**VNUHCM-University of Science**

------------------------------



**CS163 PROJECT REPORT**

**Topic:**

**Mini search engine**

**Group 6 – 20CTT1:**

1. Nguyễn Quang Long
2. Trần Minh Nam
3. Văn Hoàng Yến
4. Nguyễn Hoài Nam Phương

Contents

[***1)*** ***Design Issue*** 4](#_Toc79781039)

[*a)* *Data loading issue* 4](#_Toc79781040)

[*b)* *Query issue* 4](#_Toc79781041)

[*c)* *History issue* 4](#_Toc79781042)

[*d)* *User Interface and inputting issue:* 4](#_Toc79781043)

[*e)* *Group 100 data files* 4](#_Toc79781044)

[*f)* *Stopword and Synonym* 5](#_Toc79781045)

[***2)*** ***Data structure to solve the problem*** 5](#_Toc79781046)

[***3)*** ***Algorithm*** 6](#_Toc79781047)

[*a)* *Inserting a word into a trie:* 6](#_Toc79781048)

[*b)* *Searching a word in a trie* 7](#_Toc79781049)

[*c)* *Hashing function:* 8](#_Toc79781050)

[*d)* *Loading data* 8](#_Toc79781051)

[*e)* *Query preprocessing* 9](#_Toc79781052)

[*f)* *Special char removal when loading data file* 9](#_Toc79781053)

[*g)* *Get Intersection* 9](#_Toc79781054)

[*h)* *Get Union* 10](#_Toc79781055)

[*i)* *AND query* 10](#_Toc79781056)

[*j)* *Normal query* 10](#_Toc79781057)

[*k)* *OR query* 11](#_Toc79781058)

[*l)* *Not include query (-)* 11](#_Toc79781059)

[*m)* *intitle: query* 11](#_Toc79781060)

[*n)* *filetype: query* 11](#_Toc79781061)

[*o)* *Search for a price $* 11](#_Toc79781062)

[*p)* *Search hashtag #* 12](#_Toc79781063)

[*q)* *Search exact match (including wildcard)* 12](#_Toc79781064)

[*r)* *Search for a range of price* 12](#_Toc79781065)

[*s)* *Search with synonyms ~* 12](#_Toc79781066)

[***4)*** ***Running time*** 13](#_Toc79781067)

[***5)*** ***What to do further to optimize your algorithm*** 13](#_Toc79781068)

[***6)*** ***Is this program efficient in the case of very large text collections?*** 13](#_Toc79781069)

[***7)*** ***Sample query demo*** 14](#_Toc79781070)

[*a)* *Demo 1* 14](#_Toc79781071)

[*b)* *Demo 2* 14](#_Toc79781072)

[*c)* *Demo 3* 14](#_Toc79781073)

[*d)* *Demo 4* 14](#_Toc79781074)

[*e)* *Demo 5* 14](#_Toc79781075)

[***8)*** ***Remark and additional features*** 14](#_Toc79781076)

[*a)* *History suggestion* 14](#_Toc79781077)

[*b)* *Input rule* 14](#_Toc79781078)

[*c)* *Help command* 14](#_Toc79781079)

[*d)* *View history command* 14](#_Toc79781080)

[*e)* *Delete history command* 15](#_Toc79781081)

[*f)* *Exit command* 15](#_Toc79781082)

1. ***Design Issue***

Our project is building a mini search engine to search for documents that satisfied the users. Therefore, there are many small problems below to handle:

* 1. *Data loading issue*

We have to load nearly 11500 text files at the beginning of the program. It is a huge amount of files to store directly as a string for every document so we have to handle the storing process efficiently. Moreover, the articles may have special char so we have to remove them before move it to our program storage.

* 1. *Query issue*

We have more than ten types of queries to handle. The query is the only thing to connect the users with this program so we have to specially handle this program.

* 1. *History issue*

Every time users searching for something, we need to save that query and display it as a suggestion at the next time they enter that query.

