Ho Chi Minh University of Science

CS163 - Data Structures

Class: 19APCS2

Report on

MINI SEARCH ENGINE

19125056 - Nguyễn Phạm Tùng Lâm

19125074 - Hà Phương Uyên

19125093 - Nguyễn Đức Hoàng

19125199 - Trương Lăng Trường Sơn

August 1, 2020

0. Project introduction

In this project, you will be designing and implementing a mini search engine. You are probably familiar with Google, Bing, which are some of the most popular search engines that you can find on the Web. The task performed by a search engine is, as the name says, to search through a collection of documents. Given a set of texts and a query, the search engine locates all documents that contain the keywords in the query. The problem may be therefore reduced to a search problem, which can be efficiently solved with the data structures we have studied in this class.

1. Design issues

1.1. How are documents processed and stored?

Initial method: All the data files are stored in the same folder “data”. The program opens the file “___index.txt” - which contains the names of all the files. Each time the program gets the name of a file, it opens that file and loads each word in that file into the program. After finishing, it closes the file and opens the next file to load again, the process repeats until all the files are loaded into the program.

The drawback of this method is we have to manually type the file names into “__index.txt” everytime we add new files, which will be inconvenient in case of very large number of texts.

There is a better method which will be presented later (see 3. Algorithm)

1.2. How is the input query processed, and how to search for its keywords?

Initial method: traverse through all the files in the folder data. If we encounter a file which contains the word we want to find, we will push that file into the vector. When displaying the results, we will use that vector to print out the results.

The drawback of this method is that it costs time to traverse through every file which will be inconvenient in case of a very large number of texts.

There is a better method which will be presented later (see 3. Algorithm)

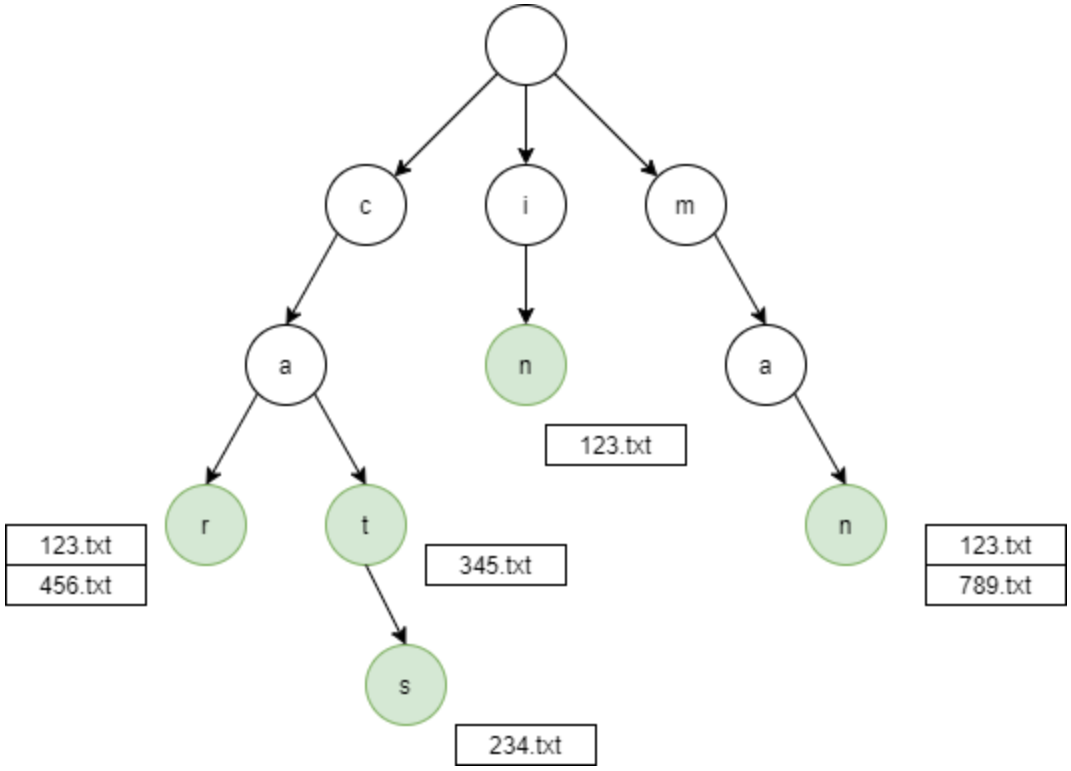
2. Data Structures

2.1. Trie

The structure of a node of the Trie:

```
class Node
{
public:
	//Marks the end of a word
	bool isLeaf;
	//A vector is used to store the names of the file that contains
the word
	vector<int> myv;
	map<char, Node*> mapkey;
};
```

In the following example of the Trie of the given Node structure, the green nodes indicate the end of a word, along with the names of the files that have the word stored in the vector.



