Tiny Search Engine

Justin Garrigus, Philip Rhodes

Github

https://github.com/justinmgarrigus/Tiny-Search-Engine

Motivation

A search engine is a type of program designed to provide a list of documents given a search query. It presents a number of computational challenges across a diverse array of problems including web crawling, indexing, and search query intent classification. This project hopes to explore these areas by creating a tiny search engine that allows users to search for a desired topic and receive an indexed, ranked list of hyperlinks that match this search.

Significance

Search engines have provided the backbone for web navigation since their standardization in the 1990s and continue to maintain a large market share on software development efforts. Be they the indexing force for generalized hyperlinks to web pages in the cases of Google, Bing, and Yahoo, or for videos in the cases of Youtube and Twitch, search engines provide unique challenges for natural language processing from the creation and maintenance of crawlers that gather information from websites to the semantic analysis that ranks and categorizes this information.

Search engines will continue to be an integral part of web interfaces for the foreseeable future, therefore this project provides its members with a foundational insight into such an important field application of NLP.

Objectives

The output of the project should be a single program—written in several different languages including Python, SQL, and C++—that at least supports the following high-level operations:

- Scraping keywords from a given website.
- Indexing a hyperlink in a database alongside its classifiers.
- Allowing the user to input a search query and optionally autocompleting their query.
- Retrieving a ranked list of hyperlinks that match the query.

The goal of this project is not to design a perfect search engine or a program that is usable in everyday life; instead, it serves as an opportunity to learn more about the domain and combining natural language processing with big data.

Features

At a minimum, the search engine should support these features:

- Indexing of a website provided via user input. The user should be able to provide a hyperlink to a website (for instance, “https://en.wikipedia.org/wiki/Natural_language_processing”) or a text file containing a list of websites, to which the program would:
    - Read the plain text from the website.
    - Associate the link with a collection of keywords and classifiers.
    - Store the link and classifiers into a database with a language like SQL.
- Allow the user to input a search query in plain english.
- Offer related link suggestions based on user searches.
    - Probabilistically for autocompletions of searches.
    - Based on semantic similarity for historical searches.
    - Based on the content of documents previously indexed.
- Retrieve a list of websites that the query may apply to.
    - Websites should be ranked by their relevance based on their content, the number of other websites that reference them, and more.

Tiny Search Engine: Increment 2

Justin Garrigus, Philip Rhodes

Introduction

Tiny Search Engine is a project that offers a portable search engine with manual indexing and search suggestions to users. It allows users to input individual hyperlinks to have indexed (or generate a list of hyperlinks to index given different rules) as well as query-based hyperlink retrieval and query-completion. The project is made of three different sections, consisting of *Indexing* (inserting hyperlinks into a database and returning a list of candidate websites based on a string query), *Ranking* (narrowing down of the initial candidate website list and ordering them based on priority), and *Prediction* (taking an incomplete query written by the user in English and offering suggestions on what they can search instead). For the second increment, the project is segmented into different executable files that provide an individual responsibility, as well as a main driver program that combines the functionality of each along with an accessible console-based display to the user. The user is capable of generating a list of hyperlinks from a starting location, indexing each into a database file, retrieving a list of candidate hyperlinks that align with a query provided in plain English (with optional auto-completion suggestions to a partial query), and ranking of candidate links by relevancy to the user's query. The result of our work is an efficient program that yields websites that justifiably relate to a provided query. Additionally, the project contains a wide breadth of test cases that allow end users to demonstrate the accuracy of the project, as well as different debug verbosity levels to ease in debugging.

Background

A search engine is a special program that provides documents to the user based on an English query. The user may type a question, like "what is natural language processing", and a ranked collection of websites will be returned that contain information relevant to the query. Search engines present special challenges in the realm of usability because of the tremendous volume of information required to be indexed for the engine to be usable: an engine which only indexes a dozen websites related to politics will be useless when asked a question about cooking, so an elegant solution needs to be devised to recursively scrape copious numbers of websites in such a way that makes retrieval simple and efficient. This section offers a few informal resources the reader can use to learn about the topic further; research papers presented in a more professional manner can be found in the *References* section at the end of this document.

Search engine *Indexing* is a relatively straightforward problem that involves a lot of different topics, like scraping, crawling, content parsing, database storage/retrieval, query translation, and more (https://moz.com/beginners-guide-to-seo/how-search-engines-operate). Spiders are programs that start at a given website (discovered via manual insertion) and discover more websites throughout the internet (https://www.cloudflare.com/learning/bots/what-is-a-web-crawler/).

Spiders form the theoretical basis of one aspect of our search engine (crawl.py), allowing the user to generate a collection of websites to index given a starting point; while real-world spiders scrape an abundance of metadata alongside target hyperlinks, our implementation maintains a simple design by splitting functionality into several linked independent sub-programs. Once websites are scraped for content and indexed into a database for retrieval, the end user can input a query in plain English, which will map to a list of candidate hyperlinks. *Ranking* then describes the process of giving each website a value based on their relevancy to the user-provided query, scoring the ones that relate strongly to the user's query and intentions very highly, while penalizing those that are only tangentially related or are otherwise untrustworthy (https://www.clickworker.com/content-marketing-glossary/search-engine-ranking/).

Auto-suggestion and auto-completion are important parts of a user experience when using a search engine, using *Prediction* algorithms to determine trailing words to an incomplete English string. Commercial search engines generally keep their algorithms for auto-suggestions a trade secret, often opting for expensive, convoluted, or otherwise obfuscated machine learning approaches from an outside perspective. For this reason, we opted instead to utilize a simplistic trie data structure (https://www.geeksforgeeks.org/trie-insert-and-search/) to help implement the autocomplete process, which efficiently stores and determines suffixes to an arbitrary string. Though the model derived from the autocompletion/prediction papers listed in the *References* section is more simplistic through a trie structure with adjoining CSV dataset, machine learning allows search engines of larger scales to perform as fast and accurately as they do.

Model & Workflow

The concept of a search engine may be ubiquitous, used by billions of people around the world, but the landscape is dominated by only a few large competitors. This makes effective, cohesive solutions to the unique problems presented difficult to perfect in a short timespan, but the design we created provides a practical experience for the user.

Indexing

The Indexing section consists of two main subprograms, each runnable by the user: `crawl.py`, which accumulates a collection of hyperlinks using a breadth-first search from a starting hyperlink, and `index.py`, which handles the storage and retrieval of these inputted hyperlinks. Each of these files are accompanied with an external documentation page created with GitHub's Wiki feature.

Circular Queue

The design the crawling program is centered around two main segments: the `CircularQueue`, which is the data structure used to store each encountered website, and the `process_website` function, which performs the steps necessary to collect a list of hyperlinks from a single source node. A graph of hyperlinks may be represented as Figure 1, where a website can embed a collection of other websites inside its HTML, connecting websites under its own domain and many others.

