

Browsing Data as a Predictor of Web Page Relevancy

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Elliot Korte

Annandale-on-Hudson, New York
May, 2014

Abstract

This project sought to find a reasonable predictor of web page relevancy. Software was created to collect and visualize a variety of data pertaining to a user's web browsing. A version of this software was used to collect data from 14 volunteers who browsed the web for three hours. The software used also surveyed volunteers about whether the pages they visited were relevant or not relevant. Comparing the data from pages which were deemed relevant by the volunteers to pages deemed not relevant we found several aspects of the data predicted page relevancy with a high degree of confidence. We concluded the best predictor was view time, or the total amount of time a user spent viewing a page.

Contents

Abstract	1
Dedication	7
Acknowledgments	8
1 Introduction	10
1.1 Proliferation of the World Wide Web	10
1.2 Structure of Browsing History	10
1.3 Impetus for this Project	12
1.4 Project Goals	14
1.5 Previous Work	15
1.6 Hypothesis	17
2 Experiment and Methods	20
2.1 Outline of Software	20
2.2 User Version	21
2.2.1 General Details	21
2.2.2 Visualization	22
2.2.3 Technical Details	26
2.3 Deployment Version	35
2.3.1 General Differences	35
2.3.2 Technical Details	36
2.4 Analysis Version	39
2.4.1 General Differences	39

<i>Contents</i>	3
2.4.2 Technical Details	41
2.5 Experimental Setup	42
3 Results	46
3.1 Summary of Results	46
3.2 Graphs of Selected Results	46
4 Analysis	60
4.1 Integrity of Experimental Data	60
4.2 Testing of Main Hypothesis	64
4.3 Testing of Other Variables	73
4.3.1 Visits	73
4.3.2 Average Visit Times	74
4.3.3 Idles	75
4.3.4 Idle Times	77
4.3.5 Loads	78
4.3.6 Load Times	78
4.3.7 Scrolls	81
4.4 Comparison of Possible Predictors	82
5 Conclusion	84
5.1 Summary	84
5.2 Limitations of Current Software	85
5.3 Further Research and Discussion	89
Bibliography	92
Appendices	94
A Additional Results	95
B Selected Source Code	118
B.1 collector.js	118
B.2 GraphBuilder.js	126

List of Figures

1.2.1	Snapshot of Chrome History	11
1.3.1	Snapshot of Chrome Extension	13
2.2.1	Snapshot of Chrome Extension graphical history without branching	23
2.2.2	Snapshot of Visualization where Gray Arrows Indicate Tab High- lights	24
2.2.3	Example of Side Panel in Visualization	25
2.2.4	Code in “collector.js” handling Google Chrome event listeners . . .	28
2.2.5	Code in “collector.js” describing a Node object	30
2.2.6	Portion of the action function in “collector.js”	31
2.2.7	Code in “collector.js” analysis version handling Google Chrome message received event	33
2.3.1	Snapshot of toolbar added to the deployment version	36
2.3.2	Code in “collector.js” deployment version handling Google Chrome message received event	37
2.3.3	Code in “collector.js” deployment version handling Google Chrome message received event	38
2.3.4	Full request.php program on the server which handled AJAX posts from the extension	39
2.4.1	Example of graph drawing too complex to read	40
2.5.1	Instructions given to research volunteers	44
3.2.1	Relevant Visits	48
3.2.2	Not Relevant Visits	48
3.2.3	Relevant Visit Times	49

3.2.4	Not Relevant Visit Times	50
3.2.5	Relevant Average Visit Time	51
3.2.6	Not Relevant Average Visit Time	51
3.2.7	Relevant Idles	52
3.2.8	Not Relevant Idles	53
3.2.9	Relevant Idle Times	54
3.2.10	Not Relevant Idle Times	54
3.2.11	Relevant Loads	55
3.2.12	Not Relevant Loads	56
3.2.13	Relevant Load Times	57
3.2.14	Not Relevant Load Times	57
3.2.15	Relevant Scroll Percentages	58
3.2.16	Not Relevant Scroll Percentages	59
4.1.1	Type Error on Read of Tab URL	60
4.1.2	Type Error on Write of Idle End Time	61
4.2.1	Bootstrapping Mean Differences for Relevant Visit Times and Not Relevant Visit Times	66
4.2.2	Code in “Analysis Version” used to perform the bootstrap com- parisons of two samples	67
4.2.3	Permutation Test Mean Differences for Relevant Visit Times and Not Relevant Visit Times	69
4.2.4	Code in “Analysis Version” used to perform the permutation test .	70
4.2.5	Bootstrapping Median Differences for Relevant Visit Times and Not Relevant Visit Times	72
4.3.1	Average Visit Time of Pages By Number of Idles	76
4.4.1	Results of Statistical Testing	82
A.0.1	Total Visits	95
A.0.2	Relevant Visits	96
A.0.3	Not Relevant Visits	96
A.0.4	No Response Visits	97
A.0.5	Total Visit Time	97
A.0.6	Relevant Visit Time	98
A.0.7	Not Relevant Visit Time	98
A.0.8	No Response Visit Time	99
A.0.9	Total Average Visit Time	99
A.0.10	Relevant Average Visit Time	100
A.0.11	Not Relevant Average Visit Time	100
A.0.12	No Response Average Visit Time	101
A.0.13	Total Idles	101
A.0.14	Relevant Idles	102
A.0.15	Not Relevant Idles	102

A.0.16	No Response Idles	103
A.0.17	Total Idle Times	103
A.0.18	Relevant Idle Times	104
A.0.19	Not Relevant Idle Times	104
A.0.20	No Response Idle Times	105
A.0.21	Total Average Idle Times	105
A.0.22	Relevant Average Idle Times	106
A.0.23	Not Relevant Average Idle Times	106
A.0.24	No Response Average Idle Times	107
A.0.25	Total Loads	107
A.0.26	Relevant Loads	108
A.0.27	Not Relevant Loads	108
A.0.28	No Response Loads	109
A.0.29	Total Load Times	109
A.0.30	Relevant Load Times	110
A.0.31	Not Relevant Load Times	110
A.0.32	No Response Load Times	111
A.0.33	Total Average Load Times	111
A.0.34	Relevant Average Load Times	112
A.0.35	Not Relevant Average Load Times	112
A.0.36	No Response Average Load Times	113
A.0.37	Total Refreshes	113
A.0.38	Relevant Refreshes	114
A.0.39	Not Relevant Refreshes	114
A.0.40	No Response Refreshes	115
A.0.41	Total Scrolls	115
A.0.42	Relevant Scrolls	116
A.0.43	Not Relevant Scrolls	116
A.0.44	No Response Scrolls	117

Dedication

This project is dedicated to my loving family. Their undying support has made this possible.

Acknowledgments

Firstly I would like to thank my project adviser Bob McGrail for the guidance, help, and insight he provided throughout this project. There were times where we disagreed about how the project should proceed, whether it be how the software should be developed, how much time should be spent on the visualization, or how the data collection should be carried out, and not once was I right and he wrong. Thanks to him I was able to accomplish meaningful research that goes beyond the development of interesting software. An additional thank you to Bob for finding all the time these last few weeks to help me edit, revise, and reedit this paper. His help has been invaluable to this paper, and throughout this project.

I would also like to thank my academic adviser Rebecca Thomas, her guidance throughout my time at Bard has been critical to my success, and the classes I have taken with her have taught me so much and left a lasting impact, inspiring a curiosity for the subject that continues to grow. She is a phenomenal professor who I am fortunate to have had the privilege to learn from. I would also like to thank Becky for her help with this paper, her careful attention to detail revealed many things I might have otherwise missed.

In addition I would like to thank Greg Landweber, his classes have instilled in me a love for math I did not always have, and have resulted in projects I remain proud of. I will remember his stories about reverse engineering and modifying old Mac OS windowing systems, the researchers at Adobe repeatedly mispronouncing “Affine Transformation”, among others. I always appreciated his willingness to offer his help and answer my questions regardless of whether or not they pertained to my class work. I am grateful to have had him as a professor.

A special thank you to Sam Hsiao who helped immensely with the statistics in this project. If it were not for him I might still be looking up different statistical

tests on Wikipedia debating which ones should be used (and being wrong about it). He helped me choose the right tests, find good references, and get the language right about my interpretation of those tests. He also offered his wisdom about statistics in general, and I learned more from him in a few meetings than I did during an entire year of AP Statistics in high school.

I would like to thank Keith O'Hara whose object oriented programming class gave me my first real programming experience. It was in this class that I fell in love with programming, and I have not looked back since. I would like to thank Trevor for creating the icon used in my extension, and for his help with some of the graphics used in this project. I would also like to thank my roommates Trevor and John for being understanding recently when I hadn't cleaned my dishes. I'd like to thank my friends Jacob and Oliver for sharing a love of computer science, and for excitedly nerding out with me and extending our learning beyond the classroom. Lastly I would like to thank my Mom, Dad and Brother, whom I love very much.

1

Introduction

1.1 Proliferation of the World Wide Web

The adoption of the World Wide Web and its proliferation of content has been tremendously rapid. The amount of information available is staggering, and in order to both gain access to this information and communicate it themselves the public has largely connected to the internet. An estimated 77% of the developed world currently uses the internet [21]. While it is apparent to those who use the internet regularly that the information available to users is immense, what they may be less aware of is the extent of the information they themselves generate while browsing the web.

1.2 Structure of Browsing History

Many internet users are familiar with the browsing history most modern web browsers provide for them. Usually it contains a list of web pages they have visited, often ordered chronologically. While this is useful, it is a scant representation all of the data being generated by a user. For example, in Google Chrome[14], which

is currently the most popular web browser in the world with 47% of users [28], when a user views their history all they see is a list of web pages that were visited listed chronologically and the time and date they loaded each page. This contains no information about the pages that they navigated to before and after a given page, how they got to a particular page (either by typing in the URL manually, or clicking on a link), or any other info that encapsulates how a user browses the web.

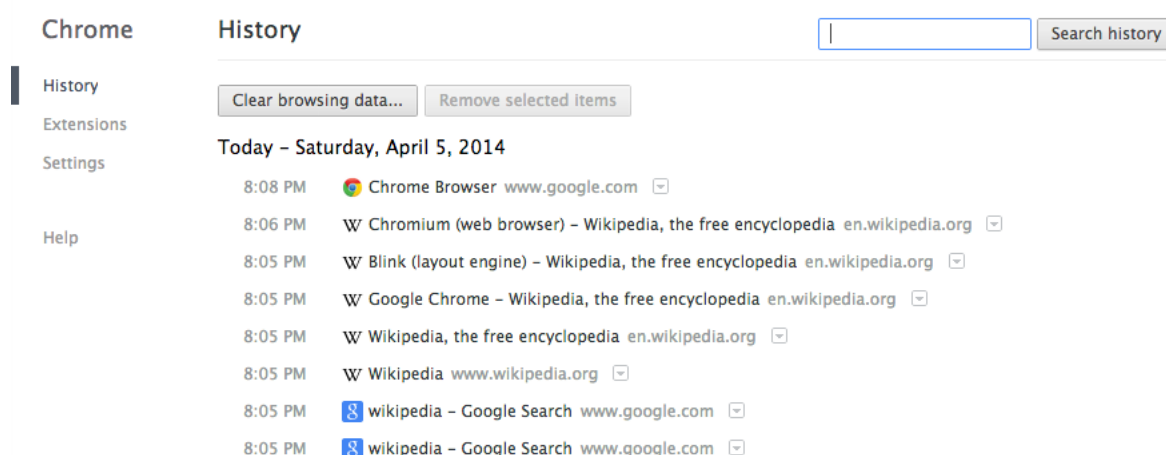


Figure 1.2.1. Snapshot of Chrome History

Only by storing all the information necessary to recreate a user's browsing session can a truly complete browsing history be stored for the user to view and analyze. There are a variety of complications in doing this, and good reasons major browsers have not implemented this functionality. Firstly, this information compromises the privacy of users. I would argue however, that the information already stored in a browser's history, all of the web pages a user visits, is the main basis for the invasion of privacy, and information regarding how a user browsed those pages does not significantly add to the harm a breach in this information would cause. The

information in question is metadata describing the pages visited by a user, and metadata is arguably less harmful than the data it describes. Secondly, there does not appear to be a great demand for this information from users. While this may be so, I would guess most users wouldn't be opposed to additional information in their browsing history, and anecdotally users I've explained this to have expressed interest in being able to have this information available to them. Lastly, additional memory and resources would have to be used in order to collect and store this information.

A natural structure for storing browsing data is a directed graph, where vertices represent web pages, and edges represent navigation between pages. This is not a new idea; since the early to mid 1990's people have been describing a variety of web related data with graphs, including the relationship between pages on the web and the links between them [17], and user web browsing data [1]. For this project browsing data was stored in a directed graph, with additional properties stored for vertices and edges describing other information about the websites visited and navigation between them.

1.3 Impetus for this Project

One of the original motivations for this project was what I felt was a lack of a comprehensive browsing history in major browsers. I wanted to create a browsing history that would allow users to view virtually all of the information they generated as they were browsing in a useful, easy to understand format. Furthermore, once this data was being stored, what benefits could this information provide the user besides the ability to see a more detailed version of their own history? I decided a browser plugin would be the best way to implement this, and after comparing browsers I chose Google Chrome because of its large user base and more modern plugin, or extension, API [14]. I was able to create a visualization of a user's browsing history,

collecting the necessary data, and while doing so contemplating how this information could benefit users in other ways. Below is a view of the same history in Figure 1.2.1 with the Chrome extension.

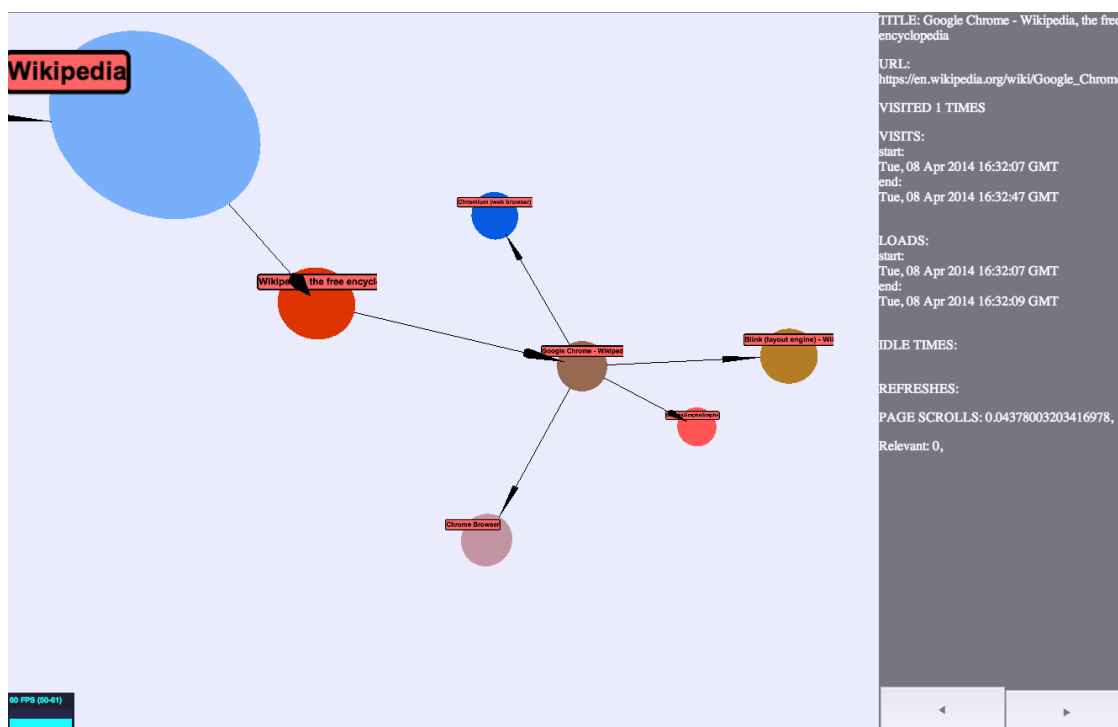


Figure 1.3.1. Snapshot of Chrome Extension

This visualization of a user's history offers several advantages over the built in Google Chrome history. It enables a user to see the order in which they navigated pages, whether pages were opened in a new tab (pages opened in a new tab fan out from and maintain the same value on the Z-axis as the page they were opened from), and a lot of additional information about each page in a panel displayed on the right. The visualization software is described in greater detail in Chapter 2.2.2.

Collecting all this information raised the question, if we were to analyze this information on a large real world dataset, what could this wealth of information possibly reveal? Perhaps trends could be found in the data that would be able to

predict user browsing patterns in some way, or maybe a trend in the data could reveal something about how people find information on the internet. While this may be the case, it would require an immense amount of data that could not feasibly be collected for this project. Instead, an interesting problem that we could potentially get data to provide evidence about was: given a set of links on a web page, how can we use this data to predict which link is most likely to be relevant to the user? Unfortunately this would still be problematic in terms of the large amount of data required, since only websites with a sufficient number of visits by different participants would be useful for the research. Ultimately we decided on a similar and no less significant problem to try and answer with this data. Given a set of user browsing data, how can we determine which pages a user likely found relevant?

1.4 Project Goals

The goal of this project is to use browsing data collected by the Google Chrome Extension to determine what web pages a user is likely to find relevant. We are not using data from an individual to predict specifically what they will find relevant, rather, we are taking data from a group of individuals to predict what the population at large will likely find relevant. Finding trends in this data has the potential to improve current solutions to real world problems. The main use of the internet, for information retrieval, can be facilitated by the knowledge of what is more likely to be relevant to users of a search program. When a user is on a web page, knowing the likelihood of a link's relevance to a user can allow a program to intelligently suggest certain pages, and hence further facilitate information retrieval. There are many potential applications that would enhance information retrieval given a large enough sample of user browsing data and the ability to use that data to predict with some degree of certainty which pages are likely to be relevant. The moral questions

posed by being able to collect all of this information from users without invading their privacy is beyond the scope of this paper. I will note however, that if a user volunteers to have their information collected, and their anonymity is well protected, as it is in this project, I see little risk.

1.5 Previous Work

Some research has been done regarding different techniques for the visualization of a user's browsing history. Some visualizations have made a tree-like structure describing a user's browsing history [1][13], however this approach does not allow for cycles in the graph describing the browsing, and will display repeated web pages in the tree when a page is browsed twice. This is not as accurate a description of a browsing history as the visualization developed for this research, since the reference made to a page that has already been browsed further up in the tree will not contain information about previous browsing done on the page. Also, the graphs drawn in these studies were two dimensional, and hence if the graph generated from a user's browsing session were not planar clearly drawing the graph's edges would pose difficulties not posed by a three-dimensional rendering [24]. Lastly, the mentioned research was conducted in the mid to late 1990's, and vast improvements in computer graphics have been made in that time [10]. Surprisingly, not much research on this topic appears to have been done in that time, and I have not been able to find a tool which visualizes a user's browsing graph for any modern browser. As discussed in Section 1.3, this was a major reason this research was conducted.

In contrast, determining the relevancy of web pages has been a well researched topic since the early to mid 1990's. Making this determination has many applications, notably in the use of Internet search engines which aim to provide the most relevant pages in relation to some input data. There is a lot of information contained

in a website, including its title, URL, hyperlinks, the tree-like structure of its tags, and the content contained in those tags. Much research has been devoted to analyzing this information to determine web page importance or relevancy [25][20][7]. Furthermore a considerable amount of research has been done to use the relationships between pages on the World Wide Web and their links to predict web page relevancy. This type of analysis serves as the foundation for the popular commercial web search engine Google [3].

There has also been research done to determine page relevancy in order to optimize Web Crawlers, or programs used to automate the retrieval of information on the internet [22]. This research, like the research done by Craswell, Hawking, and Robertson [7], looks at information about a link on a web page to determine the likelihood the page the link refers to is relevant without ever retrieving the page itself. This is done with the aim of reducing unnecessary web page retrievals in smaller scale topic-specific Web Crawlers.

The research discussed so far describe some of the more commonly accepted methods currently used to determine page relevance. All of these methods only use data that can be gleaned from the publicly accessible World Wide Web, and do not consider data gathered from user web browsing.

There has been a considerable amount of research done regarding user browsing patterns that collect information from user browsing sessions, however the aim is usually to determine how users generally find information, and not predict page relevancy. Some of this research has been devoted to finding out how often pages are navigated to using different methods including clicking a browser's forward or back button, clicking hyperlinks, submitting a form, and typing in a URL [4][16]. One study collected a long term user history for the purpose of analyzing different behaviors for web page revisitation [16]. While these studies involved the collection

of a variety of information regarding a user's browsing history, they do not attempt to use this data to make any determinations about the pages being browsed.

Another study used information collected from a user's web browsing history to try and make a personalized web search catered to that user [9]. This study implicitly makes determinations about web page relevance in deciding how to re-order search queries based on a user's browsing data, and verified their findings about which pages are relevant with test subjects. However, like the other studies mentioned, it did not take into consideration much of the data the research done for this paper primarily focuses on, notably the amount of time a user spends on a web page.

One possible reason this kind of research has not been conducted is concern for a user's privacy, as web browsing often contains sensitive information. As discussed briefly in Section 1.2, the information in question is metadata and once browsing information containing all of the web pages a user visits is collected, additional information about how navigation between those pages took place does not seem likely to significantly add to the data's sensitivity, and I see no reason the metadata collected in the studies mentioned poses a greater risk to user privacy than the metadata collected in this research. Another possible reason is the way in which researchers were collecting data did not allow for this information to be easily collected. While this may have been the case in some research, it is doubtful it has always been the case, and more likely researchers were not interested in, or did not consider this kind of information.

1.6 Hypothesis

There are several data points that can potentially be analyzed to see whether they possibly determine with a high degree of certainty the likelihood of web page relevance. The following is an explanation of the data points being collected.

Visits - the number of separate occasions in which a user viewed a web page. Each time a page gets displayed on the screen it counts as a visit.

Visit Time - The total time spent in milliseconds that a web page gets displayed. The amount of time spent during each visit is added together to obtain this value.

Average Visit Time - The average time in milliseconds a web page was displayed for. The Total Visit Time is divided by the number of Visits to obtain this value.

Idles - The number of times a user performed no action on a page for at least 30 seconds. This means there were no mouse movements, clicks, scrolls, or key presses.

Idle Time - The total amount of time a user was idle on a web page. This is calculated by adding together the amount of time (beginning after 30 seconds with no activity and ending when a user is no longer idle) of each idle that occurred on a web page.

Average Idle Time - The average amount of time a user spent idle on a web page, calculated by dividing the Total Idle Time by the number of Idles.

Loads - The number of times a page is loaded. Whenever a page is retrieved from its server this value is incremented.

Load Time - The total time in ms it a page spent loading in milliseconds, calculated by adding together the time it took for each Load to complete.

Average Load Time - the average amount of time a page took to load, calculated by dividing the Total Load Time by the number of Loads.

Refreshes - the number of times a user refreshed a page.

Scroll Percentage - the farthest a user scrolled down on a page, as a percentage of the page height. This is set to 0 if no scrolls took place, and 1 if the page was scrolled all the way to the bottom. For pages with a height shorter than the height of the Browser the Scroll Percentage is set to 0.

Each variable was hypothesized to predict page relevance, however the one that will receive the most analysis in the results section is that the Total Visit Time, not the number of visits or loads, will be the best predictor of web page relevance. In other words, the pages which are displayed for the longest amount of time will be the ones that are most likely to find relevant. The reason this was chosen to be the main hypothesis is because it seemed like the best candidate to predict relevance. There were other likely candidates including Visits, Loads, and Scroll Percentage, and these variables were also tested. In my judgment however, Total Visit Time seemed the most likely to predict web page relevance.

2

Experiment and Methods

2.1 Outline of Software

Three different versions of the Google Chrome Extension were created, a “User Version”, a “Deployment Version”, and an “Analysis Version”. The majority of development was spent on the “User Version” from which the other two versions were forked. The entire Google Chrome Extension API uses JavaScript and JSON [14][23], and hence this software was developed almost entirely using JavaScript. In addition to JavaScript, the visual components of the software use HTML [15]. The Chrome Extension API relies largely on callback functions, where instead of returning a requested value to a function call, a new function gets passed as an argument which is called and passed the requested information [14]. In example code, functions that are passed to other functions as arguments are used in this manner.

It should be noted that the information stored by this Software is not a complete history, as it does not keep all the necessary information required to completely recreate a browsing session. It does not store the amount of time a user spent

at each scroll-height, whether a navigation took place because of link click or an address being typed directly into the address bar (or some other way), or information about key presses and mouse movements. Of this information, the only bit that was considered for use in this project was the cause of a navigation, as the other information was not relevant to this research, difficult to retrieve and store, and especially intrusive to a user's privacy. Information regarding whether a navigation took place because of a link click or otherwise is available in a separate part of the Chrome Extension API that did not update as quickly as was necessary for this project [14]. Regardless, there are probably ways to be able to retrieve a lot of this information. For further information about current limitations of the software see Section 5.2.9

2.2 User Version

2.2.1 General Details

The user version is geared primarily to those who wish to be able to view a detailed and graphical version of their history during a browsing session. As a user is browsing, detailed information about their browsing is collected, including all the information used to calculate the values of the variables discussed in Section 1.6. When a user pushes a button next to the address bar a new tab is opened displaying the visualization of their history. See Figure 1.3.1 for a snapshot of the graphical history.

While this version of the extension was not used for the data collection or the analysis of the data in the experiment, since the majority of development was devoted to it, and since the data collection undergone in this version is nearly identical to the version used to collect data for the experiment, it is discussed in the most detail.

2.2.2 Visualization

When a user presses the button which opens the visualization, a directed graph describing their current browsing session gets drawn. The aim of the graph drawing algorithm was to provide an illustration of the graph that would show a path-like view of a browsing session in order to give the user a sense of the path they took while browsing. The algorithm is semi-deterministic in that the exact location in which nodes are drawn is not predetermined, however the general location in which they are drawn is. Other common graph drawing algorithms were considered, including a force-based layout where node placement is determined by a simulation in which nodes are given physical properties such as electrical charges [24], however implementing this would have been difficult, it probably would have been slower to calculate the layout, and the chaos in the system would sometimes cause nodes that were browsed in succession to be drawn far apart, which I wanted to avoid.

When links were clicked on and opened in the same tab, or an address was typed directly into the address bar, if it is a new page (recognized by the new URL) then it increases the node's value on the Z-axis. Random values within a certain range are given for the X-axis and Y-Axis, creating a zig-zag in the center of the graph from which pages opened in a new tab branch out from. The random zig-zagging is intended to allow edges to be drawn, and more easily deciphered, between any two nodes in the graph. See Figure 2.1.5 for an example of a graphical history without branching.

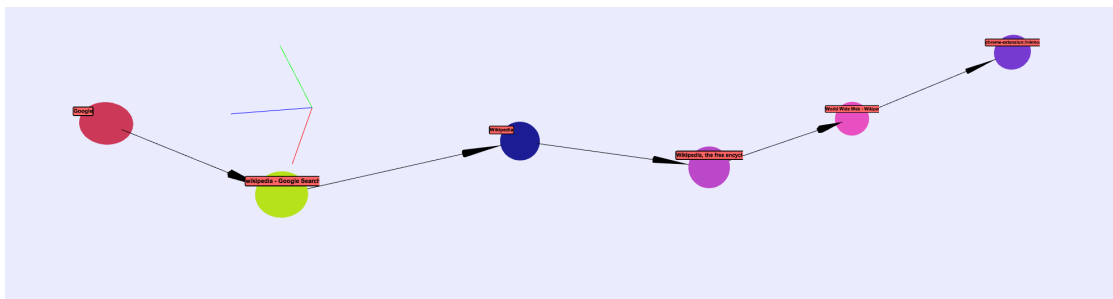


Figure 2.2.1. Snapshot of Chrome Extension graphical history without branching

When branching occurs, it is because pages are opened in a new tab. When this occurs, the pages that are opened receive the same value on the Z-axis as the page from which they were opened, and are laid out equally spaced on a circle surrounding the page they were opened from. See Figure 1.3.1 for an example of graphical history with branching.

Branching, or lack thereof, indicates whether a page was opened in a new tab, however this does not differentiate the other type of navigation, a tab highlight. The way this is displayed is with a gray arrow instead of a black one. In order for a highlight to take place, the node must have already been loaded, so no node can only contain an edge with a gray arrow. Gray arrows are drawn at a slightly different size, this is so that even when there are both a black and gray arrow between two nodes, both arrows can be seen by the user. In Figure 2.2.2 the branching of the 5 nodes surrounding “Wikipedia” indicate that they were opened in new tabs, and the gray arrows show the order in which they were highlighted.

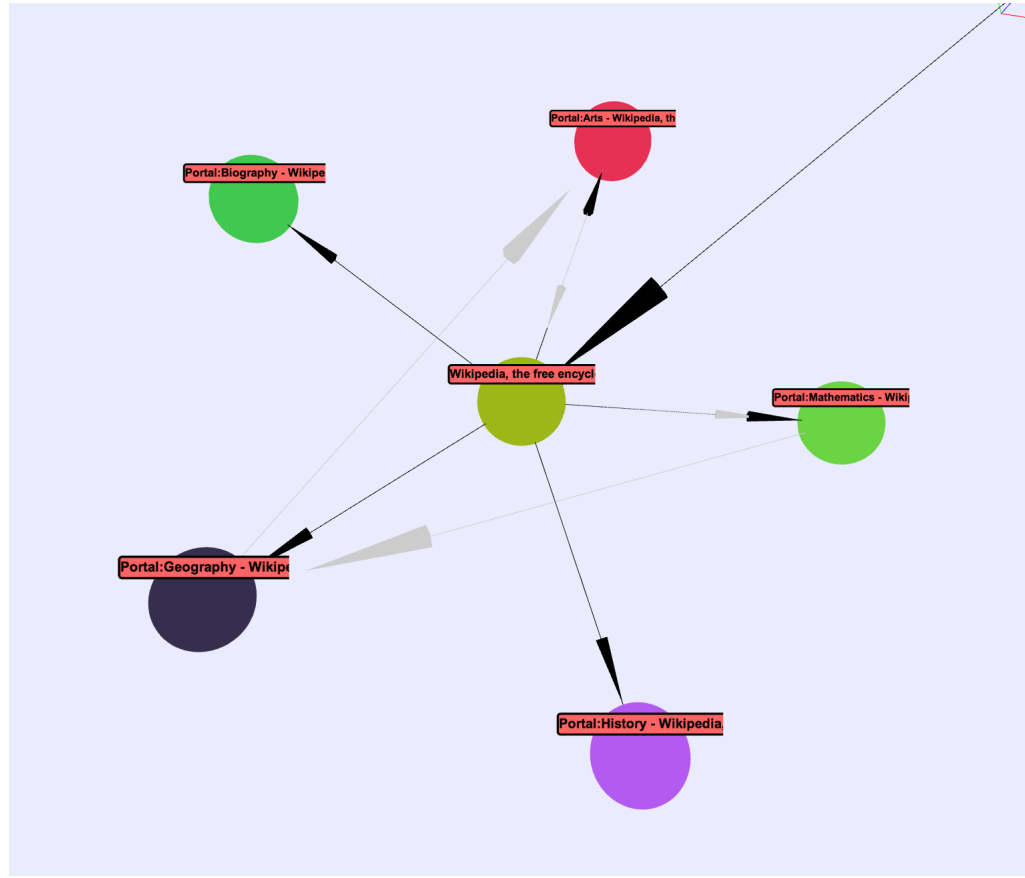


Figure 2.2.2. Snapshot of Visualization where Gray Arrows Indicate Tab Highlights

The view is centered around a particular node (Figure 2.2.2 shows a view centered around “Wikipedia”), and information about this node is displayed in a panel on the right hand side of the visualization. The node can be selected in two ways, either using the forward and back buttons to traverse through the edge list of the graph until the desired node is reached, or by clicking on the desired node. If a click does not occur on a node, and the user holds down the mouse they are able to rotate the graph around the currently selected node by moving the mouse. Using the scroll wheel on the mouse allows the user to zoom in and out, and by holding down the right click button a user is able to move the center of the graph and rotate around a different point in space when holding left click.

The side panel of the visualization displays all of the information available about a selected node, including the title and URL of the page, the number of times a page is visited, the time each visit began and ended, the number of times the page was loaded and the time each load began and ended, the number of times a user idled on the page and the time each idle began and ended, the time each refresh took place, and the maximum scroll percentage for each time the page was loaded. Figure 2.2.3 shows the side panel information for one of the pages visited during the browsing session discussed in Section 2.2.5.



Figure 2.2.3. Example of Side Panel in Visualization

It is important to note the difference between a visit and a load. A visit occurs whenever a user views a page. If a different tab is highlighted and later the user returns to the original tab, this will count as a separate visit, but not a separate

load. Loads only occur when the website is downloaded from its server over the internet. The history built into Google Chrome only keeps track of page loads, not visits as they are defined above. There are a number of issues with the graph drawing algorithm as it is currently implemented, these are discussed in Section 5.2. For the full source code of “GraphBuilder.js” see Appendix B.2

2.2.3 *Technical Details*

The actual data collection happened in a Google Chrome background script titled “collector.js”. Background scripts are run when Google Chrome is first opened, and while their code may terminate they remain in memory until Google Chrome quits, and usually continue to execute commands when some event is fired by Google Chrome. In “collector.js” almost nothing occurs when Google Chrome is first opened, however a few events trigger the function “action” which is where the data is stored and the data structures containing the data are updated. This happens when a page is loaded, when a tab or window is highlighted (clicked on by the user), and when a page finishes loading.

The functions called by these listeners account for the majority of data being collected, however there are two other events that can trigger a change in the data being stored. One is a Chrome fired event, which detects whether a user has been idle for some amount of time, which I set to 30 seconds. This event is fired both when a user is idle for 30 seconds and, if a user was idle for more than 30 seconds when they are no longer idle. In deciding the amount of time, I picked a time which a user can reasonably view and read most of a page before scrolling, but is still a fairly short span of time. This is because if a user leaves their machine, when considering “Visit Time” - “Idle Time”, I want less of this time to be considered Visit Time, even if that means an idle event is sometimes triggered while a user is

reading a long article or viewing a page for an extended amount of time of time. Later on, it can be assumed with a reasonable degree of certainty that short idle times (perhaps less than 1 minute) can be attributed to a user taking an extended amount of time to view a page, while longer idle times (perhaps 3 or more minutes) can be attributed to a user leaving their machine. See Figure 2.1.1 for the code in “collector.js” handling events fired by the Google Chrome Extension API.

The other event listener is fired when a user scrolls on a page. This is not a Chrome event, but one that is triggered by a script injected by the extension into web pages the user browses. This script sends a message to “collector.js” with the URL of the page and the scroll percentage. This event is listened for in the Message Received event handler which is fired when a separate file in the extension sends a message using the Chrome Extension API. See Figure 2.1.3 for the code in “collector.js” handling the message receive events fired by the Google Chrome Extension API.

```
1 chrome.tabs.onActivated.addListener(function (activeInfo)
2   {
3     //Listener for when a tab is highlighted
4     chrome.tabs.get(activeInfo.tabId, function(tab){
5       action(tab, "onHighlight");
6     });
7   });
8 chrome.tabs.onUpdated.addListener(function(tabId,
9   changeInfo, tab) {
10    //Listener for when a page is begins loading, or
11    //finishes loading
12    if (changeInfo.status == "loading") {
13      if (tab.highlighted) {
14        action(tab, "onCurrentUpdated");
15      }
16      else {
17        action(tab, "onOtherUpdated");
18      }
19    }
20    else if (changeInfo.status == "complete") {
21      action(tab, "completion");
22    }
23  });
24 chrome.idle.setDetectionInterval(30);
25 chrome.idle.onStateChanged.addListener(function(newState)
26   {
27    //listener for when the idle state of a page changes
28    if ((newState=="idle") || (newState=="locked")) {
29      var tempTimestamp = new Timestamp();
30      tempTimestamp.start = (new Date()).getTime();
31      prevNode.idleTimestamps.push(tempTimestamp);
32    }
33    else if (newState=="active") {
34      prevNode.idleTimestamps[prevNode.idleTimestamps.
35        length - 1].end = (new Date()).getTime();
36    }
37  });
```

Figure 2.2.4. Code in “collector.js” handling Google Chrome event listeners

Figure 2.1.1 shows how Chrome events are being handled, and most of the events trigger a call to the “action” function, which makes up the bulk of “collector.js”. The “action” function updates the data structures that store the browsing data with the necessary information. It takes two arguments; one is a “tab” object which is part of the Chrome Extension API, and the other is a string describing the type of event that happened. The “tab” object contains information relevant to the web page, including the URL and the Title. The string indicates whether the event was caused by the highlighting of a tab, a page load in the currently highlighted tab, a page load in a different tab, or a page load completion. Tab updates take place when the state of a tab changes, either from “loading” to “complete”, or vice versa, and the `tabs.onUpdated` listener accounts for all calls made to the “action” function that are not because of a tab highlight. If a website load takes place on the currently highlighted tab it usually indicates that a user clicked a link or typed in a search or website URL directly into the address bar, and the call made to the action function gets passed the string “onCurrentupdated”. If a website load took place in a different tab it is usually because the user opened a link in a new tab, and the action function gets passed the string “onOtherupdated”. If a new tab is highlighted, the `tabs.onActivated` listener is called and the action function gets passed the string “onHighlight”. Lastly if the completion of a website load takes place then the action function gets passed the string “completion”.

