mobi's customizations.

has never used before, such as the *Charset* field and options such as Requesting receipts, it would be more efficient to have those items removed from the customized view as well.

The reason for Mobi to not be able to do so is caused by how the algorithm for detecting relevant contents was designed (see Section 4.2.1). When the Action Recorder is monitoring the user's actions on the page, it detects that the user had

been looking at the form for a long time while he was drafting an email. Because of that, it assesses that the form may contain important readable contents and marks it as a highly relevant element and it is therefore included onto the customized view. This suggests that we may potentially consider having Mobi consider content elements that include form fields differently.

5.1.5 *facebook.com*

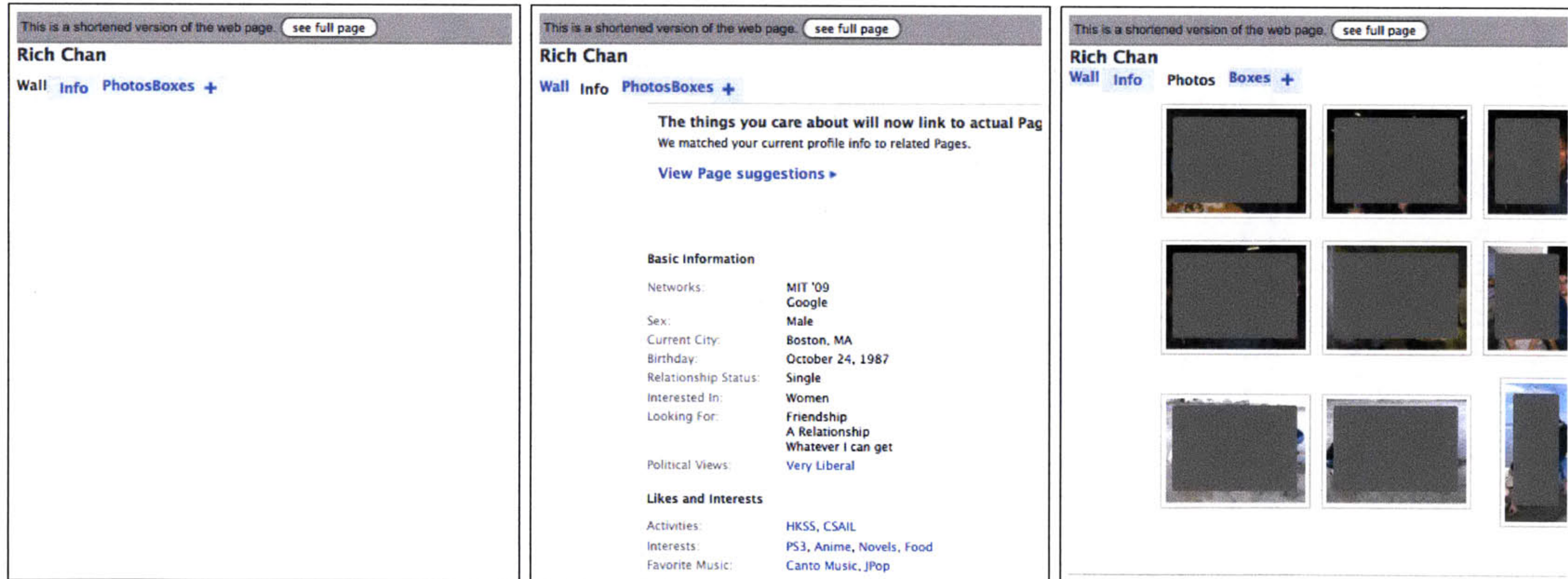
facebook.com gives an interesting example of a modern web page that is designed to be highly interactive, uses advanced CSS design techniques and has a dynamic interface that uses AJAX and loads various components asynchronously to decrease latency. As shown in Figure 5-8, Mobi succeeds in certain areas and fails in others.

By monitoring user actions, Mobi is able to detect correctly the information contents on the *Info* tab, as well as the area containing all the pictures in the *Photo* tab. When the user switches to those tabs, Mobi is able to correctly capture and display these contents. Note that switching to different tabs causes contents to be loaded from the *facebook.com* server asynchronously using AJAX. As these contents are dynamically added to the interface, Mobi successfully recognizes the relevant portion of the contents and is able to reformat the page accordingly.