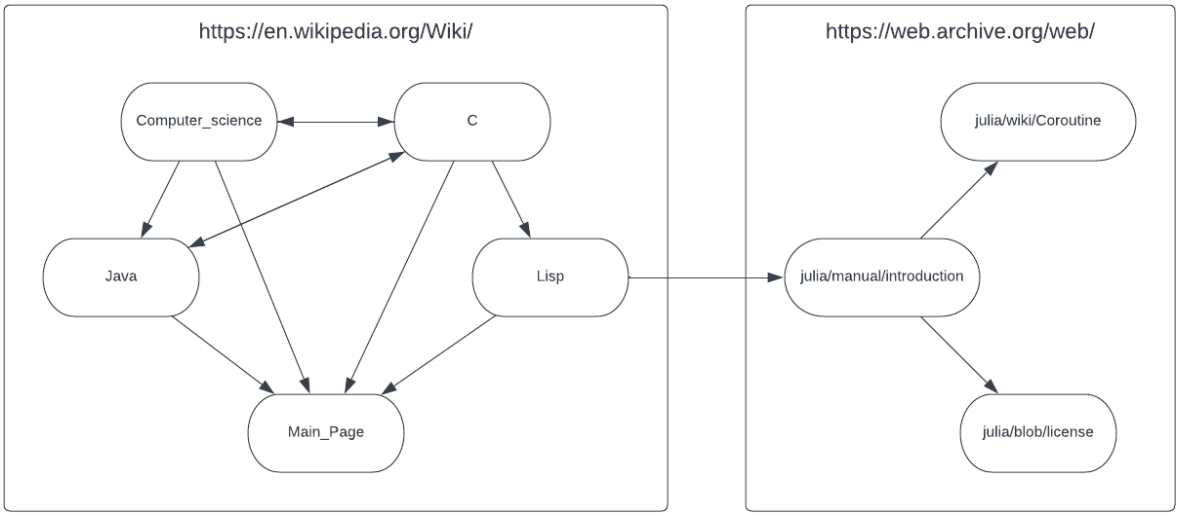

Figure 1: website connections can be visualized as a graph

Websites are collected inside queue-like data structure `CircularQueue`, which similar to a standard circular queue due to it having a fixed size, start index, and end index, but different in the fact the end index never loops back to the front and inserted elements are never overwritten. An example of the queue can be seen in Figure 2. Elements are added to `end_index` but elements are never removed; instead, `start_index` represents the item that was last "processed" by the program.

```
queue = [a, b, c, d, e, f, g, _, _, _, _, _, _, _, _]
           ^                    ^
       start_index          end_index
```

Figure 2: CircularQueue visualization

This example shows a queue that stores hyperlinks, where the queue is initialized with a length of 15 and an initial value of "`a`". The first value processed is "`a`" (advancing `start_index` to the next spot), which uncovered 6 embedded hyperlinks. The next website to be processed is the one located at `start_index`, and any more embedded hyperlinks will be added to `end_index`. This processing loop will continue until either `start_index` equals `end_index` (in which the network of websites is completely discovered) or `end_index` equals the length of the queue (in which the queue is full).

Database

The indexing program itself represents one of the more distinct aspects of the search engine project. While `crawl.py` gathers candidate hyperlinks for the engine to remember, `index.py` does the actual storage and retrieval links at the user's request. The database language chosen for this project was the relational structured query language, SQLite3. The primary reason for this decision

was its support for a collection of familiar commands to SQL programmers, as well as the fact it comes pre-installed with Python3 and allows I/O to a single database file making it attractive for portability reasons. The format of the tables represented are shown in Figure 3: `Website` represents a single website indexed, and `Token` represents a unique stem (of an english word) that appears on that website.

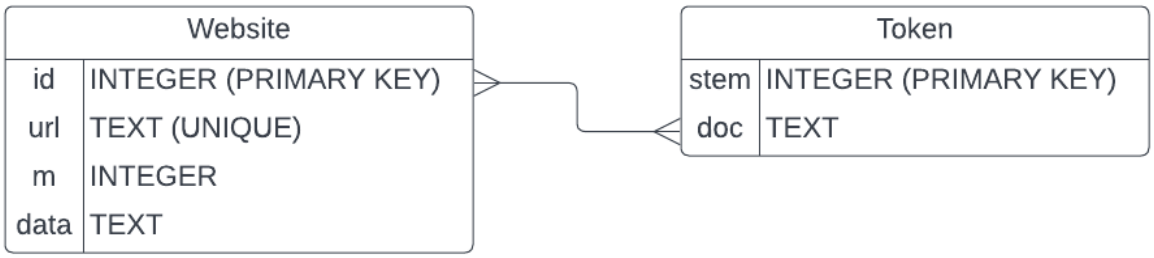

Figure 3: Website and Token tables

A consequence of SQLite3 is the lack of an array data structure, so this required a creative solution to accomplish efficient storage. The goal of the program would be to take an example website (for example, "https://archive.com/list.txt" containing the words "eggs fish bread cereal milk") and add the link to `Website` and each word to `Token`. An inserted token would contain the stem of a string, with each `doc` field representing a list of website `id`'s that contain that stem at least once, internalized as a semicolon-separated list of ints (therefore, the stem "exampl" appearing on websites 1, 2, 5, and 7 would have the `doc` value of "1;2;5;7"). This solution was tested to be marginally quicker than the alternative of creating a new table for each unique stem, making `doc` a `UNIQUE` field, and separating a single semicolon-separated list into many different cells.

Prediction

The auto-suggestion portion of the project was split along the threshold of dataset creation and data set retrieval in `prepSuggestionMaster.py` and `searchSuggestion.py` respectively.

Trie

The class `SearchSuggestion` is an example of a trie data structure. A trie is a special search tree which builds searched words from preceding characters recursively until an `endOfWord` flag is reached. In this case the stored words are the auto-suggestion and auto-completion results generated and stored in `suggestionMaster.csv`. A trie of pre-stored phrases and words is much faster than a more robust or accurate n-gram model for language. Given that the auto-suggest and auto-completion portions of the project are secondary, they should be generated quickly (not taking longer than the time spent searching for websites). Each node contains its data, a list of its child nodes, and the aforementioned boolean `endOfWord` flag.

Searches and inserts begin at the root node. The first prefix datum of the string is then checked, and the search proceeds to the child which contains this first prefix; then the second prefix character is checked, and the search proceeds to that child recursively until a node is met with a true `endOfWord` flag, at which point the desired word is built and returned to a list of results in the case of a search. When insertion is performed, a search is performed like normal until nodes are encountered which do not exist, at which point new nodes are generated and linked with a final true `endOfWord` flag appended. An example state diagram is given below in Figure 4.

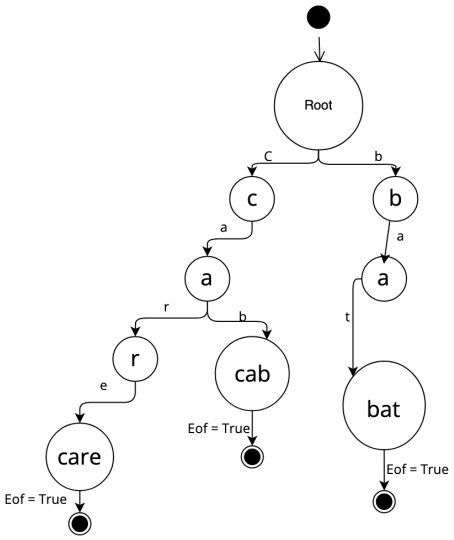

Figure 4: A state diagram of the prefix search process of a trie structure.

prepSuggestionMaster.py

The creation of a readable, compatible data set of auto-suggestions accounts for the bulk of what makes the auto-suggestions usable and partially accurate. The end-user of the search engine will not interact with prepSuggestionMaster directly, as it is strictly an ancillary backend helper function. Firstly, a list of search seeds are created in a text file called `suggestionSeeds.txt`. There are currently 205 seeds in the text file ranging among the categories indexed by the *index* portion of the project. Each of these 205 seeds are elaborated by searching for them with the help of an open source HTTP library called `Requests`. Each seed is elaborated to a list of up to eight search suggestions and autocompletions by grabbing the second argument (a list of suggestions) of the search. This list of up to eight search suggestions is then stored in a larger two-dimensional list of lists. Finally, with the help of the built-in CSV library of Python, this array is written to a CSV called `suggestionMaster.csv`. The CSV file of all of the autosuggestions is bulk loaded into an object of the `SearchSuggestion()` trie at the outset of the search engine process. A sequence diagram is given below of the setup process for the trie data structure in Figure 5.