2.2. The pros and cons of Trie

2.2.1. Pros

1. With Trie, we can insert and find strings in ***O(L)*** time where *L* represents the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)
2. Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.
3. We can efficiently do prefix search (or auto-complete) with Trie.

2.2.2. Cons

The main disadvantage of tries is that they need a lot of memory for storing the strings. For each node we have too many node pointers (equal to number of characters of the alphabet).

3. Algorithms

In this section, we will break down and describe the methods we use to index the files, process the query, search for the query and display the results.

3.1. Indexing

```
void Trie::getFileName(Node*& root, char* str, vector<string>& vt)
```

We use the file system to easily get the file name. We don't have to manually type the filename into _index.txt, because we don't need this file anymore.

With this for loop, we easily get all the filenames and push them into a vector.

```
for (auto& p : directory_iterator("data")) // data files put in folder "data"
{
    titleName = p.path().filename().string();
    vt.push_back(titleName);
}
```

Load all the filenames to a vector -> Traverse through the vector to read and load each file into the Trie using the readWord function

```
void Trie::readWord(vector<string>& ptr, Node*& root, int i)
```

Read word by word. Load all the words of a file into the Trie by reading word by word, each word is pushed into the Trie by the insert() function.

```
void Trie::insert(Node*& root, string str, int i)
```

Insert each word into the Trie.

Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark the end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

3.2. Processing queries

The user first enters a query into the program. The program first checks for the type of operators used in the query, for example “AND”, “OR”, “intile: car”... Then it checks for and removes the English stopwords in the query like “we”, “ourselves”, “me”... Next, it will operate the type of query that the user inputs.

The function :

```
void Trie::QueryOperator(Node* root, char* str, vector<string>& vt, Node* root2)
```

Will check the input and identify the query type.

Following are the types of queries used in this project.

1. AND

Search for X and Y. This will return only results related to both X and Y. Example: jobs AND gates

2. OR

Search for X or Y. This will return results related to X or Y, or both. Note: The pipe (|) operator can also be used in place of “OR.” Examples: jobs OR gates / jobs | gates

3. –

Exclude a term or phrase. In our example, any pages returned will be related to jobs but not Apple (the company). Example: jobs -apple

4. intitle:

Find pages with a certain word (or words) in the title. In our example, any results containing the word “apple” in the title tag will be returned. Example: intitle:apple

5. +

Force an exact-match search on a single word or phrase. Example: jobs +apple

6. filetype:txt

Restrict results to those of a certain filetype. E.g., PDF, DOCX, TXT, PPT, etc. Example: apple filetype:pdf

7. $

Search for a price. Put $ in front of a number. Example: camera $400.

8. #

Search hashtags. Put # in front of a word. Example: #throwbackthursday

9. “ “

Search for an exact match. Put a word or phrase inside quotes. Example: "tallest building".

10. ...

Search within a range of numbers. Put .. between two numbers. Example: camera $50..$100.

3.3. Searching

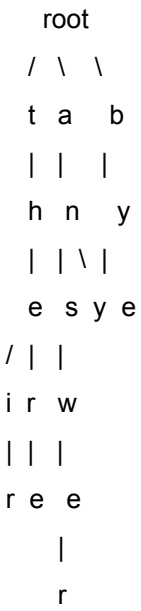
```
Node* Trie::getFile(Node* root, char* str)
```

Searches whether a word exist in the Trie and get the list of the files that contain the words.

The program searches for the query’s keywords within the Trie. If a keyword exists within the Trie, it will get the names of the files that contain the keyword.

Searching for a key is similar to insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type trie_node_t, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.

The leaf node also contains a vector. When the function inputs every word to the trie, a vector will record the file that contains the word.

THIS IS THE VECTOR, THE MOST IMPORTANT PART OF THE WHOLE PROGRAM!!

```
vector<int> myv;
```

When we input many keywords, we just combine the vectors of each word, merge it, and find the common elements of each vector. It depends on the query, for each query, we'll find a way to combine these vectors for the final result.

