WikiSleuth

Edgardo Anguiano-Sainz Pat Dale Gustav Danielsson
Gil Eisbruch Thor Laack-Veeder Gordon Loery

I. Introduction

1. The Main Problem

On Wikipedia, it's very simple to find the history of an entire page. A user can simply click the "View History" tab to see the changes that have been made to an article over time. However, it's not nearly as simple to see the history of a sentence, or a paragraph, or any arbitrary string of text. The goal of WikiSleuth is to offer a comprehensive, intuitive way for Wikipedia users to solve that problem within their browser, while on Wikipedia.

2. WikiSleuth Feature Set

2.1 Affected Revision Pane

Wikisleuth allows an editor to find the 10 most recent revisions that affect a given segment of text. The affected revision pane tells the user whether a word or phrase was added or deleted, and provides the user with information relevant to that edit, including the date of the revision, the author of the revision, and what words or phrases were changed from the text of interest. These revisions are displayed in a pane consistent with Wikipedia's design. Each entry, when expanded, can provide additional information with regards to the revision of interest including an option for the user to undo a revision.

Figure 1: WikiSleuth Affected Revisions Pane

2.2 Heatmap

WikiSleuth's heatmap colors the Wikipedia article sentence by sentence. Sentences colored blue are considered cold spots, where cold sentences are those that have not been changed for a long period of time. Sentences colored red are considered hot spots, where hot sentences are those that have been changed very recently. In this way, the Wikipedia editor can gauge how frequently the page in question has been updated, and provide insight into the accuracy of the article.

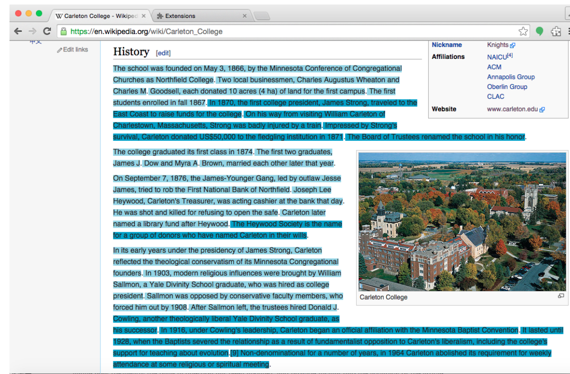

Figure 2: WikiSleuth Heatmap Feature

II. Background

Wikipedia is built by a community of editors and many of them find themselves asking questions about strange sentences or old information. It is difficult to discover when a certain part of a page was last edited, or who wrote a particular passage. Although Wikipedia makes a page's history easily accessible, as seen in the image below, there may be thousands of different entries an editor must sort through in order to find the moment a specific string was changed. Depending on the article, it could take minutes or hours to find this information. Our features, "find affected revisions" and "heatmap" would otherwise take an unreasonable amount of time to complete by a Wikipedia user, and provide valuable information to users interested in a more tailored history of an article.

1. Revisions and Affecting Revision

Wikipedia keeps track of the history of an article by storing snapshots of what the article has looked like throughout time. These snapshots are called revisions. A new revision is created and stored every time there is an edit to the article. The oldest revision of an article is the article as it existed when it was first created. The current revision of an article is what the page looks like currently on its Wikipedia page.

When a string changes between two revisions $revision_1$ and $revision_2$, we say that $revision_2$ **affects** the string. We can see exactly how the string was edited by analyzing the difference between $revision_1$ and $revision_2$. In the Affected Revision Pane, our goal is to display to the user how the highlighted string changed between the 10 most recent revisions that affect the string. In the image below, we see that the string "Most" appears in the current revision of an article that is on its 6th revision. The affecting revisions are the revisions that feature a changed version of the string.