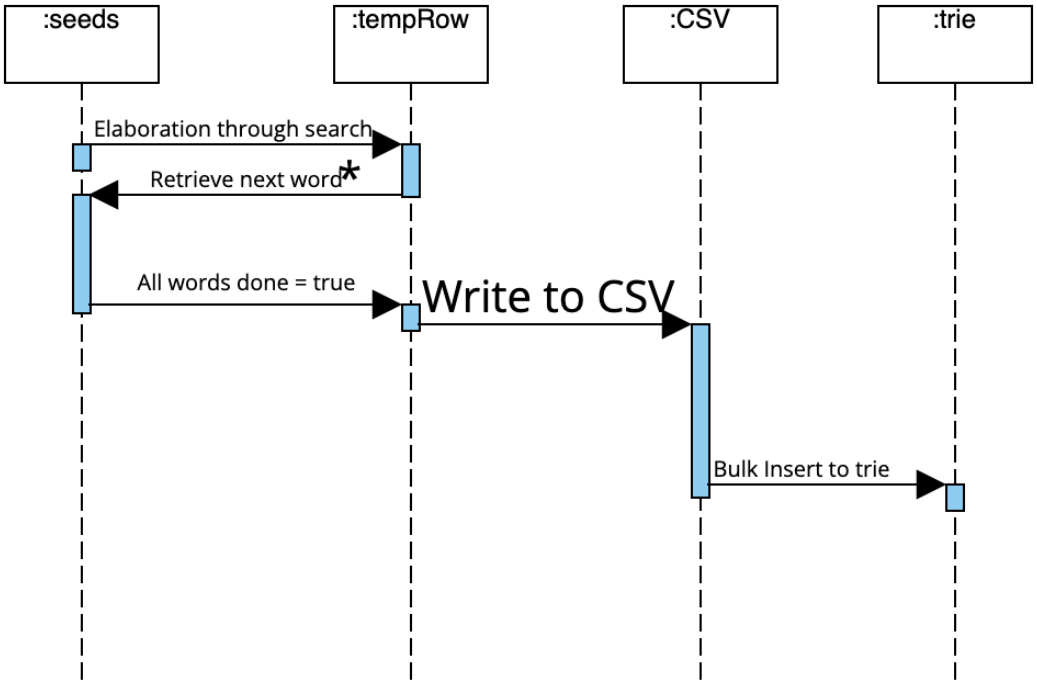

Figure 5: Sequence diagram depicting the iterative elaboration of search seeds into searchable auto-suggestions in the trie.

Little has changed in the direct model of CSV generation, except that time complexity optimizations have been made for this second increment to only append new elaborated search suggestions to the master CSV file rather than outright recreate it at the outset of each search engine instance. This is in the form of the split functions for prep, the explanation of which are elaborated in the *Implementation* section of this report.

A search suggestion driver program was added to `prepSuggestionMaster.py` was created to integrate with the indexing portion of the project, which has real time contingencies for unseen searches which have no pre-stored autocomplete suggestions. In addition, it normalizes these searches into lowercase forms for the purpose of creating more possibilities for each search elaboration.

Since the auto-suggest and auto-complete feature is secondary to the actual searches themselves, time constraints had to be taken into consideration as to not impede the flow of a user's experience. At the current scale of the database, the initialization of the auto-suggestion CSV occurs in $\theta(n^2)$ time since it is singly nested for loop, which is acceptable since it is part of the initialization process of the program and does not occur between actual searches. The largest time consideration and requirement occurs when searching and generating the list of auto-suggestion in real time during user use for searches. For this reason, the more efficient prefix search trie was used to store the autosuggestions during runtime. A trie provides a search and insertion worst case time complexity of `O(n) = O(lenKey)`, where `lenKey` is the length of the searched phrase. Given that most

searches are under 20 to 30 characters, the list generation is well within the time requirements outlined above.

Space requirements are less of a concern for the project, especially since the data is stored in plain text. A trie data structure has a space complexity of `O(n) = O(alpha*lenKey*numOfKeys)`, where `alpha` is the alphabet size (here alphanumeric characters), `lenKey` is the length of the keys (maximum length stored), and `numOfKeys` is the number of stored phrases (same as the number of elements in the CSV).

Dataset

Multiple datasets were used during the making of each section. These datasets either came from an external location and needed to be downloaded, or were generated dynamically as the program was run.

Indexing

The primary datasets that the Indexing section maintain are the `Website` and `Token` databases. When a single website is input into the program via the command-line, a collection of data is scraped from the website's HTML and stored in the SQLite3 database. Additionally, the indexer utilizes different text-parsing methods to prepare scraped content before it is input into a database, including `nltk`'s `stopwords` collection, `nltk`'s `PorterStemmer`, regular-expression matching, and `BeautifulSoup`'s html parser.

Additionally, a collection of websites were organized into text files for debugging and developmental purposes throughout the project. These text files were generated by the crawler program crawl.py, and are stored into the data folder under the following names:

- `wiki-compsci.txt`: 1000 websites describing computer-science terms, the history of computation, and domain-specific knowledge about the field, starting from "https://en.wikipedia.org/wiki/Computer_science"
- `cnn-news.txt`: 1000 websites documenting current events and political discussions, starting from "https://www.cnn.com/"
- `cnn-small.txt`: A subset of 5 websites taken from `cnn-news.txt`.
- `varied-small.txt`: 5 websites from a diverse range of sources, spanning topics such as geology, entertainment, computer science, and current events.

Prediction

The prediction dataset is split into two central files: a text file to store the "seeds" of the autosuggestions called `suggestionSeeds.txt`, and a fleshed-out master CSV storing all of the currently available auto-suggestions named `suggestionMaster.csv`.

suggestionSeeds.txt

The “seeds” of suggestionSeeds refer to individual lines of topics and partial searches contained linearly on separate lines in the text file. Currently, the file has 205 lines of seeds ranging from the same topics as the headers of many of the indexed hyperlinks found in the data sets of the indexing portion of the project. These seeds are the basis for creating and fleshing out the primary dataset for the `searchSuggestion` object by feeding them through a function called `prepSuggestionMaster`, which takes each of the lines from the seed text file and uses the HTTP request library `requests` paired with the `json` library to search for the seed and store its eight most likely autosuggestions in a one-dimensional list. The searches for which suggestion list is desired are appended to the list of search seeds to grow the data set alongside frequency of use.

lastSeed.txt

To further optimize the generation of the suggestion CSV, a function was created to only append the search suggestions added since the last instance of the search engine. When a search desires a list of auto-suggestion completions, that search is normalized and added to the `suggestionSeeds.txt`. At the start of each search engine instance, as a preprocessing behavior on the data, the cached search and line number in the `lastSeed.txt` file in the form of “`search\n linenumber`” is used as the starting index for appending new elaborated suggestions to the CSV, which is then used to populate the trie. Populating the trie happens in $\Theta(n^2)$ time, and cannot be unrolled anymore than it already has been unrolled.

suggestionMaster.csv

The one dimensional lists are then written using the `csv` library built into Python to create and maintain a two-dimensional list of lists called `suggestionMaster.csv` of all of the currently indexed autosuggestions to be passed to the search suggestion trie at run-time instantiation of the driving program of the search engine process. The algorithm for the insertion of the entire CSV currently runs in $O(n^2)$ time due to its nested loop to generate each search row, making it costly from a time perspective, but it only executes once to initialize the search suggestion object at the outset of the process. It is the largest precondition for the search suggestion feature. The suggestion matrix is not to be interacted with directly by searches, its generation and upkeep is the task of `resetPrepSuggestionMaster()` and `appendPrepSuggestionMaster()`.

