# Tiny Search Engine

Justin Garrigus, Miguel Hernandez, 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.

```
user:/tinysearchengine$ ./tinysearchengine -i https://en.wikipedia.org/wiki/Natural_language_processing --verbose
Indexing "https://en.wikipedia.org/wiki/Natural_language_processing"... Done
  Keywords:
    natural language processing
    speech recognition
    artificial intelligence
    nlp
    machine translation
    [45 others...]
  Inserting into database_hyperlinks... Done
    New size: 1567 links

user:/tinysearchengine$ ./tinysearchengine
Insert a query: natural language?
Suggestions:
  1.) natural language processing
  2.) natural language understanding
  3.) languages that are natural
  4.) natural language processing machine learning
  5.) what are natural languages
Insert a query or select an item: 1
Links:
  1.) Natural Language Processing - Wikipedia
        Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence
        concerned with the interactions between ...
  2.) What is Natural Language Processing? - IBM
        Natural language processing (NLP) refers to the branch of computer science—and more specifically, the
        branch of artificial intelligence or AI— ...
  3.) Natural Language Processing (NLP): What it is and why it matters - SAS
        Natural language processing helps computers communicate with humans in their own language and scales
        other language-related tasks. For example, NLP makes it ...
  4.) What is Natural Language Processing? An Introduction to NLP
        Natural language processing (NLP) is the ability of a computer program to understand human language as it is
        spoken and written -- referred to as natural ...
  5.) What is Natural Language Processing? Introduction to NLP
        Natural language processing (NLP) is a field of artificial intelligence in which computers analyze, understand,
        and derive meaning from human language in a ...
Insert a query or select an item: 1
  Opening "https://en.wikipedia.org/wiki/Natural_language_processing"...
```

*Figure 1: An example execution of the program.*

## 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.

More features can be added to the engine to increase its usability, including:

- Mass re-indexing over time (websites are indexed more than one time to increase the relevance of provided search terms).
- Employing machine learning to improve website ranking and query autocomplete (dynamically training a model based on the type of information a user provides).
- Exact keyword searches (queries that are surrounded in double-quotes will yield documents that contain exactly the given string of text).

# Increment 1

## Abstract

Tiny Search Engine is a project that offers a portable search engine to users with manual indexing and search suggestions. 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 first increment, the project has a functional website generation/insertion/retrieval method and an operational prediction method. For the final project submission, the program will feature multithreaded hyperlink insertion, domain-specific content retrieval when applicable (like Wikipedia-API for content originating from "https://en.wikipedia.org/wiki/"), a smarter ranking method based on website context and semantic rules, and clear natural-language text prediction that learns the user's personal writing style while increasing in accuracy over time, all combined in a single set of interconnected programs.

## Related Work

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 collection of websites will be returned that may contain information that answer the question. 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 presented a query related to 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 Related Work 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 data alongside target hyperlinks, our implementation keeps things simple by splitting functionality into several linked independent sub-programs. Once websites are scraped for content and are 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 placing value on each website, scoring the ones that relate strongly to the user's query and intentions 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. All commercial search engines generally keep their algorithms for auto-suggestions a trade secret, often opting for machine learning approaches. For further reading on the trie data structure used to efficiently store auto-suggestions in this project, consult (https://www.geeksforgeeks.org/trie-insert-and-search/) or read the article cited in the references titled "Implementation of Trie Structure for Storing andSearching of English Spelled Homophone Words"(Parmar, Kumbharana, 2017). For further reading (beyond a lay understanding), the articles titled "Efficient and Effective Query Auto-Completion" and "Autocomplete as Research Tool: A Study on Providing Search Suggestions" provide context for machine learning and other modern applications of auto-completion and auto-suggestion. Though the model derived from these papers 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.

# 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 dataset that the Indexing section maintains is the website database. When a single website is input into the program, a collection of data is scraped from the website's HTML and stored in a relational database written in SQLite3. 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, enchant's English dictionary, and BeautifulSoup's html parser.

## 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 flushed out master CSV storing all of the currently available auto-suggestions called suggestionMaster.csv.

### suggestionSeeds.txt

The “seeds” of suggestionSeeds refers 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: cnn-news.txt, cnn-small.txt, varied-small.txt, and wiki-compsci.txt. 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 an HTTP request library called “requests” paired with the “json” library, to search Google 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.