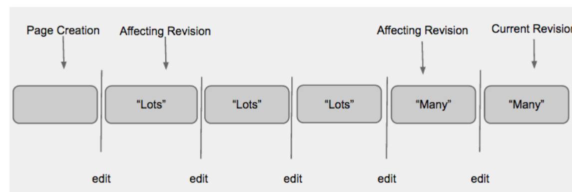

Figure 3: Revision History

2. Theoretical Background

The processes we needed, in order to achieve our features, were developed from discussions with our academic adviser, course work knowledge, and any algorithms used were implemented in groups. No article, journal, or textbook had a direct contribution to the development of WikiSleuth, but some influenced our implementation decisions.We used a modified iterative binary search algorithm, which takes the user selected text of interest and finds the first revision in Wikipedia's history that affects our text. Our modification to the binary search algorithm will be discussed later in the paper, but for more clarity on iterative binary search reference the 1971 paper by Knuth, et al. [1]. Another core component of WikiSleuth was string-to-string comparison, and we implemented a modified fuzzy string matching algorithm, which compares portions of text to another and returns a reconstructed piece of text. This paper will cover why we needed to use fuzzy string matching, but reference the 1974 paper by Wagner, et al. [2], to learn more about string-to-string comparison.

III. Methods

1. Binary Search Algorithm

1.1 Modifying Binary Search

Our program uses a modified binary search algorithm. We start by getting a list of 500 revisions and finding the midpoint. Then, if the midpoint does affect our highlighted string, we know that no revision older than the middle revision can be the revision we are looking for, so we can remove the older half of the list, excluding the middle revision, from the list of revisions being considered, because we know that the most recent revision must at least be the middle revision, if not a newer one. On the other hand, If the middle revision does not affect our highlighted string, then we know that if any changes were made to the highlighted string in any of the newer revisions, those changes would still show up in the middle revision, as both are being compared to the most recent revision. So that means that we can remove the newer half of the list of revisions, including the middle revision.

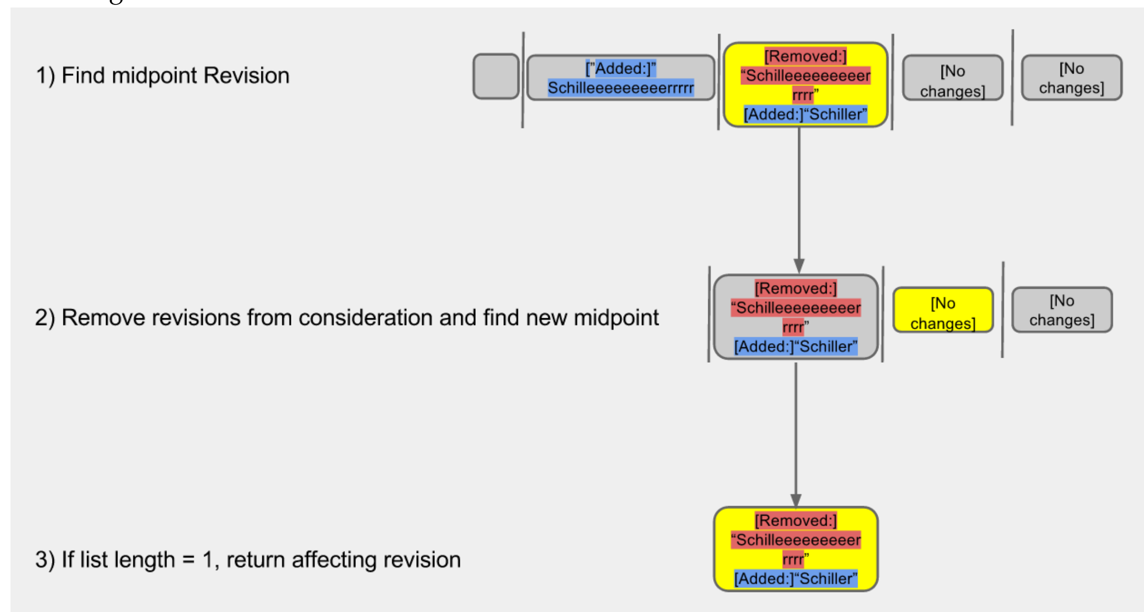

Figure 4: WikiSleuth Binary Search process

A traditional binary search algorithm divides its input size in half in every iteration. However, our algorithm sometimes divides its input size in half, and sometimes divides it in half plus one additional item. This is due to the way our algorithm goes from iteration to iteration:

If a revision doesn't affect the highlighted string we're looking at, then we know that neither it, nor any revision newer than it is the most recent revision to affect the highlighted string.

