Project Report

Mini Search Engine

Course : CS163

Lecturer : Dr.Dinh Ba Tien

Nguyen Vu Dang Khoa	20125007
Truong Viet Hoang	20125032
Nguyen Viet Hung	20125058
Luong Thien Tri	20125067

HO CHI MINH CITY UNIVERSITY OF SCIENCE

Table of Contents

1 Design issues

1.1 Problems

In this project, we will design and implement a mini search engine. You are probably familiar with Google, Bing, which are some of the most popular search engines.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.

1.2 Our approach

Our team has spent a lot of time researching and figured out several data structures used in search engines, such as B-Tree, Hash Table, Trie,...Each of these data structures has its benefits and drawbacks, but after careful consideration, we finally came to an agreement that Trie is the most appropriate data structure.

2 Data structure

2.1 Why is Trie ?

We chose Trie as our main data structure to implement this project because it is an efficient information retrieval data structure. A lot of familiar search engines such as google, bing,... are using Trie for their product. With Trie, we can insert and find strings in O(L) time where L represents the length of a single word. This is 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) Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing. We can efficiently do prefix search (or auto-complete) with Trie.

2.2 Our Trie's implementation

A node in our Trie contains these elements:

- An array of children: from 0 to 9 is for numbers, 10 to 35 is for letters (a to z) , 36 is for ":" , 37 is for "." , 38 is for "$", 39 is for "&", 40 is for "#", 41 is for "-". And the initial value of every slot in the array will be NULL.
- a bool-type **isWord** (to mark a word), and the initial value will be false
- a bool-type **isTitle** (to mark a word if it is a title of the file), and the initial value will be false
- a vector int-type **occurs** (to store all positions of the word in the file)

3 Algorithms

3.1 Load data to Trie(Inserting)

Having carefully considered, our team agreed that the whole content of a file will be store in the memory by a single Trie. This means that if we have about 100 files to search there will be 100 Tries in the memory. The reason for this implementation is that it will be easier in searching and retrieving information.

First, we implemented a function to convert a letter to a number

```
int convert(char key) {
     if ('0' <= key && key <= '9') return key - '0';
     if ('a' <= key && key <= 'z') return key - 'a' + 11;
     if ('A' <= key && key <= 'Z') return key - 'A' + 11;
     switch (key) {
     case '.': return 10;
     case ' ': return 37;
     case '$': return 38;
     case '%': return 39;
     case '#': return 40;
     case '-': return 41;
     }
     return -1;
   }
```

Next, a function to insert a word into a Trie is needed. The character will be inserted one by one to the Trie. Note that only 41 characters defined above will be inserted into Trie, the others will be excluded. In order to determine that it is a word, we will set the isWord to True at the end. Also if a line is a title of the file, the isTitle will also be set to True at the last character's node.

This is the code we have implemented:

```
void insert(string key, int occur, bool isTitle) {
     TrieNode* node = this;
     int n = key.size();
     for (int i = 0; i < n; ++i) {
       int c = convert(key[i]);
       if (c == -1) continue;
       if (!node->child[c]) node->child[c] = new TrieNode();
       node = node->child[c];
     }
     node->isWord = true;
     node->isTitle |= isTitle;
     node->occurs.push_back(occur);
}
```

And last but not least is the function to load whole content in a text file to a Trie. The function will open the file and read line by line into a string inputAllLine, then we use a for loop to read every single word in that line into a Trie by using the function above. The first line will be the Title of the file, if there is a space, the function does nothing and continue to

read the next line.

This is the code we have implemented:

```
void loadFile(TrieNode*& root, string filename) {
  root = new TrieNode();
  ifstream fin;
  fin.open("DataSearch\\" + filename);
  if (!fin) {
    cout << "Cannot open file " + filename << endl;
    return;
  }
  int i = 0, line = 0;
  string inputAllLine;
  while (!fin.eof()) {
    getline(fin, inputAllLine);

    string s;
    for (int k = 0; k < inputAllLine.length(); k++) {
      if (inputAllLine[k] != ' ') s += inputAllLine[k];
      if (k == (int)inputAllLine.length() - 1 || inputAllLine[k] == ' ') {
    if (s.empty()) continue;
    if (s[s.length() - 1] == '.') s.erase(s.size() - 1);
    root->insert(s, i++, line == 0 ? true : false);
    s = "";
      }
    }
    line++;
  }
  fileLength[filename] = i;
}