Once that action function is called, if the URL of a tab is new, then a new “node” object is created, otherwise updates are made to the “node” object with a URL matching the URL of the tab. It is assumed that this only occurs when the string passed is either “onCurrentUpdated” or “onOtherUpdated” since for a page to be highlighted or finish loading a previous call to “action” with its URL must have been made. Before the call to the “action” function terminates, the current “node” object

is stored so that the next call to “action” has a reference to the previous “node” and can update the data structures accordingly. The action function contains a fair amount of nearly redundant code, as very similar things happen for different types of events and different circumstances regarding those events. See Figure 2.1.2 for the code describing a “node” object. See Figure 2.1.4 for a portion of the “action” function.

```
1 function Node() {  
2   //properties related to the web page  
3   this.url = "";  
4   this.title = "";  
5   this.visits = 0;  
6   this.visitTimestamps = new Array();  
7   this.loadTimestamps = new Array();  
8   this.idleTimestamps = new Array();  
9   this.refreshes = new Array();  
10  this.scrollPercentages = new Array();  
11  
12  //properties used for drawing the graph  
13  this.group = 0;  
14  this.groupID = 0;  
15  this.nextGroup = 0;  
16  this.nextGroupID = 0;  
17  this.groupSize = 0;  
18  
19 }
```

Figure 2.2.5. Code in “collector.js” describing a Node object

```
1      var curNode = findNode(tab.url); //uses URL to
      determine if node already exists
2      if (curNode==null) { //this is a new page
3          //update values of node that are the same
          regardless of the event type
4          curNode = new Node();
5          curNode.url = tab.url;
6          curNode.title = tab.title;
7          var curLoadTimestamp = new Timestamp();
8          curLoadTimestamp.start = curTime;
9          curNode.loadTimestamps.push(curLoadTimestamp);
10
11         if (type=="onCurrentUpdated") {
12             //event specific properties
13             var curVisitTimestamp = new Timestamp();
14             curVisitTimestamp.start = curTime;
15             curNode.visitTimestamps.push(curVisitTimestamp)
16             ;
17             curNode.visits++;
18             curNode.scrollPercentages.push(0);
19             curNode.relevantResponses.push(0);
20             curNode.completedLoads.push(false);
21
22             curNode.group = prevNode.nextGroup;
23             groupSizes.push(1);
24             curNode.groupID = 0;
25             curNode.nextGroup = prevNode.nextGroup + 1;
26             curNode.nextGroupID = 1;
27
28             //updates previous nodes visit timestamp
29             prevNode.visitTimestamps[prevNode.
30                 visitTimestamps.length - 1].end = curTime;
31
32             //adds data to our data structures
33             edgeList.push({from: prevNode, to: curNode,
34                 type: "onCurrentUpdated", color: "black"});
35             nodeList.push(curNode);
36
37             //updates reference to previous node
38             prevNode = curNode;
39         }
40     }
```

Figure 2.2.6. Portion of the action function in “collector.js”

The data structures being used to store this information are a JavaScript array containing all of the nodes, and another array containing JavaScript objects describing edges in the graph. Originally node objects had references to the nodes they were connected to, describing the edges in the graph. However, in order to use the function “JSON.Stringify”, which is native to Google Chrome’s implementation of JavaScript, and is used to convert JavaScript objects and arrays to and from strings, the objects being stored cannot contain references to themselves, directly or indirectly. An implicit call to JSON.Stringify is made during this message passing, since the only information communicated between Google Chrome Extension messages is strings, and hence in order for this information to be passed between different parts of the program JSON.Stringify must be used. Also in order for the information to be stored permanently it must be represented as a string. Any graph which contained a cycle would violate the non-self referential property essential for the use of the JSON.Stringify function. Hence a different way to store the data was adopted, and nodes no longer contained references to other nodes, and all of the information about the graph was stored in a list of node pairs, where each element of the list contains a “from” node, a “to” node and describes an edge. In addition other information about the edge traversal pertaining to the type of navigation which occurred, and the drawing of the graph in the 3D visualization are stored in a node pair.

The Google Chrome events update the data continuously as a user browses, and when a user presses the button triggering the creation and display of the 3D visualization, a message is sent from the tab containing the visualization to “collector.js” requesting the data, and the response which is included in a callback function by the message sending function, contains the data. See Figure 2.1.4 for the code in “collec-

tor.js” handling the message receive events fired by the Google Chrome Extension API.

```

1 chrome.runtime.onMessage.addListener(
2   function(request, sender, sendResponse) {
3     if (request.greeting == "graphData") {
4       for (var i =0; i<nodeList.length; i++) {
5         nodeList[i].groupSize = groupSizes[nodeList[i].
6           group];
7       }
8       sendResponse({data: edgeList, size: nodeList.
9         length});
10    }
11    else if (request.greeting == "pageScroll") {
12      var curNode = findNode(request.url);
13      if (curNode.scrollPercentages[curNode.
14        scrollPercentages.length - 1] < request.
15        percent) {
16        curNode.scrollPercentages[curNode.
17          scrollPercentages.length - 1] = request.
18          percent;
19      }
20    }
21  };
22 });

```

Figure 2.2.7. Code in “collector.js” analysis version handling Google Chrome message received event

The visualization takes the list of edges and builds and displays the 3D graph. The drawing of the graph is done using a 3D graphics library for JavaScript called “Three.js”. “Three.js” was chosen because of its relative ease of use, not requiring extensive knowledge about WebGL, the lower level graphics API from which “Three.js” is built, as well as its high functionality and rich built in shape drawing capabilities. “Meshes” or geometric information describing a shape, were built into “Three.js” for spheres as well as arrows making it relatively easy to begin working

with these shapes, which are the only objects used in the 3D visualization, aside from the labels for the nodes [8].

All of the page layout information for the visualization was done using HTML. The program that built the 3D visualization of the graph, “GraphBuilder.js” was called by the HTML page, “graphic.html”. Upon initialization, “GraphBuilder.js” sends a request for the graph data using the Chrome Message API, and “collector.js” sends the data (see Figure 2.1.4 for code handling message received events in “collector.js”). “GraphBuilder.js” then reads in the edgeList, and adds each edge to the graph using the aforementioned graph drawing algorithm. The data on the side panel is then updated by replacing the inner text of the HTML tags displaying the corresponding data. The well known JavaScript library jQuery [5] was used to make updating the HTML tags simpler. The mouse functionality allowing the manipulation of the view of the graph is defined in an auxiliary package to “Three.js” titled “OrbitControls.js” [17], and the mouse functionality that allows the selection of nodes by clicking was done by adding a mouse listener to “GraphBuilder.js” and using functions defined in “Three.js” to detect if a three dimensional projection of the mouse click location intersects with any of the nodes.

Some of the information needed to draw the graph in the manner described in Section 2.2.2 is contained in the information about edges. This includes from which nodes arrows get drawn to and from, and the color of each arrow. The other information needed for the graph drawing is contained in each node, and includes the node title (displayed on the label drawn next to each vertex of the graph), and a few variables describing where each node should be drawn. These variables are a group, group ID, and group size.

The group is set to zero for the first node, and incremented when a page is loaded with the “onCurrentUpdated” value passed as the type of navigation to the “action”

function. In “GraphBuilder.js” a node’s group is multiplied by some constant to get its value on the z-axis so that all nodes in the same group are together on the z-axis. For every group, the first node loaded with this particular group is given a group ID of zero. When a page is loaded with the “onCurrentUpdated” value passed as the type of navigation, indicating it was opened in a new tab, the new node gets the same group value as the node it was opened from, and its group ID is incremented. This gives the node the same value on the Z-axis as the node it was opened from. The difference however is that nodes with a group ID greater than one get drawn in a circle around the node in the same group with a group ID of zero. This creates the branching of pages opened in a new tab described in Section 2.2.2. Lastly, before the data is sent to “GraphBuilder.js”, the size of each group is calculated and stored in each node’s groupSize so that the graph drawing algorithm can calculate how far apart to space the nodes on the circle. The next group, and next group ID of a node are used to calculate the group and group ID of the next node that gets processed.

2.3 Deployment Version

The deployment version is a fork of the user version that was designed specifically for data collection in the experiment portion of this research. See Section 2.3 for an outline of the experiment and how this software was used.

2.3.1 General Differences

There are a few differences between the user version and the deployment version, the first is that there is no visual component to the user’s history. While the users were fully aware of the information that was being collected from them, I did not want them to be viewing this information as they were browsing, as I did not want it to influence their browsing behavior. Hence, the button which opens the visual-

ization was removed, and the files associated with displaying this were also removed. Another difference was the addition of a toolbar asking users whether a page was relevant or not. The responses were recorded for later analysis in the analysis version. Lastly, the data being collected by the volunteers of the experiment was stored on a remote server and every four minutes this data gets posted to the server.

2.3.2 Technical Details

The toolbar asking user's whether a page is relevant was created using a script that was injected into each page the user visits, the toolbar contained the question "Is this page relevant to you?" and two buttons, one was labeled "Definitely Yes", the other labeled "Definitely No". See Figure 2.1.6 for a visual of the toolbar.

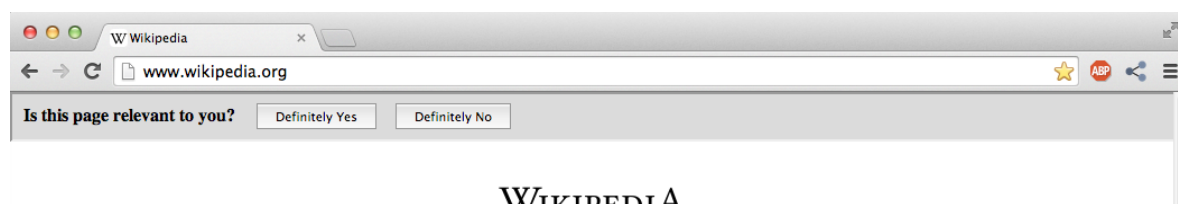


Figure 2.3.1. Snapshot of toolbar added to the deployment version

On the back-end in "collector.js" a few changes were made. The node object in the deployment version included an array, "relevantResponses", which contained a value for each time there was a page load. The value was initialized to 0, if the user clicked "Definitely Yes" it was set to 1, and if the user clicked "Definitely No" it was set to -1. The injection script containing the toolbar used the Chrome Extension message sending API to send a message to "collector.js" when either the "Definitely Yes" or "Definitely No" button was pressed. The message received listener in "collector.js" was modified to update a node's "relevantResponse" value with the value given. Also the graph data request was removed since it was no longer needed. See Figure 2.1.7 for the message listener used in the deployment version.

```
1 chrome.runtime.onMessage.addListener(  
2   function(request, sender, sendResponse) {  
3     if (request.greeting == "yesRelevant") {  
4       sendResponse({"farewell": "you are relevant!"});  
5       prevNode.relevantResponses[prevNode.  
         relevantResponses.length-1] = 1;  
6     }  
7     else if (request.greeting == "noRelevant") {  
8       sendResponse({"farewell": "you are not relevant!"});  
9       prevNode.relevantResponses[prevNode.  
         relevantResponses.length-1] = -1;  
10    }  
11    else if (request.greeting == "pageScroll") {  
12      var curNode = findNode(request.url);  
13      if (curNode.scrollPercentages[curNode.  
        scrollPercentages.length - 1] < request.  
        percent) {  
14        curNode.scrollPercentages[curNode.  
          scrollPercentages.length - 1] = request.  
          percent;  
15      }  
16    };  
17  });
```

Figure 2.3.2. Code in “collector.js” deployment version handling Google Chrome message received event

The data collected by the users being posted to the server was sent to the server using AJAX [12]. The JavaScript library jQuery was used to simplify the posting of data to the server. The posting of data occurred when the extension is first loaded, every 4 minutes after that, and whenever a window is closed. See Figure 2.1.8 for the code which posted the data on the server.

```
1 function postData() {  
2   var myData = JSON.stringify(edgeList);  
3   $.ajax({  
4     type: "POST",  
5     url: "http://www.elliotkorte.com/request.php",  
6     data: {'myData':myData, 'computerID': 99990001 *  
           20, 'newSession':newSession},  
7     success: function() {  
8       console.log("SUCCESSSSSS");  
9     }  
10  });  
11  
12  newSession = 0;  
13  setTimeout(postData, 120000)  
14 }
```

Figure 2.3.3. Code in “collector.js” deployment version handling Google Chrome message received event

On the server there was a file, “request.php” [26], which kept track of which computer in the experiment was sending the data, and then saved the data on the server in a file specific to that computer, which later gets read by the “analysis version”. A simple technique was used to ensure data posted on the server originated from this project: the computer number is first multiplied by a large prime number, and on the server only if the computer number is evenly divisible by this same number is the data posted. A variable “newSession” is also passed from the request, this is set to 1 if it is the first time the extension is making a post, and 0 afterwards. This is to ensure data from prior sessions isn’t overwritten. See Figure 2.1.9 for full code of “request.php”. See Appendix B.1 for the full source code of the version of “collector.js” used in the deployment version.

```

1  <?php
2
3      $myData = $_POST['myData'];
4      $computerID = $_POST['computerID'];
5      $newSession = $_POST['newSession'];
6      $sessionNumber = 0;
7
8      if (($computerID % 99990001) === 0) {
9          $computerID = $computerID / 99990001;
10         if ($newSession) {
11             $sessionNumber = (int) file_get_contents(
12                 $computerID . ".txt");
13             $sessionNumber = $sessionNumber + 1;
14             file_put_contents( $computerID . ".txt",
15                 $sessionNumber);
16         }
17         else {
18             $sessionNumber = (int) file_get_contents(
19                 $computerID . ".txt");
20         }
21         file_put_contents ( "computer" . $computerID . "
22             session" . $sessionNumber . ".txt", $myData);
23     }
24 }
25 ?>

```

Figure 2.3.4. Full request.php program on the server which handled AJAX posts from the extension

2.4 Analysis Version

The analysis version is a fork of the user version that was designed for the analysis of the data collected using the deployment version. See Section 4.2 - 4.4 for the analysis which used this software.

2.4.1 General Differences

The analysis version is a much farther departure from the original user version than the deployment version. The basic idea for the analysis version was, instead of

collecting data from an active browsing session, to use data that had already been collected for the visualization (see Section 2.2.3 for details on the visualization.) The hope was that having a visualization of the data collected for the experiment would facilitate making interpretations. While some of the users spent much of their time linearly in one tab, preventing the branching that complicates the graph structure making the graph drawing fairly readable, the data from most of the user's browsing sessions was too complicated for the graph drawing algorithm to render in an easily readable fashion. As can be seen in Figure 2.4.1, most of the graphs created by the visualization on the data collected were not readable.

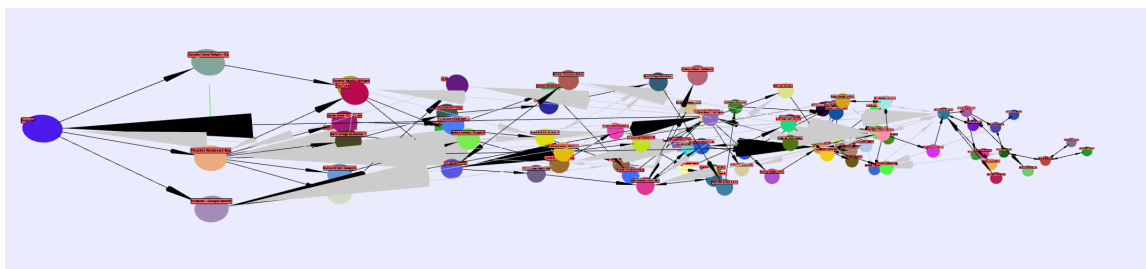


Figure 2.4.1. Example of graph drawing too complex to read

This is not to say loading the data into the visualization program was not useful, while the graph drawing was not particularly useful, using the back and forward buttons on the side panel I was able to navigate through the data contained in the graph and see all the information about each page. Navigating through the data in this manner was helpful as it allowed me to get a more general sense of the data and notice some peculiar anomalies prior to more rigorous analysis.

Most of the program is devoted to parsing through the data collected in order to make calculations, compute the datasets corresponding to the different variables discussed in Section 1.6, and perform the statistical testing used for the analysis in Section 4.2 - 4.4. Making this version of the software efficient was not a priority,

as it not being run continuously and was only intended to be used for the analysis done in this project.

2.4.2 Technical Details

The visualization outlined in Section 1.2.3 was not changed for the analysis version. The development of this version was entirely devoted to changing “graphic.html”. The only change that was made to this file pertaining to the visualization was to instead of request the graph data through the Chrome Extension Message API, it was passed directly to “GraphBuilder.js” after being loaded into “graphic.html”. The data was too large for most text editors to display and run smoothly during development. To remedy this a python script [2], “automater.py”, was used to open “graphic.html”, replace keywords in the javascript array with the browsing data collected during the experiment, and save the resulting code into a new file and open it with a web browser. On each run the program was executed using this python script. All files other than “automater.py”, “graphic.html”, and those needed for the visualization were removed.

Once the data was loaded into “graphic.html” it was parsed into a list which contained every single node from every single browsing session. Since the data was only stored as vertex pairs in a list of edges, this was not completely trivial as it was imperative the dataset was complete and did not contain duplicates, and that nodes which did not describe a web page on the internet (such as Chrome’s new tab page, or a local file) were removed. For every session duplicate URLs were kept track of, and both nodes in every edge were checked to see if they had been added to the set and that the beginning of their URL indicated they used the Hyper Text Transfer protocol (or the secure protocol) used for data communication on the World Wide

Web [11]; if both these conditions were met, the pages were added to the set and the URL was added to the list for the current session to prevent duplicates.

After the list of all the nodes in the data was constructed, it was used to generate all the data used for analysis. There were over 40 arrays constructed containing different datasets. Basic information about most of these datasets including the mean and standard deviation was printed to the console. The datasets themselves were also printed so they could be loaded into a Mathematica notebook to make histograms [27]. Finally the statistical tests outlined in Section 4.2 - 4.4 were performed. See figure 4.2.2 and 4.2.4 for the code used to run two of these tests.

While the final version of the program runs every test on every dataset, in practice most of the code running the tests was commented out, and only one or a small number of tests were run at a time. All of the code before the testing used to parse the data however, was run every time.

2.5 Experimental Setup

The experiment required the participation of volunteers in order to get browsing data for testing the hypothesis. Volunteers were Bard College students. Volunteers were recruited for the experiment using fliers posted around campus, and announcements made during classes. A small amount of compensation was offered to volunteers to attract participants. In order to sign up volunteers would have to email me, and after receiving an email outlining the details of the project and what they should be aware of, they confirmed their participation with an email reply. Prior to the experiment volunteers were encouraged to think of topics they would like to research with the time they were going to spend browsing the web, as having a particular topic they would like to research might make it easier to determine whether some page was relevant to them. This research was approved by the Bard Institutional

Review Board, and the methods for recruitment of volunteers and usage of personal information follow common practices that ensure the protection of the volunteers involved.

There were a total of 14 volunteers, and on Monday March 24th, 2014 from 10am to 1pm the volunteers browsed the web in a lab at the Reem-Kayden Center for Science and Computation at Bard College. All of the machines used by volunteers were the same model iMac [19], and were equipped with the same hardware (same processor, amount of RAM, ect.), mitigating variations in data that could be caused by differences in computer hardware. The machines ran Macintosh OS X 10.7.5 [18], and all used the same version of Google Chrome to run the extension and collect data.

Prior to the arrival of the volunteers, the “deployment version” of the Google Chrome extension was loaded onto each computer. The deployment versions were identical on every computer except for that each computer was assigned a unique integer, computer ID, for the AJAX post to the server. In the PHP program on the server the computer ID was used to distinguish browsing data from different computers, storing data from each into its own file, leading to a total of 14 different files containing browsing data from the experiment, one for each machine (see figure 2.1.9 for the PHP code handling AJAX posts to the server). Google Chrome was left open on the web page “www.google.com” for the start of the users’ browsing session. Before volunteers began browsing, they had to sign a form of consent outlining exactly what the purpose of the research was, what information was being collected from them, and some of the potential risks and benefits posed by the research. After signing the consent form, volunteers were handed and told to read the instructions shown in figure 2.3.1 prior to browsing.

Thank you for participating in my senior project! Before you begin please read these directions.

In front of you now you should see google chrome open to "www.google.com". You should also see a small toolbar that reads "Is this website relevant to you?" and two buttons that read "Definitely Yes" and "Definitely No". While you are browsing the web, if you come across a website that seems relevant to you, click on the "Definitely Yes" button, and if you find one that seems irrelevant to you, click on the "Definitely No" button. You do not need to click on one of these buttons for every website you navigate to, in fact you are only encouraged to click on one of these buttons if you have an opinion one way or the other.

When browsing, please remember and abide by the following restrictions.

DO NOT Log in to any website whatsoever (IMPORTANT).

DO NOT Use any browser besides google chrome (even for a small amount of time).

DO NOT Quit google chrome at any point, if you do this by mistake or if it crashes come get me and I'll start it back up

Hopefully you have come prepared with some topics you would like to research, if you have any questions do not hesitate to ask, now go ahead and start browsing!

Figure 2.5.1. Instructions given to research volunteers

Volunteers were instructed to click on the "Definitely Yes" button on the toolbar when they were viewing a website they found relevant, and the "Definitely No" button when viewing a website they found irrelevant. Volunteers were also instructed

not to log in to any websites. This is both to ensure their anonymity is protected and to make sure they are browsing publicly accessible websites. Volunteers were instructed not to use any browser except Google Chrome to make sure our data is not incomplete, and that all of the web browsing the user did was done using the Google Chrome Extension. Lastly volunteers were instructed not to quit Google Chrome, to ensure that no data loss occurred due to Google Chrome's lack of a quit event in the extension API as described in Section 2.2

At the end of the browsing session volunteers were verbally reminded not to quit Google Chrome, but instead to press the red close button in the upper left hand corner of the window. This triggers the window close listener in the Google Chrome extension API, which in the deployment version invokes a post of the data to the server. After users had closed the window, all of the data were downloaded from the server and stored for analysis. Once the data had been downloaded from the server, on each computer the output of the console of the Google Chrome extension was saved, cataloging any errors the extension might have had. The implications of the errors that were found are discussed in Section 4.1, Integrity of Experimental Data. After this, the software was removed from all of the computers, and the experiment concluded.

3

Results

3.1 Summary of Results

A total of 1,632 different websites were visited during the 3 hour browsing session. All nodes that were not pages from the world wide web, and did not begin with either “http” or “https” in the URL were ignored. These were mainly other Chrome pages, such as the new tab page and the settings page. Of the 1,632 websites visited, only 12 of those websites were visited by more than one user, and those 12 websites were visited by an average of 3.33 users, giving us a total of 1,672 data points. Of those data points, 204, or about 12.2% were deemed relevant, while only 42, or about 2.5% were deemed not relevant. Pages that did not finish loading were removed from the load times datasets, and for the visit time and idle time variables, some data points had to be removed due to errors, for further details see Section 4.1.

3.2 Graphs of Selected Results

The following are histograms of the data collected for variables that are going to be further analyzed. These datasets were analyzed because they exhibited a difference

in observed means that warranted further investigation. For each time a page was loaded, a separate value representing the user’s response to the question on the toolbar, “Is this page relevant to you?”, was recorded. There were no pages in which a user clicked on relevant for one load, and not relevant for a different load, hence there are no nodes which appear in both the relevant and not relevant datasets for any variable.

The captions of the histograms indicate the data they represent by specifying the variable and the dataset. In the upper right hand corner basic information about the data in the histogram is shown, including the size of the dataset, the mean, median, max, min, and standard deviation. Under each pair of graphs describing data from the relevant and not relevant dataset for the same variable, there is a brief description of the datasets and their differences. The following method was used to determine whether a particular value was an outlier. Let m be the median value of the dataset, $q1$ be the median of the set of values less than m , and $q3$ be the median of the set of values greater than m . All values greater than $q3 + \frac{3}{2}(q3 - q1)$ are weak outliers, and all values greater than $q3 + 3(q3 - q1)$ are strong outliers. This is a common approach to computing outliers [6].

Some of the graphs, in order to increase readability, had values outside a certain range omitted. In these instances details about omitted data are included in the description. All graphs display the frequency on the Y-axis in which the value on the X-axis occurred in the given dataset. See Section 1.6 for detailed descriptions of the variables.

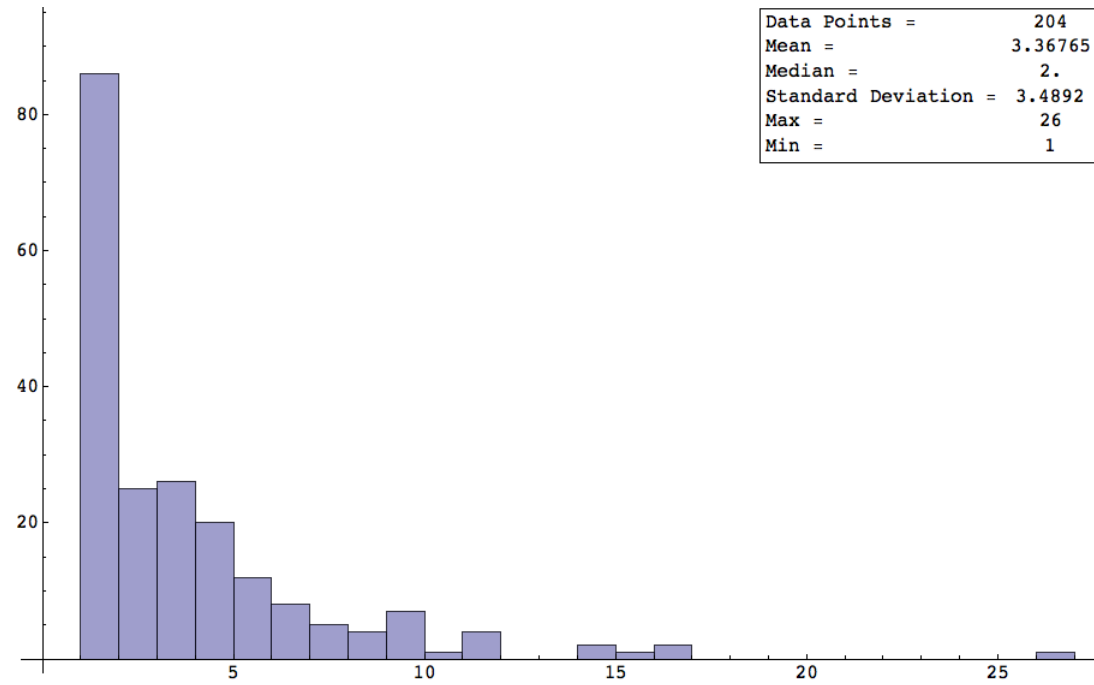


Figure 3.2.1. Relevant Visits

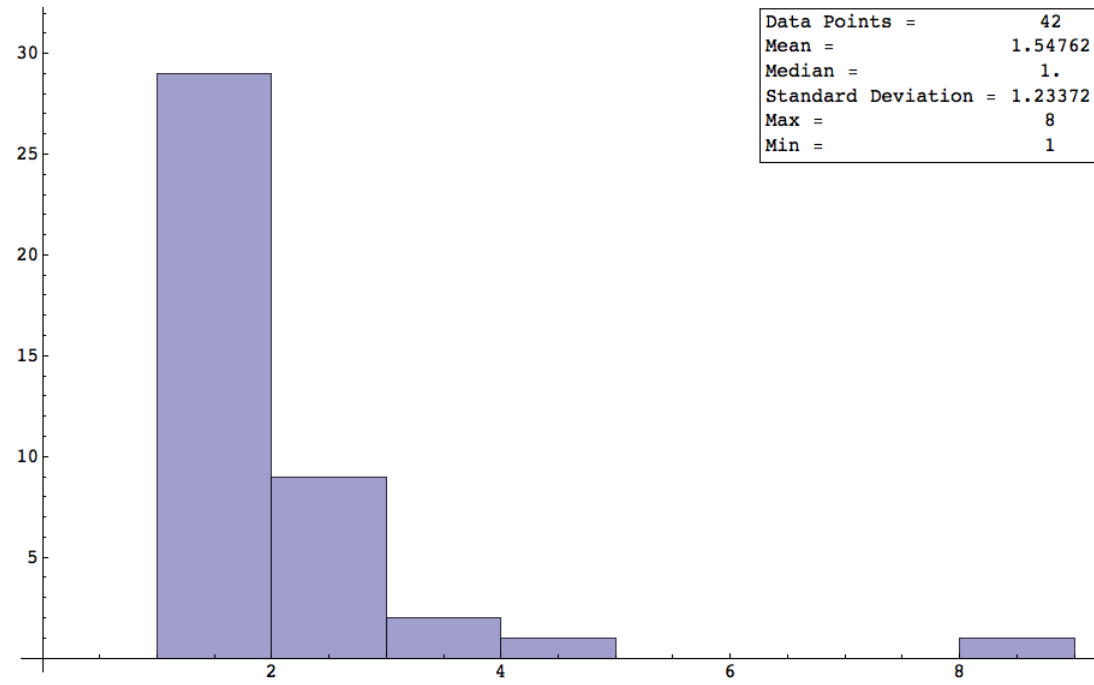


Figure 3.2.2. Not Relevant Visits

For both Relevant Visits and Not Relevant Visits pages were most frequently visited only once, however Relevant Visits had a median of 2, as opposed to only 1 for Not Relevant Visits. There was also a sizable difference in the observed means of both datasets. There was a higher percentage of outliers in the Relevant Visits data, comprising 8.8% of the values (2.9% were strong outliers). For Not Relevant Visits 4.8% of the data were outliers (2.4% were strong).

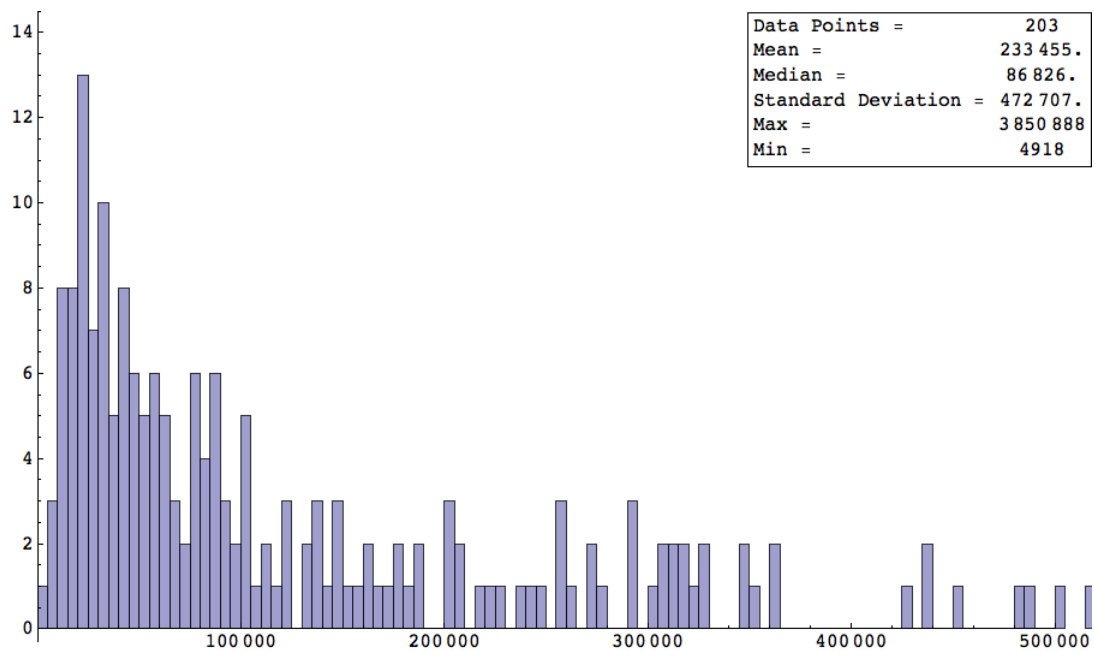


Figure 3.2.3. Relevant Visit Times

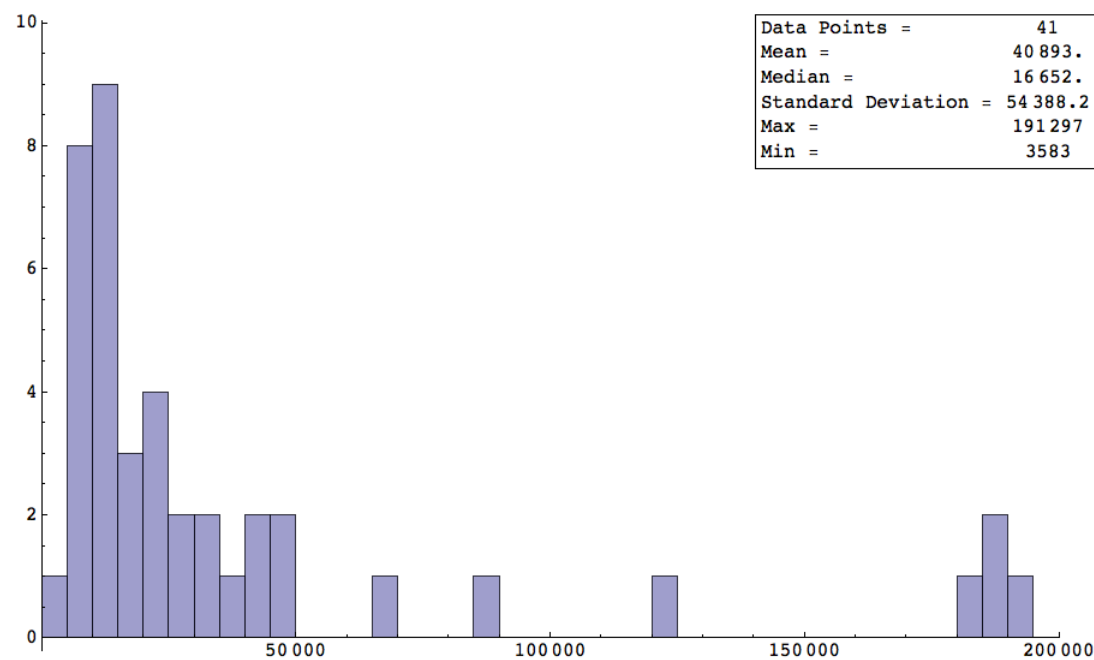


Figure 3.2.4. Not Relevant Visit Times

There was a large difference between Relevant Visit Times and Not Relevant Visit Times in both the observed mean and median. For Relevant Visit Times there were 15 results omitted from the graph averaging 1,397,475. The max not relevant visit time was only about 200 seconds, and many of the Relevant Visit Times were larger. 7.4% of Relevant Visit Times were outliers (5.4% were strong). Several of these outliers were drastically different from the rest of the data; 8 of them were more than 5 times greater than the average. While a larger percentage of Not Relevant Visit Times were outliers (12.2% were weak, 9.8% were strong), they were not as drastically different, and none were greater than 5 times the mean. This partially explains why the standard deviation for Relevant Visit Times is so much larger than the standard deviation for Not Relevant Visit Times.

