Final-Project Report

**MINI SEARCH ENGINE**

1. Introduction

Search Engine is a software system that is designed to search for information. Nowadays, Search Engine is available in many fields to deliver a suitable approach to the information. The manner might be different, depend on kind of abstraction data type (ADT) but the main core of each Search Engine is the mechanism of crawling, indexing and querying.

**Mechanism:**

* Search engines get their information by crawling from (object to object) site to site. The "spider" checks for the standard filename *Data1677.txt*, addressed to it, before sending certain information back to be indexed depending on many factors, such as the titles, content, headings, as evidenced by the standard.
* Indexing means associating words and other definable tokens found on object (files, pages,…) to their position. In addition, they are ranking to a certain way guarantee the result is the best recommendation.
* A query from a user can be a single word or a long phrase. The index helps find information relating to the query as quickly as possible.

1. Materials and Methods:

In this project, we you tries to store data. Tries is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.

We used trie to indexing the data, with the amount of data is big enough, trie show that it is efficient in term of memory because of reusing nodes.

Thus we have only one trie, e.i. all words in these files will be stored in the same trie.

struct Trie {

Node \*root = new Node;

~Trie();

void Destruct(details\*);

void Destruct(Node\*);

void Insert(const char\*,int,int); // word, file\_id, pos\_in\_content

Node \*Find(char\*);

void Show\_trie(Node \*cur,string s ="");

};

Trie has nodes. If a certain word is a complete word, in the “insert to trie” process, the linked-list of struct details \*info contain:

struct details {

int file\_id;

details \*next = NULL;

vector<int> pos;

};

* “int file\_id” : a specific number represent for filename.
* details \*next : is the pointer to the next file that contain keyword;
* vector<int> pos : is an array contain the position of that word in the current file.

Thus, with 1 Node we will know: Which file contain that word, where they are and the number of occurrences (by pos.size() method).

class Node {

public:

Node() {for (int i=0;i<256;++i) next[i] = NULL;}

Node \*next[256];

details \*info = NULL; // info is a linkedlist

int point = 0; // the times this word appeared in the whole trie

};

struct Data {

char filename[20]; // show file name ex. Data1005.txt

char \*title = NULL;

char \*content = NULL; // pointer point to the string contain file content.

};

struct Data use to show the snippets that contain keyword in the console window.

Trie-manipulate:

* Insertion:

The inserted word is lead to the suitable place, update the information (file, position, point).

void Trie::Insert(const char\* s,int file\_id,int pos) {

Node \*cur = root;

for (int i=0;i<strlen(s);++i) {

if (cur->next[s[i]] == NULL)

cur->next[s[i]] = new Node;

cur = cur->next[s[i]];

}

cur->point++;

if (cur->info == NULL) {

cur->info = new details;

cur->info->file\_id = file\_id;

cur->info->pos.push\_back(pos);

}

else {

details \*info\_cur = cur->info, \*info\_prev;

while (info\_cur != NULL && info\_cur->file\_id != file\_id)

info\_prev = info\_cur, info\_cur = info\_cur->next;

if (info\_cur == NULL) {

info\_prev->next = new details;

info\_cur = info\_prev->next;

info\_cur->file\_id = file\_id;

}

info\_cur->pos.push\_back(pos);

}

}

* Searching:
* The first iFind function to find the node that suitable with keyword and return its point.

int Trie::iFind(char \*s) {

Node \*cur = root;

for (int i = 0;i<strlen(s);++i) {

if (cur->next[s[i]] == NULL)

return false;

cur = cur->next[s[i]];

}

return cur->point > 0;

}

The second find function is return the node queried.

Node\* Trie::Find(const char \*s) {

Node \*cur = root;

for (int i = 0;i<strlen(s);++i) {

if (cur->next[s[i]] == NULL)

return NULL;

cur = cur->next[s[i]];

}

return cur;

}

The third function used to check another operant attached.

The char\* word point to the keyword, while char c is the prefix (such as #,$ and blank-space “ “ for nothing and be checked by tracking the previous tokens in content of that file. When Node found, It is returned, thus we got the linked list contain information about the keyword (where it is, position, number of occurrence), otherwise NULL be returned. Finally, “intitle” variable use to determine search processing in title or content.

Node\* Trie::Find(char c, const char\* word, Data \*&database, int intitle){

Node \*tmp = Find(word);

details \*info = tmp->info;

if (tmp == NULL) return NULL;

Node \*res = NULL;

details \*res\_info;

char \*s = new char[105000];

vector<int> pos;

while (info) {

pos.clear();

if (intitle) strcpy(s, database[info->file\_id].title);

else strcpy(s, database[info->file\_id].content);

for (vector<int>::iterator i = info->pos.begin();i != info->pos.end();++i)

if (s[\*i - 1] == c) pos.push\_back(\*i - 1);

if (pos.size() > 0) {

if (res == NULL) {

res = new Node;

res->info = new details;

res\_info = res->info;

res\_info->file\_id = info->file\_id;

res\_info->pos = pos;

res->point = pos.size();

}

else {

res\_info->next = new details;

res\_info = res\_info->next;

res\_info->file\_id = info->file\_id;

res\_info->pos = pos;

res->point += pos.size();

}

}

info = info->next;

}

delete[]s;

return res;

}

Solution for query process:

1. AND (A AND B)

We will search each term independently. One file is fine if it contains all input operand of AND operator.