* 1. *User Interface and inputting issue:*

This is what users see and interact with our program, so we also mark this as important.

* 1. *Group 100 data files*

We collected 100 data text files from <http://www.textfiles.com/> and named every file from Group06\_001.txt to Group06\_100.txt

* 1. *Stopword and Synonym*

Our search engine has to carry the stop words and synonyms. The stop words data source is <https://www.ranks.nl/stopwords>, and synonym data source is <https://www.gutenberg.org/files/51155/51155-h/51155-h.htm>.

🡺 These are problems we have to solve to complete our program

1. ***Data structure to solve the problem***

To solve this problem, we first think about how to store the information/the words of each document into the program. Each file has more than 500 words or 3000 characters so storing the information of each file as a slot in an array is not efficient in searching because if we search for a word “ABC”, we have to loop through 3000 characters x 11000 files. It costs us a huge comuter resources and time to do that process. Hence, we have to design our search engine with another data structure to store the whole library.

We choose to use trie – a retrieval data structure – because we can merge many word with same prefix together. Therefore, our storing data and searching for keywords will be efficiently optimized.



In each node of a tree, we have:

* A vector of pair “position” to store the article id and word id in the document.
* The “title” vector is for storing the id of the article that the word belong to that article title.
* “pNext” is the pointer to next node. If pNext[id] is not null, the next character in our word is the character that id represent.
* We have a bool to mark this node is the end of a word. This bool value use for searching.
* “synonym” vector to store the synonyms of the current keyword.

1. ***Algorithm***

Our trie have two process, inserting and searching.

* 1. *Inserting a word into a trie:*

There are two types of insert: insert a word in document and insert a word in the tile of the document.

The general insertion is starting from the root, we traverse each character of S, and move along the trie with the corresponding character. At the end of the string, marked the bool is true.

Time complexity:







The time complexity for searching a string S is :

* 1. *Searching a word in a trie*

Start from the root, traverse each character of S. If there is a node, then move along, else we can’t find the keyword.

Time complexity:



* 1. *Hashing function:*

To convert a char into the id of the trie node as below:

* 0 to 9 is id 0 to 9, respectively
* a to z is id 10 to 36, respectively
* # is id 37
* $ is id 38
* - is id 39



* 1. *Loading data*

We loop through each line of the \_\_\_index.txt file, each line in the file is a name of a document. Then we will open each txt file with its name and load data into our trie. We also load synonyms of each word in synonym.txt file.

We have another trie to load, a stop words trie from file stopword.txt.

* 1. *Query preprocessing*

From the raw query in the input, we design a fuction to break it into vector of words using string stream and remove all special characters, except the special chars of the query operators.

Example:

|  |  |
| --- | --- |
| User Input | Query After Preprocessing (each word in the vector is separate by ‘/ ‘) |
| ABC | abc |
| String-stream | string/stream |
| manchesTEr -UniTED | manchester/-/united |
| hamburger AND chips | hamburger/AND/chips |

We split the operator as a word and do not normalize it.

* 1. *Special char removal when loading data file*

We get each line in the data file. Then we loop through each line by char, if it is a special char, we skip it. If it is a nomal digit, we add the char into the tmp string. If it is a blank space, we add the tmp string into our trie and reset the tmp string.

To solve the problem of searching many keywords in the query, we design two functions: get intersection and get union. These two functions will also become the core of our query searching.

* 1. *Get Intersection*

To get the intersection integer value of two vector arrays:



* 1. *Get Union*

To get the union value of two vector arrays



To solve our query, we have design the following function:

* 1. *AND query*

Solving the query AND, by getting the intersection of two vector array.



* 1. *Normal query*

This type of query is the query with no operator. We will also use AND operator to solve this normal query

* 1. *OR query*

Solving query OR, using function getUnion above



* 1. *Not include query (-)*

We will search the word and get result like a normal word below. Then for every id in the not include vector, we remove it from the answer query

* 1. *intitle: query*

We get the keyword after this query and get the result vector of that keyword. Then we use function getIntersection of the result vector with this intitle vector.