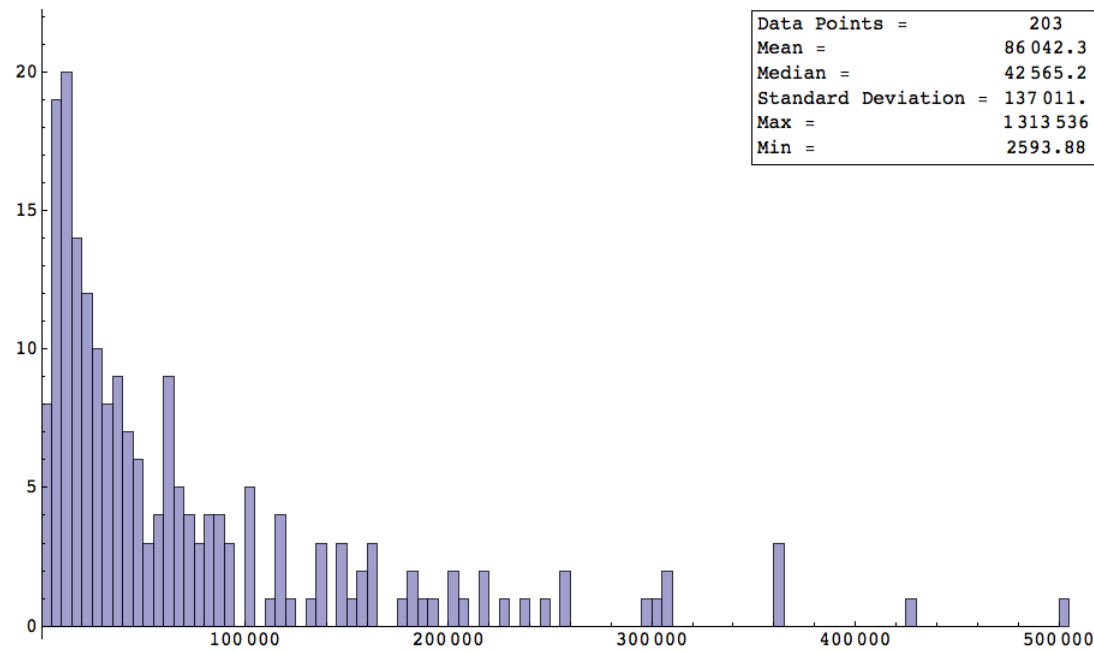


Figure 3.2.5. Relevant Average Visit Time

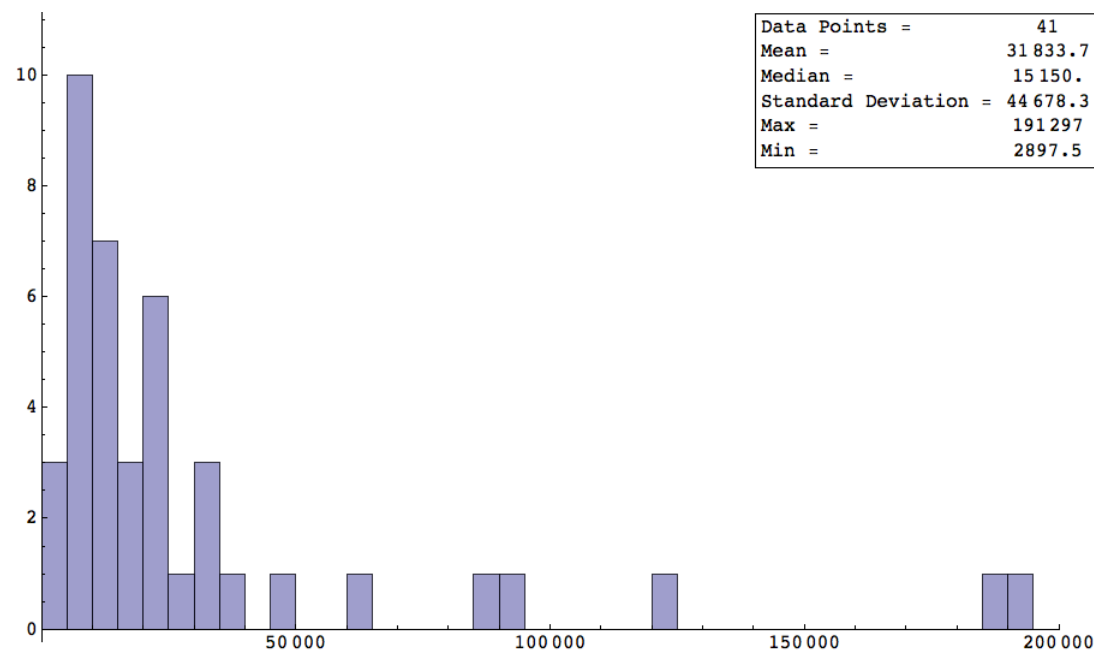


Figure 3.2.6. Not Relevant Average Visit Time

Again, there was a large difference between Relevant Average Visit Times and Not Relevant Average Visit Times for both the observed mean and median. There

were four results omitted from the Relevant Average Visit Times graph, and they averaged 802,721. Since on average relevant pages were visited more often than not relevant pages, Relevant Average Visit Times differ from Relevant Visit Times more than Not Relevant Average Visit Times and Not Relevant Visit Times. The Relevant Average Visit Times had outliers comprise 7.9% of its data, but only 2.4% were strong. This suggests that several of the strong outliers in the Relevant Visit Times dataset were visited at least twice. For Not Relevant Average Visit Times outliers make up 12.2% of its data, and 7.3% of its data were strong outliers. This is not as big a difference from Not Relevant Visits as was exhibited by Relevant Average Visit Times and Relevant Visit Times.

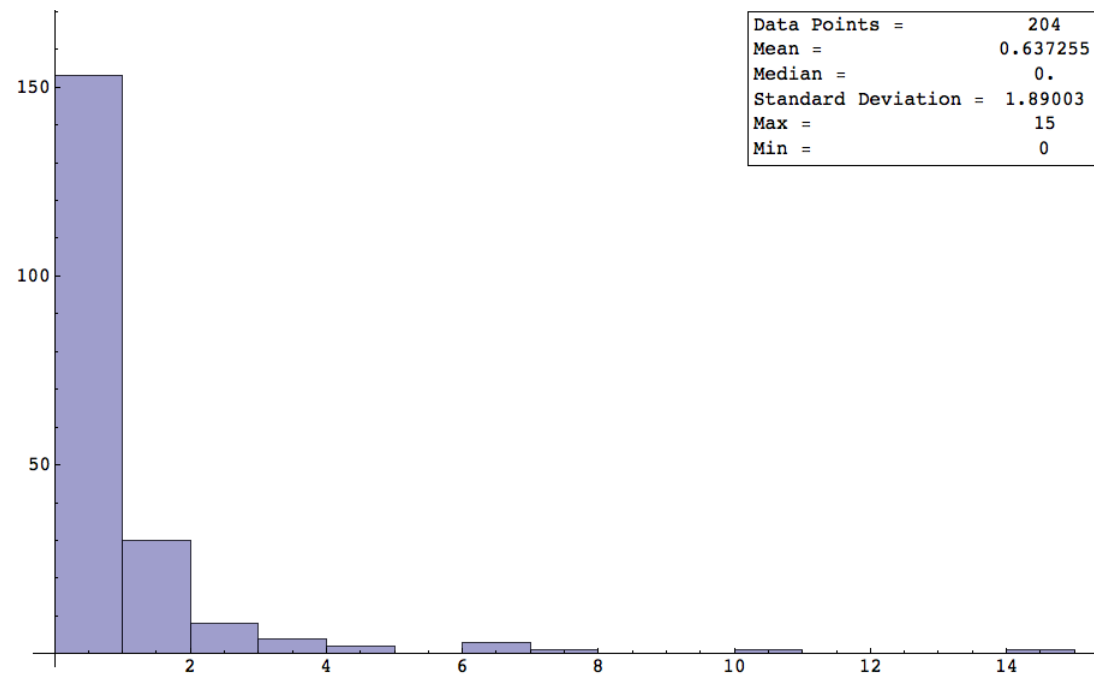


Figure 3.2.7. Relevant Idles



Figure 3.2.8. Not Relevant Idles

For Relevant Idles and Not Relevant Idles, while the observed difference in means was significant, the medians were the same and in each dataset the vast majority of the values were zero. There was however a greater majority (90.5%) of Not Relevant Idles with a value of zero, than Relevant Idles with a value of zero (75%). In the Relevant Idles dataset 6.4% of the data were outliers (3.4% strong). As discussed in Section 2.2.3 the amount of time a user had to perform no action on a page for an idle to happen was relatively short. As a result I would expect many of these outliers to be the result of several long page views, and not a user leaving the machine several times. For Not Relevant Idles every single non-zero value was a strong outlier, and only 9.5% of the values were non-zero.

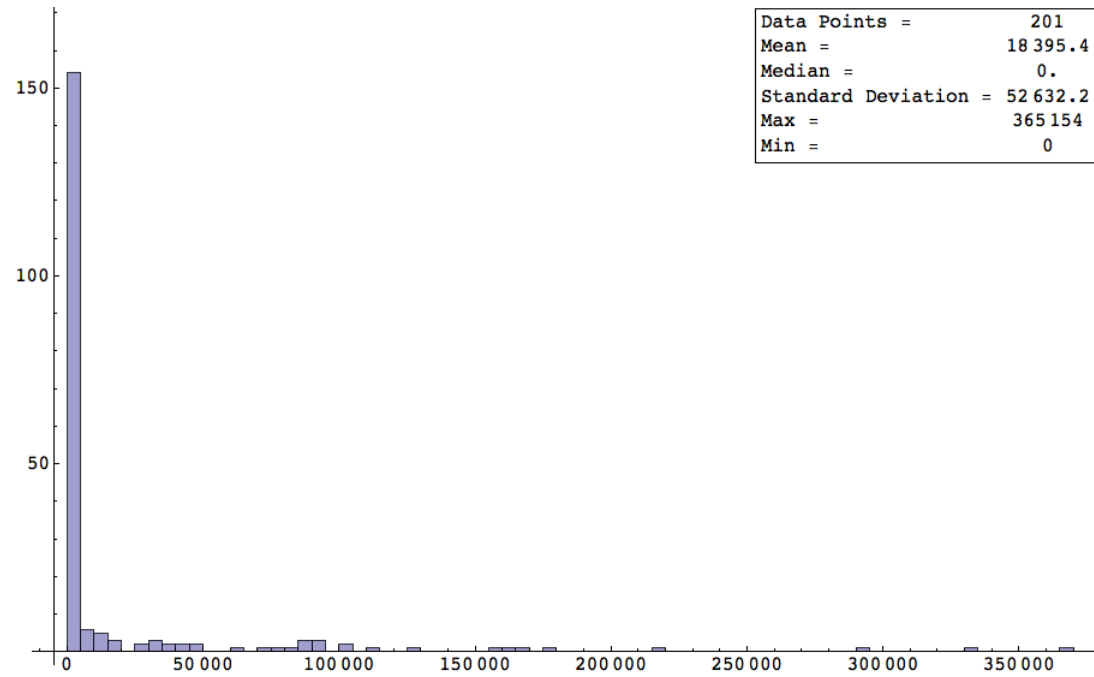


Figure 3.2.9. Relevant Idle Times

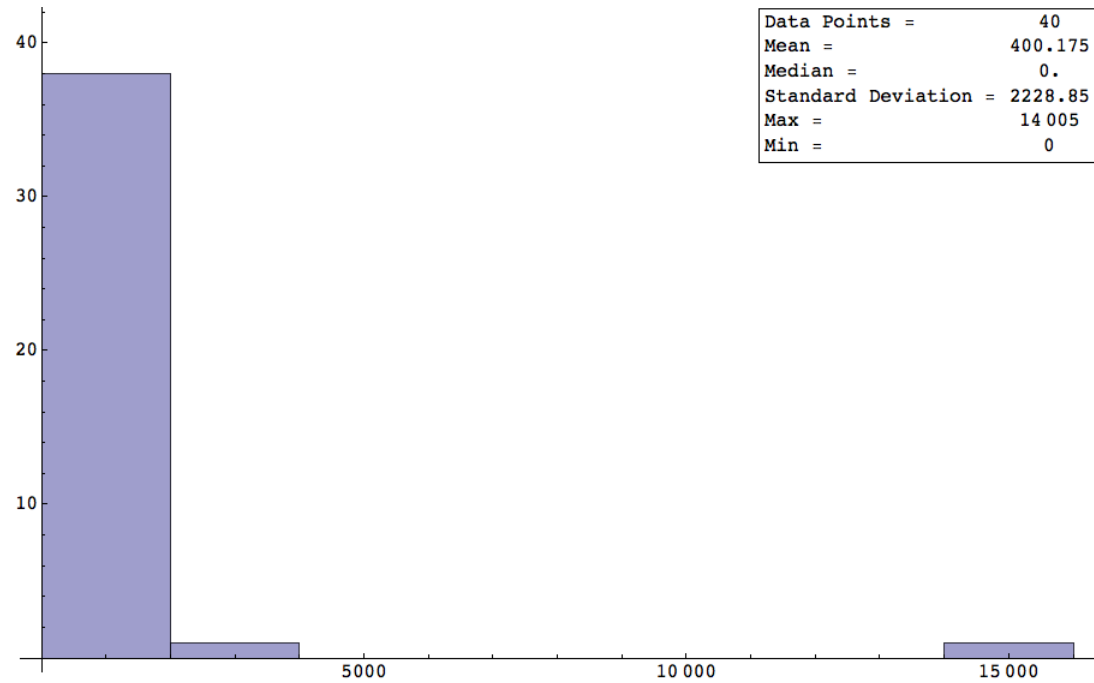


Figure 3.2.10. Not Relevant Idle Times

The observed difference in means between Relevant Idle Times and Not Relevant Idle Times was immense, yet their medians were the same. Pages which were not idled on have an Idle Time of zero, and hence a similar percentage of Idle Time Values were zero. The percentage was not exactly the same though since a small amount of Idle Times data had to be removed due to errors. Since more than 75% of values were zero for both Relevant and Not Relevant Idle Times every single non-zero value was considered a strong outlier. 23.8% of Relevant Idles were non-zero, a much larger percentage than Not Relevant Idles, which had only 5% non-zero values.

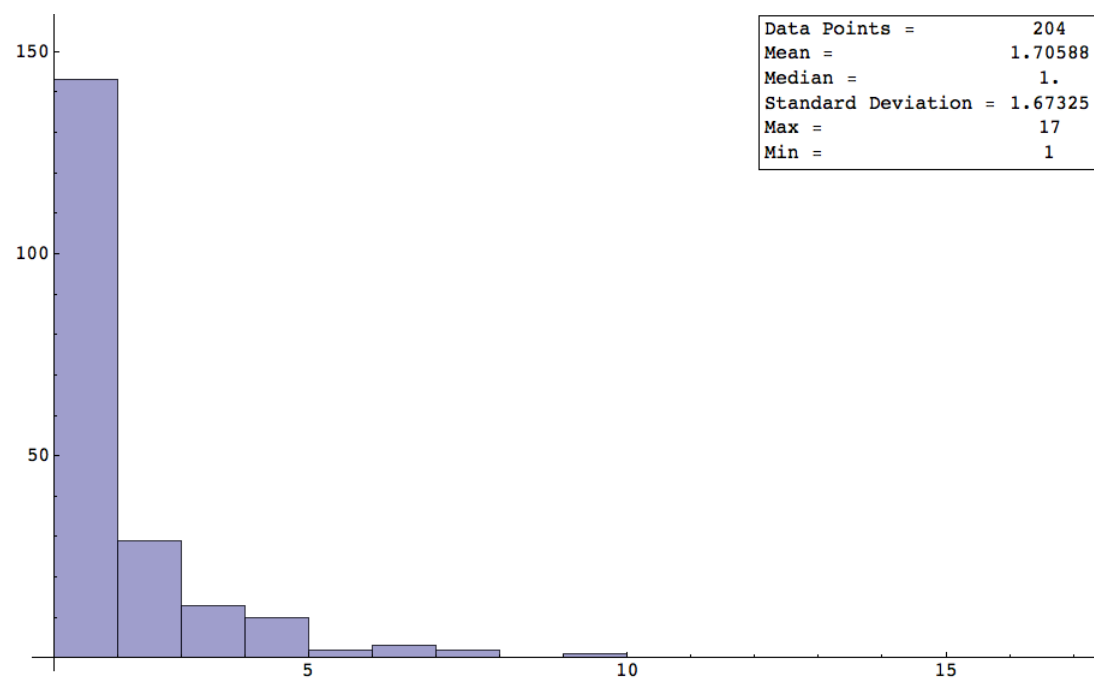


Figure 3.2.11. Relevant Loads

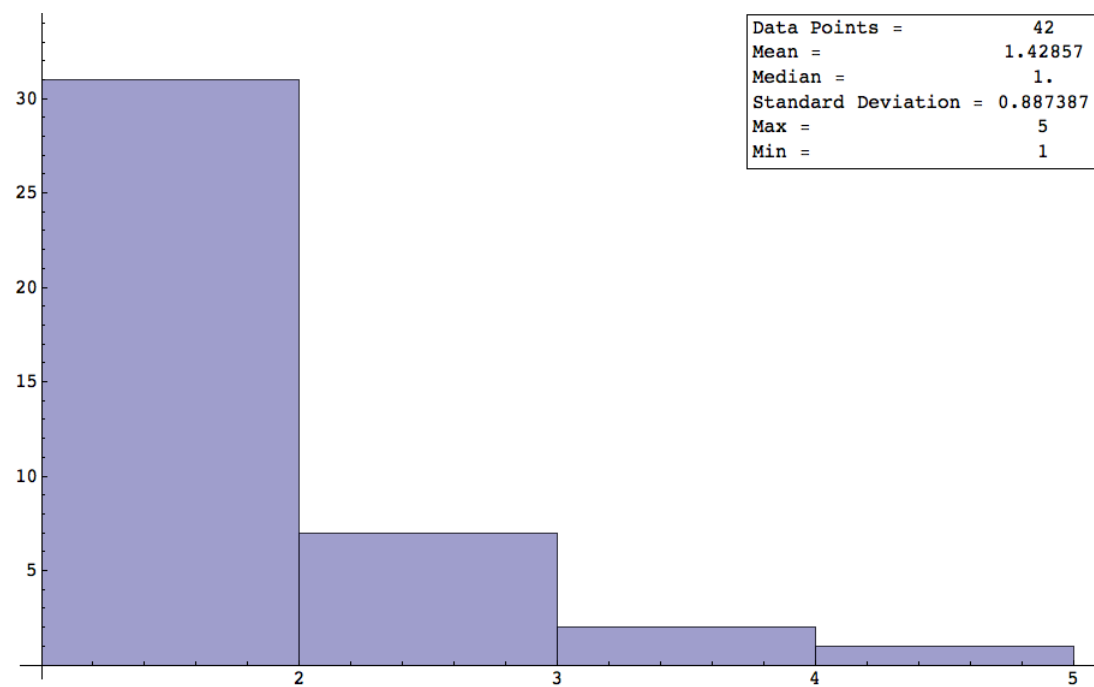


Figure 3.2.12. Not Relevant Loads

There was a small difference in the observed means between Relevant Loads and Not Relevant Loads. A large majority of both relevant and not relevant pages were only loaded once, and the percentage of pages only loaded once in each dataset is about the same. In the Relevant Loads dataset 9.3% of the data were outliers (3.4% strong), as opposed to only 4.8% of Not Relevant Loads (0% strong).

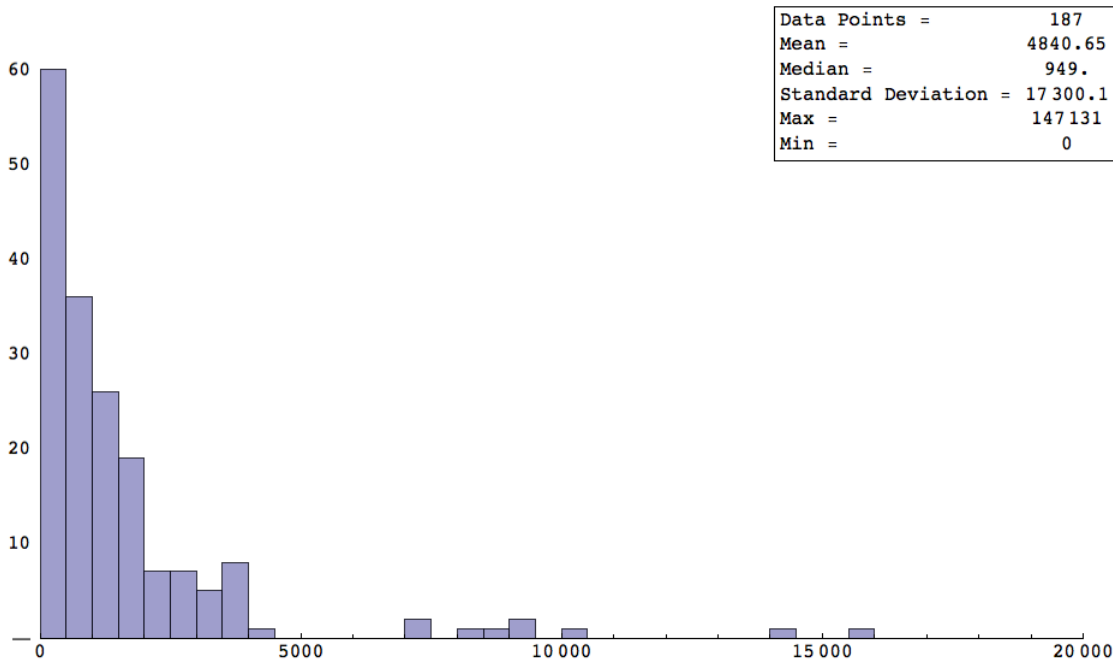


Figure 3.2.13. Relevant Load Times

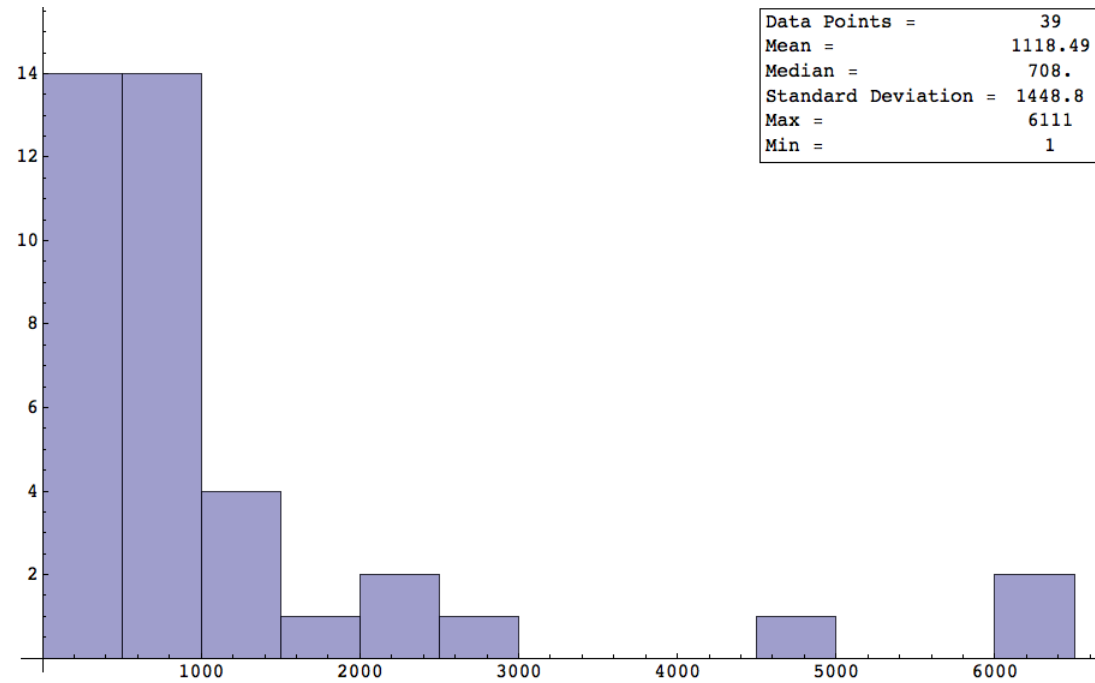


Figure 3.2.14. Not Relevant Load Times

There was a very large difference in observed means, and a much smaller difference in observed medians between Relevant Load Times and Not Relevant Load Times. There were 9 values omitted from the Relevant Load Times graph averaging 69,983. All the outliers in the Relevant Load Times dataset were strong, and they made up 9.6% of the data, a fairly large percentage. Further, many of these outliers were immensely different from most of the data, one-third of them were more than 10 times greater than the mean. For Not Relevant Load Times 12.8% of the data were outliers (7.7% strong), however none differed as drastically from the rest of the data as some of the outliers in the Relevant Load Times dataset. The implications of these outliers are discussed in Section 4.1.

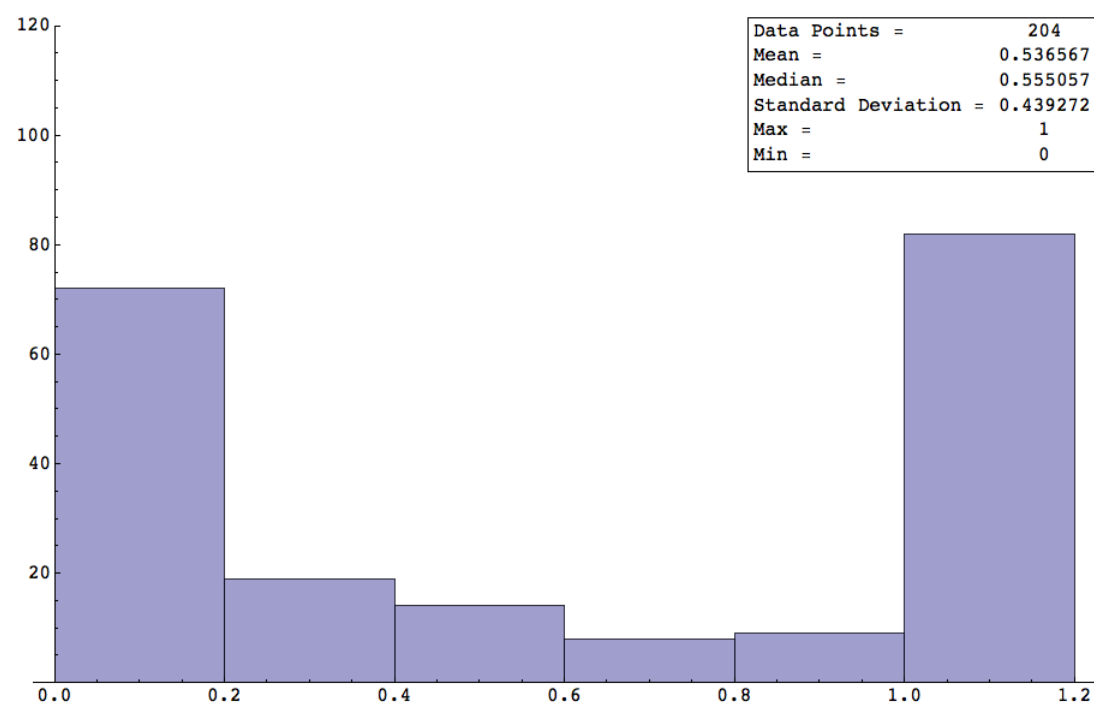


Figure 3.2.15. Relevant Scroll Percentages

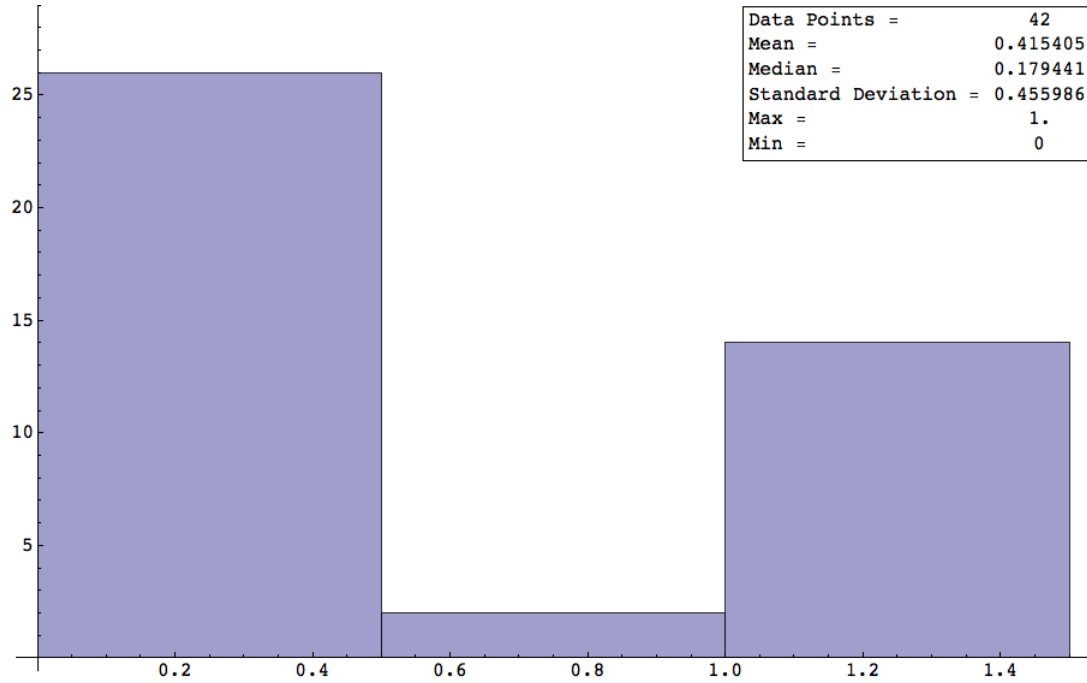


Figure 3.2.16. Not Relevant Scroll Percentages

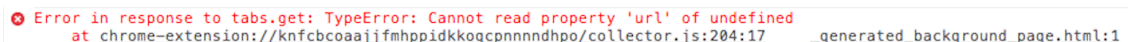
For Relevant Scroll Percentages and Not Relevant Scroll percentages there was a difference in the observed means, and a much larger difference in the observed medians. A large number of pages in both datasets were either not scrolled on at all, or scrolled all the way, giving their graphs larger values on both ends. There was both a higher percentage of Relevant Scroll Percentages with a value of one and a smaller percentage of Relevant Scroll Percentages with a value of 0, although neither difference was drastic. This was the only dataset with a constraint on the range of possible values, and there were no outliers in either dataset.

4

Analysis

4.1 Integrity of Experimental Data

For the most part our data appeared to be intact, and there were no obvious reasons to doubt the overall integrity of the data collected. The errors in the program were logged and recorded at the end of the experiment, and in 9 of the 14 browsing sessions at least one error occurred. None of these errors caused the extension to crash, and on every machine data was posted to the server even after errors occurred. There were only two different errors, the first was a type error on an attempt to read the URL of a tab. It output the following:

A screenshot of a Chrome DevTools console error message. The message is displayed in red text on a white background. It reads: "Error in response to tabs.get: TypeError: Cannot read property 'url' of undefined". Below this, in smaller text, it says "at chrome-extension://knfcbcoaajifmhppidkkogcpnnndhpo/collector.js:204:17" and "_generated_background_page.html:1".

```
⛔ Error in response to tabs.get: TypeError: Cannot read property 'url' of undefined
    at chrome-extension://knfcbcoaajifmhppidkkogcpnnndhpo/collector.js:204:17
    _generated_background_page.html:1
```

Figure 4.1.1. Type Error on Read of Tab URL

This error occurred a total of 8 times on 4 different computers. On three of the computers, it happened three times consecutively, on the other two it happened only once. The Google Chrome extension passed the value of a tab ID to the “tabs.onActivated” listener, which is called when a new tab highlight occurs, that

resulted in an undefined value in the “tabs.get” function callback. I do not know why this error occurred, and this error did not occur during testing. I speculate it occurred because a tab was highlighted and immediately closed, and in the time it took for the “tabs.get” method to call the callback function the tab had already closed, and was therefore undefined. If this were the reason, the only data that would have been affected are the visits and visit times, since the page was technically visited and viewed for a very small amount of time. This data would have been affected an insignificant amount as the error was infrequent, and the data it may have prevented from being collected was insignificant. Nevertheless, since it is not known the reason this error occurred or the impact it may have had on the data we cannot take it into consideration during our analysis.

The next error was a type error which occurred when trying to write the end value of an idle time. It output the following:

```
✖ Error in event handler for idle.onStateChanged: TypeError: Cannot set property 'end'
  of undefined
    at chrome-extension://knfcbcoaajifmhppidkkoqcpnnndhpo/collector.js:240:67
    at Function.target.(anonymous function) (extensions::SafeBuiltins:19:14)
    at Event.dispatchToListener (extensions::event_bindings:394:22)
    at Event.dispatch_ (extensions::event_bindings:378:27)
    at dispatchArgs (extensions::event_bindings:246:26)
    at dispatchEvent (extensions::event_bindings:255:7)
    at generated_background_page.html:1
```

Figure 4.1.2. Type Error on Write of Idle End Time

This error occurred a total of 7 times on 5 different computers. The reason this error occurred is because a different page was navigated to before the “idle.onStateChanged” listener was called, and the program attempted to set the idle end time on a page which had not been idled on. This suggests that the function callback does not always get called immediately after a user returns from being idle. This error did not occur during testing, however testing for the collection of Idle Time data was not as thorough as most of the other variables. This error did have

a significant impact on the data collected for Idle Times. There were 22 idle time values in which the end idle time was null, 7 of these can be attributed to this error. The other 12 of the remaining 15 were artificial idles, since they occurred at the end of the browsing session while volunteers were being instructed, presumably for more than 30 seconds, about how to complete the session among other things. We would expect idle state to change and update the data accordingly when the user moved the mouse to close the window; the fact that it did not on 12 out of the 14 machines is evidence to the amount of delay there is before the listener is called (all 14 of the last pages browsed were idled on). The cause of the remaining three null end idle times is not known, however it is likely that the same issue described above (where a delay in the program's calling "idle.onStateChanged" caused it to be called on a page that had not been idled on) happened except "idle.onStateChanged" was called on a page that had been idled on previously. This would not cause an error, since the end time of the last idle would be defined, however it would cause the page that the idle end truly occurred on to have a null end idle time. Also, the page that the listener was incorrectly called on would have an incorrect end idle time making its data invalid, and its invalidity undetectable. This happened at most 3 times.

All of this might call into question the validity of the Idle Times data. If we dismiss the artificial idles that took place at the end of the browsing session that were null, we are left with only two null end idle values on relevant pages, and only one null end idle value on not relevant pages. Of the 51 relevant pages in which an idle even occurred, two pages, or 4% had a null end idle value. In contrast, of the three not relevant pages that an idle occurred on, one page, or 33% had a null end idle value. At most three of the 53 remaining data-points being analyzed had an incorrect end idle time. During analysis of Idle Times and Average Idle Times we took this into account, and remained mildly skeptical of the gathered data's validity. It should

be noted that the described issues do not appear to have affected the data for the number of idles that occurred, as these occur during when an idle begins, and we have no evidence this occurred on a page incorrectly.

No computer had both of the aforementioned errors, only no errors, a single error, or two or more of the same error. While these were the only errors that caused an explicit error in the extension, there was evidence for other potential errors in the data.

For visit times, at the time of deployment in the window close listener the page currently being viewed did not get its view time updated, and as a result 14 of the pages visited had a null value for the end of a visit time. All 14 of these pages were the last ones being viewed by the user when they were instructed to close the window, which suggests that this was the only issue in the data for visit times, and that the rest of the Visit Times data is intact. Data loss due to the described problem only happened on one relevant page, and on one not relevant page. These were removed from the datasets and presumably did not have a significant impact.

For Load Times while there are not apparent errors there are reasons to question the integrity of the data and how accurately it represents real world data. There were many Load Times with a null end time, however this was part of the design of the software and indicates that a load did not complete. About 5.4% of all the pages were not loaded completely, this is not an unreasonable percentage, and we have no reason to believe that the data gathered for Load Times contains errors. The cause for concern is the amount of drastic outliers. Among the 187 relevant pages in which a load was completed, 18 of them, or about 9.6% were strong outliers. The median relevant page load took only 949 milliseconds, and page loads that took longer than about six seconds were considered strong outliers. The cause for concern is not merely the number of page loads that took longer than six seconds, it is the

amount more in which several took to load. Nine of these pages took more than 20 seconds to load, five took more than 60 seconds, and two even took more than 120 seconds.

After examining these pages, a few might have taken such a long time because they had a lot of information on them, such as large images. However most were foreign pages and presumably hosted on foreign servers partly explaining why they took so long to load. Still, when I tried to load the pages again from the same internet connection in the lab, while many still took a fairly long amount of time to load, most did not take nearly as long as they did during the experiment. This suggests that either the servers hosting these sites were slower or took longer to respond during the experiment, or more likely that the internet was slower for the participants since there were 14 people browsing the web on the same connection (or both). If the latter were the case, it would compromise the claim that this data is representative of real world browsing data. With real world data, Load Times would vary widely given varying connection speeds, however I would expect that most of the time these connections are not being slowed down because of the number of people using them. Nevertheless, we proceed as though they are representative of real world data despite evidence to the contrary. These concerns also apply to the Average Load Times data.

For information on how some of these errors can be avoided in future implementations of the software, see Section 5.2.