However, in the case that a revision does affect the highlighted string, we can't eliminate the middle revision we're looking at. In figure 4 (revisions are organized with oldest revisions on the left, and newest revisions on the right), we can see this happening from step 1 to step 2, where the older half of the list is removed from consideration, but the middle revision stays in the list.

Therefore, it is important to take into account a prominent edge case: if the list of revisions is a list containing two revisions, both of which affect our highlighted string, we would run an infinite loop. We would first check the middle (older) of the two revisions, which would affect our highlighted string. We would then hold on to both revisions for the next iteration of the binary search, and this process would repeat infinitely.

In order to deal with this, we use a linear search if our revision list is size two, where we simply check both revisions and return whichever one is more recent and affects the highlighted string.

1.2 Difference Algorithm

Due to the fact that we use a modified binary search algorithm to find the first revision that has affected a string, we need a way to ensure this revision is in fact the very first revision that has affected a string. To do this we use a difference (diff) algorithm found optimized for wikipedia called wikEd [4].

The diff algorithm (wikEd) is based off of Paul Heckel's paper, A technique for isolating differences between files Communications of the ACM 21(4):264 (1978). "This algorithm is word-based and uses unique words as anchor points to identify matching text and moved blocks. wikEd diff has additional code for resolving unmatched islands that are caused by common tokens at the border of sequences of common tokens or by sequence crossing-overs. From these matching data, the program extracts moved blocks and compiles a new text with added insertion, deletion, and moved blocks, and block marks." [4] This means that we are able to obtain a list of objects, which we will call fragments, that is returned from the wikEd algorithm that can be linearly iterated through which will match the same linear structure and contents of the original Wikipedia page we wanted to run the wikEd diff algorithm against.

As briefly mentioned earlier, fragments is a list of dictionary objects. The dictionary objects have two keys we care about: the type of the fragment block and the contents of that block. There are three main types of fragment blocks; equal blocks, removal blocks, and addition blocks. To simplify the explanation of these blocks we will imagine a scenario where we have an article A that is being diffed to past revision of itself named B. An equal diff type means that when comparing A to B, there is some amount of text that remained the same between A and B. A removal block means that text in article A was removed when it transitioned from a prior form B. An addition block means that text in article A was added when it transitioned from a prior form B.

A fragment block's content is composed of the longest string of text that is similar in type when two pieces of text are diffed against each other. We will use the following image to better explain this concept.