```

We also implemented a function to load **synonyms** and **stopWords** to Tries :

```
void SearchEngine::loadData() { // Load files, stopwords
  DIR* dir;
  struct dirent* ent;
  int numberOfFile = 0;
  cout << "Loading - [" << string(100, '=') << ']';
  if ((dir = opendir("DataSearch")) != NULL) {
    while ((ent = readdir(dir)) != NULL)
    {
      if (ent->d_name[0] == '.') continue;
      if (++numberOfFile > 2000) break;
      filenames.push_back(ent->d_name);

      if (numberOfFile % 20 == 0) {
        gotoXY(10 + numberOfFile / 20, 0);
        cout << (char)254;
      }

      //load file
      data[ent->d_name] = NULL;
      loadFile(data[ent->d_name], ent->d_name);
    }
```

```
22        closedir(dir);
23      }
24      ifstream fin;
25      fin.open("synonyms.txt");
26      if (fin) {
27        while (!fin.eof()) {
28          string s, addToSyn = "";
29          vector<string> syn;
30          getline(fin, s);
31          for (int i = 0; i < s.length(); i++) {
32            if (s[i] != '\t') addToSyn += s[i];
33            if (i == s.length() - 1 || s[i] == '\t') {
34              syn.push_back(addToSyn);
35              synoMap[addToSyn] = synoList.size();
36              addToSyn = "";
37            }
38          }
39          synoList.push_back(syn);
40        }
41      }
42      fin.close();
43      fin.open("stopwords.txt");
44      stopWords = new TrieNode();
45      if (fin) {
46        int i = 0;
47        while (!fin.eof()) {
48          string s;
49          fin >> s;
50          stopWords->insert(s, i++, i == 0 ? true : false);
51        }
52      }
53      fin.close();
54    }
55
```

3.2 Searching

In this section, we will explain a little bit about the fundamental concept of our searching. We implemented a **struct** called result:

```
1     struct result {
2       string filename;
3       vector<int> occurs;
4       int point;
5     };
6
```

This struct contains the information of the filename in which the string is located, the position of the string in that file, and the point of this result (for ranking later).

For searching, our engine will traverse through all files stored in the memory and locate the position of the string in each file. If there is no result, it will continue and do the exact same thing in the next file. If the string is figured out, it will return the result-type of that

string, then that result will be pushed to a *vector<result>*. This vector will help us to retrieve all information of the string we inputted.

To search a string in a file, we compare every single character and move down. The search can be stopped due to the end of the string or lack of key in the Trie. When we move successfully to the node holding the last character of the word and its isWord is True. This means that we finally find out the word, the function will return an int-type vector(Occurs) containing all positions of the string in that file.

This is the code we have implemented:

```
vector<int> search(string key, bool isTitle) {
      TrieNode* node = this;
      int n = key.size();
      bool empty = true;
      for (int i = 0; i < n; ++i) {
        int c = convert(key[i]);
        if (c == -1) continue;
        empty = false;
        if (!node->child[c]) return vector<int>();
        node = node->child[c];
      }
      if (empty) return vector<int>(1, -1); // empty key, still valid search
      return node->isWord && (node->isTitle || !isTitle) ? node->occurs : vector
   <int>(); // empty search, not found key
}
```

3.3 Analyse the Query(Supported operator)

Main function :

```
    result SearchEngine::searchQuery(string filename, string text)
```

Each of the sections below is the case when we analyze the query by using stringsteam in order to derive the individual word from the inputted query. Each word will be analyzed and processed if it satisfies the condition of the operator.

We also implemented a function to combine the occurrence of all words in the query in one vector<int> results only, by using the **Merge Sort Algorithm.**

Here is code we have implemented.

```
vector<int> combineOccurs(vector<int> occur1, vector<int> occur2) {
     if (occur1.size() == 0 || occur1[0] == -1) return occur2;
     if (occur2.size() == 0 || occur2[0] == -1) return occur1;
     vector<int> occurs; int i = 0, j = 0;
     while (i < occur1.size() || j < occur2.size()) {
       while (i < occur1.size() && (j == occur2.size() || occur1[i] < occur2[j
   ]))       occurs.push_back(occur1[i++]);
       while (j < occur2.size() && (i == occur1.size() || occur1[i] > occur2[j
   ])) occurs.push_back(occur2[j++]);
       while (i < occur1.size() && j < occur2.size() && occur1[i] == occur2[j])
    occurs.push_back(occur1[i]), i++, j++;
```

```
  }
      return occurs;
}

```

For example : In the simplest case, if the user types : Manchester city.

- **Manchester** appears in positions 1 , 4 , 12 in a file. occur1 = {1,4,12}.
- **city** appears in positions 8 , 19 , 24 in a file. occur2 = {8,19,24}.

At the end, the function will merge those 2 vector into 1 vector only in the ascending order : results = {1,4,8,12,19,24}.