4.2 Testing of Main Hypothesis

The observed difference of means between relevant visit times and not relevant visit times was 192,562 milliseconds, which is far from zero. However the datasets did not seem to fit any known distribution, which eliminated many common parametric tests

for comparing means. One common testing method which assumes no distribution is statistical bootstrapping. The basic idea behind bootstrapping is to repeatedly take samples of a dataset, allowing resampling, and computing the means of the samples to see how much they vary and how confident we can be that the true mean of our population is within a certain range [6]. Our hypothesis was that the average amount of time spent viewing a relevant page would be greater than the amount of time spent viewing a not relevant page, so if we can demonstrate the mean of our recorded Relevant Visit Times is greater than that of our Not Relevant Visit Times we will have demonstrated our hypothesis to be true. This can be done using a two sample bootstrap, where we sample the means of both our datasets and compute the difference, if our difference is greater than 0 within our desired degree of confidence then we can say our hypothesis to be true with the degree of confidence tested. After performing the test, we got the following results for our sample differences. See Figure 4.4.2 for the code used to run the bootstrapping test.

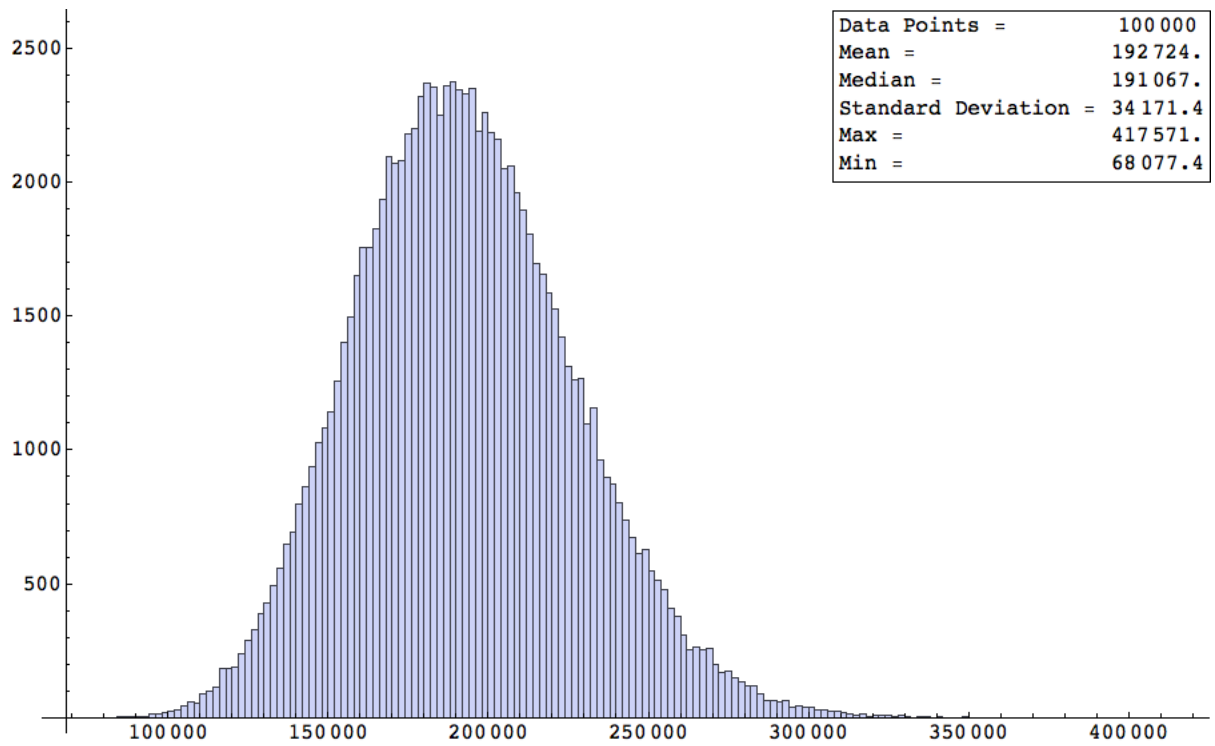


Figure 4.2.1. Bootstrapping Mean Differences for Relevant Visit Times and Not Relevant Visit Times

```

1    function performBootstrapComparison(sample1, sample2,
    confidenceInterval) {
2        if (confidenceInterval==undefined) {
3            //set a default confidence interval
4            confidenceInterval = .95;
5        }
6        var randomSamples1 = [];
7        var randomSamples2 = [];
8        var differences = [];
9        for (var i = 0; i<100000; i++) {
10           for (var j = 0; j<sample1.length; j++) {
11               randomSamples1.push(sample1[Math.floor(
12                   Math.random() * sample1.length)]);
13           }
14           for (var j = 0; j<sample2.length; j++) {
15               randomSamples2.push(sample2[Math.floor(
16                   Math.random() * sample2.length)]);
17           }
18           differences.push(computeAverage(
19               randomSamples1) - computeAverage(
20               randomSamples2));
21           randomSamples1 = [];
22           randomSamples2 = [];
23       }
24       differences.sort(function sortNumber(a,b) {
25           return a-b;
26       });
27       console.log("BOOTSTRAP DIFFS");
28       console.log(stringForMathematica(differences));
29       console.log("\n\n\n");
30       var lower = differences[Math.floor(differences.
31           length * (1-confidenceInterval))];
32       var upper = differences[Math.floor(differences.
33           length * confidenceInterval)];
34       console.log("Bootstrapping has determined with "
35           + confidenceInterval + " confidence that the
36           differences of the means is between " + lower
37           + " and " + upper);
38   }

```

Figure 4.2.2. Code in “Analysis Version” used to perform the bootstrap comparisons of two samples

The program output the following: “Bootstrapping has determined with 0.999 confidence that the differences of the means is between 102453.23705394688 and 312463.9865433137”. Even our lower estimate, 102,453.2 is well above 0, giving us an even greater than 99.95% confidence that the mean of the Relevant Visit Times is greater than that of the Not Relevant Visit Times. Furthermore, we are 99.95% confident that the average amount of time a relevant web page is visited is at least 102.45 seconds longer than a not relevant web page.

In addition to this test, to further confirm the difference of these datasets and also get a sense of how much greater than 99.95% our confidence that the means are different is, a permutation test was done. The basic idea of a permutation test is to first assume that the means of the datasets being compared are equal, and reason that since the difference in means is equal to zero, so too will the difference of the means between randomly selected subsets of a combined dataset. In other words, if we combine our two datasets and take the means of randomly selected subsets of this dataset, we would expect the difference of these means to usually be close to zero, and to get the probability the means of our datasets are the same, we compare how often the difference of the means of the random subsets lie outside the difference of our observed means [6]. After performing the test, we got the following results for our sample differences. See Figure 4.4.4 for the code used to run the permutation test.

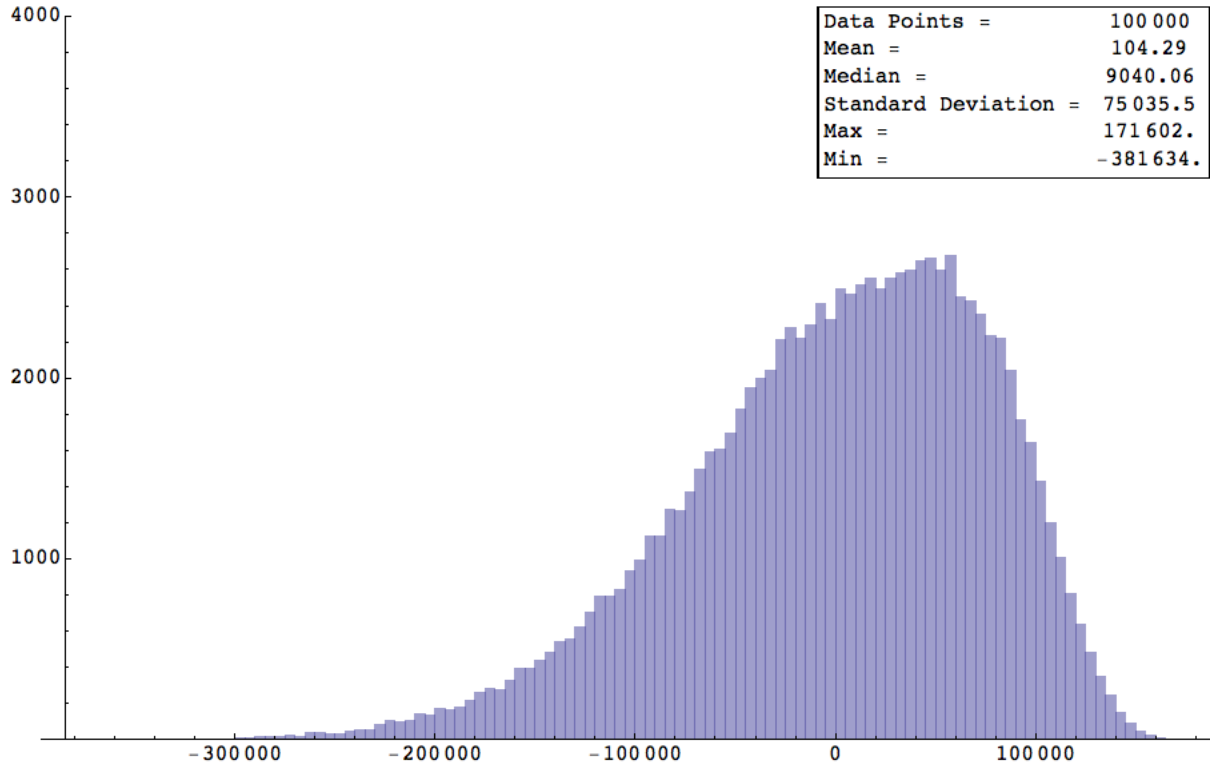


Figure 4.2.3. Permutation Test Mean Differences for Relevant Visit Times and Not Relevant Visit Times

We can see in Figure 4.2.3, that our observed mean difference, 192562, was larger than all 100,000 permutation test sample differences. To see if with a larger number of sample differences we might see a test sample difference larger than our observed difference we ran the test with larger sample differences. Even with the sample differences set to 10,000,000 still not one permutation test difference was larger than our observed difference, and the program output the following: “the permutation comparison test has determined with 0.99999990000001 confidence that the mean of the first sample is greater than the mean of the second”. Sample differences significantly larger than 10,000,000 resulted in an error, “Script on the page used too much memory”. With a permutation test using 10,000,000 samples, we can

conclude with a very strong 99.99999% degree of confidence that the average of our Relevant Visit Times is greater than the average of our Not Relevant Visit Times.

```

1 function performPermutationComparison(sample1, sample2,
   numResamples) {
2   if (numResamples==undefined) {numResamples = 100000;}
3   var fullSample = sample1.slice(0).concat(sample2.
     slice(0));
4   var averages = [];
5   for (var i = 0; i<numResamples; i++) {
6     permuteArray(fullSample);
7     averages.push(computeAverage(fullSample.slice(0,
       sample1.length)) - computeAverage(fullSample.
       slice(sample1.length)));
8   }
9   averages.sort(function sortNumber(a,b){return a-b;});
10  var observedDifference = computeAverage(sample1) -
    computeAverage(sample2);
11  console.log("PERMUTATION AVERAGES DIFFS");
12  console.log(stringForMathematica(averages));
13  console.log("\n observed difference: " +
    observedDifference + "\n\n\n");
14  var index = averages.length;
15  for (var i = 0; i<averages.length; i++) {
16    //check to see where our observed difference is
17    if (observedDifference < averages[i]) {
18      index = i; break;
19    }
20  }
21  var confidence;
22  if (index > averages.length/2)
23    confidence = (1 - (index/(averages.length+1)));
24  else
25    confidence = (index/(averages.length+1));
26  console.log("the permutation test has determined with
    " + (1-confidence) + " confidence that the mean of
    the first sample is greater than the mean of the
    second");
27 }

```

Figure 4.2.4. Code in “Analysis Version” used to perform the permutation test

One issue with means comparison is that a few outliers in one dataset can have a large impact, even though they do not reflect the behavior of the data as a whole. There were several very large values in the Relevant Visit Times dataset that may have influenced the mean testing in this manner. This is not meant to imply that the outliers observed are not legitimate data points. We simply would like to verify that the difference in means observed is not largely *because* of these outliers, and that we can be confident there is indeed a more fundamental difference between the two datasets. The observed difference between the median Relevant Visit Time and median Not Relevant Visit Time was 70,314 milliseconds, to see how widely this will vary within our 99.9% confidence interval, we ran a bootstrapping test with medians instead of with means. The program output the following: “Bootstrapping has determined with 0.999 confidence that the differences of the medians is between 36402 and 110022”. We got the following results for our median differences.

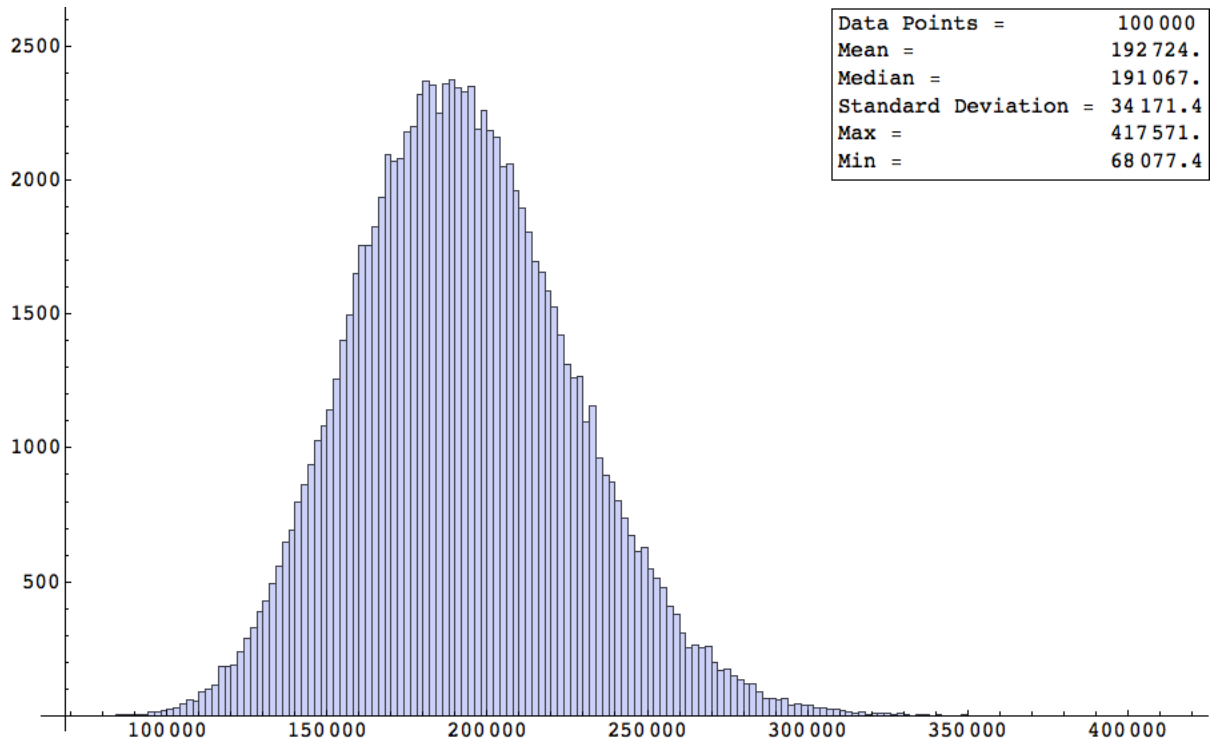


Figure 4.2.5. Bootstrapping Median Differences for Relevant Visit Times and Not Relevant Visit Times

The effect outliers had in the bootstrapping test for means is evidenced by the smaller difference values the bootstrapping test for medians produced at the same confidence interval. That being said, even the low estimate for our medians, at a high confidence interval, was significantly larger than zero. This result was confirmed with a permutation test comparing the medians. This demonstrates that the effect the outliers had on the mean difference was not so large that it might put the legitimacy of our claim, that Relevant websites are viewed for a longer amount of time than not relevant websites, into question. Hence, we can conclude with a very high degree of confidence that our hypothesis is true.

4.3 Testing of Other Variables

For all but one of other variables the observed mean of the Relevant dataset was larger than the observed mean of the Not Relevant dataset. The only variables for which this was not true, was Refreshes, where the difference was negligible. The statistical tests done for the Visit Times were also done for all the variables other than Refreshes, and Average Idle Times (where there was insufficient data) to see which ones with a high degree of confidence also determine page relevance. In addition, the results of these tests are used to make inferences about which variables best determine page relevance. For the bootstrap test a confidence of .999 was used unless the interval returned contained zero. If this happened .99 was tested, and if the interval still contained zero the confidence was decremented by .01 until the interval did not contain zero. For the permutation tests, 100,000 samples were used unless the observed difference was larger than every sample difference, in which case a larger number of samples were used. The results of every test done are not discussed, they can be found in Figure 4.4.1.

4.3.1 Visits

First we ran a bootstrapping test to compare Relevant Visits and Not Relevant Visits. We found with 99.9% confidence that the difference of the means is between 0.84663865 and 2.79271708. Like our Visit Times, we verify with greater than 99.95% degree of confidence that relevant pages are Visited more than not relevant pages. To see how much greater our confidence is we ran the permutation test. With 1,000,000 samples, the test found with 99.9987% confidence that the mean of the first sample is greater than the mean of the second". The permutation test gave us a 99.9987% degree of confidence that the mean Relevant Visits is larger than the mean Not Relevant Visits. This is a very high degree of confidence, but not quite as high

as the degree of confidence given by the permutation test for Total Visit Times. To mitigate the impact of outliers, we use the bootstrapping median test which determined with 99.9% confidence that the differences of the medians is between 0 and 2. If we use a smaller degree of confidence for the test we get the following output: “Bootstrapping has determined with 0.98 confidence that the differences of the medians is between 0.5 and 2”, and we can conclude with 99% confidence that the difference between the median Relevant Visits and median Not Relevant Visits is greater than 0.5. This shows that outliers impacted the difference in mean Visits, but not enough to lead us not to believe with a high degree of confidence that the more times you visit a page, the more likely it is to be relevant.

4.3.2 Average Visit Times

Next we ran tests for Average Visit times. The Average Visit Time for each node is calculated by dividing the visit time by the number of visits. We found Relevant Visit Times to be larger than Not Relevant Visit Times, but also the number of Relevant Visits to be larger than Not Relevant visits, putting into question whether the Average Visit Time on a relevant page would be longer than a not relevant page. The bootstrap mean comparison found with 99.9% confidence that the differences of the means is between 18920.45 and 92654.80. Again, we are more than 99.95% certain that the difference in the Average Relevant Visit Time is larger than the Average Not Relevant Visit Time. Furthermore, and we can conclude with 99.95% certainty that each view of a relevant page is on average at least 18.9 seconds longer than a view of a not relevant page. To see how much greater our confidence is that the means are different we ran the permutation test. With 100,000 samples, the test found the mean Average Relevant Visit Time to be greater than the mean Not Relevant Average Visit Time with 99.991%. The permutation test gave us a 99.991%

degree of confidence that the mean Relevant Average Visit Time is larger than the mean Not Relevant Average Visit Time. This is a very high degree of confidence, but not as high as the confidence given by the permutation test for Total Visit Times or Visits.

Anecdotally, since many of the outliers from the Relevant Visit Times were visited several times, we would expect them to have less of an impact on the Average Visit Times, however the number of outliers for both variables were similar. The bootstrap median comparison found with 99.9% confidence that the difference of the medians is between 7455.6667 and 48321.5. So we can be more than 99.95% confident the median Relevant Average Visit Time is greater than the median Not Relevant Average Visit Time, this was verified with a permutation test comparing the medians. There was still a large difference between the median and the mean, suggesting outliers still had an impact on the difference in means.

4.3.3 *Idles*

Next we compared the number of Relevant Idles and Not Relevant Idles. The bootstrap mean comparison determined with 99.9% confidence that the differences of the means is between 0.05952380 and 1.00280112. We find with greater than 99.95% confidence that the mean number of Idles on relevant pages is greater than the mean number of Idles on not relevant pages. The permutation test gives a confidence of 99.23%, a result that is not quite as high as the bootstrap, but still a very good degree of confidence. As we discussed previously in Section 2.1.1 however, since the time before an idle was triggered was set to only 30 seconds it can be reasoned that many of these idles probably were the result of an extended viewing of the page. The following bar chart shows average view time of a page by the number of times

it was idled on, the correlation serves as evidence that idles were often triggered by long page views.

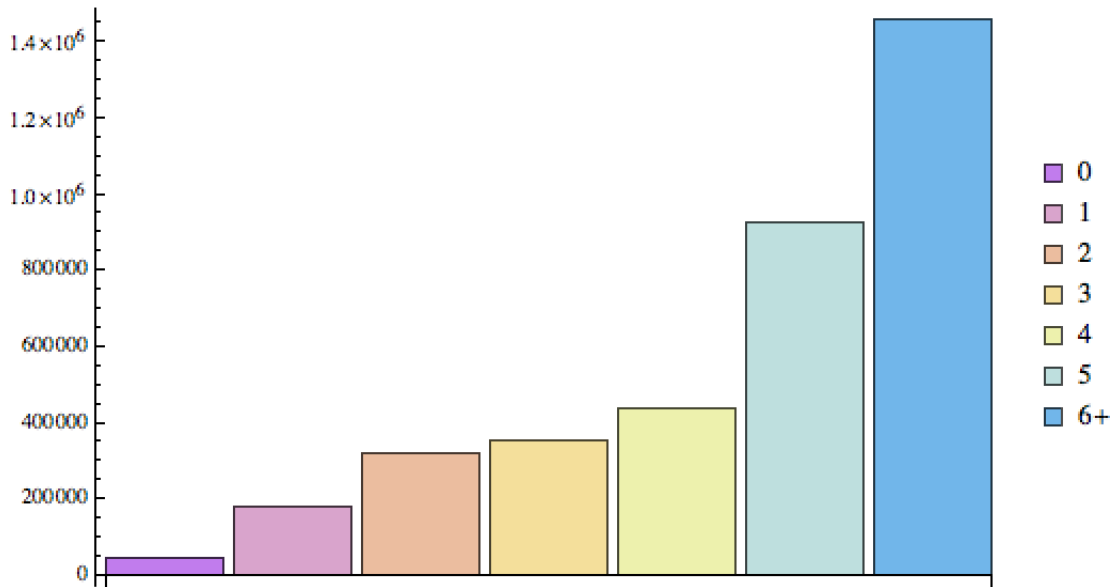


Figure 4.3.1. Average Visit Time of Pages By Number of Idles

We would like to be able to remove idles which were less than some amount of time and compare to see if the pages idled on with a greater likelihood of being the result of leaving the computer rather than a long view have as large a confidence in their difference. Unfortunately due to our lack of Not Relevant Idles data where an idle actually occurred, this test was not done. A large majority of pages were not idled on, and hence the observed median of both datasets is zero. The results of median tests confirmed they are almost certainly equal. While the test of medians does not confirm our findings for the difference of means, we can still conclude that idles is likely a predictor of page relevance.

4.3.4 *Idle Times*

Next we compared Idle Times. All but two of the not relevant pages were never idled on, giving 38 out of 40, or 95% of Not Relevant Idle Times a value of zero. In contrast, 153 out of 201, or 76.1% of Relevant Idle times were zero. This is evidence that our mean values are going to be different, however the observed median of both datasets is 0, and it would seem unlikely that the medians of the two datasets are going to be different. First we run the bootstrap mean comparison, which determined with 99.9% confidence that the differences of the means is between 8029.25 and 31041.28. Bootstrapping shows with 99.95% confidence that on average a relevant page is idled on for at least 8.029 seconds longer than a not relevant page, and with greater than 99.95% confidence that Relevant Idle Times are larger than Not Relevant Idle times. To see how much greater our confidence is we ran the permutation test. With 100,000 samples, the test determined with 99.974% confidence that the mean of the first sample is greater than the mean of the second. The permutation test gave us a 99.97% degree of confidence that the mean Relevant Idle Time is larger than the mean Not Relevant Idle Time. This is a high degree of confidence, but not as high as the confidence given for several other variables.

Although our observed medians were the same, we can use the results of the bootstrap median comparison to determine how much they may vary. The bootstrap median test output the following: “Bootstrapping has determined with 0.999 confidence that the differences of the medians is between 0 and 0”. Every single sample median was zero, and we can say with at least 99.999% confidence (since there were 100000 samples) that the medians are equal.

All we can conclude about Idle Times is that most pages are never idled on, and that Relevant Pages on average are probably idled on for longer than Not

Relevant Pages. To get a better sense of how Idle Times impact the likelihood of page relevance, it would be a good idea to omit pages where the Idle Times were zero, and see among pages that were idled on whether Relevant Idles Times are greater than Not Relevant Idle Times. This would at least yield a more meaningful median comparison. Unfortunately there were only two not relevant pages which were idled on; so we do not have sufficient data to perform this test.

4.3.5 *Loads*

Next we examine Loads. The observed means of the Relevant Loads and Not Relevant Loads were not far apart, so we would expect the degree of confidence that the mean Relevant Load is greater than the mean Not Relevant Load to be less than that of the previous variables tested. The bootstrap comparison of means found with 93% confidence that the differences of the means is between 0.01330532 and 0.54061624. Bootstrapping shows with 96.5% confidence that the mean Relevant Load is greater than the mean Not Relevant Load. Next we ran the permutation test, which found with 87.73% confidence that the mean of the first sample is greater than the mean of the second. The permutation test gave a confidence that is lower than what we deem acceptable in order to conclude that the means are different. Hence, while there is evidence, we can not conclude that relevant pages are loaded more than not relevant pages. Since there was not a verifiable difference in means, and the observed medians were the same, the results of the other tests are not analyzed further, and they can be viewed in Figure 4.4.1.

4.3.6 *Load Times*

Somewhat surprisingly Relevant Load Times and Not Relevant Load Times had a sizable difference in observed means. I would not expect Relevant Pages to take longer to load, however the observed mean of Relevant Load Times, 4840.65, was

about 3722 greater than the observed mean of Not Relevant Loads. Perhaps the reason this is the case is because relevant pages are larger in size, and contain more information. As discussed in Section 4.1 however there were several drastic outliers. To verify the observed result we run a bootstrap comparison of means, which determines with 99.9% confidence that the differences of the means is between 574.4263 and 8414.3144. Bootstrapping shows with greater than 99.95% confidence that the Relevant Load Time average is greater than the Not Relevant Load Time average. To confirm these results we use a permutation test, which unexpectedly only gives a 97.69% confidence that the mean of the first sample is greater than the mean of the second.

For the first time the permutation test contradicts the confidence of a bootstrap with test with greater than 99.95% confidence. The confidence given by the permutation test was high, but given our bootstrap results, we would expect it to be a good deal higher. I suspect this is because of the relatively large number of major outliers in the Relevant Load Times dataset, which impact bootstrapping more since it uses resampling. Not one Not Relevant Load took longer than 6.11 seconds, in contrast 18 Relevant Loads took longer than 6 seconds, 9 took longer than 20 seconds, and 5 took longer than 60 seconds. With resampling it is likely that some of these disproportionately large values are repeated in many of the sample distributions, inflating the mean difference. In the permutation test, each outlier will only appear once in the sample differences, and are equally likely to appear in either random sample, lessening their impact.

We can compare medians to get a better sense of the impact of the outliers had. We perform a bootstrap comparison of medians, which gives the following output: “Bootstrapping has determined with 0.98 confidence that the differences of the medians is between 12 and 566”. While the 99% confidence returned by the test

is high, it is lower than the mean comparison bootstrap, suggesting outliers might have made an impact. While the median bootstrapping test still gives a high enough confidence to conclude that Relevant Load Times are inherently greater than the Not Relevant Load Times, it suffers from the same inflation due to outliers as the bootstrap mean test because of the resampling, albeit less so since outliers affect means more than medians. We compare this to the permutation median test which determines with 93.98% confidence that the median of the Relevant Load Times is greater than the median of the Not Relevant Load Times. This confidence is significantly lower than the bootstrap confidence. To further bolster the claim that outliers had a large impact on the results of resampling tests. These tests yielded much smaller degrees of confidence, and clearly there is a discrepancy between the tests.

The difference in test results suggests that since outliers were prevalent in one of the datasets, resampling techniques including the bootstrap overestimate the degree of confidence we can have that the mean and median Relevant Load Times are greater than the mean and median Not Relevant Load Times. As discussed in Section 4.1, I am skeptical that the Load Times recorded accurately reflect real world data. While most of the tests we have been relying on conclude with a high degree of confidence that the load times are different, given the significant discrepancy between some of those tests, the prevalence of drastic outliers, and the other doubts this data raised we cannot conclude that this data supports Load Times being a good predictor of page relevance. It might support this claim, but further research must be done.

The interpretation and analysis of the Average Load Times was very similar to that of the Total Load times, and we concluded that given our current data we

cannot determine whether load times are a good predictor of web page relevancy. See Figure 4.3.1 for the results of the statistical tests.

4.3.7 *Scrolls*

Lastly we consider Relevant and not Relevant Scrolls. This was the only dataset in which there was a maximum possible value for elements in the set, and the only dataset in which there were no outliers. The observed mean demonstrated that in our sample relevant pages were scrolled on average about 12% more than not relevant pages. To see how significant this is we first run the bootstrap means comparison, which with 0.94 confidence determined the differences of the means to be between 0.00241225 and 0.23781113. Hence we can conclude with 97% confidence that the mean of our Relevant Scrolls is greater than the mean of our Not Relevant Scrolls. We verify this result using permutation mean test, which output the following which found with 94.72% confidence that the mean of the Relevant Scrolls is greater than the mean of the Not Relevant Scrolls. The confidence given by the permutation test is about 94.72%, lower than the confidence given by our bootstrap, but still high enough to deduce the means of the two datasets are different.

Interestingly, the observed difference between the medians of the scrolls was much larger than the observed difference between the means, this is likely because a significantly larger proportion of Not Relevant Scrolls were zero. We perform the bootstrap comparison for medians, and find with 0.93 confidence that the differences of the medians is between 0.01264980 and 0.63508916. The bootstrapping median test yields a 96.5% confidence our medians are different. Ironically the degree of confidence we have in our median difference is not greater than that of our mean difference despite our observed difference being much greater, however it is not really worse either, it is about the same. We verify the bootstrap median comparison with a permutation

median comparison, which finds with 96.547% confidence that the median of the first sample is greater than the mean of the second”. This is almost exactly the same as the confidence given by the bootstrap test. While the confidence results our tests gave were not as strong as the confidence given for several other variables, the general consistency of the tests, and their acceptably high confidence values lead us to conclude that Scroll Percentage is likely a predictor of web page relevancy.

4.4 Comparison of Possible Predictors

	Observed difference in means	Bootstrap mean confidence	Bootstrap mean interval	Permutation mean confidence	Observed difference in medians	Bootstrap median confidence	Bootstrap median interval	Permutation median confidence	predictor of relevance ?
Visits	1.82003	>99.95%	0.8466 - 2.7927	99.9987%	1	99%	0.5 - 2	>99.99999%	data strongly supports
Visit Times	192,562	>99.95%	102453 - 312464	>99.99999%	70,174	>99.95%	36402 - 110022	>99.99999%	data very strongly supports
Average Visit Times	54,210	>99.95%	18920 - 92655	99.991%	39,415	>99.95%	7456 - 48321	99.998%	data strongly supports
Idles	0.494398	>99.95%	0.06582 - 1.0063	99.23%	0	>99.95%	0 - 0	n/a	data supports
Idle Times	17,995	>99.95%	8029 - 31041	99.97%	0	>99.95%	0 - 0	n/a	data supports
Loads	0.27731	96.5%	0.01331 - 0.54062	87.73%	0	>99.95%	0 - 0	n/a	data does not support
Load Times	3,722	>99.95%	574.4 - 8414.3	97.69%	241	98%	12 - 566	93.98%	data might support
Average Load Times	3,408	>99.95%	430.2 - 7983.6	96.46%	25	78.5%	3 - 49.5	54.00%	data might support
Scrolls	0.121517	97%	0.00241 - 0.23781	94.72%	0.375616	96.5%	0.01265 - 0.63509	96.55%	data supports

Figure 4.4.1. Results of Statistical Testing

To compare which variables appear to be the best predictors, we compare the confidence our statistical tests yielded. The testing done on our experimental data confirmed the hypothesis, that Visit Times is the best predictor of web page relevance. There were other variables that the data supported as also being predictors,

although not as strongly. These were Visits, Average Visit Time, and Scrolls. There was data to support Idles and Idle Times as being predictors, however there were doubts cast on the validity of a small portion of this data (discussed in Section 4.1) and the difference was not confirmed by median testing, therefore we are not as confident Idles and Idle Times are predictors, however the data still supports the conclusion that they are. Load Times and Average Load Times were greatly affected by outliers, and while there is sufficient evidence to conclude that they are likely predictors, the doubts cast by the data and the discrepancy in test results make us reluctant to do so at this time, and further testing must be done. Lastly, while there was some evidence that Loads was a predictor of page relevance, the confidence given by our testing was not strong enough to support this claim.

5

Conclusion

5.1 Summary

This paper presented software which visualized a user's browsing history as a 3D directed graph. Each node represented a page that was browsed, and edges represented some kind of navigation between pages (see Section 2.2.2 for further details). The visualization displayed a variety of information about how a selected web page was browsed on a side panel. This information included the URL and Title of a page, Visits (amount of separate times a page was displayed), Visit Times (when each visit started and ended), Idle Times (each time an Idle started and ended), Load Times (each time a page load began and ended), Refreshes (when a page was refreshed), and Scroll Percentages (amount page was scrolled).

While there were shortcomings in the visualization, the data collection software which served as its underpinning was used for research which aimed to determine which aspects of a user's browsing history are most likely to determine web page relevance. This research used data collected from a three hour browsing session

involving 14 participants to learn what variables most likely determine web page relevance.

We found that several of the data points being collected page relevancy with higher than 95% confidence. We confirmed our hypothesis, and found that Visit Times are a predictor of relevancy with a higher degree of confidence than any other variable. We also concluded with greater than 99% confidence that Visits, and Average Visit Time are predictors of page relevance. We are also more than 95% confident that Scroll Percentages predict relevancy. While testing found with more than 99% confidence that Idles and Idle Times are relevancy predictors, this was not confirmed by comparison of medians testing, and there was some doubt about the data being fully intact. We still can conclude Idles and Idle Times are likely predictors, however we are not as convinced. For Load Times and Average Load Times the data might support a difference, however the difference observed was largely affected by drastic outliers. Given this, and the doubts that the data collected accurately reflects real world data (discussed in Section 4.1), we cannot confidently determine whether this data supports Load Times and Average Load Times as predictors of web page relevancy. While there was some evidence Loads are predictors of page relevance, the data did not support this at the 95% confidence level and we cannot conclude it is a likely predictor. Lastly, there was no evidence that refreshes are predictors of page relevance.

5.2 Limitations of Current Software

The main limitation to the current implementations of this software is the lack of long term storage for a users' history. Because of the limitations of the Chrome API, which allows a maximum of 5MB to be stored locally for use by the extension, larger datasets cannot be stored locally long term. The amount of data used to store 14

three hour browsing sessions for the experiment outlined below was over 5MB, so while it would take a good amount of browsing to reach this quota for an individual user, it is a limit that will be reached if the extension is used for a longer period of time. A solution would be to have the information stored on a server, and retrieved when the browser is restarted. This is an change I intend to make, however when the focus of this project switched from being about developing the software to collecting data to support a hypothesis, this change was not deemed a priority.

In the current implementation there are also small issues impacting the accuracy of the data collected. The most easily fixable is the absence of code in the window closing listener that would accurately update the most recent Visit Time. There are also issues with the current way Idle Times are collected. This functionality was not as well tested as other aspects of the data collection, and the unreliability of the “onIdle” listener in the Chrome Extension API was not realized until after the experiment was conducted. The errors in the data collected suggests that this event is not always fired immediately when a user returns from being an idle state. A suggested fix for this, which would require further testing, would be to add an additional reference for the page that most recently was idled on, and when a user returns from being idle use this reference to update the data in case the reference to the previously visited node has changed.

There are also limitations to the graph drawing algorithm, which often fails to draw a readable graph on larger more complex datasets. While it is true all graph drawing algorithms will ultimately fail to draw comprehensible graphs on arbitrary datasets [24], the graph drawing algorithm used in the visualization should be capable of drawing readable graphs for more datasets than it currently can. One of the main limitations in the current implementation is its lack of ability to sufficiently spread out across the 3D space. Right now it stays towards the center causing the

clustering of nodes, impairing readability (see Figure 2.4.1 for an example of this clustering). One useful feature that would at least enable users to simplify complex graphs, which has not yet been implemented, would be allowing a user to select a subset of their browsing data, perhaps chronologically, and omitting the rest of the data from the graph. This does not remedy the fact that the graph drawing algorithm does not scale well for larger datasets, and this problem will be addressed in future implementations of this software.