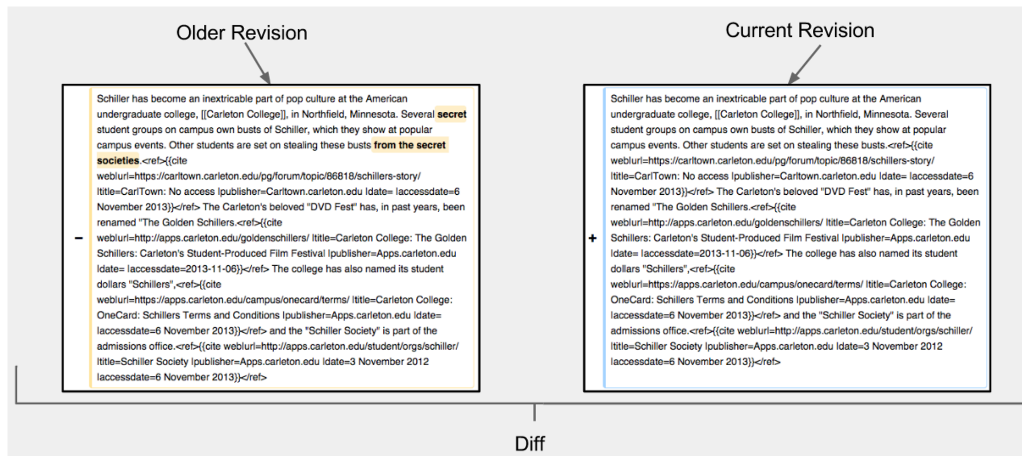

Figure 5.1: Wikipedia Diff Example

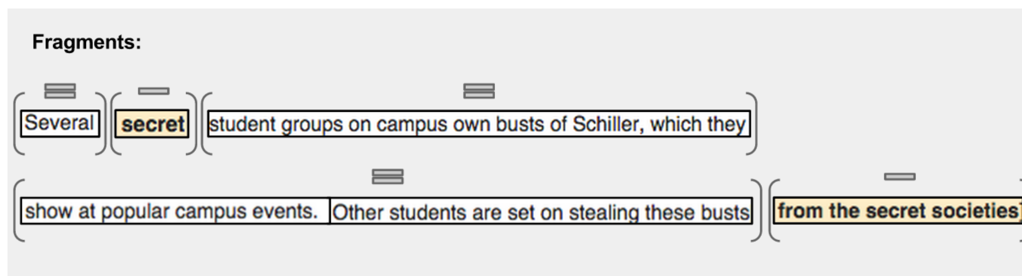

Figure 5.2: Fragment Blocks

As we can see from figure 5.1, we have a revision of a Wikipedia article diffed against an older revision of itself. In this case, the strings "secret" and "from the secret societies" were removed when this article transitioned from the newer state to its older state. wikEd will return a fragment structure that will allow us to understand that this change has occurred between these two revisions of this Wikipedia article. This structure is depicted in figure 5.2.

Figure 5.2 helps us visualize that the fragment list begins with an equal block. This means that the contents of this equal block, "Several", remained the same between the two revisions of this Wikipedia page. Following the first element of fragments, we can see that there was a removal when this article transitioned from its older state to a newer state and the contents of this removal is the string, "secret". We can see this pattern follow through the rest of the fragment structure, which parallels what we see when looking at the diff of these two revisions.

1.3 String Rebuilding

At this point, we've used binary search to locate the most recent affecting revision of the highlighted string. This is the first revision that we will show our WikiSleuth user. Our goal is to display several of the most recent affecting revisions, not just the first one. To achieve this, we continue to search backwards in the article's history. If we run binary search from this point, using the same highlighted string, we will find the very next revision, regardless of whether it affects the string or not. In the example below, we find the most recent revision where the string of interest looks different than "Many".

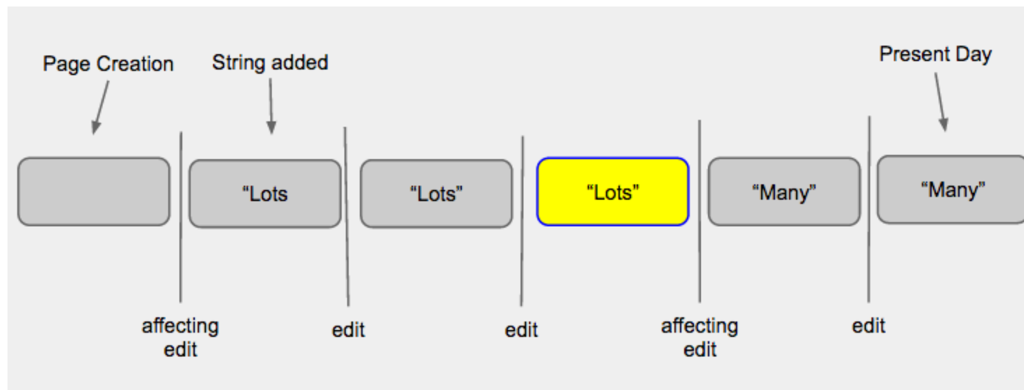

Figure 6: First affecting revision found

If we run binary search from this point, again looking for the most recent revision where the string looks different than "Many", we find the very next revision, which is not correct:

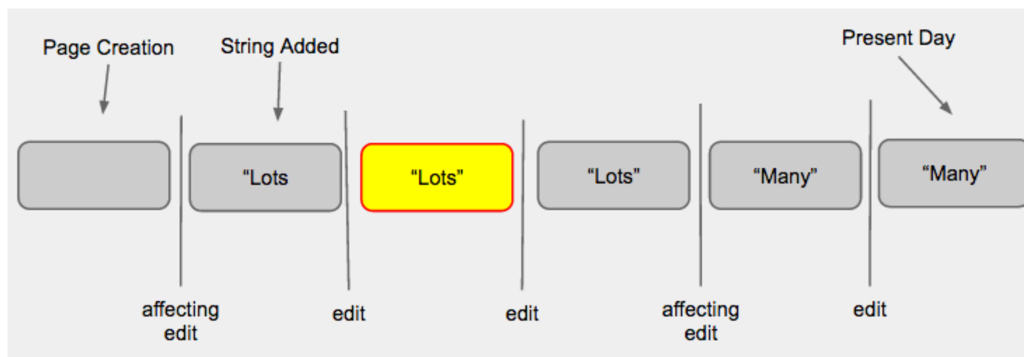

Figure 7: Incorrect second affecting revision

Instead, we want to uses binary search to find the most recent revision where the string looks different than "Lots". First, we need to find exactly what the string "Many" was before the affecting revision that we found. We call this process **string rebuilding**. To rebuild the string prior to a revision, we use the same fragment blocks that help us analyze diffs. In the following example, the highlighted string is "During the last seventeen years of his life". Binary search finds the affecting edit's fragments:

During the last [Deleted: **few**][Added: **seventeen**] years of his life

The equal blocks exist in the highlighted string exactly as they existed in the string prior to edit. The deleted word "few" does not exist in the highlighted string, but was in the string prior to edit. The added word "seventeen" appears in the highlighted string, but was not in the string prior to edit. We go through each block, either adding it to our previous string or not. In this example, we find that the string prior to the affecting edit was "During the last few years of his life".

1.4 N-Ary Search

Once we have both the first revision affecting a highlighted string, and the rebuilt version of that string before the first affecting revision, we can find any number of affecting revisions. We simply run binary search and string rebuilding one after another as many times as we'd like. We first find the most recent revision affecting a highlighted string, and then turn the highlighted string back into the version it looked like before the affecting the revision. Then, finding the first

affecting revision on the previous string is equivalent to finding the second affecting revision on the original highlighted string. We run this loop 10 times, finding the 10 most recent affecting revision. The loop breaks early if we find that the entire highlighted string was added in one revision. In this case, we've isolated the moment that the string was introduced to the article, and there is no more history to find.

2. Heatmap

While the displayed affecting revision feature gives the user detailed data about a small part of the article, the heatmap gives the user general data about the entire article. This is done with a coloring scheme that highlights sentences that were more recently edited with brighter colors and sentences that were edited earlier with darker colors.

The heatmap retrieves information necessary to determine the color of every sentence very similarly to the affected revisions data flow. Each sentence runs iterative binary search through Wikipedia's revision history array. Once the first affected revision is found, the revision id is used to make a MediaWiki API call that returns a time stamp. Using the DOM, we inject a script which changes the colors of specific text within the OuterHTML, this is available through the "document" object in JavaScript.

It quickly became evident that running this process on hundreds of sentences was time inefficient. Thus, we implemented dynamic heatmap generation and multithreading to improve the usability of the heatmap.

2.1 Dynamic Generation

WikiSleuth is meant to be seamless to the user, but gathering so much information appears like a delay to the user. To make the heatmap functional on large pages with many revisions, we made the heatmap generate dynamically. This means that instead of generating the heatmap on the entire page at once, we generate the heatmap sentence by sentence. In other words, with dynamic heatmap generation, each sentence on the page is colored as soon as the timestamp and corresponding color are calculated. This has the advantage of allowing the user to see part of the heatmap without having to wait for the heatmap calculations to be complete for the whole page. To speed the heatmap generation up even further, we used multi-threading.

2.2 Multithreading

The heatmap runs hundreds of lines from Wikipedia articles through iterative binary search and makes API calls on the received information. Multithreading gives us the ability to spread tasks, while still maintaining performance for the user. Before multithreading, WikiSleuth worked with one thread that is on one CPU core. This core handled all of the calculations and calls to run the program, but many computers now have more than one CPU core. Multithreading allows us to access the other CPU cores and assign them tasks that the main core would otherwise have to complete on its own. The tasks are distributed with a JavaScript object called a "Web Worker," which allows us to assign particular functions to a core. The heatmap utilizes the web workers by creating four new threads that run all of the required computations and calls. This increases our performance time signficantly, but we cannot continue to add workers without hindering the user's computer performance. This occurs because of a bottleneck effect, in which, our program begins to run faster but uses too much CPU power.