Analysis of Data

This section goes into more detail on the datasets from a computational perspective, providing a thorough analysis of the preprocessing that is performed before data is inserted and postprocessing after data is retrieved.

Indexing

The Indexing section is one which considers the benefits of parallelization in design. The task of indexing is largely compartmentalized, and follows a sequential approach with explicit periods of downtime shown in Figure 6.

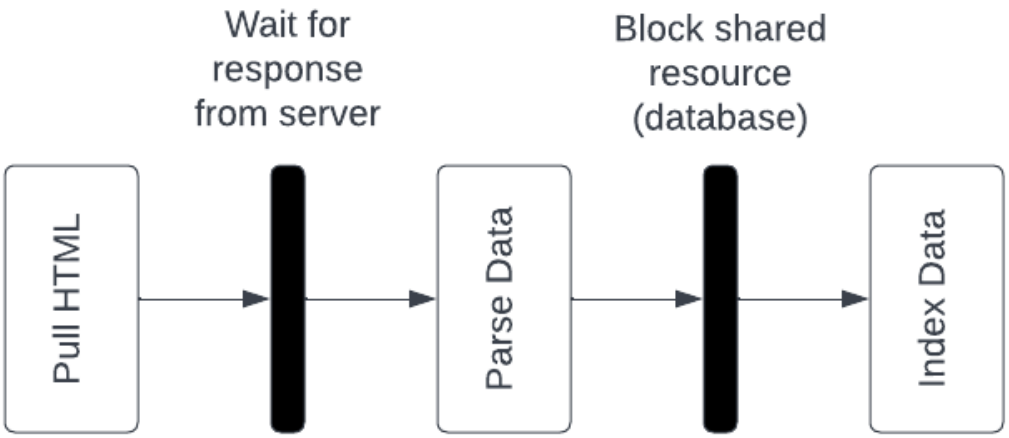

Figure 6: Sequence diagram depicting periods of downtime where the network waits for resources to be obtained

Different threads can be executing the “Pull HTML” and “Parse Data” sections at the same time, as well as the “Wait for response from the server section” which is dependent on an external server (which may take anywhere from an instant to a few seconds to respond). The “Block shared resource” and “Index Data” sections are those which can only be performed one at a time, since it involves writing to and reading data from a database. If this program was performed serially, a significant amount of time would be lost if a series of websites took a long amount of time to return responses. Alternatively, if the program was performed in parallel with a collection of *n* threads, then each of these threads could send requests to different servers and service these requests as they receive responses. This further demonstrates optimizations when provided orphan links that do not point to a server willing to service requests; instead of delaying all operations until a timeout is received, other requests can be performed in the background at the same time ensuring maximum throughput.

Hyperlinks

There are no strict rules for the type of text that can go in an a html tag, so a validation process must be performed before it is added to the hyperlink `CircularQueue`. In our case, links like “http://www.acm.org/” and “https://api.semanticscholar.org/CorpusID:3023076” are valid because they are fully qualified and contain all the necessary information to direct a web browser to an HTML page, but other links like “#cite_note-14”, “None”, and

“/wiki/Natural_language_processing” can also be retrieved from an a tag. For this reason, these requirements are set:

- Hyperlinks must be at least “partially valid” by being non-null, having a length greater than 0, not beginning with a “#”, and not containing a “<” or double-quote.
- The website’s domain is prepended to the link if it does not contain a “//” (where, for example, the domain of “https://en.wikipedia.org/wiki/Computer_science” is “https://en.wikipedia.org/”). Wikipedia articles especially violate this condition due to inter-domain redirects (i.e., articles containing links to other Wikipedia articles) omitting the domain. This condition transforms “bare” links into “fully qualifies” links, translating inter-domain redirects to global-domain hyperlinks.
- The user can optionally specify a flag to the program to ping hyperlinks before they added for extra security (`--validate`), which eliminates any dead links at the expense of additional execution time.

Prediction

The bulk of preprocessing for the data occurs in the structure below in Figure 7, which details the elaboration of suggestion seeds into the fully archived suggestion master CSV file.

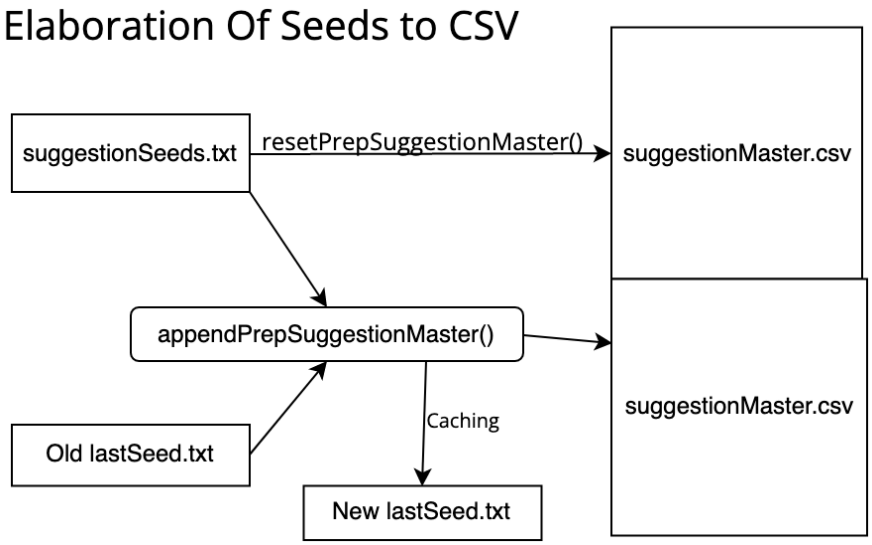

Figure 7: Diagram of the process of caching and generating the suggestion CSV from seeds.

At the start of most instances of the search engine, running the `appendSearchSuggestionMaster()` function suffices for updating the contents of the trie. For total resets or updates of the CSV, use `resetSearchSuggestionMaster()` found in `prepSuggestionMaster.py`. The only other preprocessing utilized is the case-normalization of each of the suggestion seeds. Implementing all of the searches into its lowercase form allows each potential search autocompletion to have more suggestions due to the trie structure treating different cased English letters as unique characters. The indexing portion of the project preprocess

involves removing the suffixed '?' character from the end of a search which desires elaboration. The below graph model of the prediction data shows the linear chain of data dependencies used in maintaining the trie structure. The data are all completely determined by the search suggestion seeds found in `suggestionSeeds.txt`, which initializes the cascading data flow.