Example:

When the program opens the file "Data01.txt" and it loads every word from the file. When it loads the word "new" into the trie. At the leaf node of the word "new", a vector will push the position of the file "Data01.txt" into it. When you search the word "new", the program will traverse to see if "new" is on the trie. Then, it'll take out the vector, use it for the output function.

Insert and search costs **O(key_length)**, however the memory requirements of Trie is **O(ALPHABET_SIZE * key_length * N)** where N is the number of keys in Trie. There are efficient representations of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize memory requirements of trie.

3.4. Displaying the results

```
void outPutResult(string fileName, vector<string> input, int& count, bool exact, bool intitle);
```

When the program finds the list of the position of the file that the key word is contained in, it'll traverse the list (vector) to open every file (from the position, we'll find the name). The program will only display the sentences of the file that contain the key words and highlight them.

As said above, the program gets the names of the files that satisfy the queries in the Trie. The program displays top 10 files, each with the file's name and displays the sentence inside that file which has the keywords. If there are more than 10 files that satisfy the queries, the program only shows the first 10 files.

3.5. Displaying search history

Use a vector to store the keywords that the user inputs. Each time the user input new keywords, the new keywords will be pushed into the vector. The vector stores most 5 recent queries.

4. Running time

4.1. Loading time

There are 11.370 files in the data folder.

When a user runs the program, the program first needs some time to load all the data documents into the trie, the loading time varies from 60s to 120s.

4.2. Searching time

The running time of the searching process usually takes less than 1s.

For example, user enter keyword “star wars”, using the “EXACT” operator (which force the program to search for documents containing the exact phrase “star wars”)

```
*** Please enter keyword: "star wars"
```

And the running time:

```
=> Running time: 0.19
```

5. Optimization

The loading algorithm used in this project costs a rather long time, which takes about 2 to 3 minutes, so we can optimize it by thinking of a way to load all the data files more quickly and therefore reduce the loading time to ideally less than 30s.

The Trie costs a lot of memory, so we can also optimize the memory space by implementing a better version of the Trie.

6. Problem of scalability

In the case of very large text collections, the time it takes for the program to load and store the file will cost more, but the overall time of searching won’t change considerably. In short, loading time costs more, searching time doesn’t change

7. Results

7.1. AND

```
*** Please enter keyword: bookstore AND insider


QUERY AND

***Group01News31.txt:

Whether exploring with our friends or on our own, we profited from their insider
information about such local wonders as Victoria's afternoon teas, the oyster bed
s along Washington's Samish Bay, Oregon's Columbia River Gorge and a one-of-a-kin
d Portland bookstore.
Then it was on to her top choice of a Portland tourist spot: Powell's City of Boo
ks in the center of downtown, which calls itself "the largest used and new bookst
ore in the world."
```

7.2. OR

```
*** Please enter keyword: insider OR bookstore


QUERY OR

***Group01News31.txt:

Whether exploring with our friends or on our own, we profited from their insider
information about such local wonders as Victoria's afternoon teas, the oyster bed
s along Washington's Samish Bay, Oregon's Columbia River Gorge and a one-of-a-kin
d Portland bookstore.
Then it was on to her top choice of a Portland tourist spot: Powell's City of Boo
ks in the center of downtown, which calls itself "the largest used and new bookst
ore in the world."

================================================================================
==================================================

***Group02News52.txt:

Japanese bookstore chain Yurindo and Reliable, a Japanese book and stationary sto
re chain in Hokkaido, also removed Kuroko's Basketball merchandise from their she
lves.
Several other bookstore chains such as Kinokuniya, Sanseido, Junkudo, Miyawaki, a
nd other bookstores, plan to continue carrying the manga despite receiving threat
 letters.
```

7.3. –

```
*** Please enter keyword: new -york


QUERY MINUS

***for#.txt:

Create a brand new hashtag simply by putting the hash before a series of words, a
nd if it hasn't been used before, voilá! You've invented a hashtag.
```

7.4. Intitle:

```
*** Please enter keyword: intitle: beer


QUERY INTITLE

***Group10News82.txt:

First beer, now chicken! Shortage warnings as CO2 stocks fall Supplies of chicken and beer could fa
ll due to a shortage of CO2, with warnings of a "potentially huge" effect on food production.
```