There are also parts of the program that could be implemented more efficiently. The “findNode” function in “collector.js” uses a linear search to determine whether a node already exists in our dataset. If the array storing the nodes were kept sorted, a faster logarithmic search could be used. The extra computational cost of maintaining a sorted array when a new node is inserted would not be significant, and since searches happen more often it is more important to optimize the data structure for searches rather than insertions. Since the dataset was limited to only one browsing session, the number of nodes that had to be searched was relatively small, and the inefficiency of the findNode function did not noticeably impact the performance of the program. However, if the history stored was from all the user’s previous browsing sessions this would likely have a significant impact on the efficiency of the program. This can be fixed fairly easily, and will be in future implementations.

The way in which the data structure containing the edges is sent to the server in the deployment version, and sent to “GraphBuilder.js” in the user version is not memory efficient. A reference to each node object is contained in every edge. When the edge list is converted to a string using the JSON.Stringify function, in every element of edge list the reference to each node gets replaced with text describing all the information contained in the node. Node’s are likely to appear more than once, making much of the data in the edge list redundant, and the memory inefficiency

is compounded by the fact that nodes that appear more often in the list of edges will themselves contain more information. A more efficient way to store the edges in anticipation of the data transfer would be to store the indices of the nodes in the array in which they are stored, instead of references to the nodes themselves. When the data is transferred, the array containing the nodes would be transferred separately from the edges, however since the information contained in each node is only sent once, it greatly reduces the amount of memory needed. It also does not require any additional overhead in recreating the edge list data structure, since accesses to arrays in JavaScript are a constant time operation, and the reference contained in the edges can be easily replaced with an array access given the index of the desired node. In the current implementations, since the dataset was only from one browsing session, and relatively small, the extra memory used did not noticeably impact performance, however future more robust implementations of this software will make the described change to improve memory efficiency.

Lastly, aside from the limit to the amount of memory that can be stored locally, there are a couple other limitations to the Chrome Extension API. There is no event listener for when Google Chrome is being quit [14]. This means if a user quits Google Chrome, data loss is inevitable, since the information generated between the last post to the server and the quit cannot be stored permanently before Google Chrome quits. One way to mitigate data loss would be to post data to the server more often, however this has its own drawbacks, and I see no way to eliminate data loss entirely. Fortunately for the purposes of this project data loss was avoided. Users of the software during the experiment were instructed not to quit Chrome, but to instead close the window, which on Mac OSX 10.7.5 does not quit the program [18]. It was verified by the session information posted to the server that no data loss occurred due to a premature quit. Also, there is no Google Chrome Extension

API for creating toolbars, so the toolbar used in order to ask users if a page was relevant in the deployment version of the extension had to be loaded and rendered every time a page was loaded. This causes a slight slowdown in the loading of pages, however the degree of this slowdown is difficult to test, likely insignificant, and given the somewhat unpredictable nature of loading times for web pages it is not likely to be noticed by the user. If this experiment were to be done again on a larger scale, the toolbar would have to be implemented in the same way. The only action that could be taken would be to perform additional testing in order to account for the impact rendering the toolbar might have on our data.

5.3 Further Research and Discussion

Determining web page relevancy or importance has been the topic of much research. There does not appear to have been research conducted where browsing data was used to make this determination. In comparison to other types of data which has been used to determine web page relevancy, such as information from the page itself, and information about links to a page, preliminary evidence suggests using browsing data might provide for a more accurate determination. It should be noted that this research does not propose any kind of model for using browsing data to predict web page relevancy, and it would be misleading to compare the confidence this study found about differences in datasets to the confidence found in studies which proposed and tested a model for predicting relevance. The high confidence this study found would suggest that there likely exists a good model for predicting relevance using this data, however until further research is done and a model is developed we cannot compare the findings of this study to that of studies which proposed a model for making this prediction, which comprise most of the studies which aimed to predict relevance discussed in Section 1.5.

There are different reasons page relevancy is of interest, and in some of the studies used as comparison it was evident that the relevancy of a page had a different meaning, and therefore results about the data being used to make this prediction are not comparable to our results. In other research, notably research regarding web searches, the notion of page relevancy was more similar to our notion in that it was what a user deemed relevant. It would therefore be apropos to conduct future research about using browsing data to make predictions about page relevancy for the use of web searches.

The main issues with this research have to do with the small sample size used. There were only 14 participants, who only browsed for three hours. In order to be more certain in the findings of this research the dataset would have to derive from a longer term study ideally with more participants. Research done by Obendorf et al. researched strategies regarding page revisitation using a dataset collected over a period of 52-192 days [16]. While no conclusions about the pages themselves were made, I would conjecture that pages revisited more often over a long period of time are more likely to be relevant than pages which are not revisited as often. This would mean that Loads, which in this short term study was not concluded to be a likely predictor of relevance, might have more significance in a longer term study. Furthermore the strategies of revisitation (back button, form submit, bookmark, ect.) studied [16] could also be significant in this regard. Also, a longer term study could provide sufficient data about Idle Times on not relevant pages to make a comparison; data this study lacked.

Having more data would also open the door to questions posed in Section 1.4 that could not feasibly be answered given the data collected for this study. This notably includes information about link relevancy, since we would expect there to be a larger subset of pages browsed by multiple users in a longer term, and more

expansive study. Further studies should also aim to use a more diverse population for their browsing data to better see if the results found in this study apply to the population at large. Also a smaller study using a more refined version of the data collection software could be used to better see what impact the errors in the data collected for this study had.

The results of this research substantiate the claim that browsing data can be used to predict web page relevancy, and support the need for a more expansive study to verify the claims made.

Bibliography

- [1] E. Ayers and J. Stasko, *Using graphic history in browsing the world wide web*, GVU Technical Report (1995).
- [2] Beazley D., *Python Essential Reference*, Peason Education, 2009.
- [3] Brin S. and Page L., *The Anatomy of a large-scale hypertextual web search engine*, Seventh International World-Wide Web Conference (1998).
- [4] Catledge L. and Pitkow J., *Characterizing browsing strategies in the world-wide Web*, Computer Networks and ISDN Systems **27** (April 1995), 1065–1073.
- [5] Chaffer J. and K. Swedberg, *Learning jQuery (4th Edition)*, Packt Publishing, 35 Livery Street Birmingham, UK, 2013.
- [6] Chihara L. and Hesterberg T., *Mathematical Statistics with Resampling and R*, John Wiley and Sons, Inc., Hoboken, New Jersey, 2011.
- [7] Craswell N., Hawking D., and Robertson S., *Effective site finding using link anchor information*, Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (2001), 250–257.
- [8] Dirksen J., *Learning Three.js: The JavaScript 3D Library for WebGL*, Packt Publishing, 35 Livery Street Birmingham, UK, 2013.
- [9] Gandhi R. et al., *Domain Name Based Visualization of Web Histories in a Zoomable User Interface*, Proceedings of the 11th International Workshop on Database and Expert Systems Applications (2000), 591.
- [10] Gortler S., *Foundations of 3D Computer Graphics*, MIT Press, Cambridge, Massachusetts, 2012.

- [11] Gourley D., B. Totty, M. Sayer, A. Aggarwal, and S. Reddy, *HTTP: The Definitive Guide*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2002.
- [12] Holdener III A., *Ajax: The Definitive Guide*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2008.
- [13] Matthijs N. and Radlinski D., *Personalizing web search using long term browsing history*, Proceedings of the fourth ACM international conference on Web search and data mining (2011), 25–34.
- [14] Moxon P., *Google Chrome Developer Tools*, SAPPROUK Limited, 2014.
- [15] Musciano C. and B. Kennedy, *HTML & XHTML: The Definitive Guide (6th Edition)*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2007.
- [16] Obendorf N. et al., *Web page revisitation revisited: implications of a long-term click-stream study of browser usage*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (2007), 597–606.
- [17] B. Pinkerton, *Finding what people want: experiences with the WebCrawler*, Proc. of 2nd international WWW Conference (1994).
- [18] Pogue D., *Mac OS X Lion: The Missing Manual*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2011.
- [19] Ray J., *My iMac (Mountain Lion Edition)*, Pearson Education, 2013.
- [20] Ruihua S. et al., *Microsoft research asia at web track and terabyte track of TREC 2004*, Proceedings of the Thirteenth Text REtrieval Conference (2004).
- [21] Brahim Sanou, *ICT Facts and Figures*, <http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013-e.pdf>. Accessed April 30th, 2014.
- [22] Soumen C., Punera P., and Subramanyam M., *Accelerated focused crawling through online relevance feedback*, Proceedings of the 11th international conference on World Wide Web (2002), 148–159.
- [23] Sriparasa S., *JavaScript and JSON Essentials*, Packt Publishing, 35 Livery Street Birmingham, UK, 2013.
- [24] Tamassia R., *Handbook of Graph Drawing and Visualization*, CRC Press, Boca Raton, FL, 2014.
- [25] Tao Q. et al., *A study of relevance propagation for web search*, Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval (2005), 408–419.
- [26] Tatroe K., *Programming PHP*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2013.
- [27] Wellin P., *Programming with Mathematica: An Introduction*, Cambridge University Press, Cambridge, UK, 2013.

- [28] *Top 5 Desktop Browsers from July 2008 to Jan 2014* — StatCounter Global Stats, <http://gs.statcounter.com/\#desktop-browser-ww-monthly-200807-201401>. Accessed April 30th, 2014.

Appendix A

Additional Results

The following are histograms for all datasets collected. For each variable there are four different histograms, one for the entire dataset, one for the pages which a user clicked on the relevant button, one for the pages which the user clicked on the not relevant button, and one for the pages in which a user did not click on either.