1. OR (A OR B)

We will search each term independently. One file is fine if it contains only one term of them.

1. – (e.g. Manchester -united)

For example: A -B.

We just need to search for **A** in this trie. If a file has **A** we’ll continue to search for **B** in this trie. One file is fine if it contains **A** and doesn’t contain **B**

1. + (e.g. Peanut Butter +“and Jam”)

For example: A +B.

If a file has **A** we’ll continue to search for **B** in this trie. One file is fine if it contains **A** and be ranking up if contain **B**

1. In title:

For example: intitle:debate

One file is fine if in its title contain keyword.

For example: intitle:”debate in NewYork”

One file is fine if in its title contain exact the whole phrase ”debate in NewYork”

1. Filetype:

For example: intitle:txt

First its point (pos.size() method) must bigger than zero, e.i. It contains keywords in query. Then we check the file extension, matched ones will be ranking and show in console.

1. Hashtag #

Thank to vector<int> pos we can determine the position of query. Thus, one file is fine if it contain all keyword and position of sharp(#) are ordered in font of keyword.

1. Dollar $

Thank to vector<int> pos we can determine the position of query. Thus, one file is fine if it contain all keyword and position of Dollar($) are ordered in font of keyword.

1. Exact search with speech mark “ ”:

Thank to vector<int> pos we can determine the position of query. Thus, one file is fine if it contain all keyword and position of them are ordered continuously.

File and data manipulate:

We use console-command to get the number of file, filename,…

Load and save them in data to use later show in console window).

for (int i=0;i<number\_of\_file;++i)

Free\_file(database[i].title,database[i].content);

delete []database;

void Free\_file(char\*& title, char\*& content) {

delete []title;

delete []content;

}

system("ls -F CS163-Data |grep -v / | wc -l > file.tmp");

ifstream fin("file.tmp"), fin\_2;

fin >> number\_of\_file;

fin.close();

system("rm file.tmp");

Data \*database = new Data[number\_of\_file];

system("ls CS163-Data > file.tmp");

All of them is freed when the program is closed.

void Trie::Destruct(details \*cur) {

if (cur == NULL) return;

Destruct(cur->next);

delete cur;

}

void Trie::Destruct(Node \*cur) {

if (cur == NULL) return;

Destruct(cur->info);

for (int i=0;i<256;++i)

Destruct(cur->next[i]);

delete cur;

}

Indexing:

We have an 256-element array is\_acceptable\_char. Each of these characters: $, #, @, &, \*, % is supposed to be a word. So their is\_acceptable\_char values are 1. Digits from 0 to 9 have is\_acceptable\_char values which are 2. The rest characters have is\_acceptable\_char values which are 0.

int is\_acceptable\_char[256] = {0};

is\_acceptable\_char['$'] = is\_acceptable\_char['#'] = is\_acceptable\_char['@'] = 1;

is\_acceptable\_char['&'] = is\_acceptable\_char['\*'] = is\_acceptable\_char['%'] = 1;

for (int i='0';i<='9';++i) is\_acceptable\_char[i] = 2;

We create two tries, one contains title of documents, the other contains content of documents.

Trie T\_title; Trie T\_content;

Data from files have been stored in database array. For each file, we enter titles to T\_title (trie for titles) first, then enter contents to T\_content (trie for contents). The entering process of titles and contents are the same. For a string from titles or contents, we split it into words and add every single word to the corresponding trie. So how do we split a string into words? If a character has its is\_acceptable\_char value = 1, it is supposed to be a word, we just need to add it to the trie. Else if that character is a letter of English alphabet or a digit, we create a temporary string (tmp) to contain that character. Next, while the next character is still a letter or a digit or a apostrophe ( ' ) or a hyphen ( - ) which is used to join two words together to make a new one, we continue to add that character to the last of tmp. When we can not continue, tmp is a word which is splitted from the string, we add tmp to the trie.

for(int i=0;i<number\_of\_file;i++) {

//Title

int len=strlen(xau);

string tmp;

for(int j=0;j<len;j++)

if (isascii(xau[j]))

{

if (is\_acceptable\_char[xau[j]] == 1) {

tmp="^";

tmp[0]=xau[j];

T\_title.Insert(tmp.c\_str(),i,j);

}

else if (isalnum(xau[j])) {

tmp=tolower(xau[j]);

int k=j;

while (j<len-1)

if (isalnum(xau[++j]) || xau[j]=='\'' || xau[j]=='-')

tmp+=tolower(xau[j]);

else break;

if (is\_acceptable\_char[xau[j]] == 1) {

string temp="^";

temp[0]=xau[j];