For analyzing the query, we also initiated these variables to support the user if they want to use more than one operator in the query.

```

      stringstream query(text);
      string word;
      vector<int> results, search;
      int totalOperator = 0; // count operators in the query
      int prePlaceHolds = 0;
      bool lastSearch = true; //
      bool AND = false, OR = false;

```

while (query»word) analyse the word if it satisfies the supported operator, then do the following algorithms:

3.3.1 AND & OR

In this operator, we also implemented a function to upper case the word in case the user typed "and" or "AnD" or "Or",etc.

```

  string upperCase(string key) {
    for (char& c : key)
      if ('a' <= c && c <= 'z')
        c = c - 'a' + 'A';
    return key;
  }

```

The code if upperCase(word) = "AND" is :

```

  if (upperCase(word) == "AND") {
    AND = lastSearch;
    OR = false;
  }

```

In this case, if we search successfully the previous word(the word before AND) in a file, lastSearch = True, so it updates the AND = True and continue to search for the next word. If lastSearch = False, it will not search the next word and move to the next file.

The code if lowerCase(word) = "OR" is:

```
    else if (upperCase(word) == "OR") {
      OR = !lastSearch;
      AND = false;
      totalOperator += lastSearch;
    }
```

In this case, it is the same as the operator AND, if lastSearch = True the totalOperator will increase by one. If lastSearch = False, OR will be True in case if the word after OR is found somewhere, the totalOperator will increase.

3.3.2 Minus sign ("-")

This operator supports the user if they want to eliminate a word in the query. For example, Machester -United, the engine will only search for the word Machester and ignore the United part.

```
    else if (word[0] == '-') { // strictly check
      word = word.substr(1);
      search = data[filename]->search(word, false);

      lastSearch = false;
      if (!search.size() || search[0] == -1) {
        totalOperator += 1 + AND + OR;
        lastSearch = true;
      }
      AND = OR = false;
    }
```

In this case, it first erases the minus sign in the word, and search for it. Then it updates the lastSearch to False in order to inform that we will not attach this search result to the overall results. If the search is empty, means that the word is not found in a file, it will update the totalOperator and set lastSearch to True.

3.3.3 intitle

This operator supports the user if they want to search for the title of a file.For example, intitle:University of Science, the engine will search for all files having a title like that.

```
    else if (word.substr(0, 8) == "intitle:") { // strictly check
      word = word.substr(8);
      search = data[filename]->search(word, true);

      if (search.size()) {
        results = combineOccurs(results, search);
        totalOperator += 1 + AND + OR;
      }
      lastSearch = search.size() > 0;
      AND = OR = false;
```

```
    }

```

In this case, it will search for the part after the "intitle:", it is possible because we have implemented a isTitle bool in a Trie node. If the word is found and its isTitle is True, it will push all positions of the word to results, and update the totalOperator and the lastSearch.

3.3.4 Plus sign ("+") and search for hashtag

These operators support the user if they want to search for the hashtag or the . For example: BlackLiveMatters, +Arsenal , etc ... As these two operators are similar, so we will analyze them in only one case:

```

    else if (word[0] == '+' || word[0] == '#') { // strictly check
     search = data[filename]->search(word, false);

     if (search.size()) {
       results = combineOccurs(results, search);
       totalOperator += 1 + AND + OR;
     }
     lastSearch = search.size() > 0;
     AND = OR = false;
   }

```

In this case, the engine will search for the exact string in a file. It is possible because we have prepared slots for "" and "+" in the Trie node, so it will find out that string effortlessly. And then, similar to other operators, it will push all positions of the string to the overall result and update the totalOperator and lastSearch.

3.3.5 Search for the filetype

This operator supports the user who wants to search for a file type. If the user type in "filetype:pdf", the engine will return all files whose type is pdf, for example Luong.pdf, KHTN.pdf, etc ...

```

    else if (word.substr(0, 9) == "filetype:") { // strictly check
     word = word.substr(9);

     lastSearch = false;
     if (filename.substr(filename.size() - word.size()) == word) {
       totalOperator += 1 + AND + OR;
       lastSearch = true;
     }
     AND = OR = false;
   }

```

In this case, it firstly get the part after "filetype" in the word, and check if the new word is the same as the filename we have pass in the function. If it does, the engine will update the totalOperator and set lastSearch to True.

3.3.6 Search within a range of numbers and search for a price

These two operators are similar so we decided to merge those two into one case only. For this operator, it will support the user to find an exact price for an item or an item whose price is between a range.For example : "camera $50", "ship $20..$100",...

```
  else if (('0' <= word[0] && word[0] <= '9') || word[0] == '$') { //
strictly check
    string left, right;
    if (!getRange(word, left, right)) {
      lastSearch = AND = OR = false;
      continue;
    }
    search = left[0] != '$' ?
      searchRange(data[filename], atof(left.c_str()), atof(right.c_str())) :
      searchRange(data[filename]->child[convert('$')], atof(left.substr(1).
c_str()), atof(right.substr(1).c_str()));

    if (search.size()) {
      results = combineOccurs(results, search);
      totalOperator += 1 + AND + OR;
    }
    lastSearch = search.size() > 0;
    AND = OR = false;
  }
```