* 1. *filetype: query*

Because all data files are txt so it will play as a normal query

* 1. *Search for a price $*

This query will be considered as a string of USD dollar and combined with the previous result vector using getIntersection function

Our trie node will have a node that present character #

* 1. *Search hashtag #*

As same as a normal query, we search this query and combined with previous result vector using getIntersection function

Our trie node will have a node that present character $

* 1. *Search exact match (including wildcard)*

We will split the input query like normal, but keeping the double quote marks. Then, when searching, if we meet a word with the beginning character is the opening double quote, we will jump into the special IF case. We jump out this special case when we reach the word that it ends with ending double quote.

The result of the vector will be combined with the previous result using getIntersection function.

* 1. *Search for a range of price*

$money1..$money2

I have a function isRangeMoney(string s) to consider if a string in query is a range money query.

Because a range of price have two character $ so whenever it satisfied this condition, I will split it into two numbers. I run for each integer value from money1 to money2 and getUnion for each result vector.

Finally, I use getIntersection function between the previous result vector and the range money result vector.

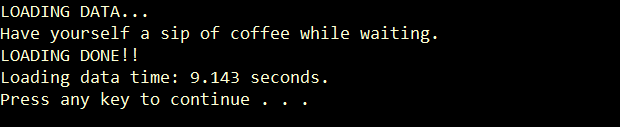
* 1. *Search with synonyms ~*

~keyword

In this query, we search for the string after character ~ in the synonym trie. We got the vector of string (which is the synonym for keyword) and add it to query vector. We continue the search like normal.

1. ***Running time***

At first, when we open the program, it takes us from about nearly 10 seconds (on Nguyen Quang Long computer) by loading 11500 files.



For searching, we get the result immediately if the query is short (below 1ms). In case that the query is so long such as (a AND b AND c OR d $500 “limit time” intitle:E), it will take us . We show you the list of documents that is satisfied with your input query. You will choose which documents you want to see the data inside.

1. ***What to do further to optimize your algorithm***

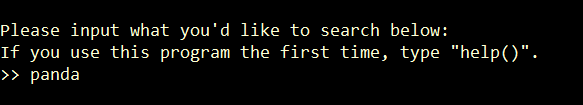
With loading time, we cannot optimize more. However, we still think about after searching for a query, should we save that result vector. In the future, when user type the same query, we do not have to search again. We just use our saved result vector.

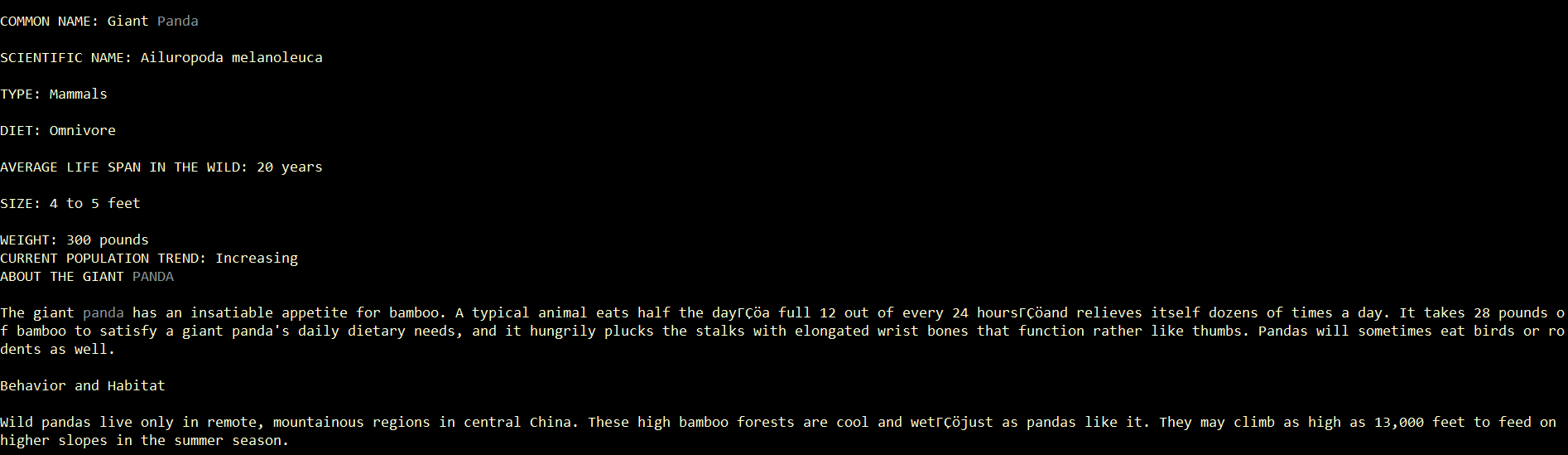
1. ***Is this program efficient in the case of very large text collections?***