On the *Wall* tab of the profile page, however, Mobi fails to display anything (other than the links for switching to different tabs). This prevents the user from accessing functions such as viewing wall posts and adding comments without switching back to the full page mode.

The reason behind this is the failure of the relevance assessment algorithm to correctly recognize the wall messages and comment input form as relevant contents. This is caused by the way *facebook.com* designs its AJAX interface for loading the contents of the *Wall* tab: in order to allow JavaScript that is included onto the page asynchronously after the contents of the *Wall* finishes loading, the element containing the wall posts is assigned a unique ID every time the contents of the *Wall* tab updates. Because of the way action history data references elements via XPaths (see Section 4.2), changing the ID will cause references to elements previously recorded by the

(a) The interface on *facebook.com*'s profile pages, with elements determined to be relevant marked in green.



(b) *facebook.com*'s interface after applying Mobi's customizations.

Figure 5-8: Evaluation of Mobi running on *facebook.com*, showing the elements marked as relevant and the corresponding customized version using this data.

Action Recorder to fail. As such, the relevance assessment algorithm fails to apply history data that references the *Wall* elements, and those elements will consequently never be recognized as relevant.

5.1.6 *expedia.com*

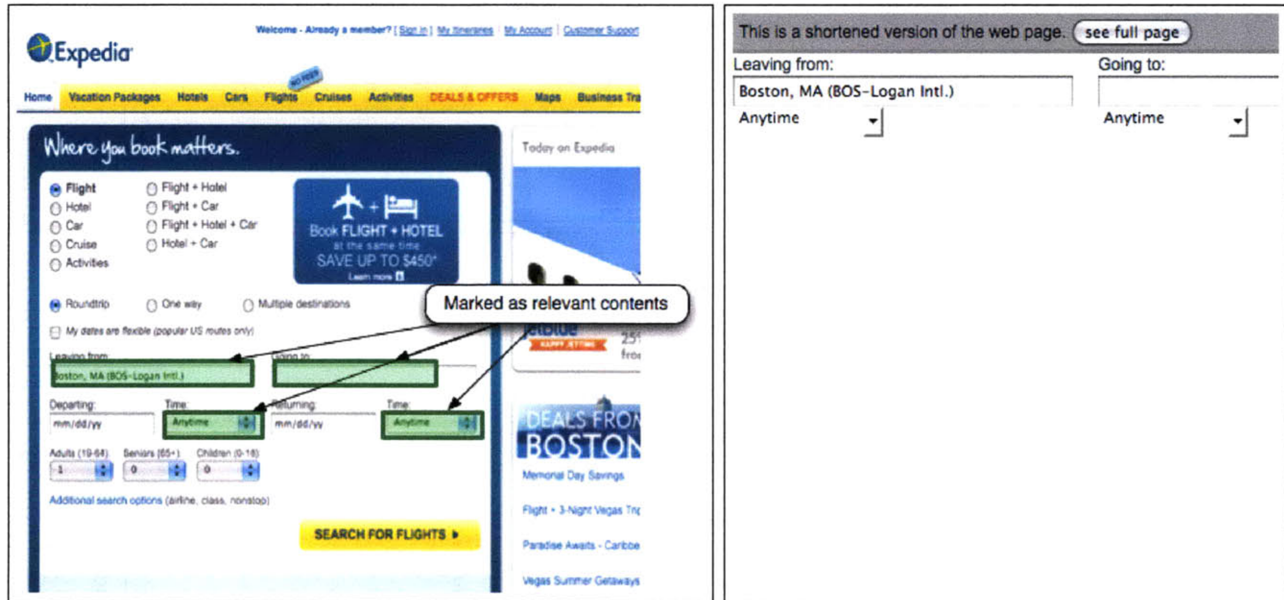


Figure 5-9: Evaluation of Mobi running on *expedia.com*, showing the elements marked as relevant and the corresponding customized version using this data.

expedia.com gives another example where Mobi fails to successfully include all contents necessary for performing the site's functions. As shown in Figure 5-9, Mobi fails to detect the *Search for Flights* button as a relevant element despite the fact that we did demonstrate clicking on it to the Action Recorder. This prevents the user from being able to perform the page's function (for searching flights) since he is unable to click on the button in the customized view.