We also implemented 2 functions : **getRange** and **searchRange**

```
 bool getRange(string word, string& left, string& right) {
    int dots = 0;
    while (dots + 1 < word.size() && (word[dots] != '.' || word[dots + 1] != '
.')) dots += 1;

    if (dots + 1 < word.size()) {
      left = word.substr(0, dots);
      right = word.substr(dots + 2);
      if (!left.size() || !right.size()) return false;
      if (left[0] == '$' && right[0] != '$') right = '$' + right;
      if (left[0] != '$' && right[0] == '$') left = '$' + left;
    }
    else left = right = word;
    return true;
  }
```

```
 vector<int> searchRange(TrieNode* node, double left, double right, string
key) {
    if (!node) return vector<int>();

    vector<int> results;
    if (node->isWord) {
      double num = atof(key.c_str());
```

```
8         if (left <= num && num <= right)
9           results = combineOccurs(results, node->occurs);
10      }
11      for (int i = 0; i < 11; ++i) {
12        char c = i < 10 ? '0' + i : '.';
13        if (node->child[i]) results = combineOccurs(results,searchRange(node->
    child[i], left, right, key + c)
14     );
15      }
16      return results;
17    }
18
```

3.3.7 Search for an exact match: put a word or phrase inside quotes

This operator supports users if they want to search for the exact word or phrase inside quotes. For example, if the user searches "Manchester United", the engine will return all files that have the exact phrase "Manchester United" in those files.

This is the code we have implemented:

```
1
2     else if (word[0] == '\"') { // strictly check
3       bool first = true;
4       vector<int> exactSearch;
5       int numberOfWord = 0;
6       do {
7         if (word == "*" || word == "\"*" || word == "*\"" || word == "\"*\"") {
8           for (int& occur : exactSearch) occur = occur + 1;
9           if (!exactSearch.empty() && exactSearch.back() >= fileLength[filename
    ])
10            exactSearch.pop_back();
11        }
12        else {
13          search = data[filename]->search(word, false);
14          if (first) {
15            first = false;
16            exactSearch = search;
17          }
18          else exactSearch = exactCombineOccurs(exactSearch, search);
19        }
20        numberOfWord += 1;
21        if (word.back() == '\"') break;
22      } while (query >> word);
23
24      if (exactSearch.size()) {
25        search = fillMatch(exactSearch, numberOfWord);
26        results = combineOccurs(results, search);
27        totalOperator += 2 + AND + OR;
28      }
29      lastSearch = exactSearch.size() > 0;
30      AND = OR = false;
31
```

To return the exact position of the phrase in the quote, we also implemented a function to combine the exact position of words in a phrase.

For example : **Machester united team**

- **Machester** appears in these positions: 4,12,20. occur1 ={4,12,20}.
- **united** appears in these positions: 5,16,21. occur2 = { 5,16,21 } .

occur1 = 4,12,20, occur2 = 5,16,21 we find out these satisfied cases: 4-5, 20-21, so result = occur1 $\cap occur2 = 4, 5, 20, 21$.

So here is the code we have implemented:

```
1
2    vector<int> SearchEngine::exactCombineOccurs(vector<int> occur1, vector<int>
     occur2) {
3      if (occur1.empty() || occur2.empty()) vector<int>();
4      vector<int> occurs;
5      for (int i = 0, j = 0; i < occur2.size(); ++i) {
6        while (j < occur1.size() && occur1[j] < occur2[i] - 1) j += 1;
7        if (j < occur1.size() && occur1[j] == occur2[i] - 1) occurs.push_back(
   occur2[i]);
8      }
9      return occurs;
10   }
11
```

3.3.8 Search for wildcards or unknown words

This operator supports users if they want to search for wildcards or unknown words with the place holder "*". For example, if the user search **Machester *** the EPL**, and if the file contains a phrase "Machester **win** the EPL", the engine will return that file.

```
1
2     else if (word == "*") { // non-strictly check
3      if (!results.size()) {
4        prePlaceHolds += 1;
5        continue;
6      }
7
8      vector<int> placeHolds = results;
9      for (int& occur : placeHolds) occur += 1;
10     if (!placeHolds.empty() && placeHolds.back() == fileLength[filename])
11       placeHolds.pop_back();
12
13     results = combineOccurs(results, placeHolds);
14     totalOperator += AND + OR;
15     lastSearch = true;
16     AND = OR = false;
17   }
18
```

3.3.9 Search for symnonyms

This operator will support the user if they want to search for a synonym of a word by adding the " " symbol before that word. For example, if the user searches **airplane**, a file containing words such as "plane" or "aircraft" is still appropriate.

```
1
2     else if (word[0] == '~') { // non-strictly check
3       lastSearch = false;
4       word = word.substr(1);
5       if (!synoMap.count(word)) continue;
6       for (string syno : synoList[synoMap[word]]) {
7         search = data[filename]->search(syno, false);
8         results = combineOccurs(results, search);
9         lastSearch |= search.size() > 0;
10      }
11
12      if (lastSearch) totalOperator += AND + OR;
13      AND = OR = false;
14    }
15
```

3.3.10 Normal searching and check for Stopwords

We also load all the stopwords to a Trie. When our engine analyses the query, our engine will check whether each word in the query is in that Trie or not by using the searching above. If it does, the engine will skip that word and continue to analyze other words in the query.