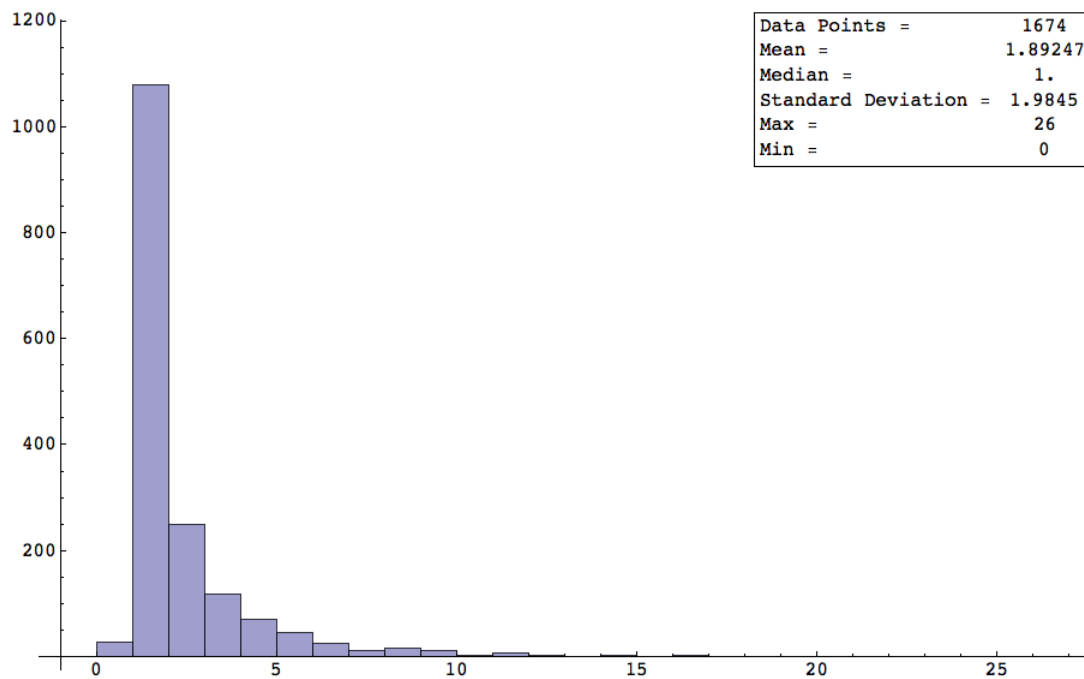


Figure A.0.1. Total Visits

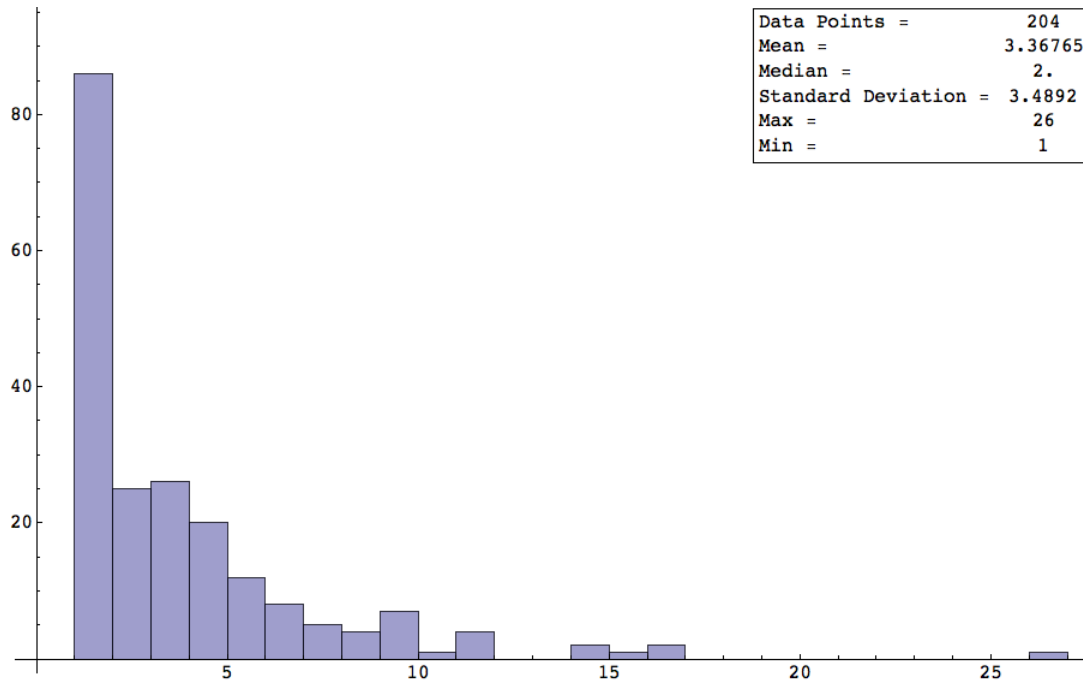


Figure A.0.2. Relevant Visits

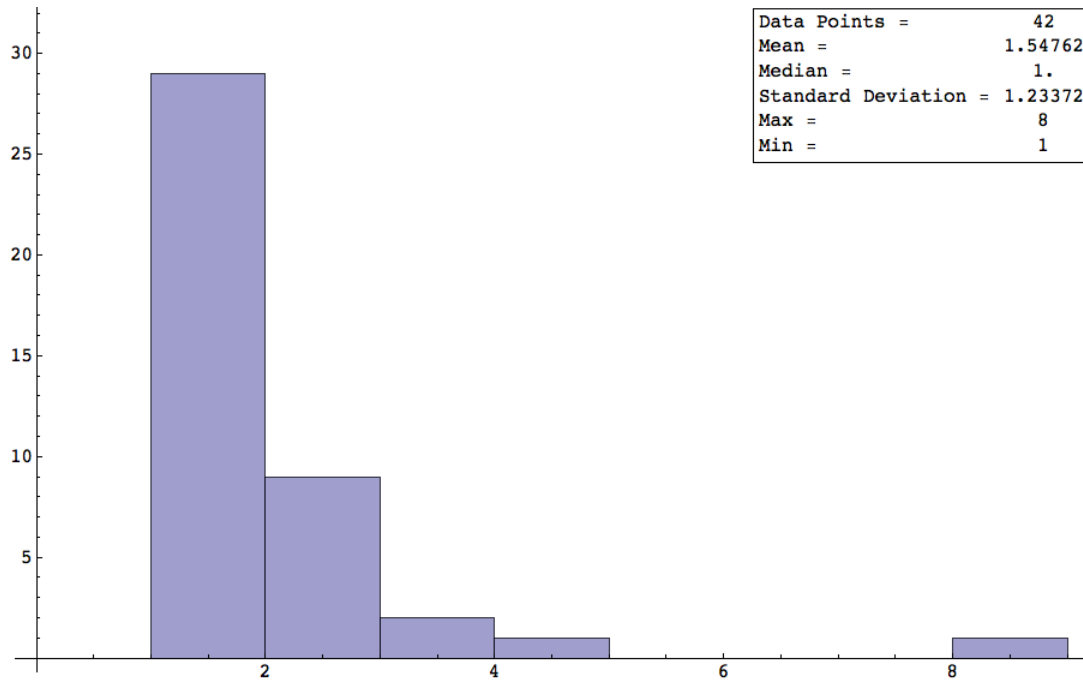


Figure A.0.3. Not Relevant Visits

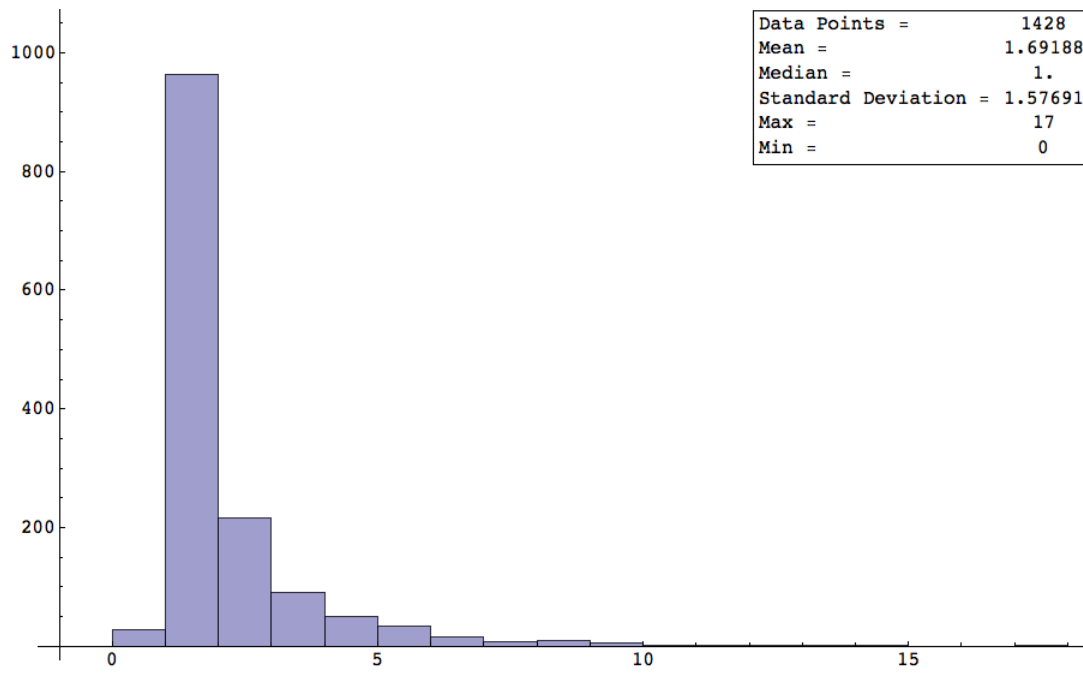


Figure A.0.4. No Response Visits

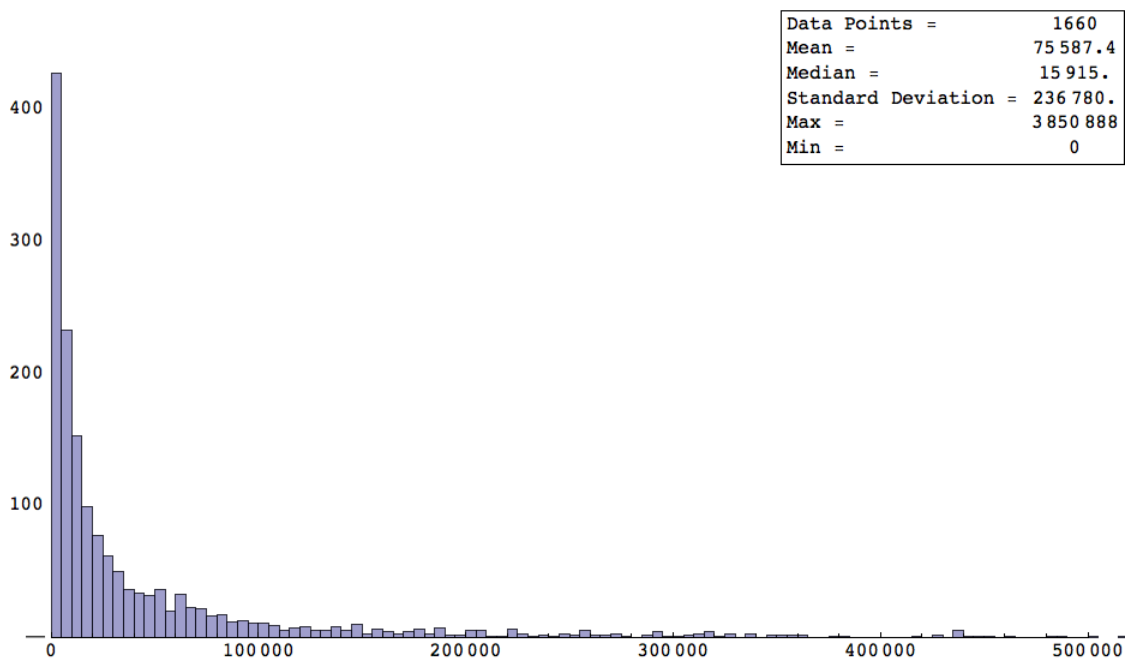


Figure A.0.5. Total Visit Time

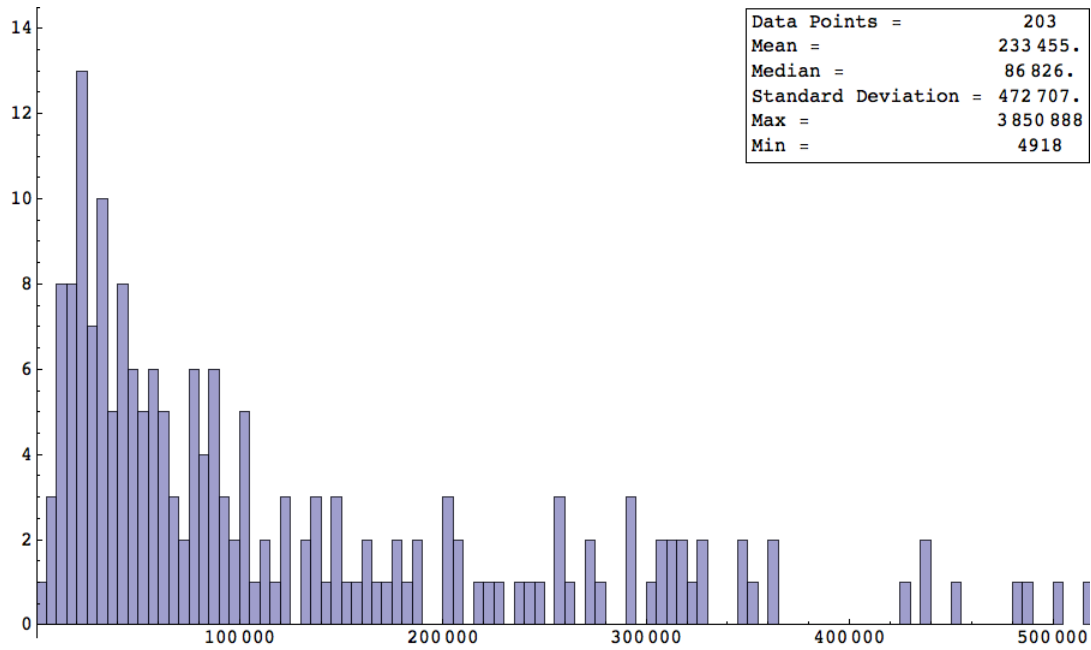


Figure A.0.6. Relevant Visit Time

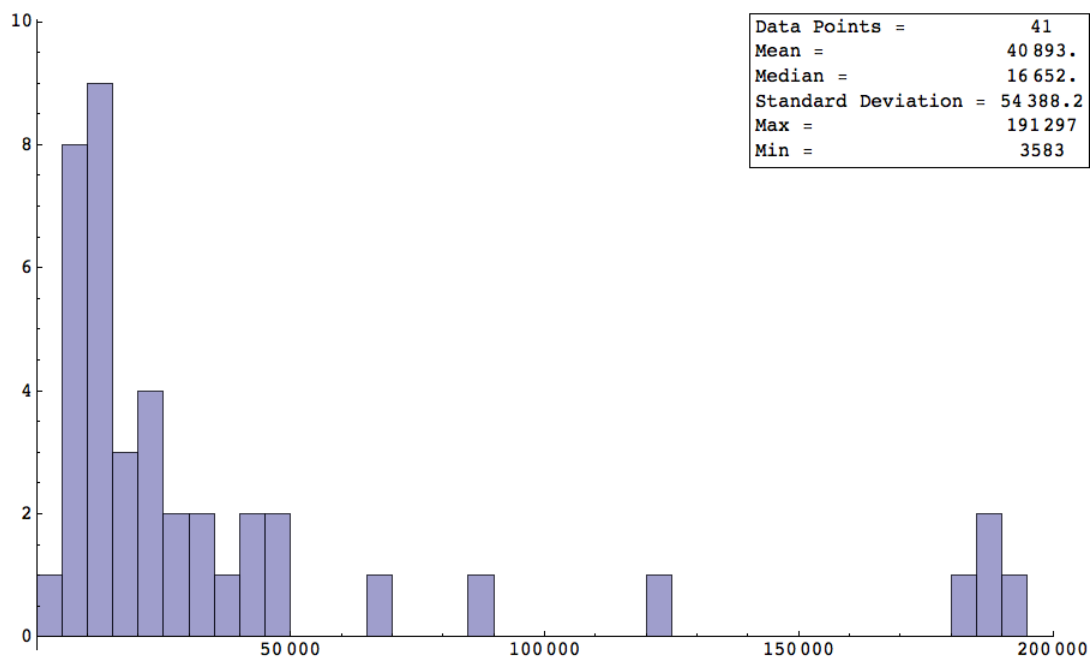


Figure A.0.7. Not Relevant Visit Time

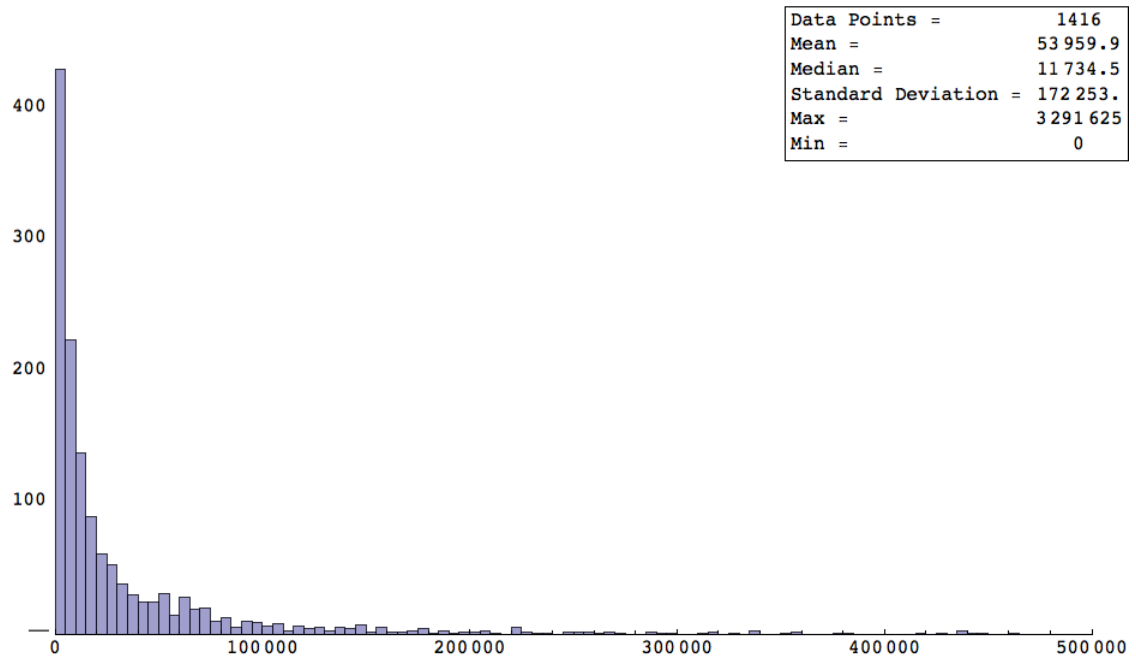


Figure A.0.8. No Response Visit Time

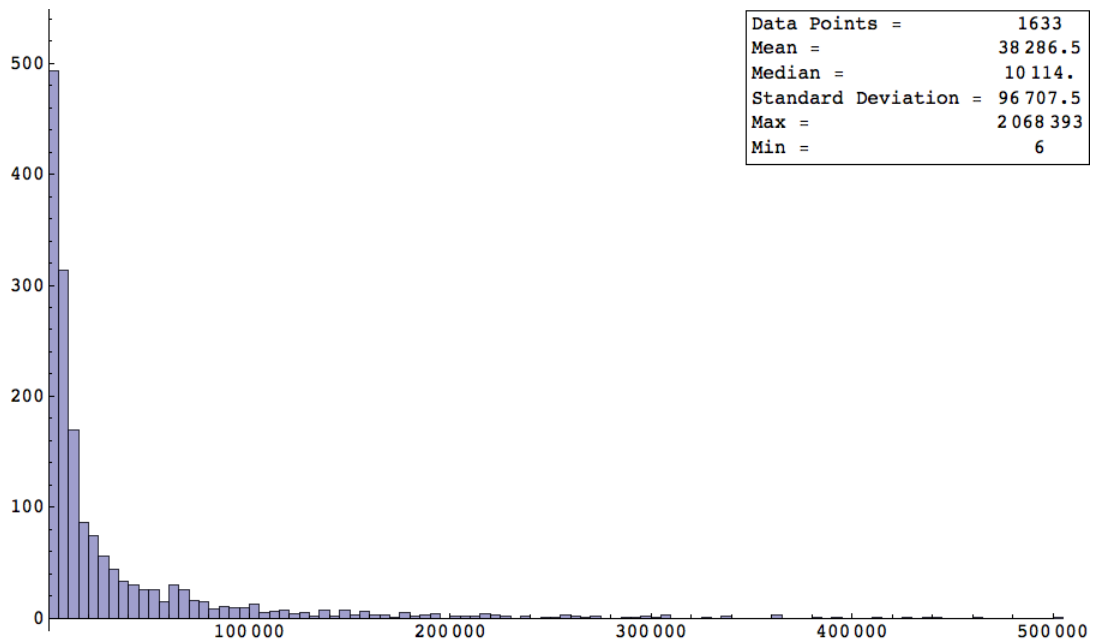


Figure A.0.9. Total Average Visit Time

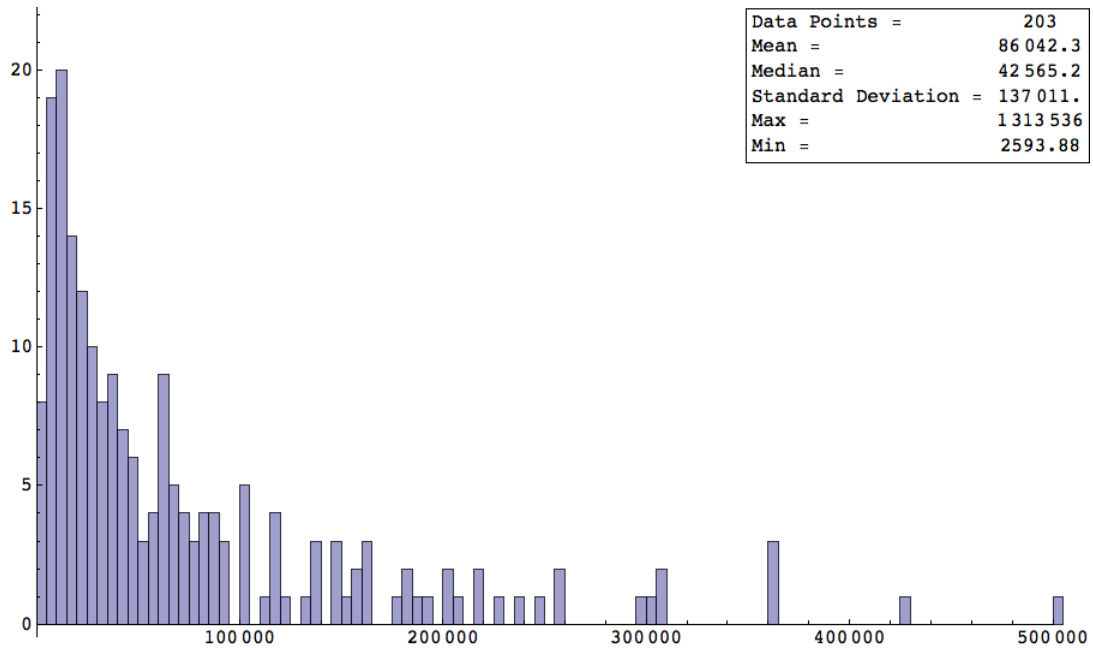


Figure A.0.10. Relevant Average Visit Time

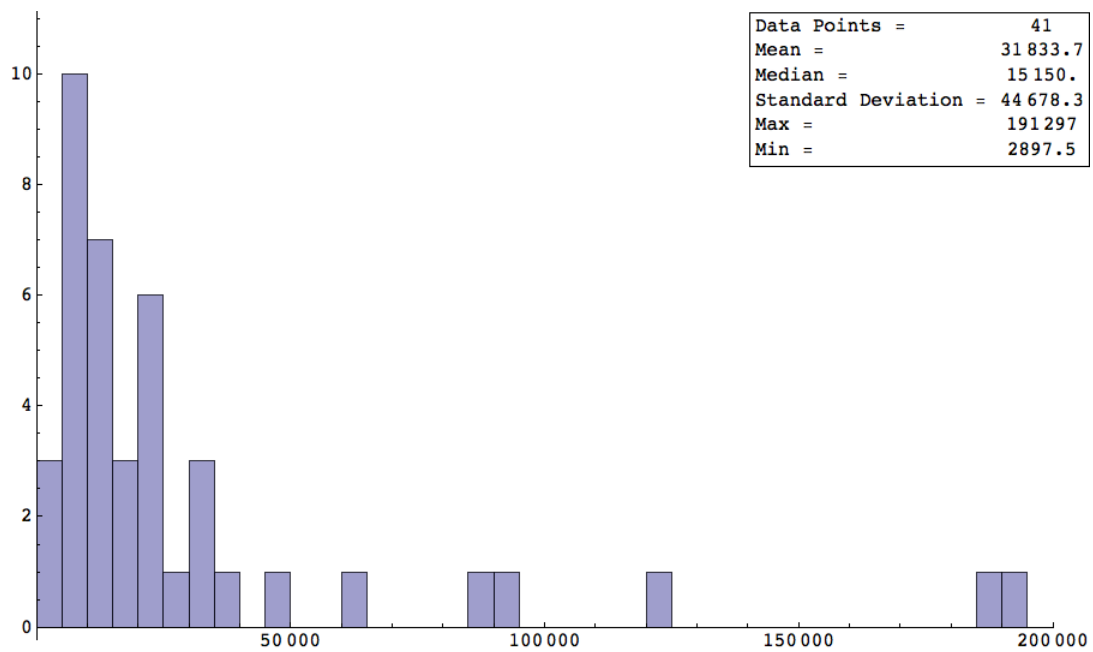


Figure A.0.11. Not Relevant Average Visit Time

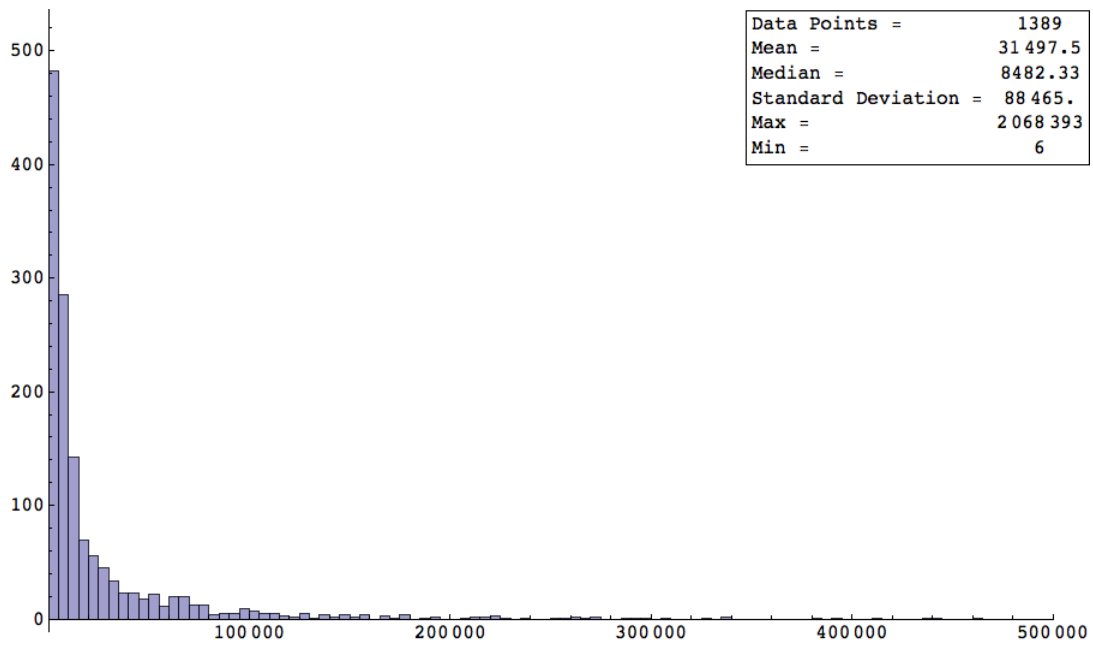


Figure A.0.12. No Response Average Visit Time

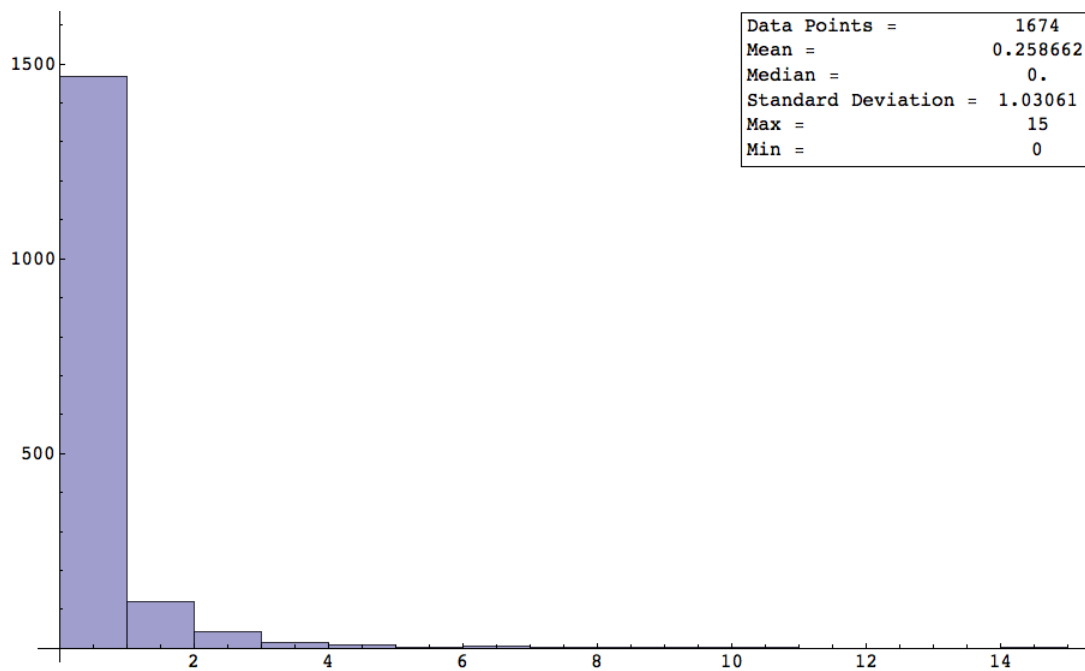


Figure A.0.13. Total Idles

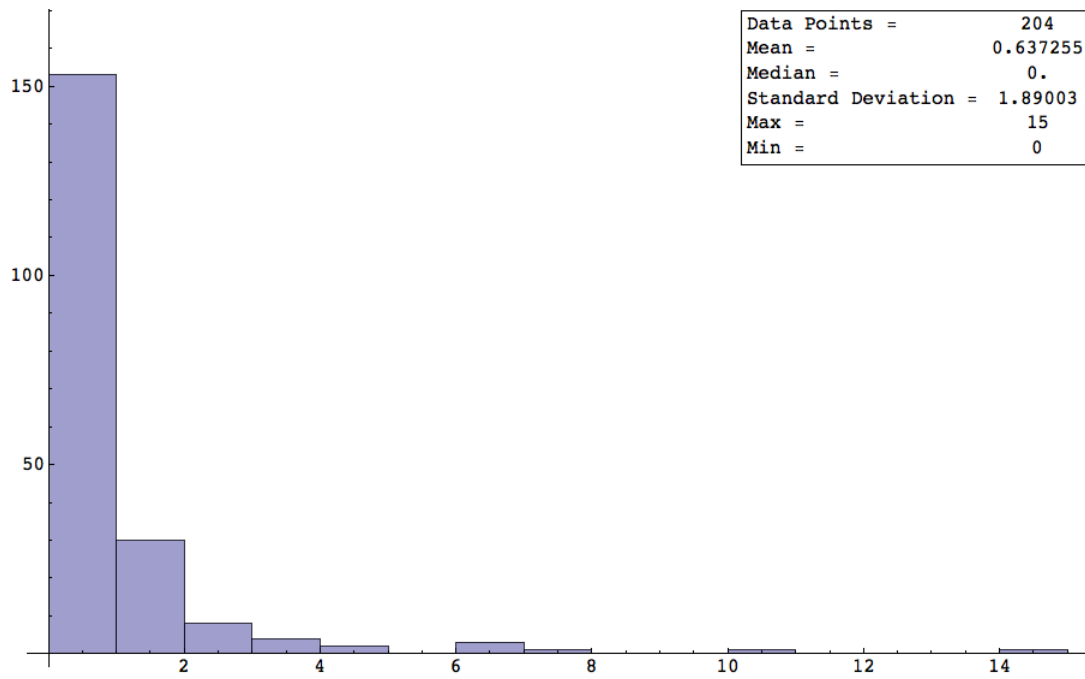


Figure A.0.14. Relevant Idles

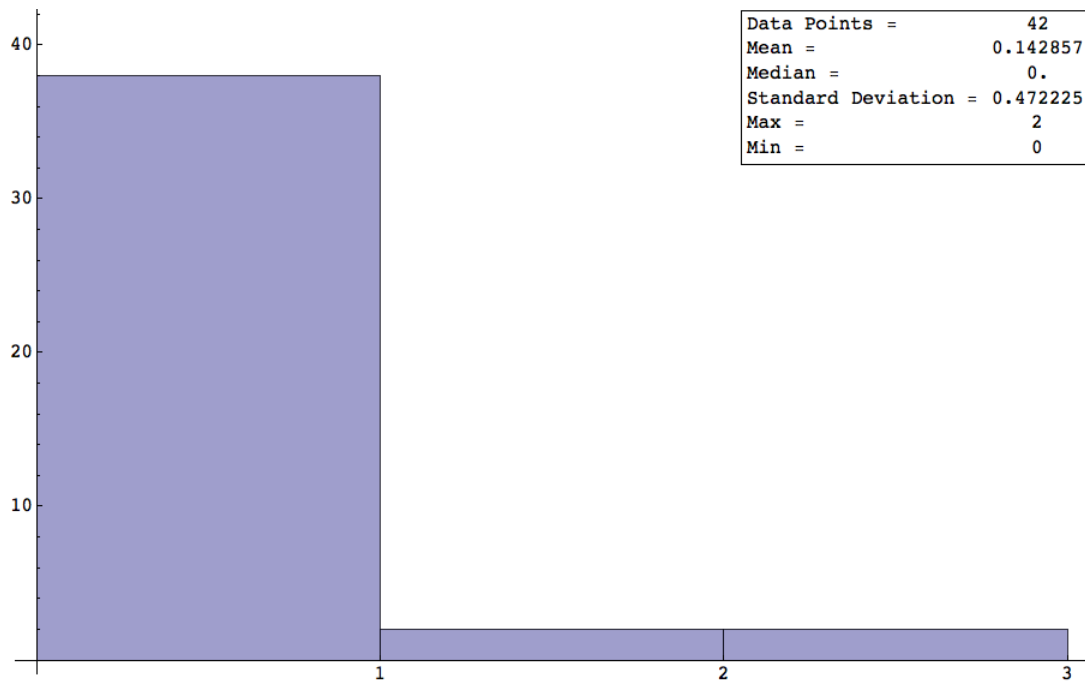


Figure A.0.15. Not Relevant Idles

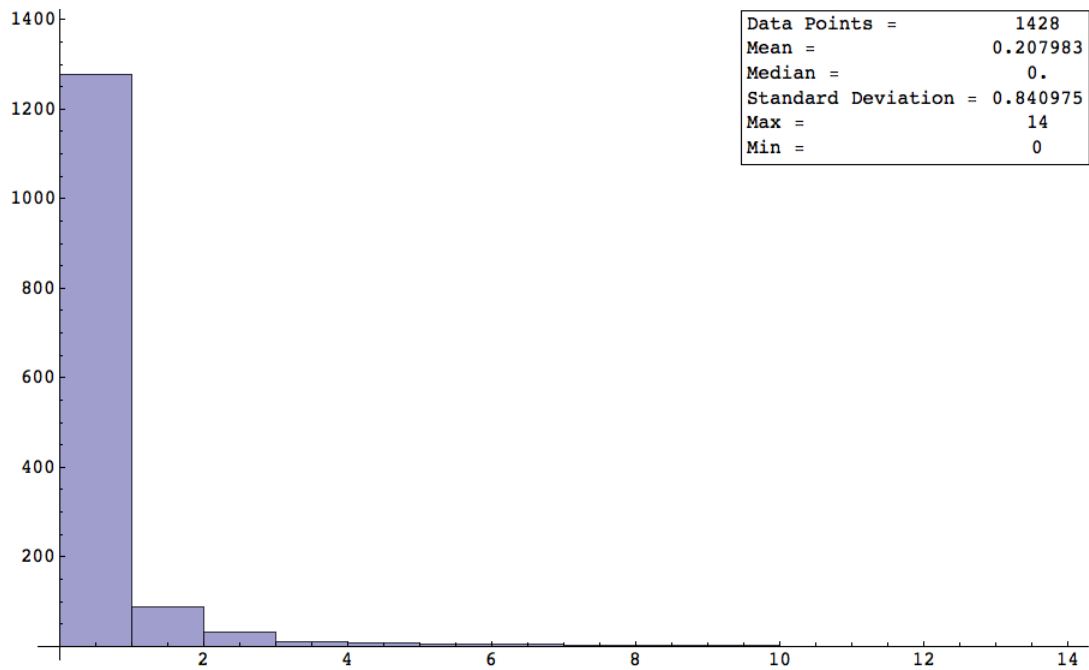


Figure A.0.16. No Response Idles

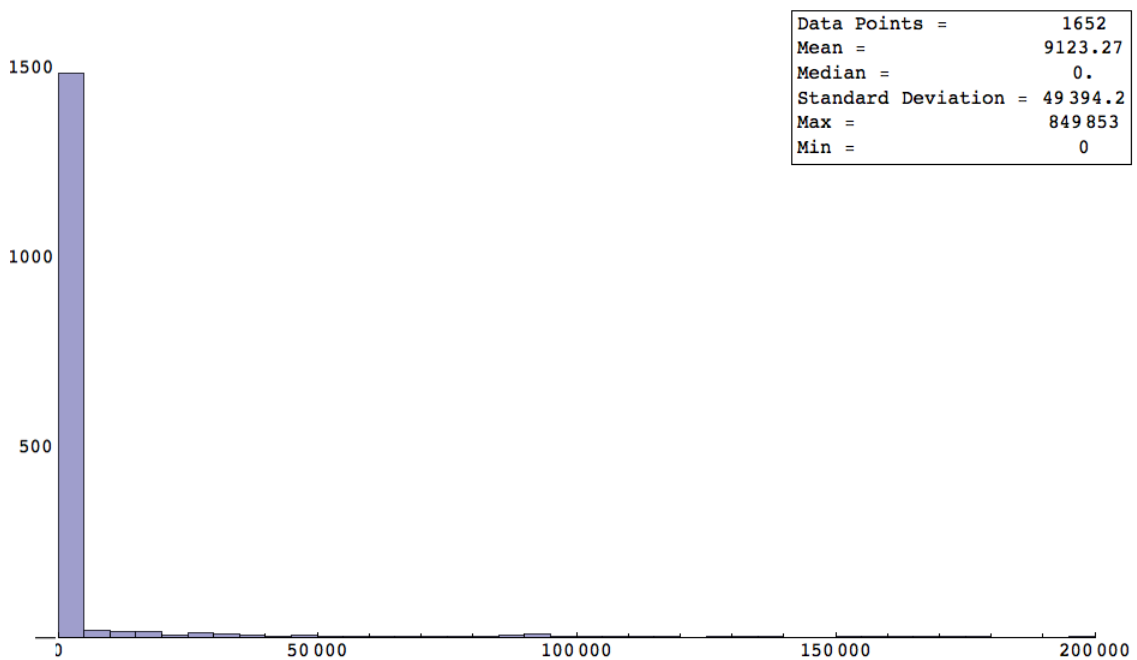


Figure A.0.17. Total Idle Times

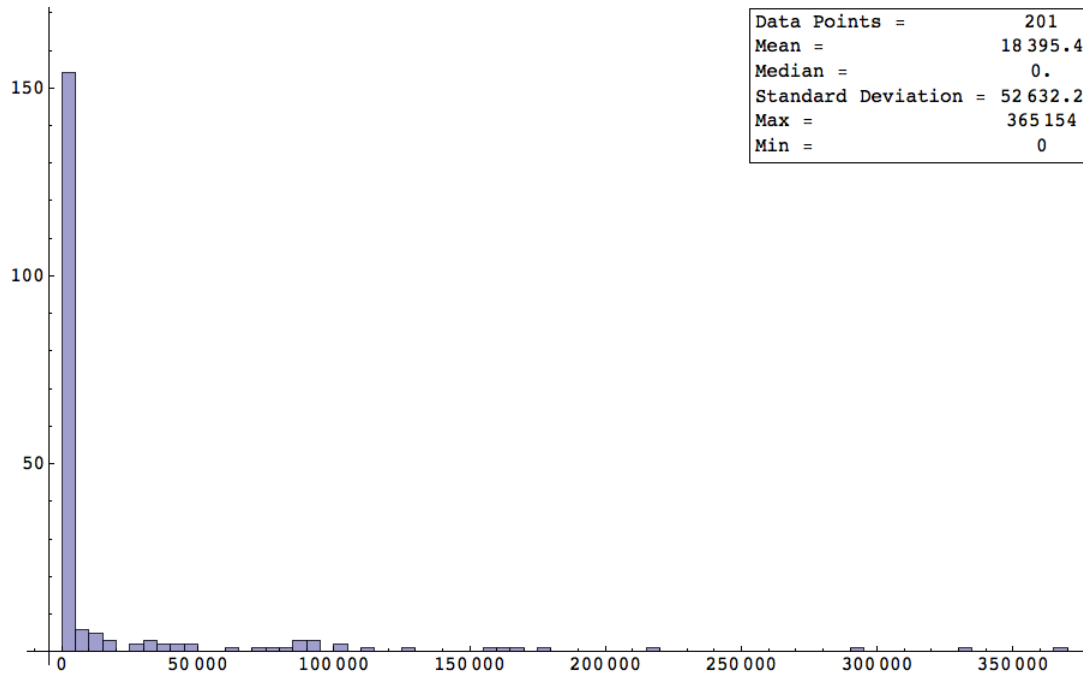


Figure A.0.18. Relevant Idle Times

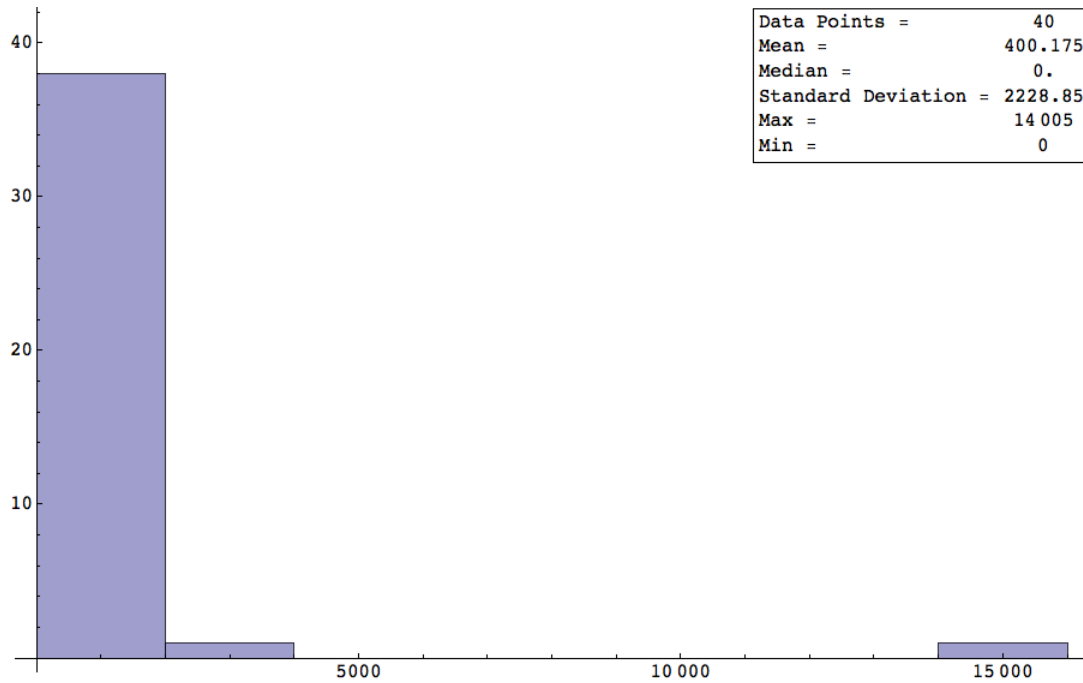


Figure A.0.19. Not Relevant Idle Times

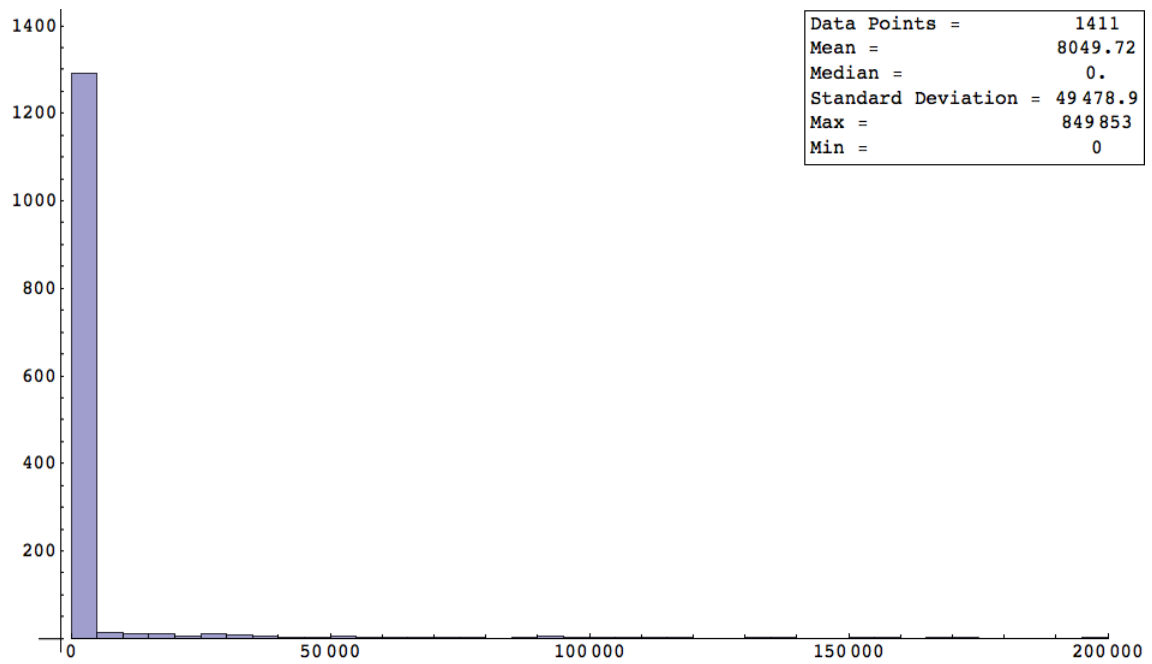


Figure A.0.20. No Response Idle Times

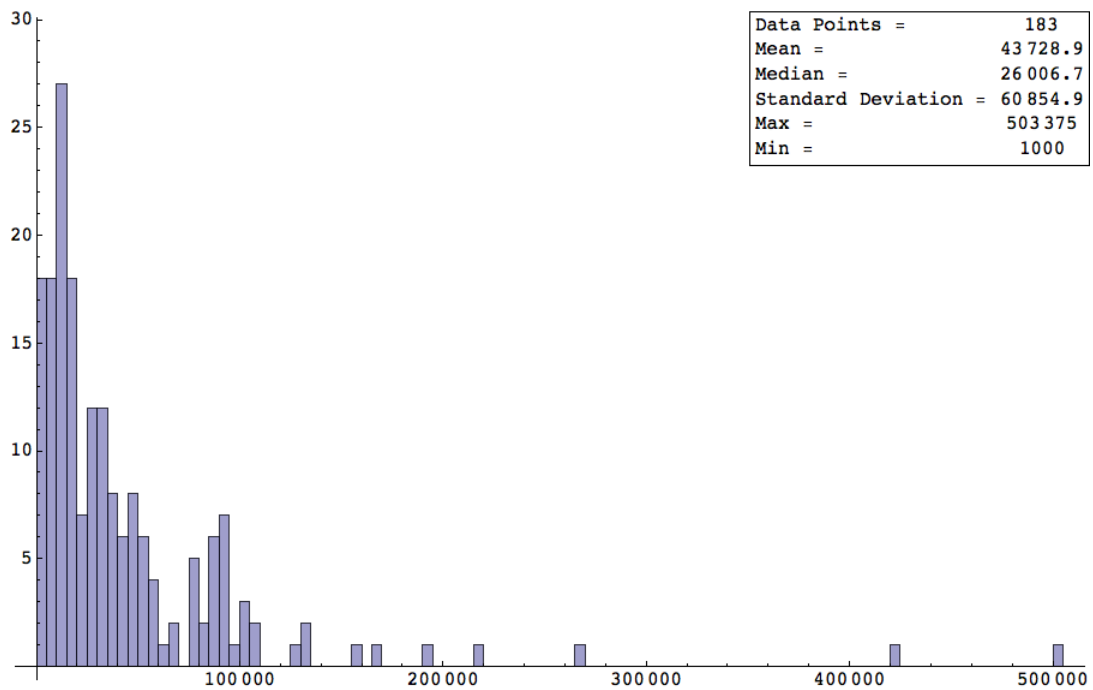


Figure A.0.21. Total Average Idle Times

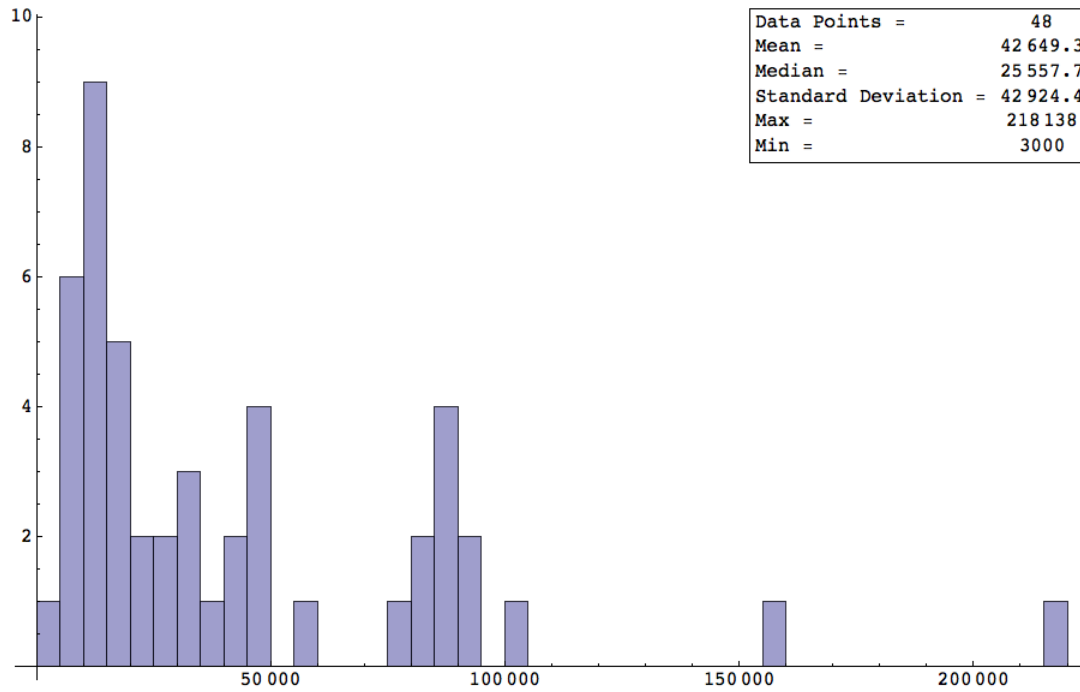


Figure A.0.22. Relevant Average Idle Times

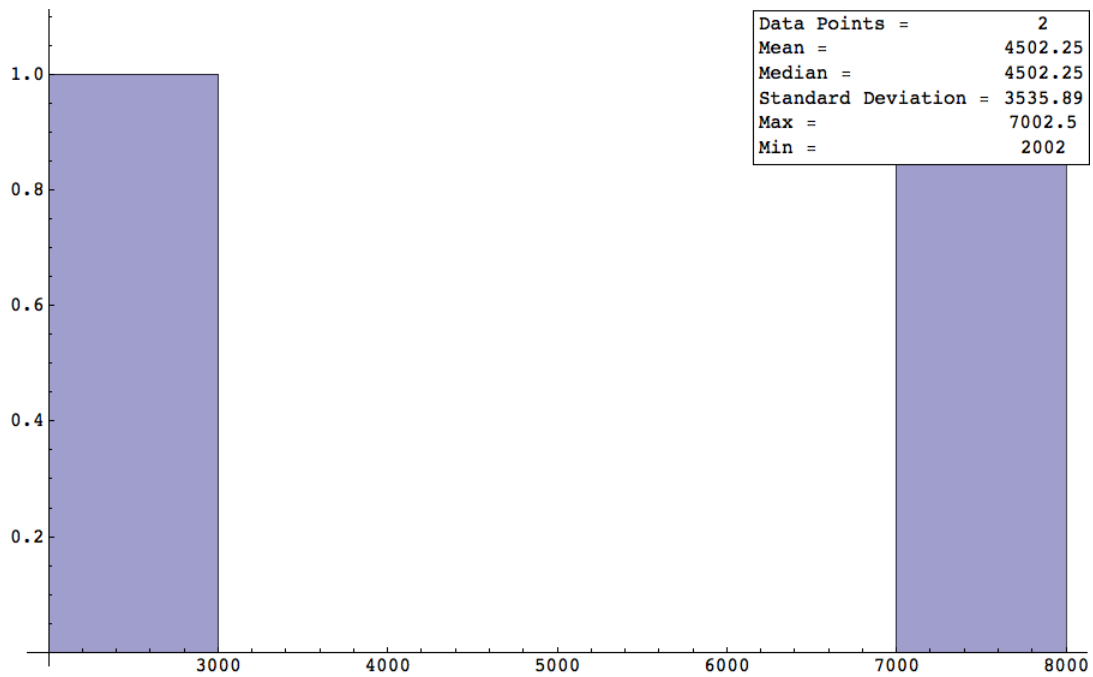


Figure A.0.23. Not Relevant Average Idle Times

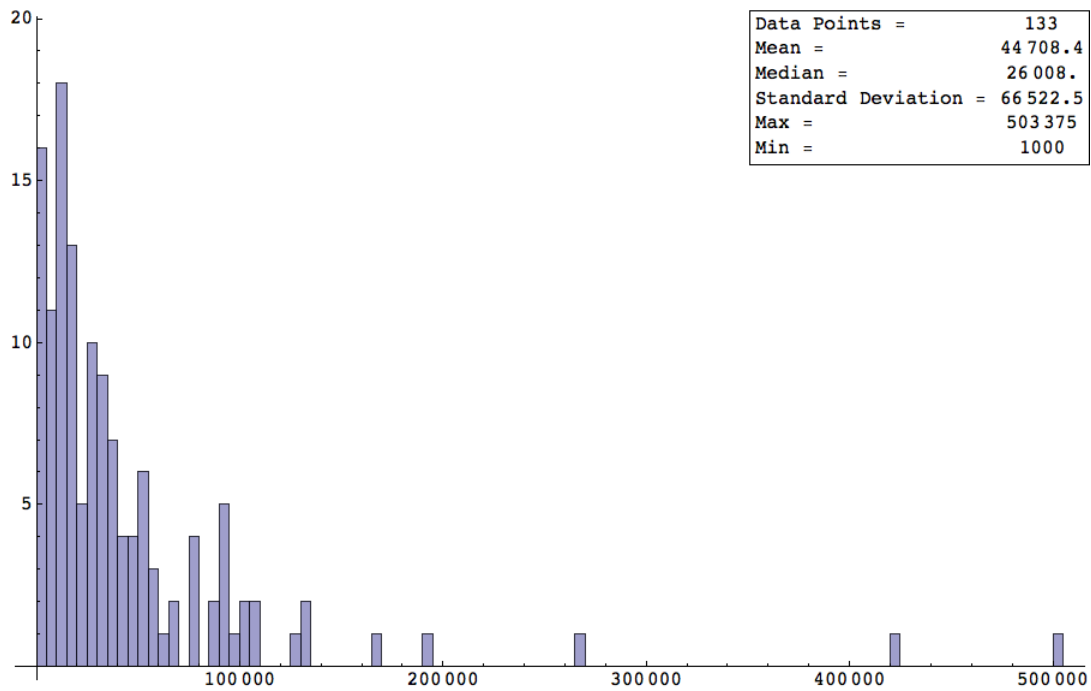


Figure A.0.24. No Response Average Idle Times

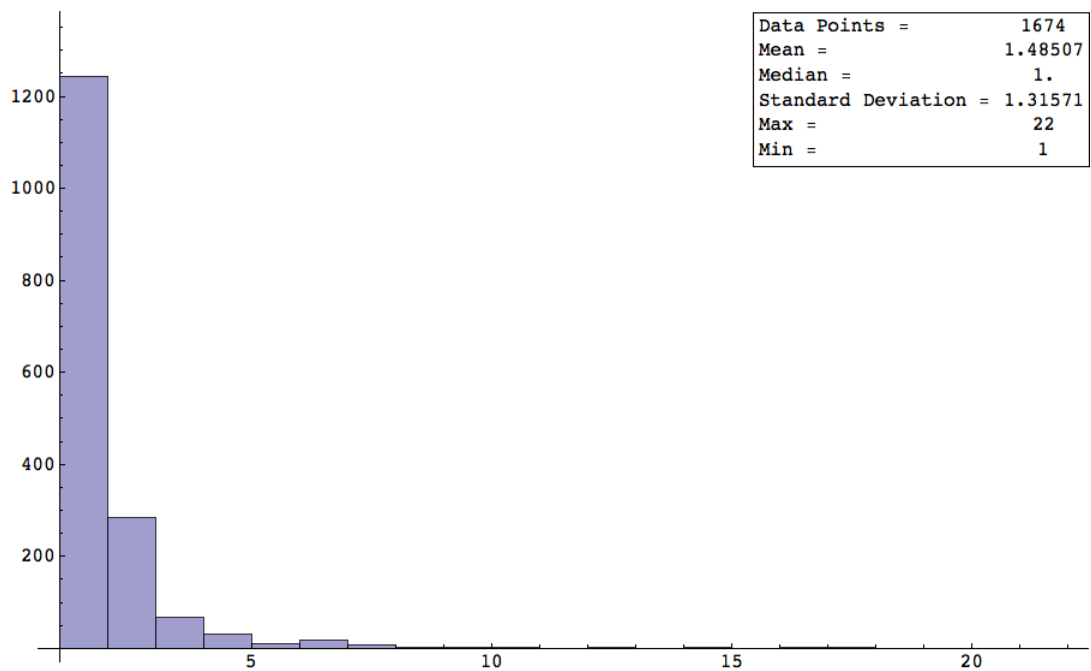


Figure A.0.25. Total Loads

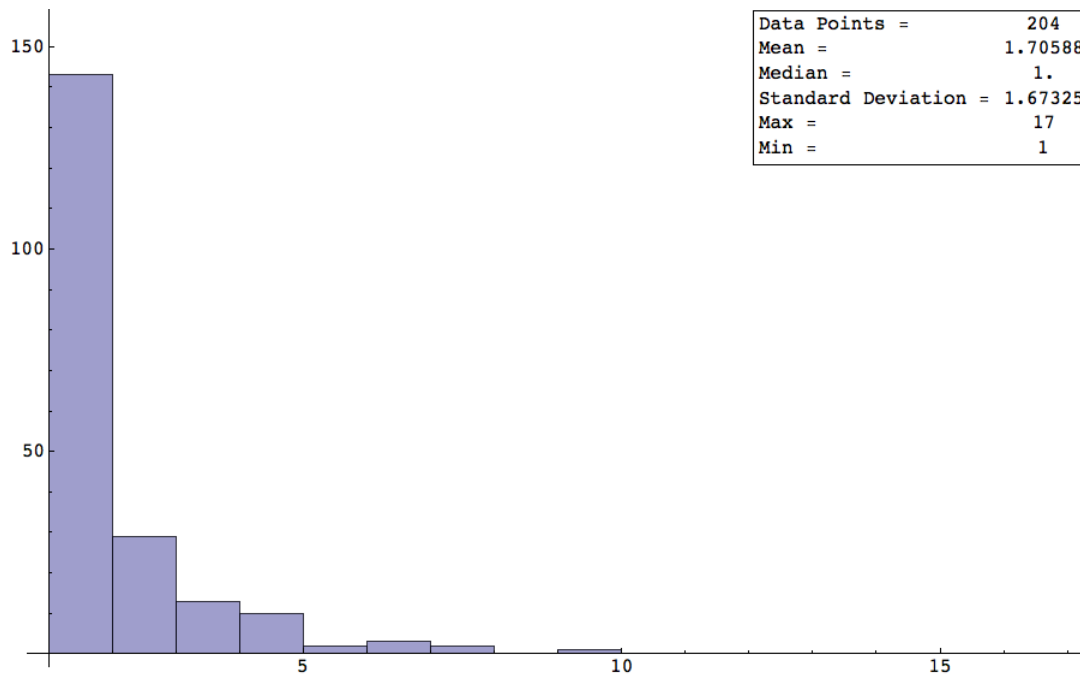


Figure A.0.26. Relevant Loads

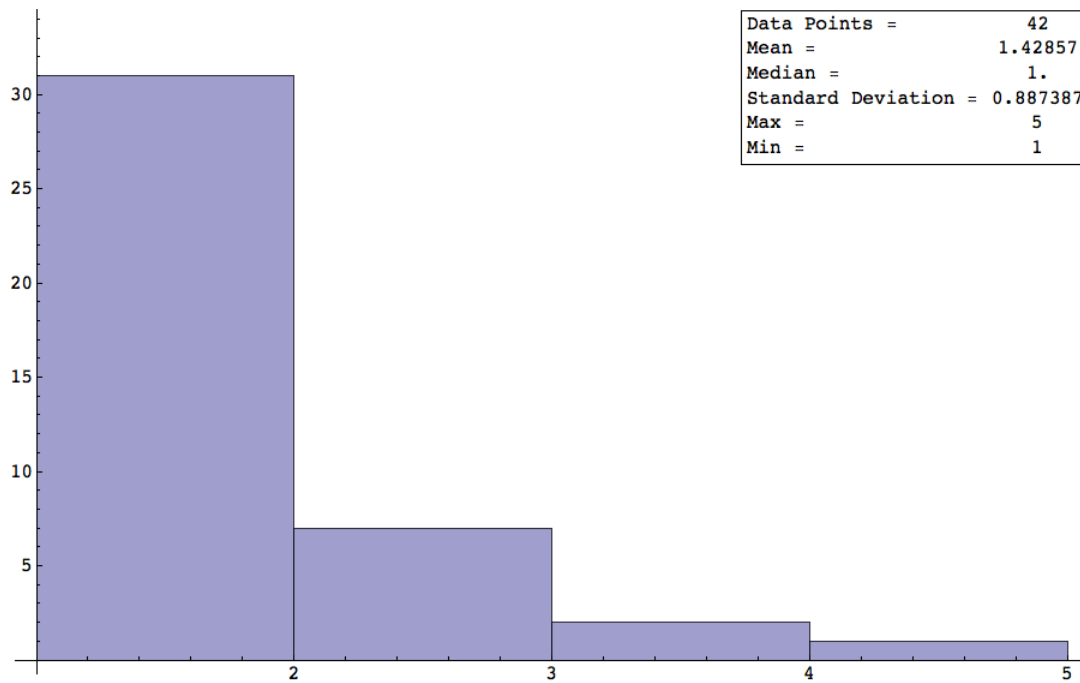


Figure A.0.27. Not Relevant Loads

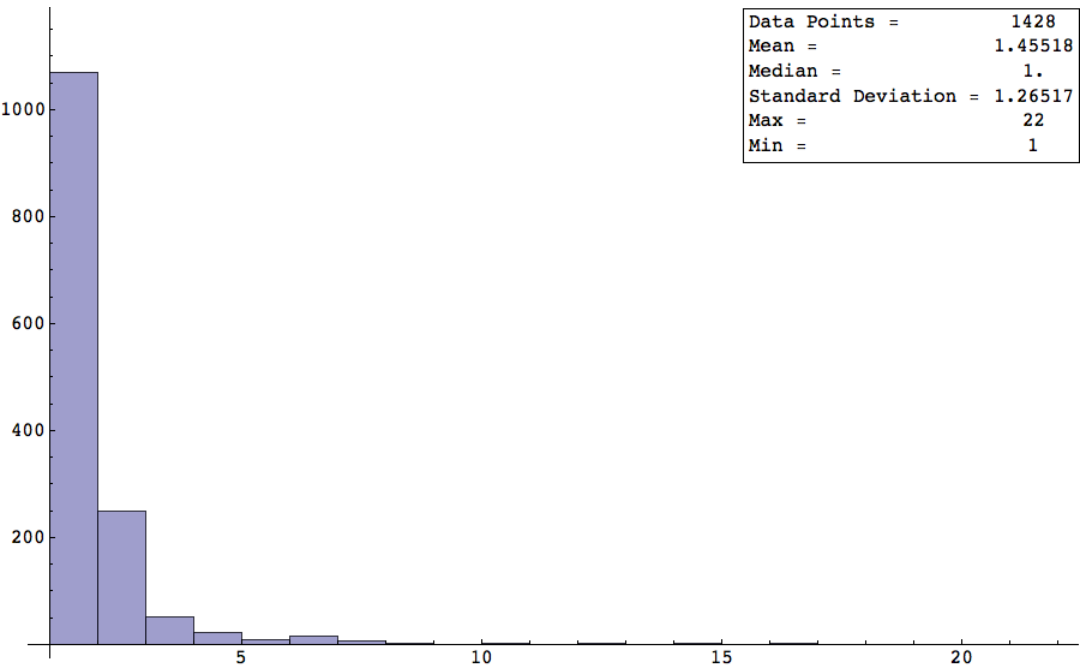


Figure A.0.28. No Response Loads

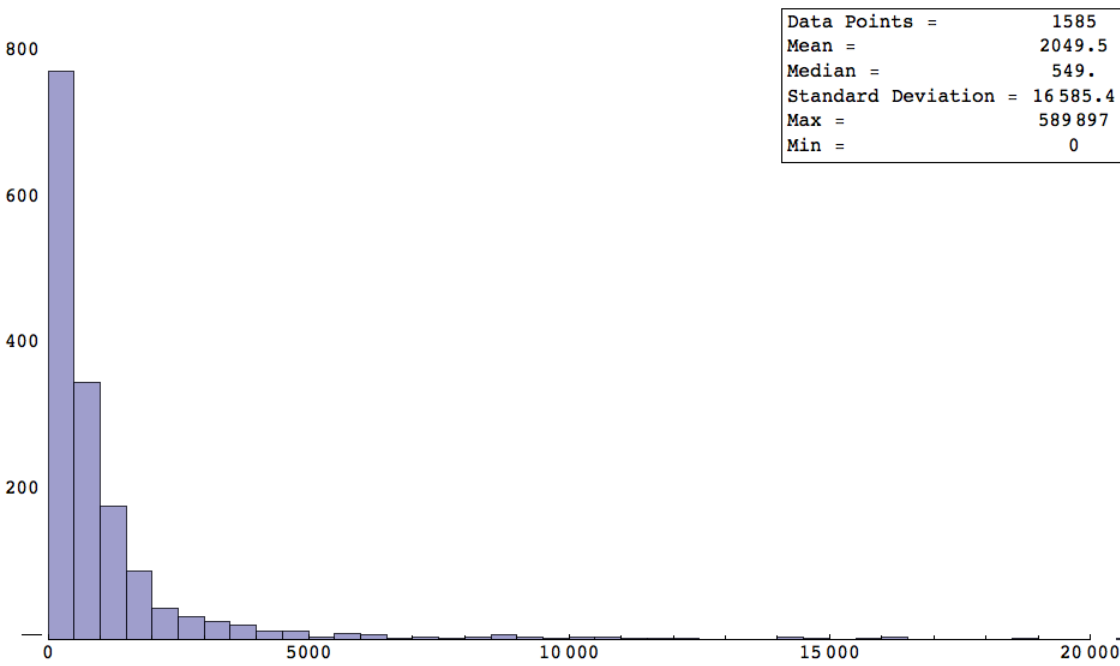


Figure A.0.29. Total Load Times

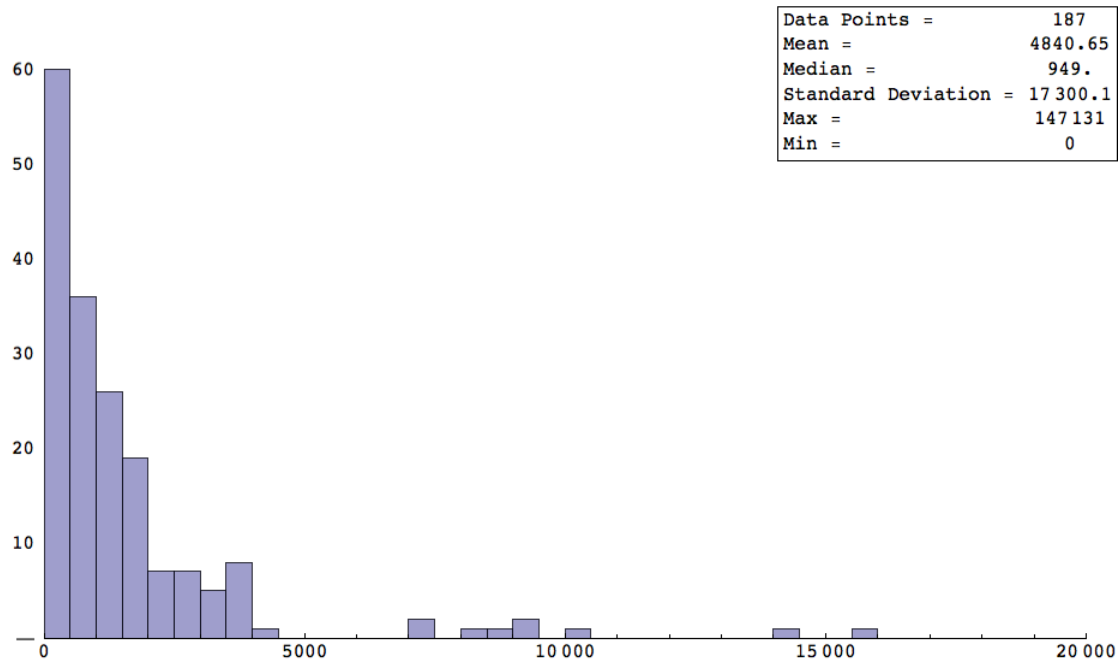


Figure A.0.30. Relevant Load Times

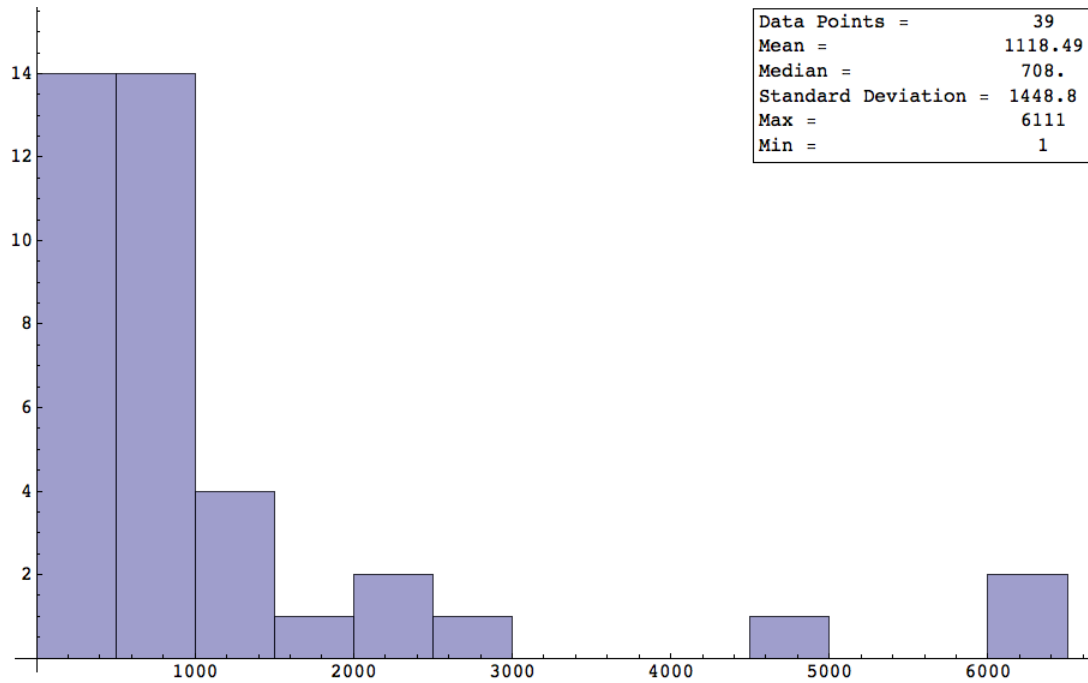


Figure A.0.31. Not Relevant Load Times

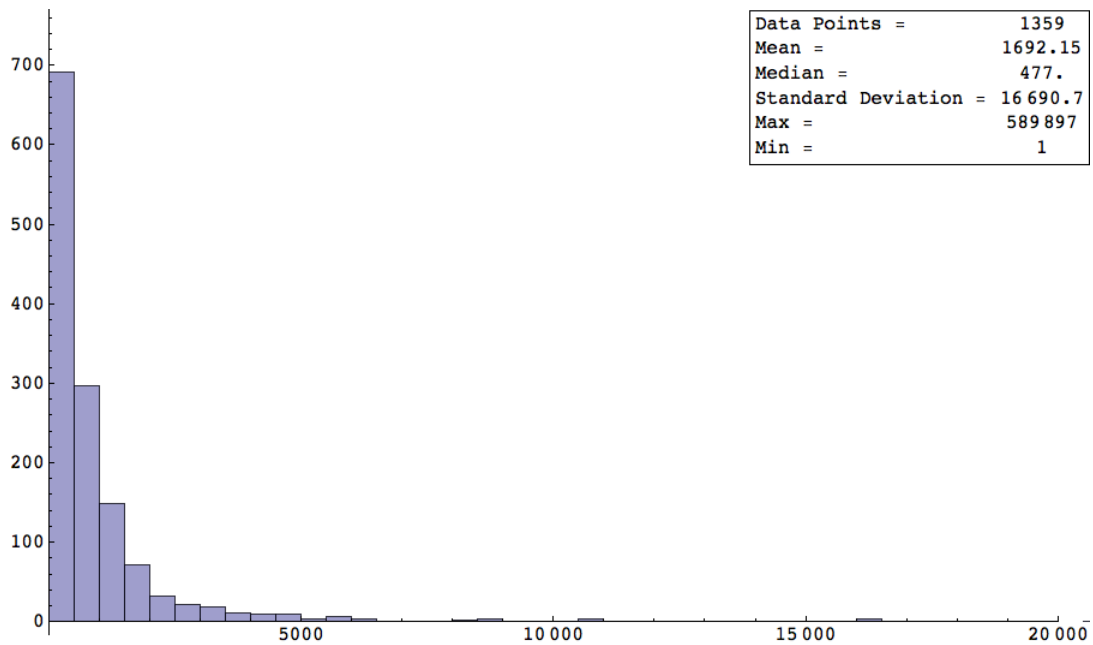


Figure A.0.32. No Response Load Times

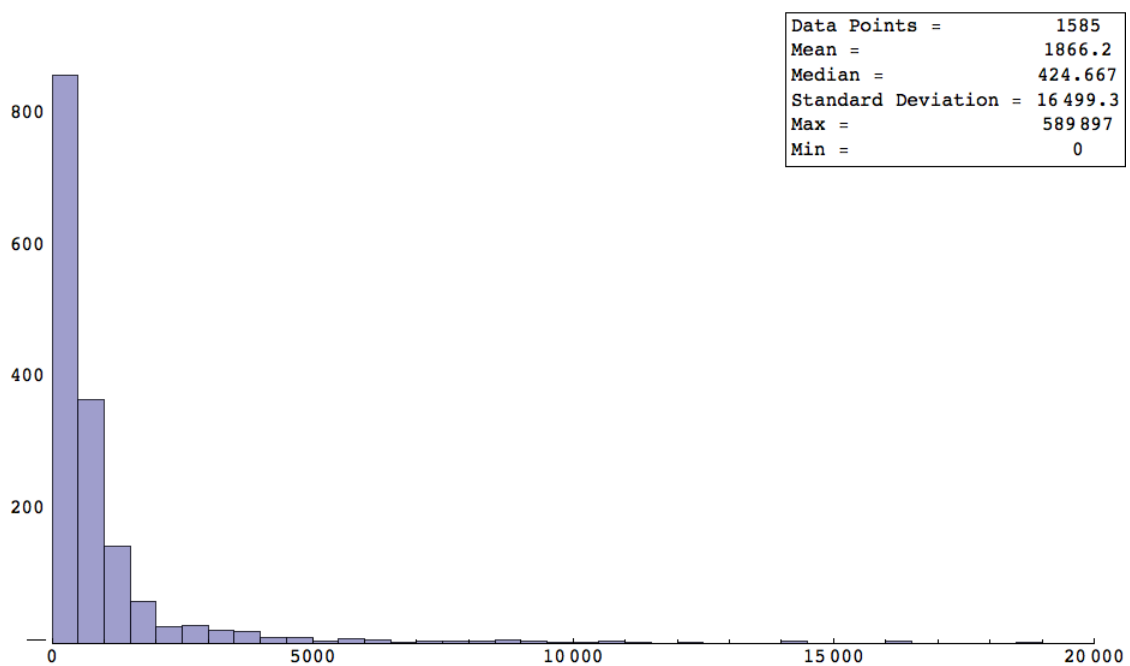


Figure A.0.33. Total Average Load Times

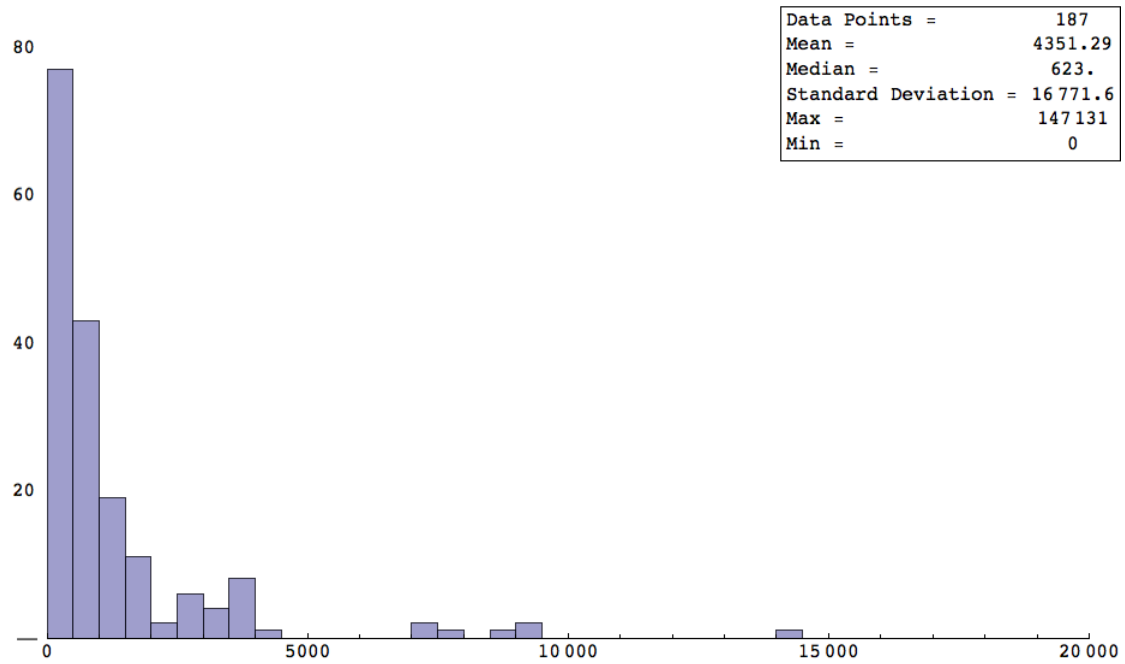


Figure A.0.34. Relevant Average Load Times

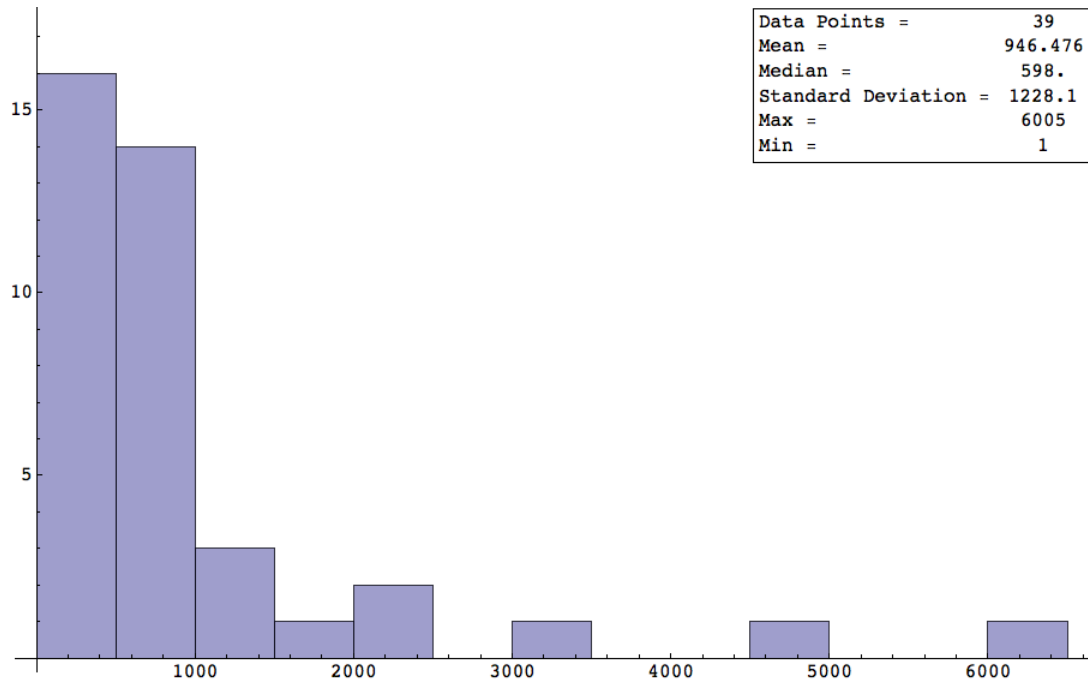


Figure A.0.35. Not Relevant Average Load Times

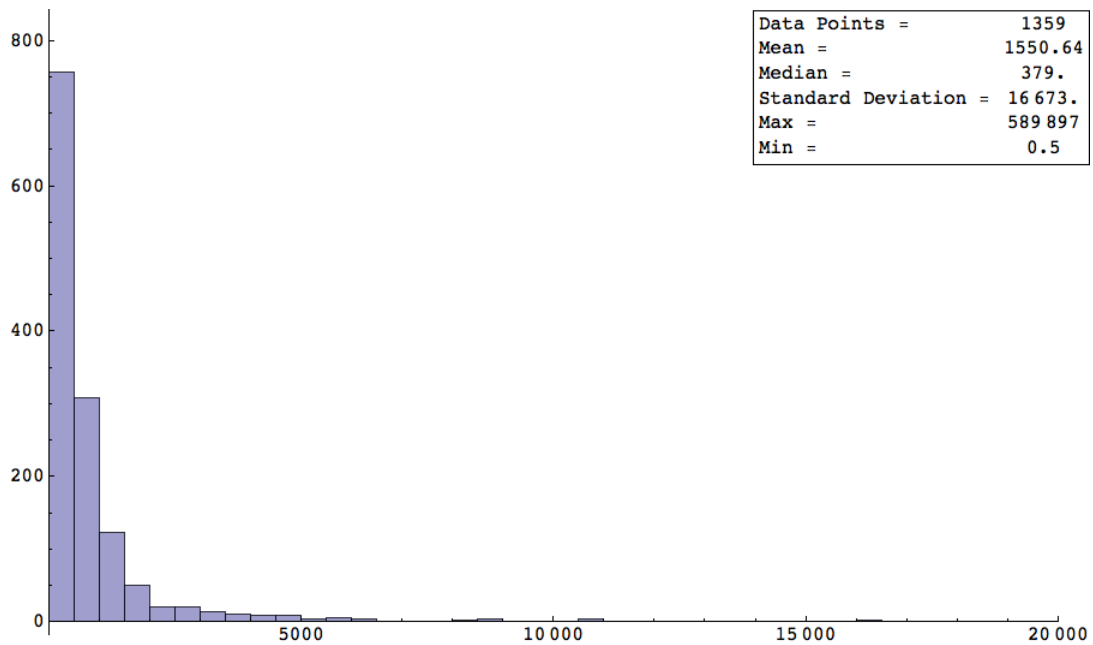


Figure A.0.36. No Response Average Load Times

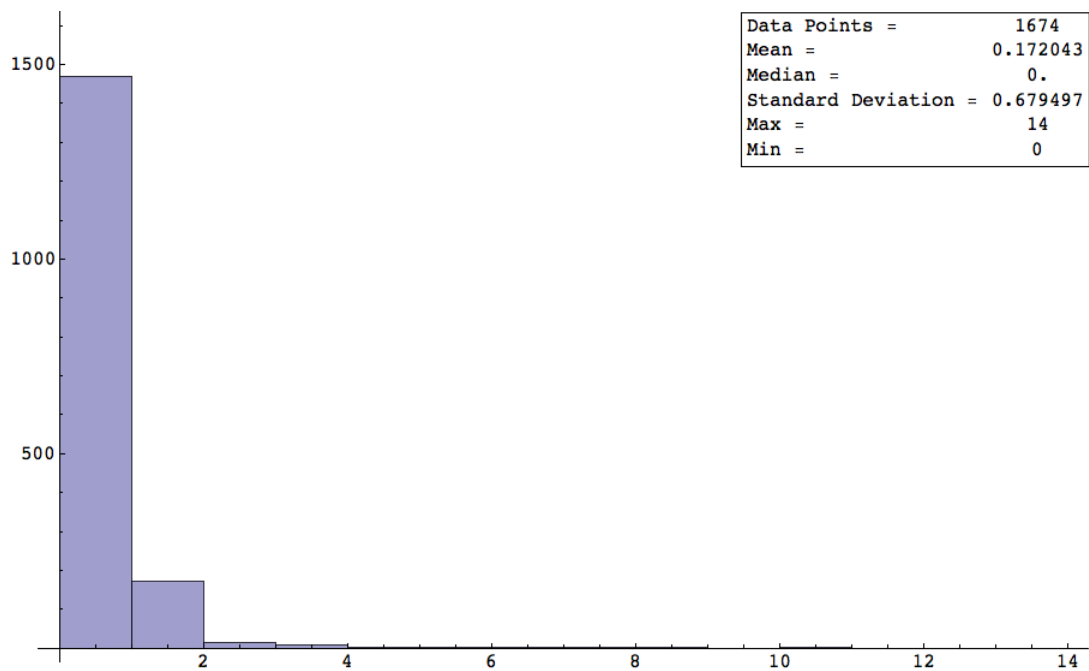


Figure A.0.37. Total Refreshes

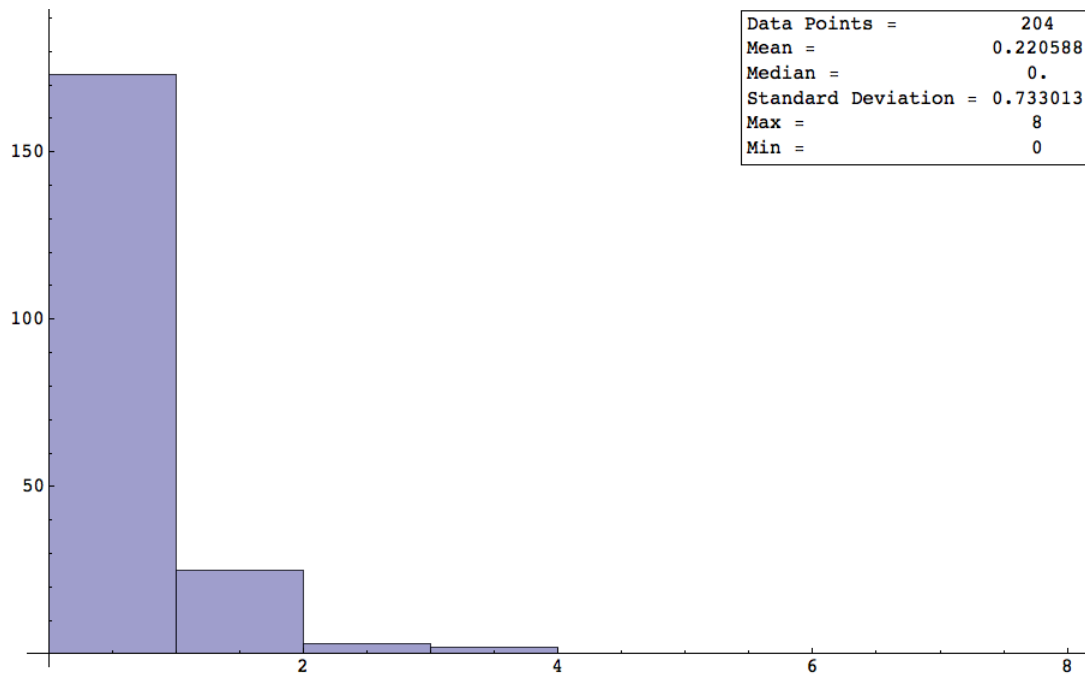


Figure A.0.38. Relevant Refreshes

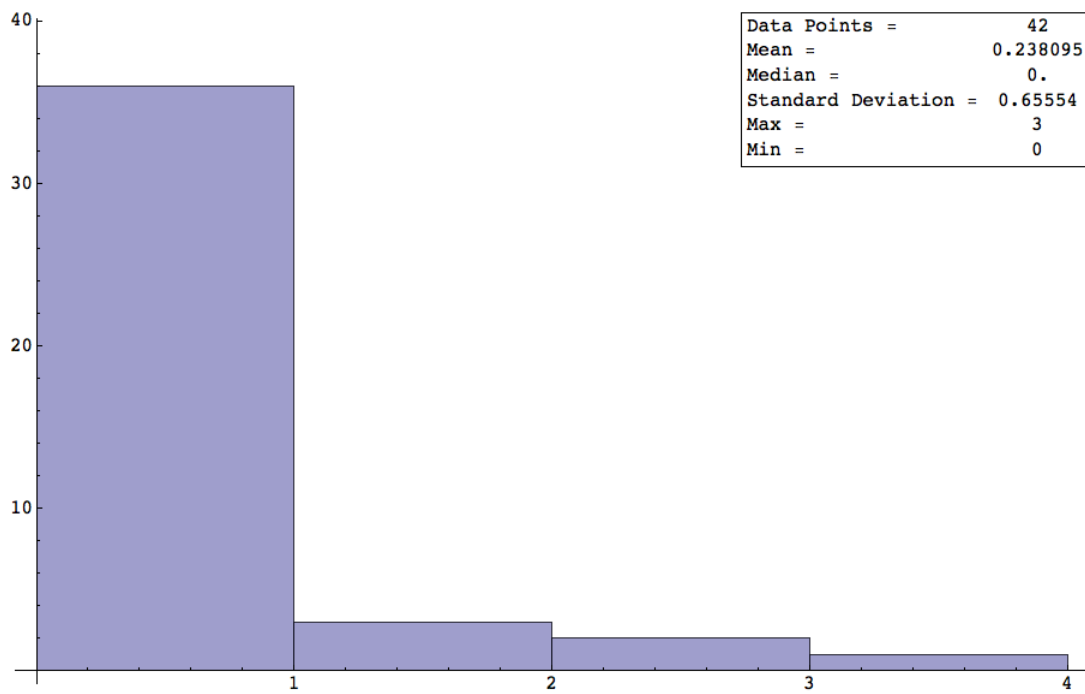


Figure A.0.39. Not Relevant Refreshes

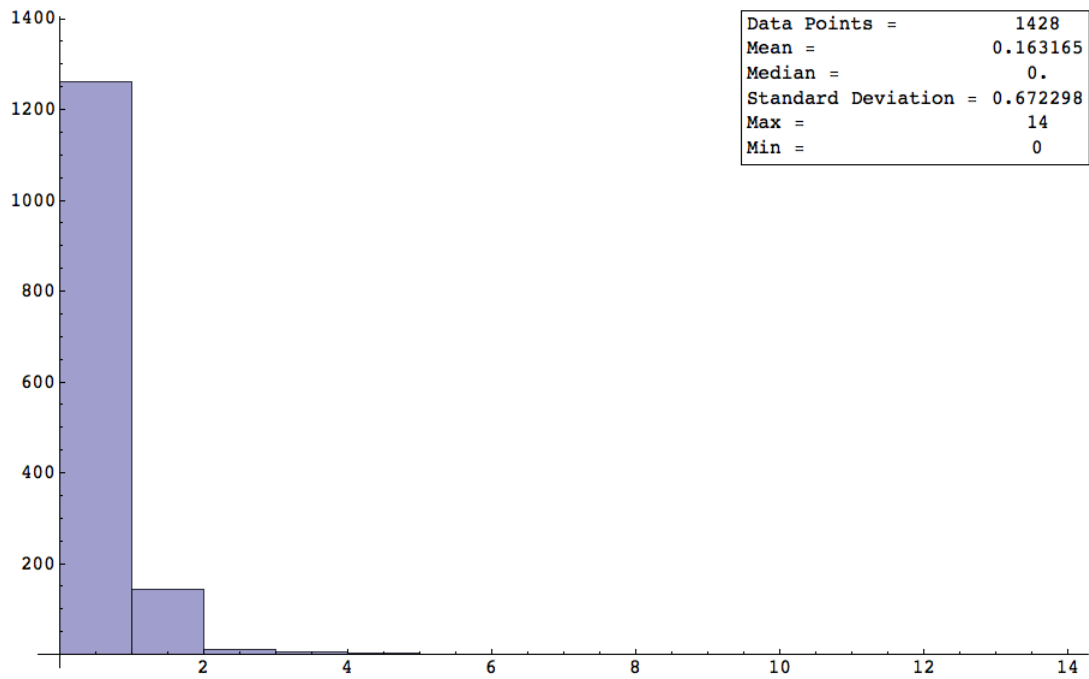


Figure A.0.40. No Response Refreshes

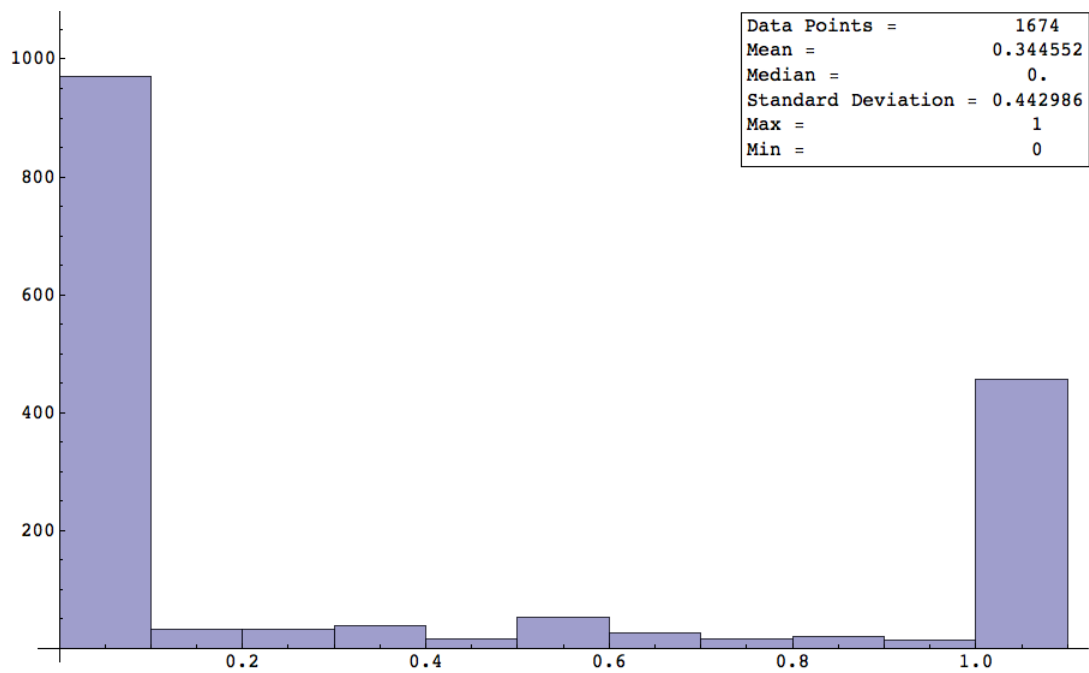


Figure A.0.41. Total Scrolls

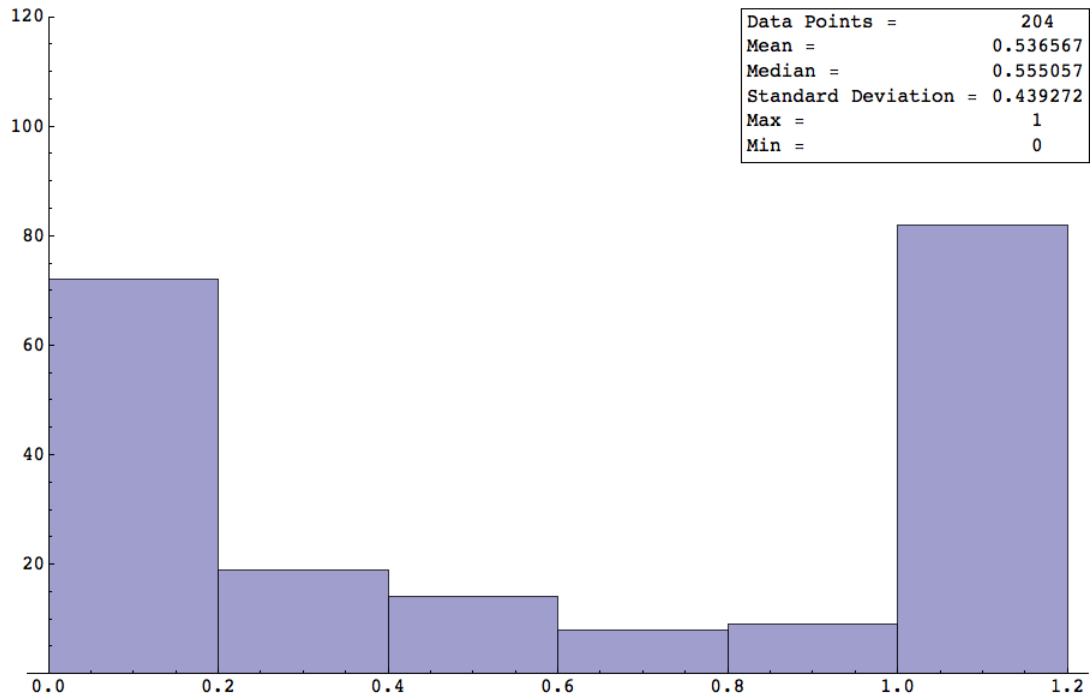


Figure A.0.42. Relevant Scrolls

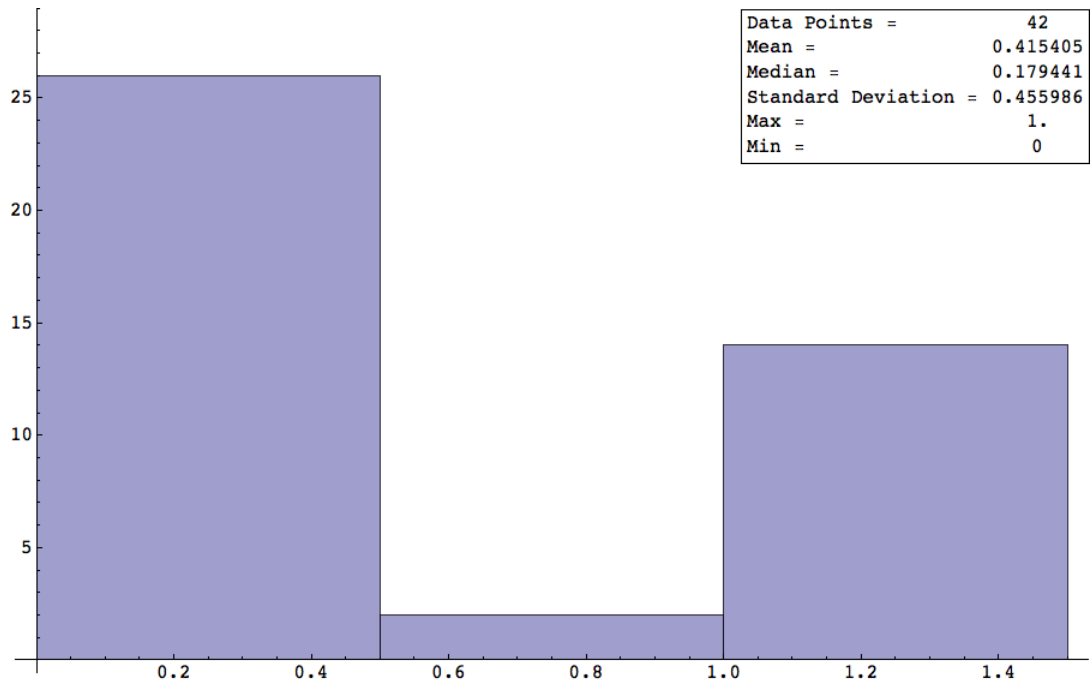


Figure A.0.43. Not Relevant Scrolls

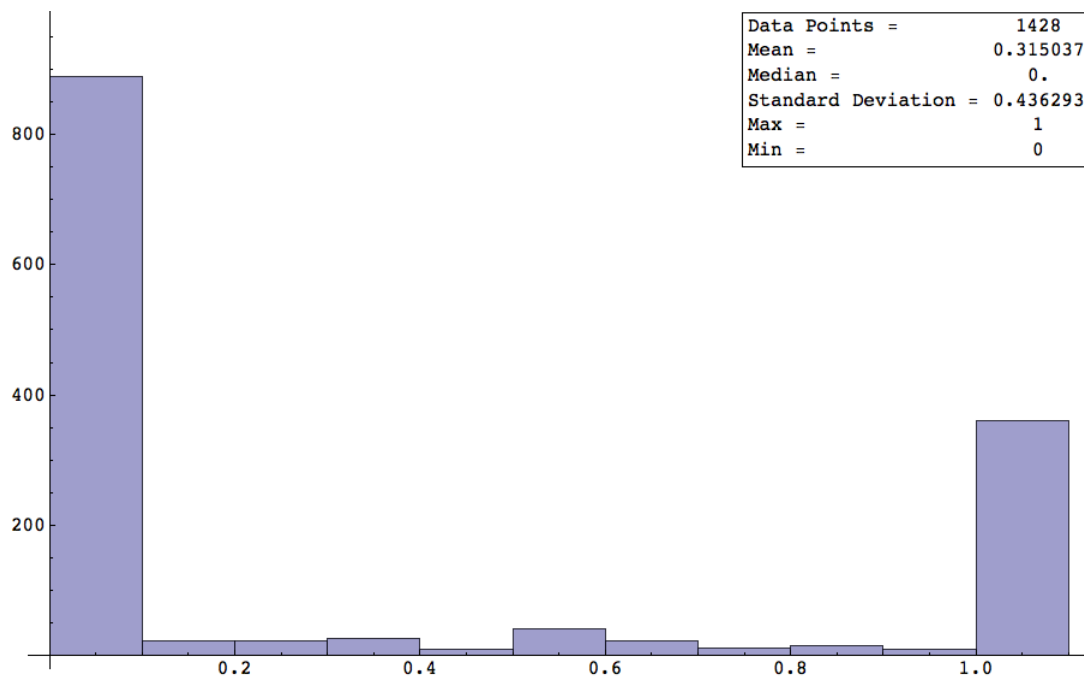


Figure A.0.44. No Response Scrolls

Appendix B

Selected Source Code

B.1 collector.js

This is the version of “collector.js” used in the deployment version outlined in Section 2.3.

```
1  /*
2  * Author: Elliot Korte
3  */
4  var edgeList = new Array(),
5  nodeList = new Array(),
6  prevNode = null,
7  prevVisitTimestamp = null,
8  prevLoadTimestamp = null,
9  groupSizes = new Array(),
10 newSession = 1;
11
12
13 function Timestamp() {
14     this.start = null;
15     this.end = null;
16 }
17
18
19 function Node() {
20     //properties related to the webpage
21     this.url = "";
22     this.title = "";
```

```
23     this.visits = 0;
24     this.visitTimestamps = new Array();
25     this.loadTimestamps = new Array();
26     this.idleTimestamps = new Array();
27     this.refreshes = new Array();
28     this.scrollPercentages = new Array();
29     this.relevantResponses = new Array();
30
31     //properties used for drawing the graph
32     this.group = 0;
33     this.groupID = 0;
34     this.nextGroup = 0;
35     this.nextGroupID = 0;
36     this.groupSize = 0;
37
38 }
39
40 function findNode(url) {
41     //returns node with the url given, or null if there is
42     //none
43     for (var i = 0; i<nodeList.length; i++) {
44         if (url == nodeList[i].url) {
45             return nodeList[i];
46         }
47     }
48     return null;
49 }
50
51 function action(tab, type) {
52     //meat of the collector, gets called whenever a page is
53     //highlighted, or loaded
54     var curTime = (new Date()).getTime();
55
56     if (prevNode==null) {
57         //this is the first page load
58         var curNode = new Node();
59         curNode.url = tab.url;
60         curNode.title = tab.title;
61         var curLoadTimestamp = new Timestamp();
62         curLoadTimestamp.start = curTime;
63         var curVisitTimestamp = new Timestamp();
64         curVisitTimestamp.start = curTime;
```



```
63     curNode.visitTimestamps.push(curVisitTimestamp);
64     curNode.loadTimestamps.push(curLoadTimestamp);
65     curNode.visits++;
66     curNode.scrollPercentages.push(0);
67     curNode.relevantResponses.push(0);
68
69     curNode.group = 0;
70     groupSizes.push(1);
71     curNode.groupID = 0;
72     curNode.nextGroup = 1;
73     curNode.nextGroupID = 1;
74
75     nodeList.push(curNode);
76     prevNode = curNode;
77
78 }
79 else {
80     if (type=="completion") {
81         //find node that finished loading and update values
82         var curNode = findNode(tab.url);
83         if (curNode==null) {
84             console.log("COMPLETED NODE NULL, SHOULD NEVER
85                 HAPPEN\n");
86             return;
87         }
88         curNode.title = tab.title;
89         curNode.loadTimestamps[curNode.loadTimestamps.
90             length - 1].end = curTime;
91         return;
92     }
93     else {
94         var curNode = findNode(tab.url);
95
96         if (curNode==null) { //this is a new page
97
98             curNode = new Node();
99             curNode.url = tab.url;
100             curNode.title = tab.title;
101             var curLoadTimestamp = new Timestamp();
102             curLoadTimestamp.start = curTime;
103             curNode.loadTimestamps.push(curLoadTimestamp);
```

```

103
104     if (type=="onCurrentUpdated") {
105         var curVisitTimestamp = new Timestamp();
106         curVisitTimestamp.start = curTime;
107         curNode.visitTimestamps.push(curVisitTimestamp)
108         ;
109         curNode.visits++;
110         curNode.scrollPercentages.push(0);
111         curNode.relevantResponses.push(0);
112
113         curNode.group = prevNode.nextGroup;
114         groupSizes.push(1);
115         curNode.groupID = 0;
116         curNode.nextGroup = prevNode.nextGroup + 1;
117         curNode.nextGroupID = 1;
118
119         prevNode.visitTimestamps[prevNode.
120             visitTimestamps.length - 1].end = curTime;
121
122         edgeList.push({from: prevNode, to: curNode,
123             type: "onCurrentUpdated", color: "black"});
124         nodeList.push(curNode);
125         prevNode = curNode;
126     }
127     else if (type=="onOtherUpdated") {
128
129         curNode.group = prevNode.group;
130         groupSizes[prevNode.group]++;
131         curNode.groupID = prevNode.nextGroupID++;
132         curNode.nextGroup = prevNode.nextGroup;
133         curNode.nextGroupID = 0;
134
135         curNode.scrollPercentages.push(0);
136         curNode.relevantResponses.push(0);
137
138         nodeList.push(curNode);
139         edgeList.push({from: prevNode, to: curNode,
140             type: "onOtherUpdated", color: "black"})

```

```

141         console.log("NEW NODE HIGHLIGHT SHOULD NEVER
142                     HAPPEN\n");
143     }
144 }
145 else if (curNode === prevNode) { //this node is a
    refresh
146
147     curNode.refreshes.push(curTime);
148     var curLoadTimestamp = new Timestamp();
149     curLoadTimestamp.start = curTime;
150     curNode.loadTimestamps.push(curLoadTimestamp);
151
152 }
153 else { //this node already exists
154
155
156     if (type=="onCurrentUpdated") {
157         var curLoadTimestamp = new Timestamp();
158         curLoadTimestamp.start = curTime;
159         curNode.loadTimestamps.push(curLoadTimestamp);
160
161         var curVisitTimestamp = new Timestamp();
162         curVisitTimestamp.start = curTime;
163         curNode.visitTimestamps.push(curVisitTimestamp)
164         ;
165         curNode.visits++;
166         curNode.scrollPercentages.push(0);
167         curNode.relevantResponses.push(0);
168
169         prevNode.visitTimestamps[prevNode.
170             visitTimestamps.length - 1].end = curTime;
171
172         edgeList.push({from: prevNode, to: curNode,
173             type: "onCurrentUpdated", color: "black"});
174         prevNode = curNode;
175     }
176
177     else if (type=="onOtherUpdated") {
178         var curLoadTimestamp = new Timestamp();
179         curLoadTimestamp.start = curTime;
180         curNode.loadTimestamps.push(curLoadTimestamp);

```

```

178         curNode.scrollPercentages.push(0);
179         curNode.relevantResponses.push(0);
180
181         edgeList.push({from: prevNode, to: curNode,
182             type: "onOtherUpdated", color: "black"});
183     }
184     else if (type=="onHighlight") {
185         prevNode.visitTimestamps[prevNode.
186             visitTimestamps.length-1].end = curTime;
187
188         var curVisitTimestamp = new Timestamp();
189         curVisitTimestamp.start = curTime;
190         curNode.visitTimestamps.push(curVisitTimestamp)
191         ;
192         curNode.visits++;
193
194         edgeList.push({from: prevNode, to: curNode,
195             type: "onOtherUpdated", color: "gray"});
196
197         prevNode = curNode;
198     }
199 }
200
201 chrome.tabs.onActivated.addListener(function (activeInfo)
202 {
203     //Listener for when a tab is highlighted
204     chrome.tabs.get(activeInfo.tabId, function(tab){
205         action(tab, "onHighlight");
206     });
207 }));
208 chrome.tabs.onUpdated.addListener(function(tabId,
209     changeInfo, tab) {
210     //Listener for when a page is begins loading, or
211     finishes loading
212     if (changeInfo.status == "loading") {
213         if (tab.highlighted) {
214             action(tab, "onCurrentUpdated");

```

```

213     }
214     else {
215         action(tab, "onOtherUpdated");
216     }
217 }
218 else if (changeInfo.status == "complete") {
219     action(tab, "completion");
220 }
221 });
222
223 /* Uncomment to reenale visualization button
224
225 chrome.browserAction.onClicked.addListener(function() {
226     chrome.tabs.create({'url': chrome.extension.getURL('
227         graphic.html')}, function(tab) {
228     });
229 });
230 */
231
232 chrome.idle.setDetectionInterval(30);
233
234 chrome.idle.onStateChanged.addListener(function(newState)
235 {
236     if ((newState=="idle") || (newState=="locked")) {
237         var tempTimestamp = new Timestamp();
238         tempTimestamp.start = (new Date()).getTime();
239         prevNode.idleTimestamps.push(tempTimestamp);
240     }
241     else if (newState=="active") {
242         prevNode.idleTimestamps[prevNode.idleTimestamps.