The failure to capture the *Search for Flights* button is caused by how *expedia.com* uses unconventional HTML elements as interactive elements. Instead of actually using an INPUT button element, the *Search* button is in fact a DIV element with background images, with a listener programmed in JavaScript listening to it click events. Because the Action Listener only captures events on elements such as input

elements and links, clicks on the image are not detectable and the relevance score for the *Search* button is never incremented.

5.2 Summary

Table 5.1 provides a summary of how Mobi works over a variety of different websites, evaluated under aspects such as:

- whether the layout and styles of the customized view remain consistent (*layout*);
- whether the customized view successfully hides the irrelevant contents and thereby improves the efficiency of the page (*efficiency*);
- and whether the customized view is able to successfully retain all the relevant contents for the page to serve its original function (*functionality*).

5.3 Discussions

As seen in Table 5.1, Mobi works well on a majority of the websites tested. Only two websites have functionalities that cannot be provided by the customized view (*facebook.com*, whose problems regarding dynamic contents as described in Section 5.1.5 prevent functions such as posting wall messages; *expedia.com* has interactive elements that are actually DIV elements, as described in Section 5.1.6).

While the rest of the websites have customizations that function properly, various issues and limitations of the Mobi system's customizations techniques are discovered. The following section discusses these limitations.

5.3.1 Limitations

As a by-product of removing irrelevant contents from the page, it is unavoidable to modify the placement and positioning of various elements in the page. Because of that, the styles of the original page that was designed for the original view may

Website	Layout	Efficiency	Functionality
google.com	good	good	good
wikipedia.org	good	good	good
webmail.mit.edu	good	okay	good
nytimes.com	okay	good	good
facebook.com	okay	good	bad
ebay.com	good	good	okay
craigslist.com	good	good	okay
cnn.com	okay	good	good
flickr.com	good	good	good
weather.com	good	good	okay
cnet.com	good	good	good
ehow.com	okay	good	good
amazon.com	good	good	good
yelp.com	good	good	good
yahoo.com	good	good	good
expedia.com	good	good	bad
imdb.com	good	okay	good
mapquest.com	okay	good	good
reference.com	good	good	good
usps.com	good	good	good
stellar.mit.edu	good	good	good

Table 5.1: Summary of evaluating Mobi over different web sites.

no longer apply well on the customized view. This is especially the case for web sites with creative interfaces, either for aesthetics or optimization purposes. For instance, web sites that uses CSS to position elements at exact locations; web sites with elements that overlap; and web sites that have backgrounds with assumptions about the locations of various elements are all likely going to have customizations that fail to preserve the original layout.

The evaluation also shows weaknesses that exist in the algorithm for detecting relevant elements. The *functionality* issues are often caused by the algorithm not recognizing elements that should be included onto the view (false negatives); while the *efficiency* issues are caused by the algorithm overly estimating the relevance of too many elements (false positives).

These types of limitations clearly require further exploration and will be discussed in more details in Section 7.

Chapter 6

Conclusion

Mobi provides a system that automatically customizes web pages for mobile devices without any user intervention. Web interfaces designed for desktop use are made to be more efficient and easier to use on mobile devices, without sacrificing the functionalities of their original desktop versions. Using data that it collects from desktop browsers passively, Mobi is able to provide customizations that are specific to the tasks relevant to the users.

As presented by this thesis, the design and implementation of Mobi encounter various challenges, including the need to design customization techniques under the constraints of mobile devices such as small screens and difficult input methods; the need for these customizations to work over designs and layouts that differ greatly from website to website; the various limitations of web browsers and HTML specifications; as well as the need of a method to accurately assess how relevant various elements on a web page are to the user. While Mobi clearly still has flaws, as shown by the evaluation in Section 5, the current implementation of Mobi shows promising results and I expect high potentials from future iterations.

6.1 Contributions

In summary, Mobi makes the following contributions:

- The design and implementation of a mobile web page customization system that

works on all major smart phones today.

- The design of various customization techniques applicable to mobile devices.
- Methods for assessing the relevance of various elements on a web page using data about user behaviors on the web page in a desktop environment.
- A qualitative evaluation of this automatic customization technique applied over many different web pages.

Chapter 7

Future Work

7.1 Further developments

While the first working prototype of Mobi is complete, many areas of the system deserve further exploration and development.

As mentioned in Section 5.3.1, more research needs to be done to develop better methods in keeping the layout and design of the page consistent when applying customizations onto the page. Keeping the design consistent with the original page allows users to better make use of knowledge they had already about the familiar desktop interface, potentially making the interface easier to learn and use.