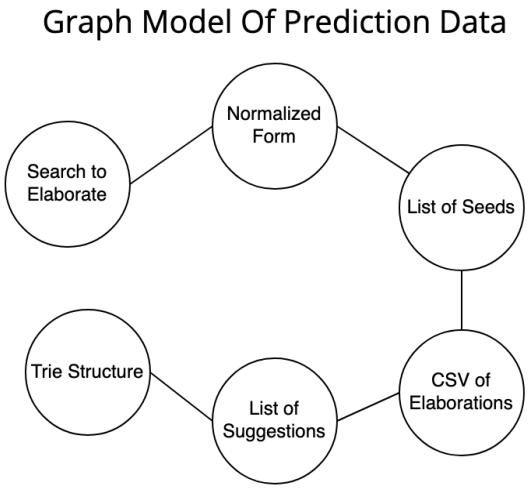

Figure 8: Graph model of data dependencies in prediction elements.

Implementation

After design work was completed for the project, we each programmed our part using our models and interfaces as blueprints. We chose Python to implement the majority of the project due to its ease of writing and abundancies of NLP modules, as well as Python's ability to be executed without needing to be compiled.

Indexing

Functionality for the scraper was condensed into a single wrapper function named `process_website`, whose goal is to perform the scraping operation for a single website. The main driver function will invoke this repeatedly until either of the queue's two exit conditions are met (the queue is full or no more websites can be indexed). Websites are pulled with `urllib` (which initiates a request with the website's server and pulls an HTML page from it) and parsed with `BeautifulSoup` (which applies a set of syntactical rules to extract meaningful information from the text). In `crawl.py`, the only output of an execution is a collection of hyperlinks, so actual in-depth scraping of website content is reserved for the `index.py` file; in this case, the only tagged content searched is for the "`a`" html tag which specifies an embedded hyperlink. The program iterates through each website obtained from this loop and checks for validity, inserting it to the queue if it is considered valid, which is determined by the set of rules elaborated previously.

SQLite3 is syntactically identical to other popular relational database languages so storage and retrieval is performed through standard SELECT, INSERT, and UPDATE calls. The routine for storage inside of `index.py` begins relatively similar to the `process_website` routine in `crawl.py`; a request is sent to a given website domain for a particular HTML file which `BeautifulSoup` parses, but the entirety of the plaintext is read instead of just the hyperlink "`a`" tags. Although regardless of the parser used, HTML is a very verbose language with no singular clear way of representing text so a huge amount of metadata is retrieved along with the ideal text. This is ignored in the current version of the program; certain stems may be indexed for all input websites (like the stem "ref" which appears in every HTML document, causing a user query of "ref" to return every website), but it gives acceptable results in most cases. Regardless, "words" are identified in the parsed html based off of connected stretches of alphabetic characters since `BeautifulSoup` does not have a utility for retrieving a list of individual words; the text is fed through a simple regex "`[a-zA-Z]+`" and each match is interpreted as a single word.

Retrieval

The final phase of the index.py program can be visualized as a reverse of the previous section, where the user's input query (an English string of characters like "what is the fastest car in the world") is scraped for stems, and each stem is SELECT-ed in the `Token` table for `doc`'s that contain the stem. In order to narrow down the search results slightly, stopwords are also removed from the query (so that the previous example query becomes "fastest car world"). The final result of the program is a semicolon-separated list of unique website ID's, representing the websites that contain one or more stems from the user query. It should be noted that a sufficiently large database consisting of tens of thousands of indexes may return hundreds of websites for a given user query; it is up to the *Ranking* section to narrow down this list and determine which specific websites most closely apply to the user's intentions.

Prediction

All program files were implemented with Python due to its ease of testing and large swaths of NLP modules already available. Further details of the implementation can be seen in the Github linked above under `searchSuggestion.py` and `prepSuggestionMaster.py`. The requests library (https://pypi.org/project/requests/) was an essential open source import which allowed for the elaboration through search pulling, and the CSV library likewise streamlined the dataset creation process. The implementation of the search trie in `searchSuggestion` was modeled after the base algorithm with some additional dependencies and text processing like stripping whitespace and normalizing test case for preprocessing.

The interface for adding to the trie is explored in the demo video provided, but is accomplished through three commands:

- `trie.insert(string)`
- `trie.batch_insert([list of strings])`
- `trie.search(string)` => returns a list of possible suggestions from the trie.

The other group of functions for the project involve the maintenance and preprocessing of the CSV containing all of the autocompletions for any given search string for the indexed data. Improvements for Increment 2 of this report were done to allow this initialization process to happen in $\Theta(n)$ time rather than in $\Theta(n^2)$ time. This was done by unrolling the nested for-loop in `prepSuggestionMaster()` originally in the form of:

```
1   for-each seed in suggestionSeeds:
2       elaborate
3       for-each elaboration in elaborations:
4           append to list
5       append list as row to CSV
```

Pseudocode 1

This nested loop was modified into the new form which takes advantage of a temporary variable for each row, preprocessed before hand into a list, so that the second for loop is not required to be nest, but in line with the mother for loop:

```
1   for-each seed in suggestionSeeds:
2       elaborate
3       append list as row to CSV
```

Pseudocode 2

This process of loop unrolling greatly improved the time performance during the initialization process. The second large improvement to performance during initialization was the introduction of a separate `appendPrepSuggestionMaster()`, which treats the CSV as a semi-static device rather than completely replacing it each time a new search engine process is created. Requiring a text file with the cached final row at the last initialization process of the search engine, the CSV has new rows of search suggestions appended to it rather than having an entirely new CSV created:

```
1   open(lastSeed.txt)
2   read(lastSeed.txt)
3   lastplace = lastSeed[1]
4   if lastplace != actualLastPlace:
5       nextlastplace = actualLastPlace
6       while lastplace != actualLastPlace:
7           elaborate
8           append list as row to CSV
9           lastplace++
10  write(lastSeed.txt).(nextlastplace)
```

Pseudocode 3

Finally, the last set of changes to the prediction implementation were done to create an integration driver function for the search suggestion system called `searchSuggestionDriver`, which

accepts a string which requires search suggestion autocompletions. The search is then normalized and searched for. If a readily available list of suggestions has not been stored already, suggestions are elaborated in real time. The pseudocode for this driver program is below in Pseudocode 4.

```
1    searchSuggestionDriver(searchString)
2    normalize(searchString)
3    searchSuggest(searchString)
4    if suggestions are empty:
5         elaborate search
6         cache search to suggestion seeds for the future.
7    return ListOfSuggestions
```

Pseudocode 4

Results

After programming the requirements, connecting the interfaces to each other, and testing the result for accuracy, the current program displays an adequate level of success in generating hyperlinks for the user, retrieving hyperlinks based on a query, and suggesting completions to the user's query. The program is entirely usable in its current state and shows promising results for the final submission.