Our data has nearly 11500 files, each file has above 2500 characters (500 words). The loading process with this data set takes us only 10 seconds, so I think with bigger dataset, this program can work well for loading

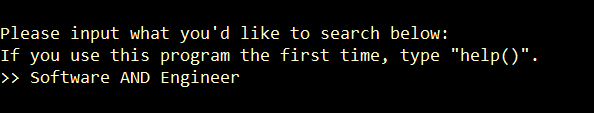
For searching, each word we get the id vector and we merge those vectors together according to the query operator so our program may work well with huge data set.

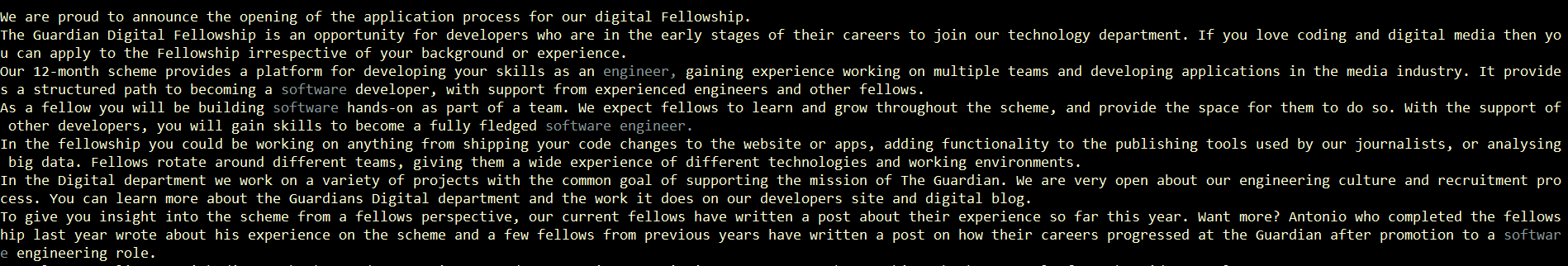
1. ***Sample query demo***
   1. *Demo 1: Normal keyword*