### 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 insertion 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 prepSuggestionMaster().

# Design

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.

### crawl.py

The design of this 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.

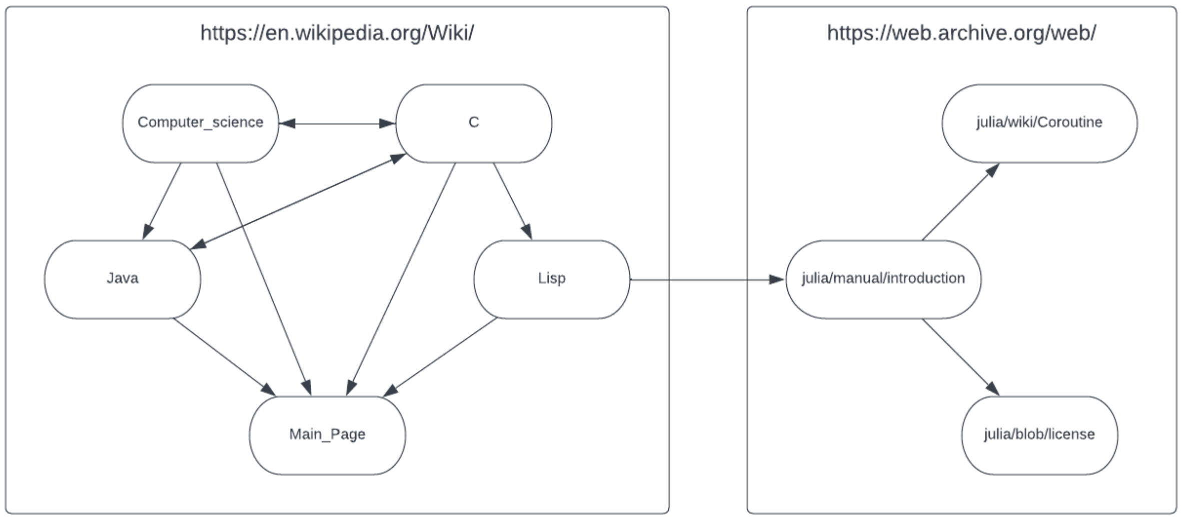

https://en.wikipedia.org/Wiki/
Computer_science
C
Java
Lisp
Main_Page
https://web.archive.org/web/
julia/wiki/Coroutine
julia/manual/introduction
julia/blob/license

*Figure 1: website connections can be visualized as a graph*

## CircularQueue

Websites are collected inside a queue-like data structure, 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. 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).

## process_website

The goal of this function 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. 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 represents 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.

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 queue. 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 "<".
- 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 "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, which eliminates any dead links at the expense of additional execution time.

## index.py

The indexing program itself represents the more idiosyncratically distinct aspects of the search engine program. 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. This program is split into three parts: the database, storage, and retrieval.

### Database

The database language chosen for this project was SQLite3. While primarily being relational (and therefore supporting a collection of familiar commands to SQL programmers), it also 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.

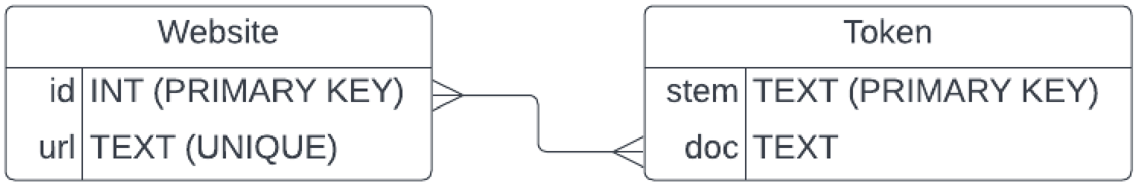


*Figure 3: Website and Token tables*

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 input token would be the stem of a word, but each doc field represents 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”). The alternative to this solution is either storing doc as a BLOB (where each ID occupies a 4-byte chunk of data) or a FOREIGN KEY to a unique table for that new stem; for the next increment, profiling of each of these solutions will reveal the most optimal solution to this problem.

## Storage

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 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 Storage, 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 Tokens 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

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.