Indexing

The *Indexing* section has sufficiently implemented the two main subprograms. Due to the faithfulness of the program adhering to the design specified, there was no ambiguity on the accuracy or efficiency of the algorithms, so testing was the primary cause for concern in guaranteeing the implementation followed the design specifications. The two programs (`crawl.py` and `index.py`) can be run via command-line with the following commands:

- `python3 crawl.py <start> <limit> [-o <output>] [-t <timeout>] [--verbose] [--validate]`
- `python3 index.py {-w <website> | -q <query> | -d <document>} [-t <timeout>] [--verbose]`

Different variables shown above trigger different events from happening, like the `--validate` flag which requires websites be pinged to verify they work before they are added to the hyperlink queue. Additionally, while `index.py` presents multiple different execution paths (either insertion into the database or retrieval from the database), the path taken can be triggered by the presence of either `-w`, `-q`, or `-d`. This introduces some concern about the accuracy of command-line arguments, so a focus was placed on removing ambiguity from the command-line and providing clear expectations to the user for what will occur when different arguments are passed, accomplished with the help of ample testing.

For `crawl.py`, it supports the generation of a list of hyperlinks given a user's input query: after using a command like "`python3 crawl.py https://en.wikipedia.org/wiki/Computer_science 1000 -o csci.txt --validate`", this generates a newline-separated list of 1000 hyperlinks starting from the Computer Science Wikipedia page placed in the "`csci.txt`" file, with each discovered hyperlink first pinged before being added to the output.

```
1   $ python3 -m unittest src/index.py -v -b
2   test_cla_00_no_args (src.index.CommandLineArgumentTest) ... ok
3   test_cla_01_no_main_direction (src.index.CommandLineArgumentTest) ... ok
4   test_cla_02_multiple_main_direction (src.index.CommandLineArgumentTest) ... ok
5   test_cla_03_unknown_single_option_1 (src.index.CommandLineArgumentTest) ... ok
6   test_cla_04_unknown_single_option_2 (src.index.CommandLineArgumentTest) ... ok
7   test_cla_05_unknown_double_option (src.index.CommandLineArgumentTest) ... ok
8   test_cla_06_missing_document (src.index.CommandLineArgumentTest) ... ok
9   test_cla_07_missing_database (src.index.CommandLineArgumentTest) ... ok
10  test_cla_08_timeout_not_numeric (src.index.CommandLineArgumentTest) ... ok
11  test_cla_09_timeout_negative (src.index.CommandLineArgumentTest) ... ok
12  test_cla_10_multiple_website_specified (src.index.CommandLineArgumentTest) ...
13  ok
14  test_cla_11_multiple_query_specified (src.index.CommandLineArgumentTest) ...
15  ok
16  test_cla_12_multiple_docs_specified (src.index.CommandLineArgumentTest) ... ok
17  test_cla_13_multiple_db_specified (src.index.CommandLineArgumentTest) ... ok
18  test_cla_14_multiple_timeout_specified (src.index.CommandLineArgumentTest) ...
19  ok
20  test_cla_15_verbose_misspelled (src.index.CommandLineArgumentTest) ... ok
21  test_document_database_increase (src.index.UniqueTests) ... ok
22  test_document_query (src.index.UniqueTests) ... ok
23  test_document_rank_multi (src.index.UniqueTests) ... ok
24  test_document_rank_single (src.index.UniqueTests) ... ok
25  test_index_document_multi (src.index.UniqueTests) ... ok
26  test_null_database (src.index.UniqueTests) ... ok
27  test_timeout (src.index.UniqueTests) ... ok
28  test_website_database_increase (src.index.UniqueTests) ... ok
29
30  ----------------------------------------------------------------------
    Ran 24 tests in 1.486s

    OK
```

Figure 9: Execution of crawl.py

For `index.py`, it accurately supports the two main requirements for implementation: insertion of hyperlinks into the database (where a command like "`python3 index.py -w csci.txt`" adds every hyperlink from the `csci.txt` file into the database) and querying from the database (where the command "`python3 index.py -q 'what is natural language processing'`" returns a semicolon-separated list of values representing indexes pointing to hyperlinks that reflect the content of the website). Insertion can be seen in Figure 10:

```
1   $ python3 src/index.py -w data/links/varied-small.txt --verbose
2   Reading "https://www.cnn.com" ... Obtained
3   Inserting 2523 stems into data/table.db ... Done
4   Reading "https://www.polygon.com" ... Obtained
```

```
Inserting 946 stems into data/table.db ... Done
Reading "https://en.wikipedia.org" ... Obtained
Inserting 2016 stems into data/table.db ... Done
Reading "https://www.trains.com" ... Obtained
Inserting 992 stems into data/table.db ... Done
Reading "https://www.dictionary.com" ... Obtained
Inserting 1318 stems into data/table.db ... Done
Table(website) size: 5
Table(token) size: 5644
```

Figure 10: Execution of index.py -w

Ranking is performed with a user query. The query is preprocessed the same way as websites are (tokenized and stemmed), and each resulting stem performs a `SELECT` into the database to obtain a list of website `id`'s that contain at least one of the words in the query. The resulting collection of websites are then each iterated through and scored based on a tf-idf metric:

```
website_count = SELECT COUNT OF websites
for-each website in query_results:
    token_dict, m = SELECT token_dict, m FROM website
     term_sum = 0
     for-each token, freq in website:
     if token in query:
          term_count = SELECT term_count FROM tokens
          tf  = freq / m
          idf = log2(website_count / term_count) + 1
          term_sum += tf * idf
return top 10 items from websites.sort(best term_sum)
```

Pseudocode 5

After indexing, retrieval and subsequent ranking can occur, seen below:

```
python3 src/index.py -q 'what are mountains' --verbose
4
[0.443] https://www.dictionary.com/browse/mountain

python3 src/index.py -q 'how many computers are there' --verbose
2;4
[3.372] https://en.wikipedia.org/wiki/Computer_science
[0.006] https://www.dictionary.com/browse/mountain
```

Figure 11: Execution of index.py -q

In order to demonstrate the accuracy of the subprograms, each portion comes equipped with a wide breadth of tests that sufficiently cover every error, conditional, and path the program can take. The testing library used was unittest, which comes pre-installed by default on Python3. This offers a tremendous level of control to the user and automates a lot of the testing process through the use of cookie-cutter function templates. Tests can be run in bulk with "`python3 -m unittest -s src -v -b`", and the current version on GitHub accurately executes each of its tests. The tests