3. Wikipedia's API

3.1 Definition

An API is an Application Programming Interface. An API allows developers to quickly access raw data in a parseable objects,format from a website. For our project, we utilize MediaWiki, Wikipedia's API.

3.2 MediaWiki Requests

Figure 8: MediaWikiRequest For Last 500 Revisions of Schiller Wiki Page

In general, MediaWiki Requests take the form of URL's, which are split into two parts. The endpoint, provides the location of Media Wiki's API. The action, the second component of the URL request, tells MediaWiki what data we need to use, and in what format in which to send it to our application. For the purposes of our project, we almost always use the "query" action to send queries for certain revisions (which we specify in the parameters), and we request data back in a JSON format by using the "format" parameter available by MediaWIki. Additionally, MediaWiki has several "properties" that allowed us to access specific information like "timestamps". For reference, we used MediaWiki's documentation page[3], and tested the API calls by running them in our Google Chrome Browser.

3.3 MediaWiki Responses

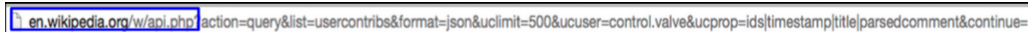

Figure 9: Endpoint of MediaWiki

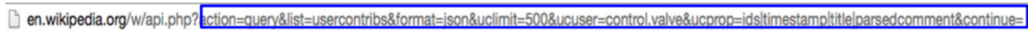

Figure 10: MediaWiki Action

MediaWiki Responses for our project are formatted in JSON, which we parse through using Javascript's built in JSON parser. For the affected revisions pane, we parse through the MediaWiki Response in order to make a revision list to run our binary search algorithm on. This list, a dictionary structure, has a reference to its own revision id, and the id of its parent revision, that is, the revision that occurred just prior to it. For our heatmap, timestamps are translated from ISO format to UTC format.

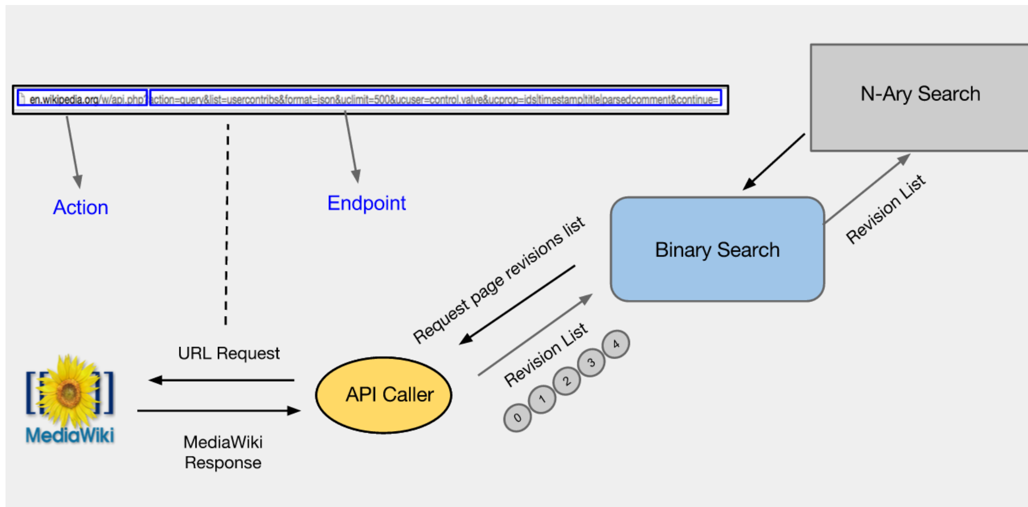

Figure 11: Wikipedia API Flow

4. Software Architecture

The Wikipedia webpage is the view, and sends the highlighted portion of text and URL of the webpage to our master class, which is equivalent to our controller. The controller sends the highlighted string to the core search algorithm, encased in the model, which requests a list of 500 revisions from our MediaWiki Caller class. The parameters of that call are based on the title of the Wikipedia Page. The MediaWIki Caller returns to WikiRev Finder a list of 500 revisions, of which Wiki Rev Finder runs our binary serach algorithm to determine the first affecting revision. Once we have the first affecting revision, we run our string rebuilding algorithm to transform the highlighted string back into the form it had before the first affecting revision. We can then run the binary search algorithm again on this new highlighted string to find the second revision affecting the original highlighted string. We repeat this process until we have up to ten affecting revisions, or until we can no longer find revisions that affect the highlighted string.

