Search Engine

Alessandra Cid, Lucas Emanuel Resck and Lucas Moschen

June 2019

Abstract

For this project our group developed a search engine that searches for a word, or multiple words, in the English portion of the Wikipedia corpus. In order to do that, we constructed a trie in C++ to use as our data structure. We used Python to pre-process the corpus and then inserted it into the trie. Our search is also done in C++. If we have more than one word, we compare the pages that are common to all words and only return them. If a user misspells a word when searching, our program returns suggestions for alternative results.

Relevant references

This is the link to the group's Github repository containing all of the project's source code: `https://github.com/lucasresck/Search-engine`. In order to run the program you should download the files on this link (`https://bit.ly/2WvZOFj`):

1. The folder "separated pages": this folder contains the Wikipedia pages separated in different files and ordered by alphabetical order;
2. The file "titles_ordered.txt": list of titles of Wikipedia pages in alphabetical order;
3. The file "serialization.txt": contains the serialization of the trie composed by the Wikipedia pages;
4. The folder "cleaned pages": this folder contains the files used to build the trie. It is not necessary to download it to run the engine, but it can be downloaded in case a user wants to change the way the pages are processed.

You should unzip the "separated pages" folder and make sure that you have the following packages:

1. iostream
2. vector

3. fstream
4. string
5. ctype.h
6. bits/stdc++.h
7. iomanip
8. chrono

The final step is to run the engine.exe file while in the same folder as the "separated pages", the "titles_ordered.txt" file and the "serialization.txt".

For a description on how the project was developed and the results obtained, this is the link to the group's video : `https://youtu.be/iaAe9gT1-Y0`.

Description

The Wikipedia corpus used in this project was downloaded[1] as several different text files. Each file contained one or more Wikipedia pages. The first step in the project was to pre-process, in Python, this data. Some non ASCII and ASCII symbols were removed, such as "+", "{" and "]", but all of the numbers and the symbols "%", "&", "-", "@" and " ' " were maintained. All of the accents from the words that had them were also eliminated and the letters in upper case were transformed into lower case.

The way the different Wikipedia pages were stored in each text file was also changed. Taking advantage of the "doc id" tag identifying the begging of each page, we separated the pages and stored 10.000 Wikipedia pages per file.

For the data structure, the group first chose to build a suffix tree[2]. A suffix tree is a compressed trie[3] that stores all the suffixes of a given text. The first letter of each suffix starts as a node from the root of the tree and the other letters are the child of that node. If a node only has one child, the child and parent node are compressed into a single node. This structure was built[4] by going through the text files that corresponded to the Wikipedia pages and inserting it's suffixes. After building this structure and a search function for it, the group concluded that it would take too much time to insert the Wikipedia corpus into the suffix tree. The group decided, then, to build a trie.

[1] As assigned for this project, the data was downloaded from: `http://www.cs.upc.edu/~nlp/wikicorpus/`.

[2] As explained in: MIT, Advanced Data Structures, Session 16: Strings. Video available at: `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/lecture-videos/session-16-strings/`.

[3] A full explanation of what is a trie is going to be given in the next paragraph.

[4] The algorithm to build the suffix tree was based on: `https://www.geeksforgeeks.org/generalized-suffix-tree-1/`

A trie is a tree that stores different words[5]. In this case, all the words in the Wikipedia corpus. Each letter of a word in the trie is the child of the letter that came before it in the word. For this project, the trie was constructed[6] by going through the words in the text files. Each word in the file that wasn't already in the trie by the time it was reached, was inserted into it. In order to identify the Wikipedia pages that contained each word, a pointer to an array named docs was placed in each node. These arrays stored numbers identifying each of the pages in which a certain word appears. They could be accessed through the node related to the last letter in the word.