included range from simple argument variations (to ensure no end user can crash the program with an invalid parameter) to more complex logical tests (which includes testing that websites are added properly to the queue, that the correct websites are retrieved in specific documents, that websites are scraped the correct stems they should contain and indexed correctly, and that websites are ranked in the correct order based on relevancy to a supplied test query). The program passes each of these tests, ensuring that the program is not only capable of indexing, retrieving, and ranking websites, but also readily available to demonstrate its ability to do so with a single command.

```
$ python3 -m unittest src/index.py -v -b
test_cla_00_no_args (src.index.CommandLineArgumentTest) ... ok
test_cla_01_no_main_direction (src.index.CommandLineArgumentTest) ... ok
test_cla_02_multiple_main_direction (src.index.CommandLineArgumentTest) ... ok
test_cla_03_unknown_single_option_1 (src.index.CommandLineArgumentTest) ... ok
test_cla_04_unknown_single_option_2 (src.index.CommandLineArgumentTest) ... ok
test_cla_05_unknown_double_option (src.index.CommandLineArgumentTest) ... ok
test_cla_06_missing_document (src.index.CommandLineArgumentTest) ... ok
test_cla_07_missing_database (src.index.CommandLineArgumentTest) ... ok
test_cla_08_timeout_not_numeric (src.index.CommandLineArgumentTest) ... ok
test_cla_09_timeout_negative (src.index.CommandLineArgumentTest) ... ok
test_cla_10_multiple_website_specified (src.index.CommandLineArgumentTest) ...
ok
test_cla_11_multiple_query_specified (src.index.CommandLineArgumentTest) ...
ok
test_cla_12_multiple_docs_specified (src.index.CommandLineArgumentTest) ... ok
test_cla_13_multiple_db_specified (src.index.CommandLineArgumentTest) ... ok
test_cla_14_multiple_timeout_specified (src.index.CommandLineArgumentTest) ...
ok
test_cla_15_verbose_misspelled (src.index.CommandLineArgumentTest) ... ok
test_document_database_increase (src.index.UniqueTests) ... ok
test_document_query (src.index.UniqueTests) ... ok
test_document_rank_multi (src.index.UniqueTests) ... ok
test_document_rank_single (src.index.UniqueTests) ... ok
test_index_document_multi (src.index.UniqueTests) ... ok
test_null_database (src.index.UniqueTests) ... ok
test_timeout (src.index.UniqueTests) ... ok
test_website_database_increase (src.index.UniqueTests) ... ok

----------------------------------------------------------------------
Ran 24 tests in 1.486s

OK
```

Figure 12: Execution of the test suite for index.py

Prediction

When a user inputs a search with a suffixed '?' question mark character, the preceding search string is passed to `SearchSuggestion.search(precedingString)`, and a list of possible auto-suggestions is returned. From the search menu, the user can then choose one from this list of possible suggestions and autocompletions. Testing was done for empty strings and exceedingly long string outliers with no major execution errors in search and insertion time for the trie. An example of the output of a given search suggestion can be found below in Figure 13.

```
suggestions = ss.search('news')
for alternates in suggestions:
  print(alternates)
```

```
news today
news google
news nation
news 9
news history search
news history today
news history apple
news without opinions
news page with opinions
news article with opinions and assertions
```

Figure 13: Example output of a search suggestion

Several time complexity improvements were made between the work of increment 1 and increment 2 of this report, as can be seen in the following succession of figures. Figure 14 shows the previous implementation of the sole function `prepSuggestionMaster()`'s run time for prepping an entire CSV from scratch during initialization. Its $\Theta(n^2)$ time complexity was a serious hindrance on the performance of the search engine during process initialization. It ran on average in 30 seconds for the storage of 208 elaborated search suggestion seeds.

```
start = time.time()
prepSuggestionMaster()
end = time.time()
print("Time: ",end - start)
```

```
Time:  29.556596994400024
```

Figure 14: Slow processing of data with initial design in $\Theta(n^2)$ time complexity.

To alleviate this, the nested for-loop found in prepSuggestionMaster() was unrolled using dynamic programming techniques—essentially caching each row as a separate temp list—to improve its time complexity to $\Theta(n)$. On an even longer list of seeds (215), the new resetPrepSuggestionMaster() runs on average in less than 10 seconds, which is a run time reduction of over 66% in most cases. The results of this reduction can be seen in Figure 15.

```
start = time.time()
resetPrepSuggestionMaster()
end = time.time()
print("Time: ",end - start)

Time:  10.062649726867676
```

Figure 15: Speeding up performance through loop unrolling of initial design to produce Θ(n) time complexity.

Further reductions in speed were made by assuming that the CSV itself as a static object, rather than one which needed to be reinitialized with every new search engine process. Thus the use of the cached lastSeed.txt was pivotal in creating a cache based implementation of seed elaboration to append to the master CSV file. Unless there is an extremely large bulk data appended during the lifespan of a search engine process, this append function performs much faster in Θ(n) time complexity, but with an average much smaller n than resetPrepSuggestionMaster(). Figure 16 shows that this append normally happens in the order of milliseconds to deciseconds.

```
start = time.time()
appendPrepSuggestionMaster()
end = time.time()
print("Time: ",end - start)

best ways program
[]


Time:  0.0013957023620605469
```

Figure 16: Speeding up performance even further through caching and assumption of a semi static CSV.

The average time for a batch of 10 test searches is 0.032 seconds or 3.2 centiseconds per search to provide a list of suggestions. This is well within the acceptable limit agreed upon at the outset of this project. Half of the searches in the test batch do not have search suggestion seeds, and require elaboration through contingency measures, which is the largest contributor to a longer processing time. Figure 17 below shows these test search times. Lastly, Figure 18 below shows the time for an average run of populating the trie structure using the master CSV file. The trie structure prep happens currently—at the scale of the engine at the time of this report— in under a second on average.

```
for test in testBatch:
  suggestionTimeTest(test)
```

```
Time:  0.057304859161376 95
Time:  0.034968376159667 97
Time:  0.043416976928710 94
Time:  0.036957502365112 305
Time:  0.001067876815795 8984
Time:  0.053053140640258 79
Time:  0.000131130218505 85938
```

Figure 17: Times for providing search suggestions for a batch of searches, half of which have never been encountered.

```
start = time.time()
ss = SearchSuggestion()
with open('/content/suggestionMaster.csv') as file_obj:
      readCsv = csv.reader(file_obj)
      for row in readCsv:
        ss.batch_insert(row)
end = time.time()
print("Time: ",end - start)
```

```
Time:  0.07348775863647461
```

Figure 18: Time to completely prep the Trie data structure.

Project Management

In its current form, the project is entirely usable as a functional search engine. It does not perform perfectly due to the simplicity of the ranking method, but it yields sufficiently accurate results at a moderate delay even when thousands of websites are indexed. For these reasons, the project can be considered a resounding success. Below, a detailed description of each group member's work is provided, along with a percentage representation of how much their part constitutes the final project.

Justin Garrigus (Indexing and Ranking): 75% of the Project

These sections constitute the main functionality of the search engine concept. Everything that was planned to be completed has been completed successfully to an extremely positive degree; the results display excellent progress has been made in developing a successful search engine. Usage of

the code can also be implemented both directly in Python by linking the modules, or indirectly by utilizing the command-line interface. Additionally, the entire program comes equipped with a very thorough testing suite that guarantees the code to be error-free for any range of syntax errors for passed arguments or logical errors inside the functions.

Philip Rhodes (Prediction): 25% of the Project

All of the previous work for the search suggestion and autocomplete system still stands, but it has been optimized for both space and time complexity. The recursive solution suggested in the work to be complete was ultimately unnecessary and provided an inadequately slow solution to the problem of unseen searches to the suggestion system. Likewise, a machine learning approach produces inhuman answers despite being trained and tested on datasets of suggestions, and it is much slower than a simple trie structure for the scale of this project. Instead, a system of real time search suggestion generation was preferred for its small time and space requirements.

As seen above, the old `prepSuggestionMaster()` program was optimized from a $\Theta(n^2)$ time complexity to two new functions, called `resetPrepSuggestionMaster()` and `appendPrepSuggestionMaster()`. Both of these have been optimized to run in $\Theta(n)$ and $\Theta(b)$ time complexities respectively, where `n` is the number of all of the search suggestion seeds found in the master text file, and `b` is the number of new search suggestions added during the laster instance which is computed using `n - a`, where `a` is the line number of the cached suggestion seed in the cache file `lastSeed.txt`. This is a marked improvement in the initialization time for the CSV file, even at this scale of a project. As seen in the analysis section above, the time to generate the CSV went from about twenty to thirty seconds on average, to an average of ten seconds for a total reset and an average of one millisecond for append operations. The append function makes use of a cached line number and the last known search suggestion seed to grow the CSV accordingly. If for some reason, the CSV is lost, one must only run the reset function outlined above.