Once we have calculated the ten most recent affecting revisions, we send them back to our controller, which converts the list of revisions into a CSS Pane via a helper function. The controller sends the ten most recent revision CSS Pane back to the view, which updates the view.

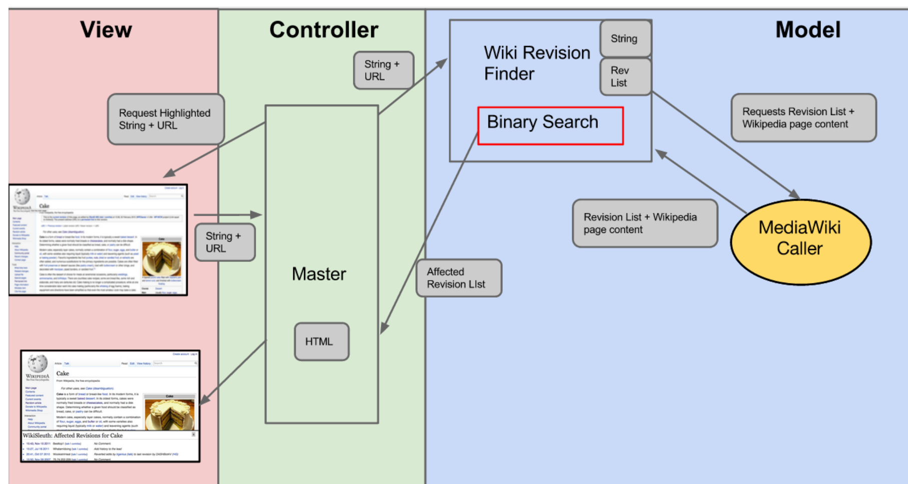

Figure 12: WikiSleuth Software Architecture

5. Evaluation

5.1 Timing Test for Finding Revision

The test was run by finding random Wikipedia articles, retrieving a random sentence within that article, then running our core searching algorithm and recording how long it took to generate the revisions. While this occurs we are collecting the run time for each individual search. For every sentence, we expected to see that the second, third and following revision searches would still run at similar times to the first search. Though, we would expect faster run times in these cases because we would be working with a smaller list.

The test was conducted manually. Wikipedia provides the ability to search for random articles, and the sentences chosen in each article were done as randomly as possible. We selected twenty random articles, and WikiSleuth reported the total search time after each search. After the 20 tests were completed, the average time was calculated. An aspect of this process that could not be controlled was a page's popularity. Our small sample size unintentionally encased many newer topics, which had fairly small pages, and thus the run times did not reflect the actual run time for

the average page. We could have only checked extremely popular topics, and this would have resulted in a much slower average run time then expected.

We decided not to perform automatic tests for a number of reasons. For one, the MediaWiki API does not support retrieving content for a random page in the same way it does retrieving content for a general page. This means that the only way to get the content of a random page via a program is to get all of the HTML at once. We attempted to parse through this HTML using jQuery, but were not successful in getting useful, random sentences from a random article, due to a number of errors that occurred, as well as difficulty integrating the tests with our existing code. Because manual testing would provide results that would be just as accurate, we decided to use a manual testing approach.

5.2 Timing Test for Heatmap Generation

As with the revision testing, programmatic timing testing on the heatmap was not possible given our time frame. We did, however, run a manual timing test for the heatmap. This test consisted of using Wikipedia's random page feature to select 25 random pages. Pages with fewer than 5 sentences were skipped. All of the data was generated on the same machine. While running the heatmap, we recorded the number of revisions on the page, the number of sentences and the time it took for complete heatmap generation.

6. Results

6.1 Affected Revision Test

The functions to find the first revision affecting a highlighted string took on average 1.9 seconds. Finding each additional revision after the first one also took about 1.9 seconds.