T\_title.Insert(temp.c\_str(),i,j);

}

T\_title.Insert(tmp.c\_str(),i,k);

if (!isalnum(tmp[tmp.size()-1])) {

tmp.erase(tmp.end()-1);

T\_title.Insert(tmp.c\_str(),i,k);

}

}

}

//Content

len=strlen(chuoi);

for(int j=0;j<len;j++)

if (isascii(chuoi[j]))

{

if (is\_acceptable\_char[chuoi[j]] == 1) {

tmp="^";

tmp[0]=chuoi[j];

T\_content.Insert(tmp.c\_str(),i,j);

}

else if (isalnum(chuoi[j])) {

tmp=tolower(chuoi[j]);

int k=j;

while (j<len-1)

if (isalnum(chuoi[++j]) || chuoi[j]=='\'' || chuoi[j]=='-')

tmp+=tolower(chuoi[j]);

else break;

if (is\_acceptable\_char[chuoi[j]] == 1) {

string temp="^";

temp[0]=chuoi[j];

T\_content.Insert(temp.c\_str(),i,j);

}

T\_content.Insert(tmp.c\_str(),i,k);

if (!isalnum(tmp[tmp.size()-1])) {

tmp.erase(tmp.end()-1);

T\_content.Insert(tmp.c\_str(),i,k);

}

}

}

}

We entered stopwords from a file and add them to a trie (T\_stop).

Trie T\_stop;

fin.open("stopwords.txt");

while (getline(fin,String))

T\_stop.Insert(String.c\_str(),0,0);

fin.close();

Query process:

The user enters a query. The query is stored by a string. If the string is “quit\_”, the program stops. If the query is AND, flag = 1, if the query is OR, flag = 0,… (each type of queries has a corresponding flag value). We split the string into words. We add information like flag, words which have been splitted,… to the Query function. This function searches in all files and returns rating values. rating is an array contains number of appearances of words in files. If rating[i]<0, file i does not contain words. Base on rating array, we can know which file contains words.

string s, s2, ss; int flag, first, n\_input, intitle = 0;

string file\_type = "";

char \*TITLE = new char [10];

strcpy(TITLE,"intitle:");

Trie\* TT;

vector<string> input;

while ((cout << "> ") && (getline(cin,String)) && (String != ":quit\_")) {

if (String == "\n" || String == "" || Check\_all\_space(String)) continue;

TT = &T\_content; intitle = 0;

ss = String;

input.clear();

ZERO(rating);

while (Next\_token(String,s)) input.push\_back(s);

for (vector<string>::iterator i=input.begin();i!=input.end();++i)

ToLower(\*i);

//

first = true; flag = -2;

n\_input = input.size();

for (int i=0;i<input.size();++i) {

if (flag == -1) {

if (input[i] == "and" || input[i] == "AND") flag = 1;

if (input[i] == "or" || input[i] == "OR") flag = 0;

}

if (T\_stop.iFind(input[i].c\_str())) continue;

if (input[i] == "AND" || input[i] == "OR") continue;

// intitle:

if (input[i].length() > 7) {

int is\_ok = 1;

for (int j=0;j<8;++j)

if (input[i][j] != TITLE[j]) is\_ok = 0;

if (is\_ok) {

intitle = 1, TT = &T\_title;

Next\_token(input[i],s2,':');

}

}

if (first) {

if (n\_input > 1 && input[i+1] == "and") flag = 1;

else flag = 0;

Query(flag,\*TT,input[i].c\_str(),rating,number\_of\_file,database,intitle);

first = 0;

flag = -1;

}

else {

if (flag == 0 || flag == 1)

Query(flag,\*TT,input[i].c\_str(),rating,number\_of\_file,database,intitle);

else {

if (input[i][0] == '-') {

input[i].erase(0,1);

flag = -oo;

}

else if (input[i][0] == '+') {

input[i].erase(0,1);

flag = 2;

}

else flag = 0;

Query(flag,\*TT,input[i].c\_str(),rating,number\_of\_file,database,intitle);

}

flag = -1;

}

}

1. Result:

* In this project, we apply the Search Engine support 9 query types, corresponding to these operant:
* AND
* OR
* –
* +
* In title
* Filetype
* $ (Money, price information)
* # (Hash tag)
* Speech mark for Search exact (“ ”)

In addition, we deliver a manner to query by using hybrid method that more than 2 operant in 1 query input.

E.g. “Donald Trump” -debate AND “Hillary Clinton” filetype:html

“Donald Trump” AND “Hillary Clinton” AND “Barack Obama”

* Memory cost: About 400MB.

1. References:

[1] <https://bynd.com/news-ideas/google-advanced-search-comprehensive-list-google-search-operators/>

[2] <https://supple.com.au/tools/google-advanced-search-operators/>

[3] <https://support.google.com/websearch/answer/2466433?hl=en>