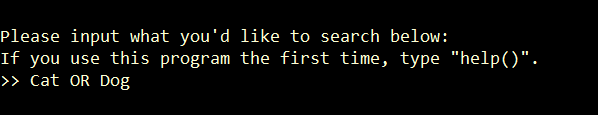


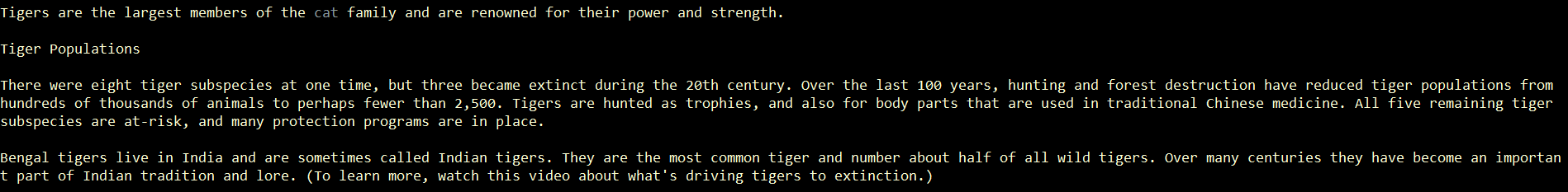
* 1. *Demo 2: AND query*



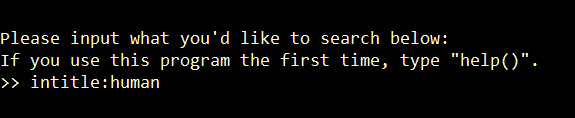


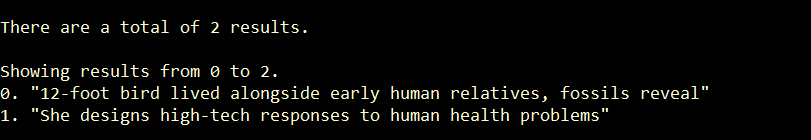
* 1. *Demo 3: OR query*



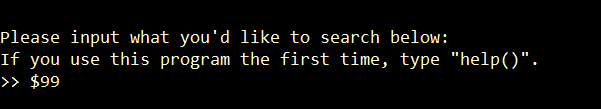


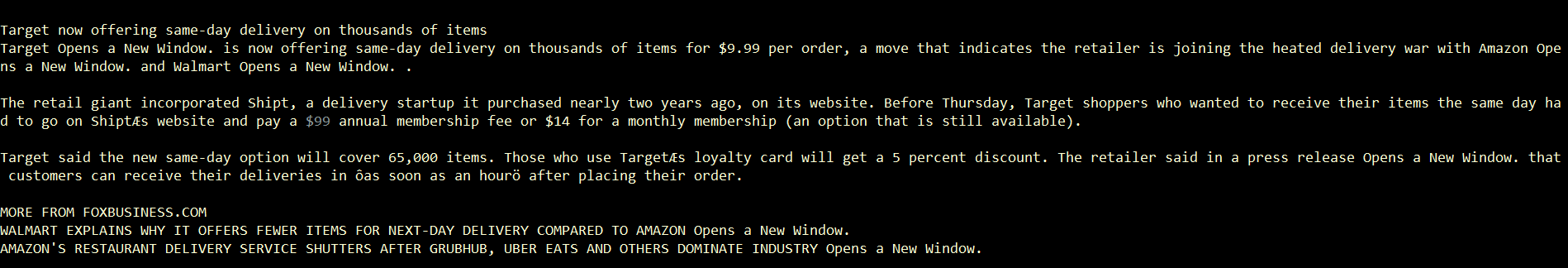
* 1. *Demo 4: intitle query*





* 1. *Demo 5: Money query*





1. ***Remark and additional features***
   1. *History suggestion*

I use a history.txt file to store all the inputted queries.

Whenever the user enters a character from a keyboard, it will \_getch that char and searching for up to 10 queries that start with the matching char.

To print out the history suggestion, we have to design the console, interact with the cursor to choose the place to output and back to the input line after printing the suggestion.

This process will stop when user press Enter (equivalent to character ‘\r’) and we get an input query from the user to use in the next process.

* 1. *Input rule*

If you input query, you cannot type blank space at the beginning, the backspace will make no sense. You cannot also type two consecutive backspace.

* 1. *Help command*

You can type “help()” to get instructions how to use this program.

* 1. *View history command*

You can type “viewHistory()” to view all the queries you have searched.

* 1. *Delete history command*

You can type “clearHistory()” to delete all the queries you have searched.

* 1. *Exit command*

You can type “exit()” instead of enter query to exit the program.