243             length - 1].end = (new Date()).getTime();
244     }
245 });
246
247 chrome.runtime.onMessage.addListener(
248     function(request, sender, sendResponse) {
249         if (request.greeting == "yesRelevant") {
250             sendResponse({"farewell": "you are relevant!"});
251             prevNode.relevantResponses[prevNode.
252                 relevantResponses.length-1] = 1;

```

```
251     }
252     else if (request.greeting == "noRelevant") {
253         sendResponse({"farewell": "you are not relevant"});
254         prevNode.relevantResponses[prevNode.
            relevantResponses.length-1] = -1;
255     }
256     else if (request.greeting == "pageScroll") {
257         var curNode = findNode(request.url);
258         if (curNode.scrollPercentages[curNode.
            scrollPercentages.length - 1] < request.
            percent) {
259             curNode.scrollPercentages[curNode.
                scrollPercentages.length - 1] = request.
                percent;
260         }
261     };
262 });
263
264 function postData() {
265     var myData = JSON.stringify(edgeList);
266     $.ajax({
267         type: "POST",
268         url: "http://www.elliottkorte.com/request.php",
269         data: {'myData':myData, 'computerID': 99990001 *
            20, 'newSession':newSession},
270         success: function() {
271             console.log("SUCCESSSSSS");
272         }
273     });
274     newSession = 0;
275     setTimeout(postData, 120000)
276 }
277
278 chrome.windows.onRemoved.addListener(function(windowId) {
279     postData();
280 });
281 setTimeout(postData, 120000);
```

B.2 GraphBuilder.js

GraphBuilder.js was used to visualize graphs and display information in the User Version and Analysis Version

```
1  /*
2  * Author: Elliot Korte
3  */
4  var renderer = null,
5  scene = null,
6  camera = null,
7  container = null,
8  isMouseDown = false,
9  bigObject = null,
10 windowWidth = 0,
11 windowHeight = 0,
12 clock = null,
13 nodes = [],
14 mouse = new THREE.Vector2(),
15 objects = [],
16 offset = new THREE.Vector3(),
17 projector = new THREE.Projector(),
18 objectSelected, objectIntersected,
19 currentObject = null,
20 edgeList = null,
21 curEdgeIndex = 0,
22 curNode = null;
23
24 function getRandomColor() {
25     var letters = '0123456789ABCDEF'.split('');
26     var color = '#';
27     for (var i = 0; i < 6; i++ ) {
28         color += letters[Math.round(Math.random() * 15)];
29     }
30     return color;
31 }
32
33 function initializeBuilder(c) {
34
35     container = c;
36     windowWidth = window.innerWidth;
37     windowHeight = window.innerHeight;
38 }
```

```
39     renderer = new THREE.WebGLRenderer( {antialias:true}
40         );
41
42     renderer.setSize(windowWidth, windowHeight);
43     renderer.setClearColor(0xeaecfe, 1);
44     container.appendChild( renderer.domElement );
45
46     // Create a new Three.js scene
47     scene = new THREE.Scene();
48
49     // Put in a camera
50     camera = new THREE.PerspectiveCamera( 60, window.
51         innerWidth / window.innerHeight, 1, 20000 );
52     camera.position.set( 0, 0, 600 );
53
54     // EVENTS
55     THREEEx.WindowResize(renderer, camera);
56     THREEEx.FullScreen.bindKey({ charCode : 'f'.charCodeAt
57         (0) });
58
59     // MOUSE CONTROLS
60     controls = new THREE.OrbitControls( camera, renderer.
61         domElement );
62
63     // STATS
64     stats = new Stats();
65     stats.domElement.style.position = 'absolute';
66     stats.domElement.style.bottom = '0px';
67     stats.domElement.style.zIndex = 100;
68     container.appendChild( stats.domElement );
69
70     bigObject = new THREE.Object3D(); //object all
71         visible objects get added to
72
73     //create a visible x, y, and z axis
74     var material = new THREE.LineBasicMaterial({ color: 0
75         xff0000, linewidth: 1 });
76     var geometry = new THREE.Geometry();
77     geometry.vertices.push(new THREE.Vector3(120,0,0));
78     geometry.vertices.push(new THREE.Vector3(0,0,0));
79     var xAxis = new THREE.Line(geometry, material);
80     bigObject.add(xAxis);
```



```
74     material = new THREE.LineBasicMaterial({ color: 0
75         x00ff00, linewidth: 1 });
76     geometry = new THREE.Geometry();
77     geometry.vertices.push(new THREE.Vector3(0,120,0));
78     geometry.vertices.push(new THREE.Vector3(0,0,0));
79     var yAxis = new THREE.Line(geometry, material);
80     bigObject.add(yAxis);
81
82     material = new THREE.LineBasicMaterial({ color: 0
83         x0000ff, linewidth: 1 });
84     geometry = new THREE.Geometry();
85     geometry.vertices.push(new THREE.Vector3(0,0,120));
86     geometry.vertices.push(new THREE.Vector3(0,0,0));
87     var zAxis = new THREE.Line(geometry, material);
88     bigObject.add(zAxis);
89
90     addMouseHandler();
91     scene.add(bigObject);
92 }
93
94 function getNode(url) {
95     for (var i = 0; i < nodes.length; i++) {
96         if (nodes[i].url == url) {
97             return nodes[i];
98         }
99     }
100     return null;
101 }
102
103 function getGroupCenter(n) {
104     for (var i = 0; i < nodes.length; i++) {
105         if ((nodes[i].groupID == 0) && (n.group == nodes[
106             i].group)) {
107             return [nodes[i].x, nodes[i].y, nodes[i].z];
108         }
109     }
110 }
111
112 function generateGraph(data, dataSize) {
113     //takes edge pair list from collector.js and
114     generates the graph
```

```

112     edgeList = data;
113
114     var first = new node(data[0].from, data[0].from.group
        , data[0].from.groupID, data[0].from.groupSize,
        data[0].from.infoString, getRandomColor());
115     first.initialize();
116     nodes.push(first);
117     first.setCoords( 0, 0, (first.group * -300) + 300 );
118
119     var second = new node(data[0].to, data[0].to.group,
        data[0].to.groupID, data[0].to.groupSize, data[0].
        to.infoString, getRandomColor());
120     second.initialize();
121     nodes.push(second);
122
123     second.setCoords( (Math.random()*400 - 200), (Math.
        random()*400 - 200), (second.group * -300) + 300
        );
124
125     addEdge(first, second, data[0].color);
126
127     for (var i = 1 ; i<data.length; i++) {
128         first = getNode(data[i].from.url);
129         second = getNode(data[i].to.url);
130         if ((first!=null) && (second!=null)) {
131             addEdge(first, second, data[i].color);
132         }
133         else if (first!=null) {
134             second = new node(data[i].to, data[i].to.
                group, data[i].to.groupID, data[i].to.
                groupSize, data[i].to.infoString,
                getRandomColor());
135             second.initialize();
136             nodes.push(second);
137             if (second.groupID == 0) {
138                 second.setCoords( (Math.random()*400 -
                    200), (Math.random()*400 - 200), ((
                    second.group * -300) + 300));
139             }
140             else {
141                 var angle = (2*Math.PI)/(second.groupSize
                    - 1);

```

```

142         coordList = getGroupCenter(second);
143         second.setCoords( 250*Math.cos(2*Math.PI*
            second.groupID/(second.groupSize - 1))
            + coordList[0], 250*Math.sin(2*Math.
            PI*second.groupID/(second.groupSize -
            1)) + coordList[1], ((second.group *
            -300) + 300));
144     }
145     addEdge(first, second, data[i].color);
146 }
147 else if (second!=null) {
148     first = new node(data[i].from, data[i].from.
        group, data[i].from.groupID, data[i].from.
        groupSize, data[i].from.infoString ,
        getRandomColor());
149     first.initialize();
150     nodes.push(first);
151     if (first.groupID == 0) {
152         first.setCoords( (Math.random()*400 -
            200), (Math.random()*400 - 200), (
            first.group * -300) + 300);
153     }
154     else {
155         var angle = (2*Math.PI)/(first.groupSize
            - 1);
156         coordList = getGroupCenter(first);
157         first.setCoords( 250*Math.cos(2*Math.PI*
            first.groupID/(first.groupSize - 1)) +
            coordList[0], 250*Math.sin(2*Math.PI*
            first.groupID/(first.groupSize - 1)) +
            coordList[1], ((first.group * -300) +
            300));
158     }
159     addEdge(first, second, data[i].color);
160 }
161 else {
162     console.log("THIS SHOULD NEVER HAPPEN");
163 }
164 }
165 updateLabels();
166 curEdgeIndex = edgeList.length - 1;
167 curNode = getNode(edgeList[curEdgeIndex].to.url);

```

```

168     cameraTarget(curNode);
169     updateSidebar(curNode.relatedNode);
170     $("#forwardButton").click(goForward);
171     $("#backButton").click(goBackward);
172 }
173
174 function node(relatedNodey, g, gID, gS, infoSt, c) {
175     //object representing graphical node, reference to
176     //node it represents is contained in this.
177     relatedNode
178     this.color = c;
179     this.relatedNode = relatedNodey;
180     this.name = this.relatedNode.title;
181     this.url = this.relatedNode.url;
182     this.x = 0;
183     this.y = 0;
184     this.z = 0;
185     this.mesh = null;
186     this.label = null;
187     this.group = g;
188     this.groupID = gID;
189     this.groupSize = gS;
190     this.infoString = infoSt;
191     this.edgeVectors = [];
192
193     this.initialize = function() {
194         var material = new THREE.MeshBasicMaterial({color
195             : c});
196         var geometry = new THREE.SphereGeometry(32, 32,
197             32);
198         this.mesh = new THREE.Mesh(geometry, material);
199         this.mesh["nodey"] = this;
200         this.label = makeTextSprite( this.name, {
201             fontsize: 24, backgroundColor: {r:255, g:100,
202             b:100, a:1} } );
203         bigObject.add( this.label);
204         bigObject.add(this.mesh);
205         objects.push(this.mesh);
206     };
207
208     this.setX = function(xx) {
209         this.x = xx;

```

```

204         this.mesh.position.x = xx;
205     };
206     this.setY = function(yy) {
207         this.y = yy;
208         this.mesh.position.y = yy;
209     };
210     this.setZ = function(zz) {
211         this.z = zz;
212         this.mesh.position.z = zz;
213     };
214     this.setCoords = function(xx,yy,zz) {
215         this.x = xx;
216         this.mesh.position.x = xx;
217         this.y = yy;
218         this.mesh.position.y = yy;
219         this.z = zz;
220         this.mesh.position.z = zz;
221         var zAdd = (Math.abs(zz) * (-15 /
222             240) + 15);
223         this.label.position = (new THREE.
224             Vector3());
225         updateLabels();
226     };
227     this.updateLocation = function() {
228         this.x = this.mesh.position.x;
229         this.y = this.mesh.position.y;
230         this.z = this.mesh.position.z;
231         for (var i = 0; i<this.edgeVectors.
232             length; i++) {
233             console.log(this.edgeVectors[i]);
234             this.edgeVectors[i].set(this.mesh.
235                 position.x, this.mesh.
236                 position.y, this.mesh.position.
237                 z);
238         }
239         this.label.position = (new THREE.
240             Vector3());
241         updateLabels();
242     };
243 };

```

```

239
240 function updateLabels() {
241     var cameraDistance = Math.sqrt(Math.pow(camera.
        position.x,2) + Math.pow(camera.position.y,2) +
        Math.pow(camera.position.z,2));
242     var scaledDownCameraX = (camera.position.x );
243     var scaledDownCameraY = (camera.position.y);
244     var scaledDownCameraZ = (camera.position.z);