No marked changes were made to `searchSuggestion.py`, but in addition to the optimizations of `prepSuggestionMaster.py`, a third function was added as the driver for the search suggestion process itself. It calls a pre-created search suggestion object named `ss`, then normalizes the passed string to all lowercase, then performs the trie search. If the search suggestions are empty or the search is unseen without close enough suggestions, the program adapts by providing search suggestions in real time, then adding them to the suggestion seed text file. All of the source code for the prediction function can be found in `searchSuggestion.py` and `prepSuggestionMaster.py`. Everything has been tested thoroughly for time and outliers, as can be seen in the video demonstration. For the scale and timeline of this project, this solution is quite effective and accounts for 100% of the work needed.

Miguel Hernandez: 0% of the Project

Miguel has not contributed to the project. As such, the section he was originally assigned to work on (Ranking) was needed to be completed by Justin instead.

Issues and Concerns

This section details any unsolved complications that arose during the development of the project as well as further work that may be done in the future.

Indexing and Ranking (Justin Garrigus)

Due to the breadth and effectiveness of the test suite at uncovering errors, there are no bugs with the current build of the project. The project performs excellently, indexing and retrieving everything it needs to, but in the future work can be done in improving its performance. On a database with 1000 entries, the program takes about 5 seconds to gain a ranked search result. This may be improved with a better database management application (one that is ideally nonrelational due to the requirement that lists be stored) or a better table schema (as data is currently stored with `TEXT` instead of a possibly faster `BLOB` type).

Prediction (Philip Rhodes)

- Though the time complexity for the CSV generation has been greatly reduced through some loop unrolling to go from $\Theta(n^2)$ to $\Theta(n)$ time complexity, the fresh CSV generation of the CSV reset function in prepSuggestionMaster.py, may take on the order of 30 seconds for CSVs of over 100,000 elements. Though the scale of the project prevents this to an extent, it is indeed still a consideration for extensions.
- Despite further normalization techniques provided in the suggestion driver function, out of character errors may still occur. It is assumed that the incoming text is plain English with small ascii elements, though non-valid characters should be discarded normally, it is possible that not every unsupported character was considered.
- The trie structure used in the retrieval of suggestions slows down on average linearly with the average length of stored phrases. With a large enough number of exceedingly long searches, prohibitive slowdown may occur.
- Some searches for too specific or too unseen a topic, may net fewer than 5 suggestions.

References

This section features a collection of research papers that were utilized when designing and implementing the project, along with a summary of the paper, what the paper discovered, how it was used by the research team, and how we used the ideas in our own search engine.

Gog, S., Pibiri, G.E., & Venturini, R. (2020). Efficient and Effective Query Auto-Completion. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. https://www.semanticscholar.org/paper/Efficient-and-Effective-Query-Auto-Completion-Gog-Pibiri/432ec69c262cd11cc6c9766c1f0f5b5fe0410f28.

The above paper is a study of several different data structure implementations to categorize

(Query Auto-Completions) or QACs. Of those explored, dictionaries, completions (a prefix trie), and inverted index lists, the competition trie proved most vital to efficient auto-completion tasks. This information corroborated the findings of Parmar and Kuhbharana's paper above, and was an incentive for using a trie structure.

Navarro, G. (2022). Indexing highly repetitive string collections, part II. *ACM Computing Surveys, 54(2), 1–32*. https://doi.org/10.1145/3432999.
This paper features a survey of indexing methods focused mainly on compressed representations of strings. Indexing involves scraping text and representing it in a database, and compression methods make it so only certain portions of the text need to be represented rather than the entire text. The survey describes programming designs that form the basis of indexing algorithms, including classic text indexes (suffix trees, suffix arrays, and CDAWGs) and pattern matching methods. Additionally, the paper notices the repetitiveness that some of these methods yield, and new algorithms which capitalize on the repetitiveness to offer further compression of the data. This paper was used to inspire the indexing design employed in our own search engine, giving a new perspective on how fast speeds can be gained from traditional indexing methods.

Parmar, Kumbharana. Implementation of Trie Structure for Storing and Searching of English Spelled Homophone Words. International Journal of Scientific and Research Publications, Volume 7, Issue 1, January 2017. https://www.ijsrp.org/research-paper-0117/ijsrp-p6102.pdf.
The paper explores the application of trie data structures in general before diving into the specific application of tries for homophone spelling recognition. It differs from the one-dimensional implementation of the trie for this project in that it requires a two-dimensional structure tagging system to differentiate homophone contexts. It was an amazing paper to learn a basic implementation of the trie structure, and was utilized thoroughly.

Turgunbaev, R. (2021, September). Metadata in Data Search. *In "ONLINE-CONFERENCES" PLATFORM (pp. 93-96)*.
https://www.researchgate.net/profile/Rashid-Turgunbaev/publication/363534664_Metadata_in_Data_Search/links/63214ea90a70852150f16a40/Metadata-in-Data-Search.pdf.
This paper offers a brief overview of how metadata is used in search engines. Primarily, metadata is a useful resource that can improve the accuracy of searches and can route the user to documents better than they would otherwise, but metadata can be incomplete, inaccurate, and complicated to use. This paper also gives an overview on metadata standards like the Resource Description Framework which encodes the descriptions of certain formats of resources in a machine-readable form. Metadata is scraped and associated with hyperlinks in the design of our own search engine, and this paper informed the way we utilized metadata. Additionally, it offered new perspectives on the types of metadata we could be using in the future; some tags and data segments are complex to parse and difficult to use, but they can offer valuable information about a document if scraped correctly.

Ward, D., Hahn, J., & Feist, K. (2012). Autocomplete as Research Tool: A Study on Providing Search Suggestions. *Information Technology and Libraries*, *31*(4), 6-19. https://doi.org/10.6017/ital.v31i4.1930.

The above paper focuses less on the exact implementation of autocompletions, and more so on the need and context of autocompletion as a tool for reducing spelling errors, speeding up the research process, and building confidence when researching an unfamiliar topic in the context of a library system. It found that users equipped with a search help function in the form of auto-completion were able to find related papers and topics faster than users without these ancillary processes. Overall the paper provided vital context for the importance of including a search help system in a search engine, and determined some of the functionalities in the project.

Zhang, X., Zhan, Z., Holtz, M., Smith, A. M. (2018). Crawling, indexing, and retrieving moments in videogames. *Proceedings of the 13th International Conference on the Foundations of Digital Games*. https://doi.org/10.1145/3235765.3235786.

This paper approaches the problem of indexing and retrieving specific moments from recorded sessions of video games. This would allow a user to input a query about a point in a video game to the artificial-intelligence model and retrieve a clip or image depicting what they were looking for. The results of the model yielded unique identifications of moments that were not seen from traditional visual-based search engine. This problem is very unique and does not strictly apply to our own search engine (especially since our engine is text-based and theirs is visual-based), but the new perspectives offered informs a new way of visualizing the indexing problem. Additionally, the paper mentions methods of indexing like clustering and data transformations which are applicable to our own project, where NLP conversions would be used instead of image conversions.