6.2 HeatMap Test

The results of the heatmap test are shown in Figure 13. The results are ordered by length of time for heatmap generation starting with the longest. Note that the time is in seconds.

No.	Article Name	Revisions	Sentences	Time
1	Manuel Senante Martinez	59	105	149.9
2	Thunderbirds Machines	418	139	138.3
3	Ken Dryden	804	103	97.79
4	Gun Bound	2348	99	89.44
5	June 2009 Washington Metro Train Collision	752	81	86.49
6	KFSM	221	76	62.08s
7	Demographics of South Dakota	48	66	48.81
8	Mary and Eliza Freeman Houses	84	55	42.5
9	Bermuda Hogges	302	28	42.2
10	Al-Nour Party	354	46	38.92
11	Del Monte Forest	86	52	35.38
12	Lawrie McMenemy	191	26	34.28
13	Always (Blink 182)	273	32	31.59
14	Parklife	546	36	29.91
15	Born To Do It	214	64	21.64
16	I Go Dye	71	25	18.64
17	Alexander Falconbridge	93	22	19.71
18	Musee des Arts Decoratifs, Paris	49	22	17.19
19	Concession (politics)	28	17	8.33
20	I Made a Game with Zombie in it!	86	18	16.12
21	Garelochhead Railway Station	77	17	11.33
22	Wilhelmi Malmivaara	21	23	11.85
23	Lurkmore	109	13	10.87
24	John A. Johnson (politician)	15	9	4.75
25	Corydoras Evelynae	18	10	3.80

Figure 13: Heatmap timing test results

7. Analysis

7.1 Factors Contributing to Running Times

For the core search algorithm (finding the first affecting revision), we believe that the majority of this running time results from the large number of calls made to Wikipedia's API.

One might expect the running time for finding subsequent affecting revisions after the first one to decrease, since we are operating on a smaller list of revisions than we were while searching for the first affecting revision. However, for any time gained by searching a smaller list of revisions, we also lose an equivalent amount of time by rebuilding the highlighted string for each step of the n-ary search.

For the heatmap, the results show that the number of sentences appears to be a more influential factor on the length of generation compared to the number of revisions. Generally, the number of sentences and the length of time are more correlated while the number of revisions tends to be more distributed as the length of time increases. One notable outlier was the page on which it took the longest to generate the heatmap. This suggests there are other significant factors at play. We predict another significant factor is the amount of memory being used on the machine by Chrome or by any other program.

7.2 Progress

Based on our results, we can see that implementing some form of caching would have an immediate, tangible performance boost, especially when running the heatmap. Due to the amount of time we spend pulling data from MediaWiki, any time we can save by cutting out unnecessary, repetitive API calls will result in a performance boost. We implemented a version of this in WikiSleuth, but had to disable it due to errors it caused in the search algorithm.

We also saw a large boost in the performance of the heatmap from multiprocessing. By utilizing more of the computer's processing power at once, we were able to gain dramatic decreases in the running time of the heatmap.

8. Conclusion

WikiSleuth started with the task of finding the history of any piece of text and the editor responsible for its current state or previous development. WikiSleuth has successfully been able to answer this problem, and has been able to give a quick page overview and editor's edit synopsis from one central location. But now that these features were feasible to create, we know that there is more that can be done with the software and the APIs. Some questions that could be interesting and solvable for future WikiSleuth developers include: "Has there been an editing war on this article?", "Can I find a better way to locate bad editors?", or "Can I make WikiSleuth faster and more accurate?". We are happy to say that this project was successful, and we are thankful for the support the Carleton College computer science department provided and especially the guidance from our adviser Jadrian Miles.

References

[1] Knuth, D. E. (1971). Optimum Binary Search Trees. *Acta Informatica*, 1.1 14-25. Web.

[2] Wagner, Robert A., and Fischer, Michael J. (1974). The String-to-String Correction Problem. *Journal of the ACM*, 21.1 168-73. Web.

[3] Wikipedia, "API: Main Page" Wikipedia, Web.

[4] Wikipedia, "User:Cacycle/diff" Wikipedia, Web.