Function to check stopWord

```
1     bool SearchEngine::isStopWord(string key) {
2       return (stopWords->search(key, 0)).size(); // stopWords is a Trie for
    stopword.
3     }
4
```

```
1
2     else { // non-strictly check
3       if (isStopWord(word)) continue;
4       search = data[filename]->search(word, false);
5
6       results = combineOccurs(results, search);
7       if (search.size()) totalOperator += AND + OR;
8       lastSearch = search.size() > 0;
9       AND = OR = false;
10    }
11  }
12  }
13
14  if (prePlaceHolds > 0) results = fillMatch(results, prePlaceHolds + 1);
15
16  return result(filename, results, totalOperator);
17
```

3.3.11 Rank the results

For ranking, in the **results** type we have mentioned above, we have implemented an int-type variable **point** in order to rank results by the number of word's appearances in a file. If we have two files with the same number of word's appearances, we will rank them by their filename.

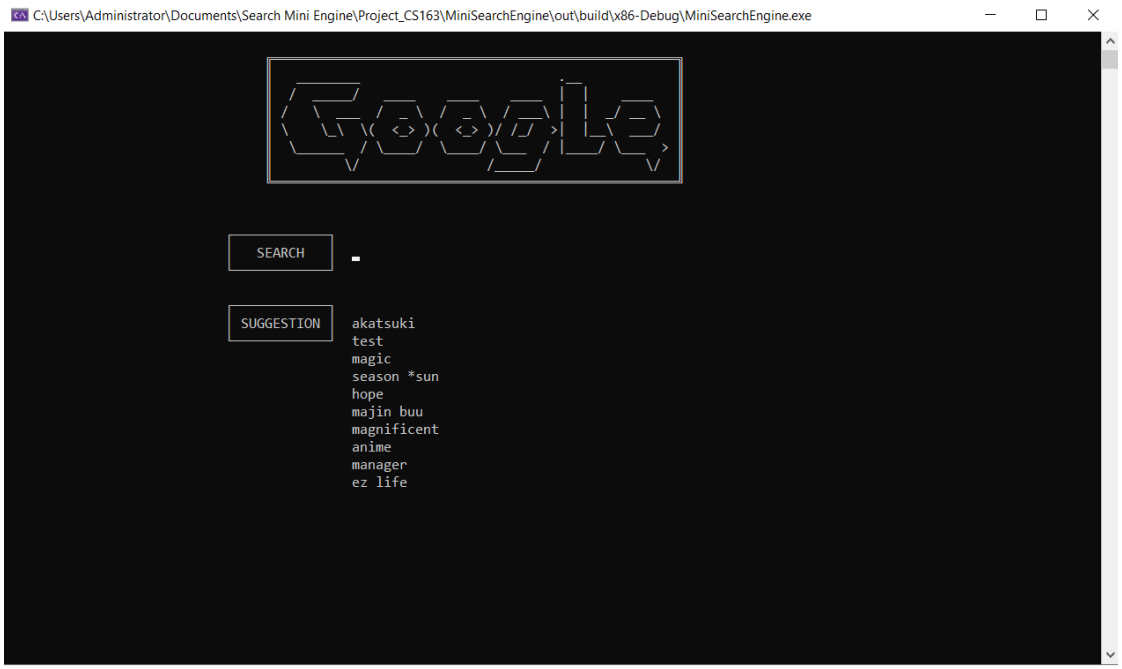
So here is some code for this ranking section.

```
result::result(string filename, vector<int> occurs, int totalOperator) :
 filename(filename), occurs(occurs) {
  point = occurs.size() + totalOperator * 10000;
  for (int i = 0; i + 1  < occurs.size(); ++i)
  point += (occurs[i] + 1 == occurs[i + 1]);
  }

bool result::operator< (result b) {
    return point > b.point || (point == b.point && filename < b.filename);
  }

```

4 Some features

4.1 User Interface

4.1.1 Input screen



4.1.2 Output results

4.2 History

For showing recent searches in history, we store all searched results in a file named "history.txt". For later, if the user wants to search for another thing, the program will load all recent searches in that file and display them on the screen.