### searchSuggestion.py

The class SearchSuggestion is an example of a trie data structure. A trie is a 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 not take longer than the searches themselves (required to take no longer than a second to perform for each search). Each node contains its data, a list of its child nodes, and a boolean endOfWord flag.

Searches and inserts begin at the root node. The first prefix datum of the string is then checked, 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 time the desired word is built and returned to a list of results in the case of a search. In the case of an insertion, the final prefix is deposited at its final node and the final node's endOfWord flag is set to true. An example state diagram is given below in Figure 4.

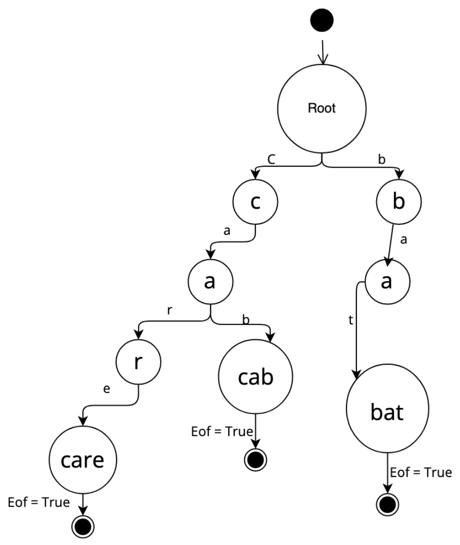


*Figure 4: A state diagram of the prefix search process of a trie structure.*

### 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 through Google with the help of an open source HTTPs library called Requests. Each seed is elaborated to a list of up to eight search suggestions and autocompletions by grabbing the argument of the Google 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 SearhSuggestion() 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.

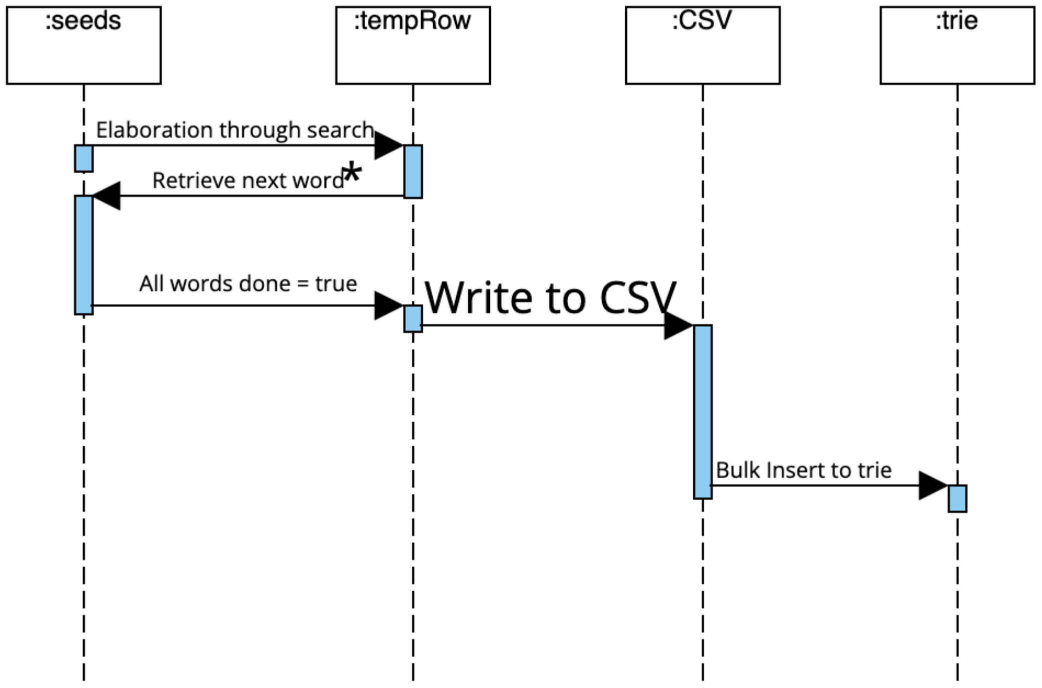


*Figure 5: Sequence diagram depicting the iterative elaboration of search seeds into searchable auto-suggestions in the trie.*

## Analysis

Certain considerations about the project needed to be made both before and during the implementation phase. We wanted to emphasize planning before programming with this project, and placed a big focus on fully understanding the topic and developing diagrams and interfaces before any code was actually created. For this reason, we gained considerable foundational knowledge that helped us create the program, informing us on the topics of search engine creation, big-data indexing, web scraping, text prediction, and inter-program communication.