While the algorithm for recognizing relevant contents produces promising results for many of the websites we tested, it is clearly still in need of more work and development. It is overly aggressive for certain web pages and also too conservative at times. More in-depth analysis should be done to evaluate the success and accuracy of the algorithm. The algorithm may also potentially benefit from machine learning techniques.

A more technical quantitative evaluation of the system should also be done to better assess the success and feasibility of Mobi. The evaluation as described in Section 5 is very formative. As the project matures through further development and iterations, more technical user studies should be performed to guarantee unbiased conclusions about the success of the system.

7.2 User control

Considering the many heuristics and automations involved in Mobi, there may be benefits in providing users with some manual control over how Mobi works. Early iterations of Mobi, for instance, includes a prototype of an interface that allows users to add and remove contents from the customized view on their mobile device. Information about these additions and removals can be saved at the proxy server and reapplied upon subsequent visits to the page.

This feature, however, makes the interface more complicated and is eventually removed for that reason. With a more suitable design, though, this may be a good way to increase the usability of the customized interface as it gives users direct control over what they see.

7.3 Browser capabilities

The capabilities of web browsers and specifications such as HTML and CSS [15] are changing rapidly. Some of the problems faced during implementation, as described in Chapter 4, can be mitigated as advancements in newer browsers are developed.

There are discussions of CSS 3, for example, to include features such as zooming [1] that would allow specific portions of DOM elements to be rescaled. Part of the reason that we abandoned the enlarge-in-place prototype, as described in section 3.2.1, was partly because of the lack of a method for doing this.

7.4 User behavior data

The user behavior data collected by the Action Recorder may potentially be used for many other purposes as well. Similar to the approach in Creo and Adeo [9], for example, such data may provide a basis for creating automation techniques that would make working on a mobile device, with its difficult input methods, more efficient.

Another potential idea is to create a corpus of shared data between multiple users of Mobi. For example, data collected from one user's browser may be used by other

users as well. This way, customizations may be created for a web page even if the user has never been to the page before. This approach will likely run into issues such as privacy and security, but is a great potential method for further increasing Mobi's robustness in automatically creating customizations for the mobile web.

Bibliography

[1] Css zoom. http://www.css3.com/css-zoom/. [Online Document].

[2] Greasespot. http://www.greasespot.net/. [Online Document].

[3] Platypus. http://platypus.mozdev.org/. [Online Document].

[4] Project joey - mozillawiki. https://wiki.mozilla.org/labs/joey. [Online Document].

[5] Squid: Optimizing web delivery. http://www.squid-cache.org/. [Online Document].

[6] Nilton Bila, Troy Ronda, Iqbal Mohomed, Khai N. Truong, and Eyal de Lara. Pagetailor: Reusable end-user customization for the mobile web. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys), 2003.*

[7] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *UIST, 2005.*

[8] Alexander Faaborg. A goal-oriented user interface for personalized semantic search. Master's thesis, Massachusetts Institute of Technology, 2006.

[9] Alexander Faaborg and Henry Lieberman. A goal-oriented web browser. In *Proceedings of the SIGCHI conference on Human Factors in computing systems, 2006.*

[10] Mozilla Foundation. Dom:element.getboundingclientrect. http://developer.mozilla.org/en/docs/dom:document.elementfrompoint. [Online Document].

[11] MIT CSAIL UID Group. Chickenfoot. http://groups.csail.mit.edu/uid/chickenfoot/. [Online Document].

[12] Google Inc. Google ajaxslt. http://goog-ajaxslt.sourceforge.net/. [Online Document].

[13] J. Nichols and T. Lau. Mobilization by demonstration: Using traces to re-author existing web sites. In *Proceedings of IUI'2008.*

[14] Jacob Nielsen and Kara Pernice. *Eyetracking Web Usability.* New Riders Press, 2009.

[15] W3C. Cascading style sheets: Current work. http://www.w3.org/style/css/current-work. [Online Document].

[16] W3C. Cascading style sheets level 2 revision 1 (css 2.1) specication. http://www.w3.org/tr/css21/. [Online Document].

[17] W3C. Document object model (dom). www.w3.org/dom/. [Online Document].

[18] W3C. Xml path language (xpath) version 1.0. http://www.w3.org/tr/xpath/. [Online Document].