245     var curNode = null;
246     for (var i = 0; i < nodes.length; i++) {
247         curNode = nodes[i];
248         var scalar = 40 / Math.sqrt(Math.pow(
            scaledDownCameraX - curNode.x, 2) + Math.pow(
            scaledDownCameraY - curNode.y, 2) + Math.pow(
            scaledDownCameraZ - curNode.z, 2) );
249         curNode.label.position.x = curNode.x - (scalar *
            (-scaledDownCameraX + curNode.x));
250         curNode.label.position.y = curNode.y - (scalar *
            (-scaledDownCameraY + curNode.y));
251         curNode.label.position.z = curNode.z - (scalar *
            (-scaledDownCameraZ + curNode.z));
252     }
253 }
254
255 function addEdge(node1, node2, c) {
256     //draws an arrow from node1 to node2
257     var co;
258     if (c=="gray") {
259         co = 0xcccccc;
260     }
261     else if (c=="black") {
262         co = 0x000000;
263     }
264     var material = new THREE.LineBasicMaterial({color: co
        , linewidth: 3 });
265     var geometry = new THREE.Geometry();
266     var firstVector = new THREE.Vector3(node1.x,node1.y,
        node1.z);
267     node1.edgeVectors.push(firstVector);
268     geometry.vertices.push(firstVector);
269     var secondVector = new THREE.Vector3(node2.x,node2.y,
        node2.z);

```

```

270     node1.edgeVectors.push(secondVector);
271     geometry.vertices.push(secondVector);
272
273     var direction = new THREE.Vector3().subVectors(
274         secondVector, firstVector).normalize();
275     if (c=="gray") {
276         var arrow = new THREE.ArrowHelper(direction,
277             firstVector, computeDistance(node1, node2) -
278             64, co);
279     }
280     else if (c=="black") {
281         var arrow = new THREE.ArrowHelper(direction,
282             firstVector, computeDistance(node1, node2) -
283             32, co);
284     }
285     bigObject.add(arrow);
286 }
287
288 function computeDistance(n1, n2) {
289     return Math.sqrt(Math.pow(n1.x - n2.x, 2) + Math.pow(
290         n1.y - n2.y , 2) + Math.pow(n1.z - n2.z, 2))
291 }
292
293 function makeTextSprite( message, parameters )
294 { //this function authored by Lee Stemkoski
295     if ( parameters === undefined ) parameters = {};
296     var fontface = parameters.hasOwnProperty("fontface")
297         ?
298         parameters["fontface"] : "Arial";
299     var fontsize = parameters.hasOwnProperty("fontsize")
300         ?
301         parameters["fontsize"] : 18;
302     var borderThickness = parameters.hasOwnProperty("
303         borderThickness") ?
304         parameters["borderThickness"] : 4;
305     var borderColor = parameters.hasOwnProperty("
306         borderColor") ?
307         parameters["borderColor"] : { r:0, g:0, b:0, a
308             :1.0 };
309     var backgroundColor = parameters.hasOwnProperty("
310         backgroundColor") ?

```

```

299         parameters["backgroundColor"] : { r:255, g:255, b
           :255, a:1.0 };
300     var canvas = document.createElement('canvas');
301     var context = canvas.getContext('2d');
302     context.font = "Bold " + fontsize + "px " + fontface;
303     var metrics = context.measureText( message );
304     var textWidth = metrics.width;
305     // background color
306     context.fillStyle = "rgba(" + backgroundColor.r + "
           , " + backgroundColor.g + ", "
307                                     + backgroundColor.b + "
                                           , " + backgroundColor
                                               .a + ")";
308     // border color
309     context.strokeStyle = "rgba(" + borderColor.r + ", " +
           borderColor.g + ", "
310                                     + borderColor.b + ", " +
                                           borderColor.a + ")";
311                                     ;
312     context.lineWidth = borderThickness;
313     roundRect(context, borderThickness/2, borderThickness
           /2, textWidth + borderThickness, fontsize * 1.4 +
           borderThickness, 6);
314     context.fillStyle = "rgba(0, 0, 0, 1.0)";
315     context.fillText( message, borderThickness, fontsize
           + borderThickness);
316     var texture = new THREE.Texture(canvas)
317     texture.needsUpdate = true;
318
319     var spriteMaterial = new THREE.SpriteMaterial(
320         { map: texture, useScreenCoordinates: false } );
321     var sprite = new THREE.Sprite( spriteMaterial );
322     sprite.scale.set(100,50,1.0);
323     return sprite;
324 }
325
326 function roundRect(ctx, x, y, w, h, r) {
327     // function for drawing rounded rectangles
328     ctx.beginPath();
329     ctx.moveTo(x+r, y);
330     ctx.lineTo(x+w-r, y);

```



```
331     ctx.quadraticCurveTo(x+w, y, x+w, y+r);
332     ctx.lineTo(x+w, y+h-r);
333     ctx.quadraticCurveTo(x+w, y+h, x+w-r, y+h);
334     ctx.lineTo(x+r, y+h);
335     ctx.quadraticCurveTo(x, y+h, x, y+h-r);
336     ctx.lineTo(x, y+r);
337     ctx.quadraticCurveTo(x, y, x+r, y);
338     ctx.closePath();
339     ctx.fill();
340     ctx.stroke();
341 }
342
343 function addMouseHandler()
344 {
345     var dom = renderer.domElement;
346     dom.addEventListener( 'mousemove', onMouseMove, false
347                          );
347     dom.addEventListener( 'mousedown', onMouseDown, false
348                          );
348     dom.addEventListener( 'mouseup' , onMouseUp, false);
349 }
350
351 function onMouseDown(event)
352 {
353     event.preventDefault();
354     isMouseDown = true;
355     var vector = new THREE.Vector3( ( event.clientX /
356                                     window.innerWidth ) * 2 - 1, - ( event.clientY /
357                                     window.innerHeight ) * 2 + 1, 0.5 );
356     projector.unprojectVector( vector, camera );
357
358     var raycaster = new THREE.Raycaster( camera.position,
359                                         vector.sub( camera.position ).normalize() );
359
360     var intersects = raycaster.intersectObjects( objects
361                                         );
361
362     if ( intersects.length > 0 ) {
363         //if mouse click happened on a node
364         curNode = intersects[0].object.nodex;
365         cameraTarget(curNode);
366         updateSidebar(curNode.relatedNode);
```

```
367         for (var i = 0; i<edgeList.length; i++) {
368             if (curNode===getNode(edgeList[i].to.url)) {
369                 curEdgeIndex = i;
370             }
371         }
372         if (curNode===getNode(edgeList[0].from.url)) {
373             curEdgeIndex = -1;
374         }
375     }
376 }
377
378 function onMouseUp (event) {
379     event.preventDefault();
380     isMouseDown = false;
381 }
382 }
383
384 function onMouseMove( event ) {
385     event.preventDefault();
386     if (isMouseDown) {
387         updateLabels();
388     }
389 }
390
391 function goForward() {
392     var foundNext = false;
393     var startIndex = curEdgeIndex;
394     while (!foundNext) {
395         if (curEdgeIndex<0) {
396             if (edgeList[++curEdgeIndex].type != "
onOtherUpdated") {
397                 curNode = getNode(edgeList[curEdgeIndex].
to.url);
398                 cameraTarget(curNode);
399                 updateSidebar(curNode.relatedNode);
400                 foundNext = true;
401             }
402         }
403         else if (curEdgeIndex<(edgeList.length - 1)) {
404             if (edgeList[++curEdgeIndex].type != "
onOtherUpdated") {
```

```

405         curNode = getNode(edgeList[curEdgeIndex].
406             to.url);
407         cameraTarget(curNode);
408         updateSidebar(curNode.relatedNode);
409         foundNext = true;
410     }
411     else {
412         curEdgeIndex = startIndex;
413         break;
414     }
415 }
416 }
417
418 function goBackward() {
419     var foundNext = false;
420     var startIndex = curEdgeIndex;
421     while (!foundNext) {
422         if (curEdgeIndex>0) {
423             if (edgeList[--curEdgeIndex].type != "
424                 onOtherUpdated") {
425                 curNode = getNode(edgeList[curEdgeIndex].
426                     to.url);
427                 cameraTarget(curNode);
428                 updateSidebar(curNode.relatedNode);
429                 foundNext = true;
430             }
431         }
432         else if (curEdgeIndex==0) {
433             if (edgeList[curEdgeIndex].type != "
434                 onOtherUpdated") {
435                 curNode = getNode(edgeList[--curEdgeIndex
436                     + 1].from.url);
437                 cameraTarget(curNode);
438                 updateSidebar(curNode.relatedNode);
439                 foundNext = true;
440             }
441         }
442         else {
443             curEdgeIndex = startIndex
444             break;
445         }
446     }

```

```

442     }
443 }
444
445 function cameraTarget(node) {
446     controls.target.set(node.x, node.y, node.z);
447 }
448
449 function updateSidebar(relatedNode) {
450     $("#title").html("TITLE: " + relatedNode.title + "<br>");
451     $("#url").html("URL: " + relatedNode.url + "<br>");
452     $("#visits").html("VISITED " + relatedNode.visits + " TIMES <br>");
453
454     var visitTimestampString = "";
455     for (var i = 0; i<relatedNode.visitTimestamps.length; i++) {
456         visitTimestampString += "start:<br> ";
457         visitTimestampString += (new Date(relatedNode.visitTimestamps[i].start)).toUTCString();
458         visitTimestampString += "<br>end:<br>";
459         visitTimestampString += (new Date(relatedNode.visitTimestamps[i].end)).toUTCString() + "<br>";
460     }
461     $("#visitTimes").html("VISITS:<br> " + visitTimestampString + "<br>");
462
463     $("#loads").html("LOADED " + relatedNode.loadTimestamps.length + " TIMES <br>");
464     var loadTimestampString = "";
465     for (var i = 0; i<relatedNode.loadTimestamps.length; i++) {
466         loadTimestampString += "start:<br>";
467         loadTimestampString += (new Date(relatedNode.loadTimestamps[i].start)).toUTCString();
468         loadTimestampString += "<br>end:<br> ";
469         loadTimestampString += (new Date(relatedNode.loadTimestamps[i].end)).toUTCString() + "<br>";
470     }

```

```

471     $("#loadTimes").html("LOADS:<br>" +
        loadTimestampString + "<br>");
472
473     $("#idles").html("IDLED ON " + relatedNode.
        idleTimestamps.length + " TIMES <br>");
474     var idleTimestampString = "";
475     for (var i = 0; i<relatedNode.idleTimestamps.length;
        i++) {
476         idleTimestampString += "start:<br>";
477         idleTimestampString += (new Date(relatedNode.
            idleTimestamps[i].start)).toUTCString();
478         idleTimestampString += "<br>end:<br> ";
479         idleTimestampString += (new Date(relatedNode.
            idleTimestamps[i].end)).toUTCString() + "<br>"
            ;
480     }
481     $("#idleTimes").html("IDLE TIMES:<br>" +
        idleTimestampString + "<br>");
482
483     var refreshesString = "";
484     for (var i = 0; i<relatedNode.refreshes.length; i++)
        {
485         refreshesString+= relatedNode.refreshes[i] + ", "
            ;
486     }
487     $("#refreshes").html("REFRESHES: " + refreshesString
        + "<br>");
488
489     var scrollsString = "";
490     for (var i = 0; i<relatedNode.scrollPercentages.
        length; i++) {
491         scrollsString+= relatedNode.scrollPercentages[i]
            + ", ";
492     }
493     $("#scrolls").html("PAGE SCROLLS: " + scrollsString +
        "<br>");
494
495 }
496
497 function runIt() {
498     // Render the scene
499     requestAnimationFrame(runIt);

```

```
500
501     renderer.render( scene, camera );
502
503     controls.update();
504     stats.update();
505
506 }
```