One topic of interest was that of performance: most of the operations performed are not necessarily dependent on language (meaning a search engine written in C would perform at similar speeds to one written in Python), which let us place a larger emphasis on libraries in informing our language choice. Despite this, search engines need to comb through a tremendous amount of data in order to be operational, so we needed to take great care in designing code that would maximize efficiency.

## 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.

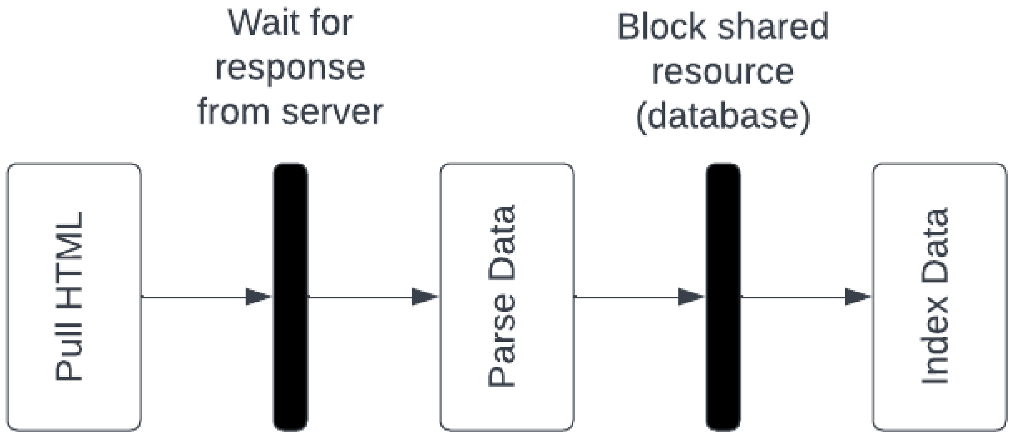

Wait for response from server
Block shared resource (database)
Pull HTML
Parse Data
Index Data

*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 get 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.

## Prediction

Since the auto-suggest and auto-complete feature is secondary to the actual searches themselves, care and additional consideration for time constraints had to be taken to not impede the flow of a user's experience. The performance considerations still apply to the prediction section. Further testing has to be done to determine the speed differences between a compiled language such as C and an interpreted language like Python. However, 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 that 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 are 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).

# Implementation

After design work was completed for the project, each of us 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 availability of NLP dependencies, as well as Python's ability to be executed without needing to be compiled.

## Indexing

The Indexing section features a faithful recreation of the plans created beforehand, performing line-for-line implementations of concepts presented in different sources. For this reason, there was no ambiguity on the accuracy or efficiency of the design, so testing was the primary cause for concern in implementation. 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>] [-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 or -q. 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.

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. The actual tests performed include 11 compiler tests (which verify the accuracy of the commands being run, i.e. disallowing the user from passing in an unknown path to the output flag or a non-integer to the timeout flag) as well as 4 content tests (which perform more intensive operations, including verification that cyclical website scrapings will not cause repeated links to be added to the crawling queue). After running each test in bulk with python3 -m unittest -s src -v -b, we can receive confirmation that any code changes made do not modify the intended output of the program.

## 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 Google 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 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.