To build this in C++, pointers were used. Each pointer had 128 potential "child" pointers corresponding to a specific ASCII character. If a person wants to verify if a specific word is in the trie, they can start from the root of the trie (pRoot) and visit the child (pChild) corresponding to the ASCII character of each of the letters in the word. If they were able to reach the node corresponding to the last letter in the word, it means that in at least one Wikipedia page that word exists. Accessing the array docs from that point allows the person to see the number corresponding to those pages.

In order to access this trie outside the scope of the original code in which it was constructed, the group performed a serialization of the trie, inserting it into a text file[7]. This serialization started from the root and accessed the child nodes. If the node existed, the ASCII character corresponding to that node was inserted into the file. A dot (".") was used to indicate that that the number corresponds to a node. A plus sign ("+") followed by a number indicates the size of the array of documents. In this case, the numbers corresponding to the pages were separated by commas (","). After this, that node's child was accessed and the same process was performed again. When a node did not have any more child nodes a "-" sign was written in the document. Following the logic through which the serialization was developed, a deserialization function was built.

In the search environment, the first step of the program is to perform the deserialization. After that, the user can enter a query. The query is pre-processed, removing accents and lowering the case of letters in upper case. Then, each word in the query is identified and searched, by accessing the child of each of the word's letters in the trie. This search is done in linear time, varying on the length of what was searched. If the word searched exists in one or more Wikipedia page, the number corresponding to those pages is inserted into an array. If that word does not exist, the program runs a minimum edit distance algorithm to identify similar expressions and returns the ones that appear the most in Wikipedia pages. This way, the search returns a popular word similar to the one searched by the user.

[5]As explained in: MIT, Advanced Data Structures, Session 16: Strings. Video available at: `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/lecture-videos/session-16-strings/`.

[6]The algorithm to build the trie was based on: `https://www.geeksforgeeks.org/trie-insert-and-search/`

[7]The algorithm to build the serialization and the deserialization was based on: `https://www.geeksforgeeks.org/serialize-deserialize-n-ary-tree/`

If a query has more than one word, each time the program searches for a new word it compares the previous document array that it had with the new searched one and only keeps the pages that are common to both. As the page numbers are ordered in each array, this is done in linear time in relation to the number of documents.

After the search ends, the program returns the number of results that were found and identifies the title of each page found in the search. The title of the 20 first results are then displayed in alphabetical order. The user is able to open in details an specific page that returned as a result or to see the title of 20 more pages found in the search.

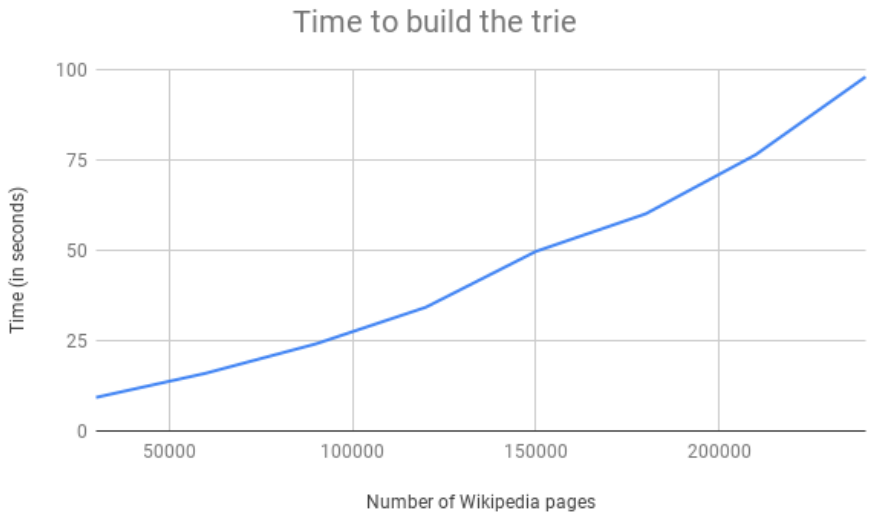
Results

Originally, the group downloaded 164 documents. The division of pages in those files was changed, in order to store 10.000 Wikipedia pages per file. That resulted in 136 files, corresponding to 1.359.870 Wikipedia pages.