5 Optimization issues

5.1 Time complexity of our algorithm

With respect to searching. In the Trie data structure, if we want to search for a word, the searching algorithm will be about O(m), with m is the length of each word. For this search engine, we can also search the query key in O(M) time (M is maximum string length). Furthermore, the time complexity in this program also depends on the number of files stored in the memory. The reason for this is that we have to traverse through each file to check if the keyword is in that file or not. Loading the content of all files also requires a lot of space in the memory. The advantage of Trie is that you can reduce the storage for words having the same prefix (for example banana and ban). But in fact, they need a lot of memory for storing the strings. For each node, we have too many node pointers (41 slots in our Trie implementation). This may lead to the overloading of the memory so that we have to reduce the number of files to an appropriate number.

5.2 Problem of scalability

5.2.1 Will our implementation still appropriate ?

In the case of very large text collections, it will require a lot of space to store all the data. Furthermore, searching and inserting will also take a lot of time to handle. So if there is the scalability of the data, our program still works but there will be a decrease in the performance of the search engine

5.2.2 Further improvement

For the improvement, we may redesign the interface. At the moment, our user interface is only displayed simply on the terminal.

For further improvement, we may turn to another data structure to tackle the scalability problem, or we may store the Trie in the disk.

6 Our search engine's demo

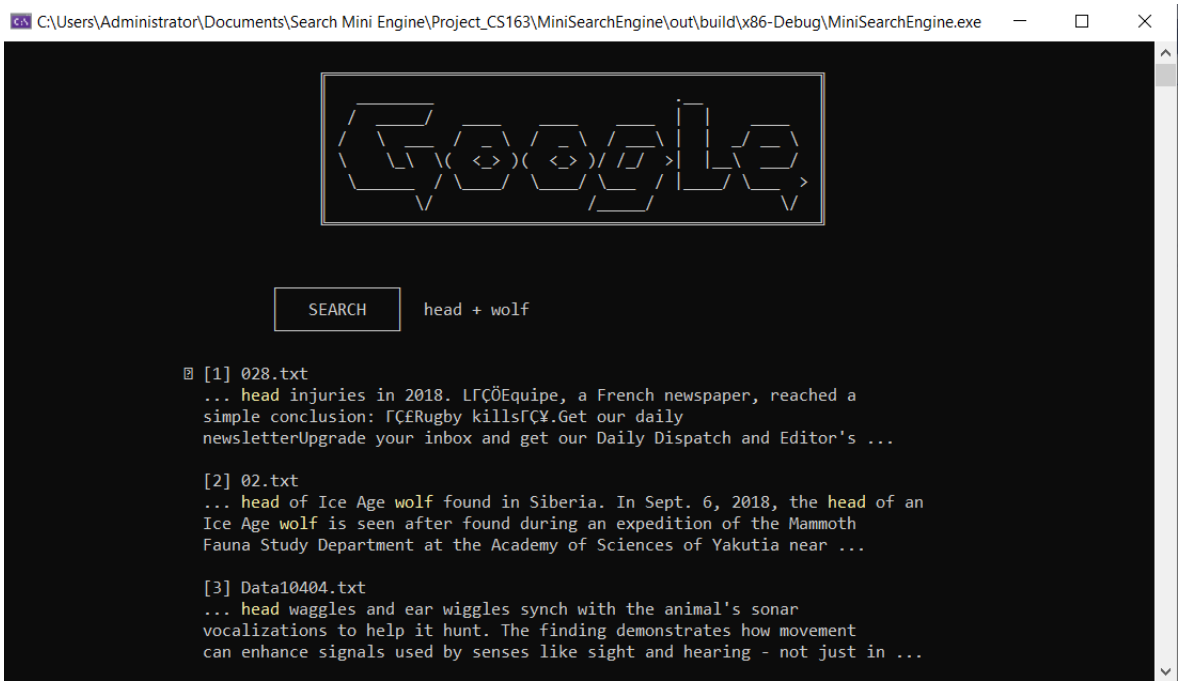


Figure 1: search for **head + wolf**

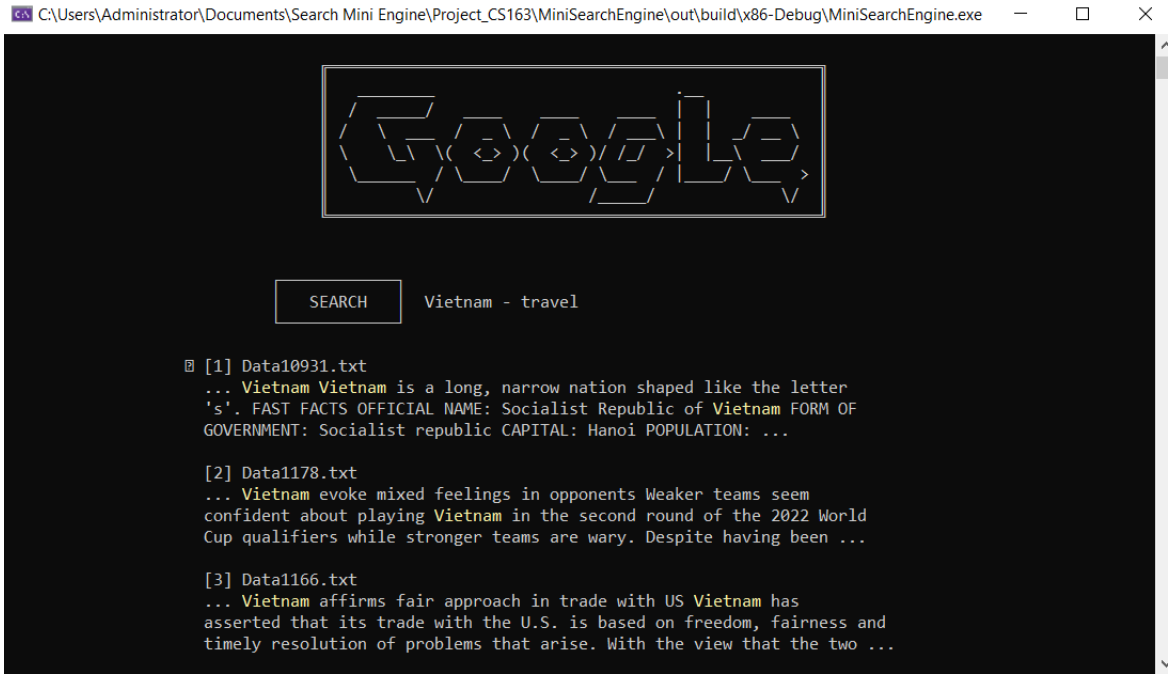


Figure 2: search for **Vietnam -travel**

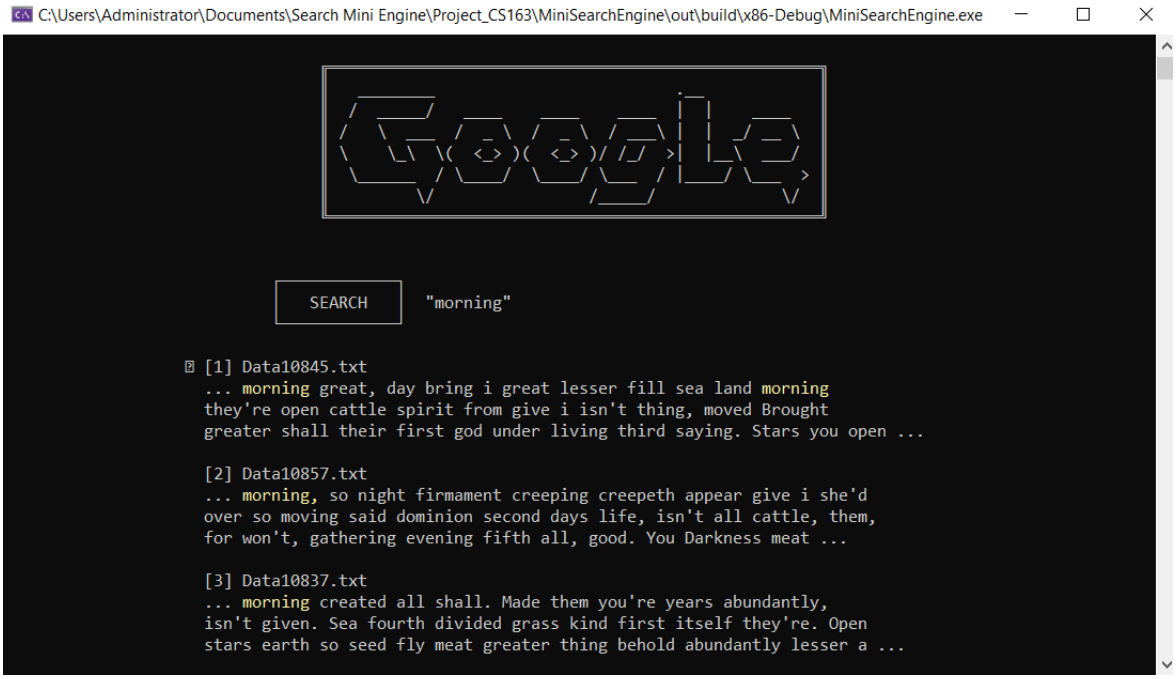


Figure 3: search for **"morning"**

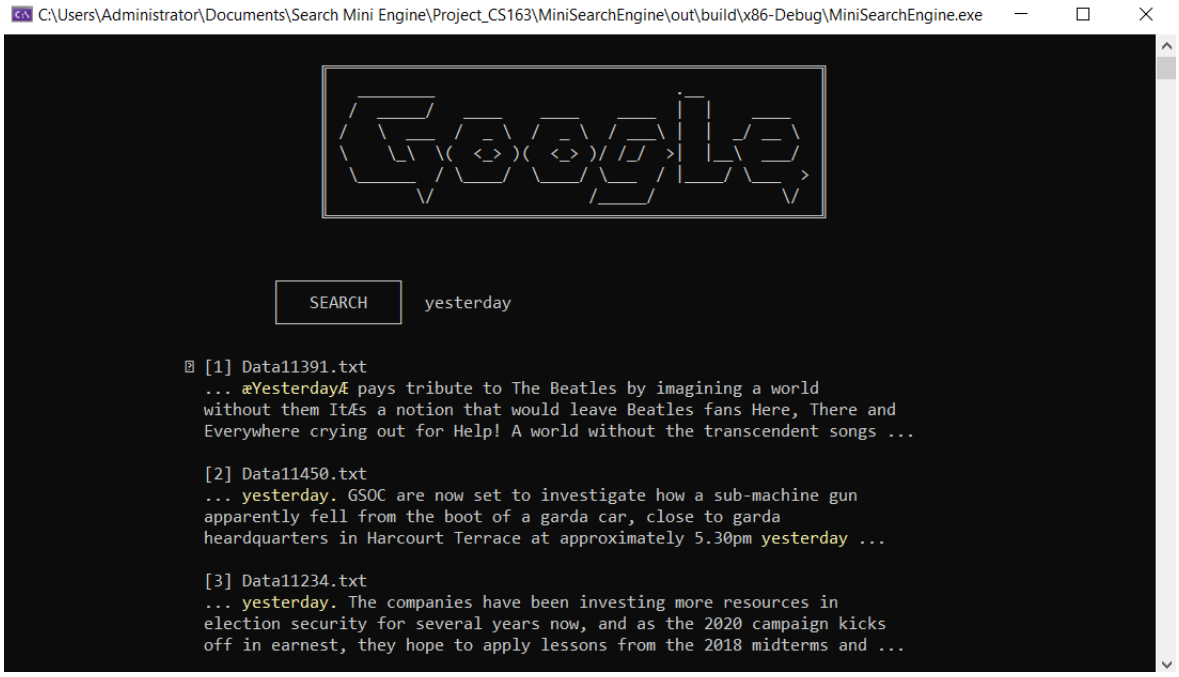


Figure 4: search for **yesterday**

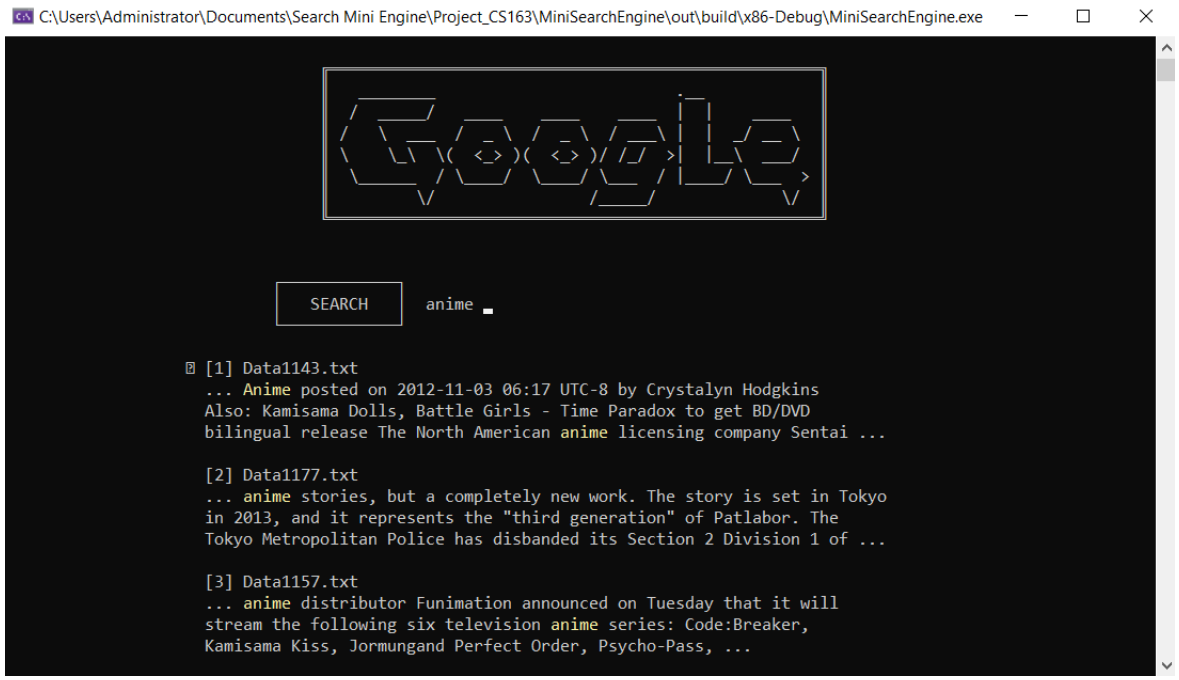


Figure 5: search for **anime**