# Results

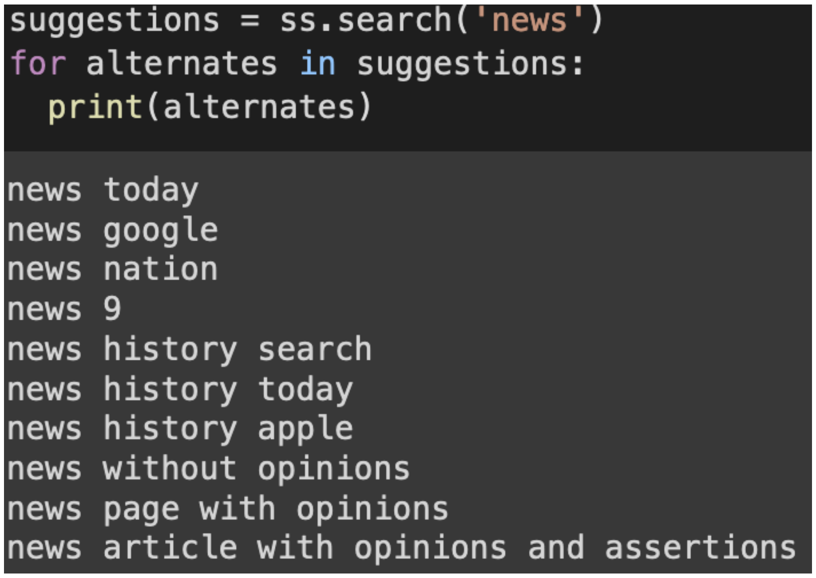
After programming the requirements, connecting the interfaces to each other, and testing the result for accuracy, the current program displays an adequate level of success in generating hyperlinks for the user, retrieving hyperlinks based on a query, and suggesting completions to the user's query. The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

## Indexing

The Indexing section has completed the two main subprograms to an acceptable degree. 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 in the home directory, with each discovered hyperlink first pinged before being added to the output. 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 may represent the content of the website). Finally, 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.

## 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 7.



```
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 7: Example output of a search suggestion*

## Work Completed

By the first project increment, the project seems to be around 70 percent complete. The Ranking section has not been started due to a team member not responding to messages, but the Indexing and Prediction sections are both nearing completion.

### Indexing (Justin Garrigus): 90% Complete

The Indexing section is nearly complete. It successfully supports all functionality required, and is missing only certain test cases demonstrating its accuracy. Websites are capable of being efficiently scraped and output by the thousands, and the content is able to be stored and retrieved with english strings easily. Furthermore, documentation already exists to explain to end users how to utilize the functions to create and modify indexes.

### Ranking (Miguel Hernandez): 0% Complete

For the first project increment, Miguel has yet to communicate with the group. They have not contributed to the GitHub, to the video, or to this document.

### Prediction (Philip Rhodes): 75% Complete

What has currently been completed for the prediction section is the implementation of the storage data structure for the auto-suggestions and auto-completions in the form of a prefix search trie implemented in searchSuggestion.py, and the dataset generation tool for the auto-suggestions in the form of prepSuggestionMaster.py. This is a large bulk of the work for this section of the project, accounting for at least 75% to 80% of the feature. See the section below for more on how it will be elaborated.

## Work to be Completed

The Indexing and Prediction sections are nearly complete and feature minor additions that must be added before the final project deadline, but the Ranking section has not been started yet due to miscommunication on a team member's part. While the project in total is less than 70% complete for this reason, the project as a whole will be completed by the deadline and will be demonstrably accurate and usable.

### Indexing

All the required functionality has been complete, but the final missing 10% mentioned previously represents test cases that demonstrates the ability of the program to work despite edge-cases in website content or user action. The addition of these test cases would finish this section, but more work can still be done to improve it, including semantic analysis of scraped websites and categorization to be included in the database, website-specific API utilization like the Wikipedia-API, and minimization of websites retrieved from queries based of syntactical rules. There are no issues or concerns involved with this section; it has been completed as planned.

## Ranking

Since Miguel has not yet contributed to this section, Justin and Philip will assume that he will continue to not contribute for the final submission. Therefore, the work to be done for this section include translation of website indexes to URLs via SQLite3 calls, scoring of websites based on relevancy to the topic presented in a query, and presentation of ranked websites to the user. Since Miguel was supposed to do research on this topic, a concern is that the interfaces and databases created will not be sufficient to solve the ranking problem, requiring changes to be made to the Indexing section.

## Prediction

*Work to be completed*

- Add a recursive CSV generation feature that breaks down seeds and can generate suggestion data sets of desired sizes.
- Work with indexer/search designer to extend the search seed system to have a cached seed list for different users (determined by linux process owner user) alongside the base list.

*Issues, Concerns, or Bugs*

- Possible slowdown with larger and larger CSVs. Over about 50,000 data elements, the auto-suggest feature might be prohibitively slow. Work will be done on this.
- 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

# 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.