Regarding space, the serialization file takes 1,73 GB of space. The "Titles Ordered" file takes 25 MB and the "Separated pages" folder 3,19 GB.

When running our search tool, the deserialization takes approximately 378,486 seconds. Searching for a certain word is very fast, taking about 0,000005 seconds.

The group also analyzed the time that it took to build the trie with a different number of pages, one of the initial problems that we faced when using the suffix tree. First we built the tree with 30.000 pages, then with 60.000, repeating this until 240.000 pages. The result was the time displayed in the graph bellow:



As it is clear by the graph, the time grows as the number of pages gets

higher. We expected the line in the graph to be a smooth one, but there seems to be a slight change around 150.000. We believe that this might be due to other tasks that were being performed by the computer at that time.

Limitations

Although the group was able to successfully perform the task of searching the Wikipedia corpus, there are some limitations to the final result. Cleaning the documents and doing the serialization and the deserialization are processes that take time and could be done faster. For the serialization, a binary document could be used and so could other C++ packages, such as Boost, that make it more efficient. The intersection of pages, for queries with more than one word, could also be developed in place. Finally, an HTML interface could have been developed for the search, making it more user-friendly.

Future work

For future work it would be interesting to improve on the mentioned limitations of the project. Although the documents only have to be cleaned once, it would be good to do it faster. There are Python libraries that could be used to improve that, such as the regular expressions one ("re"). If, while cleaning the data, we also split the words in the text using Python, the process of constructing the trie would take less time. But, this would add more time to the cleaning process. It would be interesting to measure which one is faster.

As was mentioned, we could take advantage of packages such as Boost, to make the serialization process faster. We were not able to use the package for this version, because group members could not install it on Windows computers.

The HTML interface would also be a great update. It directly influences the impression that users have of our engine, so it would make the experience of searching in it much better.

The alternative result suggestion, when a word is typed wrong, is a process that would be interesting to improve as well. The place where letters are located in the keyboard could indicate common mistakes between two keys in the keyboard. A list could also be made of common mistakes made by users when typing. Finally, it would be interesting to take advantage of other algorithms, such as machine learning ones, in order to identify similar words.

Another idea would be to expand the project's database, including searches in the Wikipedia corpus of different languages. Ultimately, we could even search databases that are outside of Wikipedia.

Conclusion

In conclusion, the group developed a tool that is able to search in less than one second the English portion of the Wikipedia corpus and return the

Wikipedia pages in which that word was found. The user, then, is allowed to open any of the pages that returned as a result. If the user types a word that is not on Wikipedia, the program returns five word suggestions that are similar to the one searched.

In order to do that, the group used Python to clean the text, removing accents and symbols, and changed the way the pages were stored in files. Then, a trie was built in C++ in order to store this data. In this trie, a user can access an array containing the pages in which each word in the corpus appears. Based on this data structure, the group developed the search and the word suggestion, also in C++.

The final result was a fast engine, but it had limitations. Cleaning the texts and doing the serealization, processes that happen before the search, take a lot of time and could be done faster. In the future, it would be interesting to improve on that, to develop an HTML interface and a more precise alternative result suggestion.

Distribution of work in the group

The main decisions for the project were taken by the group and all members helped, by making suggestions, thinking of new approaches and working on code, on parts of the project that they were not primarily responsible for. Individually, each were responsible for the following parts:

- Lucas Moschen was responsible for pre-processing the data, building the overall search interface and writing the function for alternative result suggestion in case of misspelled queries.
- Lucas Emanuel Resck built the data structure for the suffix tree and the trie and edited the group's video presenting the project. Lucas Moschen and Alessandra worked on initial versions of the serialization and Lucas Resck was responsible for the final one.
- Alessandra built the word identification for the query and the search in the suffix tree and the trie. Alessandra also wrote this report for the project.