7.5. +

```
*** Please enter keyword: grand +canyon


QUERY AND

***Group01News32.txt:

ùIrene Lechowitzky Million-Dollar Colorado Landscapes The route: Durango to Grand
 Junction, Colo.,
After switchbacks over three passes, the road straightens on the far side of the
range where the terrain transitions; the canyon lands and Bookcliffs around Grand
 Junction resemble Utah more than the Rockies.
Catch happy hour in Palisade, Colorado's little corner of wine country adjacent t
o Grand Junction, for sangria at the laid-back Red Fox Cellars, where dusty hikin
g boots are welcome.
For dinner, drop into Grand Junction's Bin 707 Foodbar for farm-to-table fare tha
t changes with the seasons.
```

7.6. Filetype:

```
*** Please enter keyword: filetype: txt

QUERY FILETYPE

***for#.txt:

Which characters can you include in a #hashtag?
```

7.7. $

```
*** Please enter keyword: $10


QUERY PRICE

***Group01News13.txt:

goods and services produced and sold in China totaled $223 billion, compared with
 a minuscule $10 billion of Chinese products made and sold in the United States,
Peterson's Lardy wrote last week.
```

7.8.

```
*** Please enter keyword: #mileycyrus


QUERY HASTAG

***for#.txt:

Using a popular hashtag that has nothing to do with your brand (for example, incl
uding #MileyCyrus in a tweet about cheap airfare) makes you look like a spammer a
nd will hurt your credibility.
```

7.9. “ “

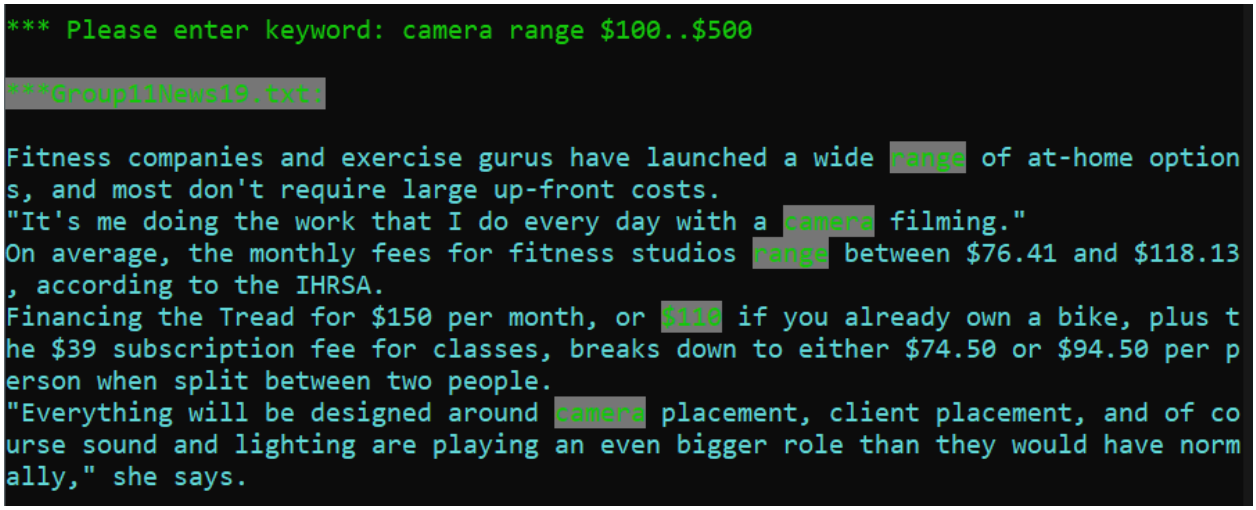
```
*** Please enter keyword: "star wars"


QUERY EXACT

***Group04News80.txt:

Star Wars: The Last Jedi now playing at home: Everything we know The newest Star
Wars film can now be in your home movie collection.
But you didn't expect Disney to stop telling the story of The Last Jedi with just
 the movie, right? Since the first trailer's premiere at the Star Wars Celebratio
n in April through adaptations of the story into a novelization and comic book, L
ucasfilm has been enlarging the latest tale of a galaxy far, far away.
Rogue One: A Star Wars Story is currently available on Netflix too.
The Last Jedi is the longest Star Wars film so far, with a runtime of 2 hours, 30
 minutes.
```

7.10. ...



8. References

Google Search Operators: https://ahrefs.com/blog/google-advanced-search-operators/

English stopwords: https://gist.github.com/sebleier/554280