
# Analysis of Algorithms Project Report

Course Code: CMP3005

Course Name: Analysis of Algorithms

Faculty: Engineering and Natural Sciences

Department: Computer Engineering

Group Members: Mehmet Ali Kısacık (1904537)
Mert Oğuz (1904634)
Ramazan Berkay Demircioglu (1729788)

Date: 02.01.2022

# PROBLEM DEFINITION

In this project we are required to find solution for the problem which is returning an answer for the question which the users enters to the program. The answer should be found in the text which is “The Truman Show Script” and the each answer per question is contained in one of the sentences of the script , an answer cannot be found in two of the sentences. Main requirement for the project is finding the answer correctly and making it fast as possible. Two programming languages were given in this project Java and C++.As from examples of this problem in web there are a lot of complex programs which find the answer of given question in much deeper methods such as “Deep Learning” or “Machine Learning” but this project doesn’t focus on these parts , purpose is the finding the right answer in a simple way in the scope of the course CMP3005.

# METHODOLOGY

## PROGRAMMING LANGUAGE

We have used C++ because if you can solve the problem C++ is faster compared to Java in general. C++ is more close to machine code , it is one of the reasons that why it is faster than Java.

## ITERATION

To solve the problem first we have considered the script that was given for us to find the answer , our aim was to iterate through every word in text and store them in a data structure which we have designed. A for loop with constant complexity is used to make the upper case of words lower case because in our data structure word “document” and “Document” are not the same words because of the ASCII value difference between upper and lower case letters.
Related code:

```
for(int i = 0;i<word.length();i++){
   if(word[i] > 64 && word[i] < 91){
      word[i] = word[i] + 32;
   }
}
```

Because the generic C++ iteration operator takes every strings between blank lines some strings like “adopt.” , “time.”” were shown as a word , we had truncate the punctiations at the end of the strings , we used if statements to do so and the newSentence property will be set to true.

Related code:

```
if (size >1 && (word[size-1] == '.'  || word[size-1] == '!' || word[size-1] == '?')  ){
   string modifiedWord(word.begin(),word.end()-1);
   word = modifiedWord;
```

```
    newSentence = true;     }
else if(size >1 && word[size-1] == 34 && (word[size-2] == '.' || word[size-2] == '!' ||
word[size-2] == '?') ){
    string modifiedWord(word.begin(),word.end()-2);
    word = modifiedWord;        newSentence = true;
}
```

# STOP WORD CHECKING

**Check function** is the function that returns boolen type value for example “the” is the stopword , check(“the”) returns true. Each word is checked whether it is a stopword before it is stemmed. Stop words are most common words in a language. The reason the stop words don’t get inserted in data structure is they are so common that they can even be in every sentence. That makes them redundant for sentence finding. Check functions complexity is constant because the arrays are predefined and there can be at most 26 inner loop comparisons for extreme case letter ‘W’.

Related Code:

```
if (check(word)) {
    if(newSentence){
        sentenceNum++;
        newSentence = false;
    }
    continue;
}
```

Check Function is so long because there are one array and one if/else if statement per letter , only one example for letter ‘O’ is shown below , the array for the words starts with ‘O’ are defined like this:

```
string o[14] =
{"of","off","on","once","only","or","other","ought","our","ours","ourselves","out","over","own"
};
```

Then the checking occurs for each letter:

```
    else if (str[0] == 'o'){
            for(int i = 0; i < sizeof(o)/sizeof(string); i++){
                if (o[i] == str){
                    return true;
                }
            } }
```

**Stemming** is used for every iterated word. Oleander Stemming Algorithm is used in this project which has constant time complexity for English language.

Related Code:

```
word = stem(word);
stemming::english_stem<> StemEnglish;
string stem(string word){
   wstring w(word.begin(),word.end());
   StemEnglish(w);
   string str(w.begin(),w.end());    return str; }
```

# SENTENCE NUMBERS

Every sentence is being numbered in this project numbered , for example the first sentences number is 0 , third sentences number is 2. In truncation part for the strings if the last index of word is '!' , '.' Or '?' the newSentence property will be set to true. At the end of each iteration if newSentence is true sentenceNum will be incremented by 1.

Related code:

```
        if (newSentence){
           sentenceNum++;
           newSentence = false;
        }
```

# NODES

The words and their properties are stored in nodes in this project because the binary tress will be used for data structure . Along with the left and right pointers which link to their left and right nodes , a string word property and sentence list property are stored. Sentence list uses generic lists in C++ this list stores the sentence numbers of the words which the words are used.

```
struct Node
{
   string word;
   Node *left = nullptr;
   Node *right = nullptr;
   list<int> sentenceList;
};
```

Create node function is below:

```
Node* CreateNode(string word,int sentenceNum)
{
   Node* node = new Node();
   node->word = word;
   node->left = nullptr;
   node->right = nullptr;
   node->sentenceList.insert(node->sentenceList.end(), sentenceNum);
   return(node);
}
```

# DATA STRUCTURE

**The data structure** is basically an two dimensional array which stores 26x26 Binary Tree root nodes, the number 26 comes from the number of the letters in English alphabet. The words will be stored in terms of their first two letters because the only one letter words in English are "I" and "a" and they are stop words , the structure doesn't store these words as well as two letter words which are either stop words or super rare words (will be explained later in insertion part). The diving methodology makes the searching and inserting algorithms faster compared to only one Binary Tree approach. The data structure is defined as "**Struct**" , we didn't choose to use "Class" because a struct would be enough for our data structure , we didn't need to use inheritence or private variables or functions. The structure consists of one field which is array and one insert , one search which uses two other inserting and searching function.

26x26 Array of the structre defined as below:

```
Node* wordArr[26][26];
```

# INSERTION

We have two functions regarding the insertion , first one struct function other one is public function . Insertion will occur after dividing the words regarding their first two letters. In first design of the algorithm the words that has two letters were stored and worked on Mac without and problem but after testing the program on Windows reaching the index two of two letterd word has given an error so we have changed the code so the words with two letter don't be inserted. In the case of one words as explained before there are only two words which are one letter and both of them are stop words. Also the first index on string will be checked in first inserting function whether it is letter or not , if it is not letter it wont be added because it wouldn't be a valid word.

Regarding code for the first insert function in WordStruct:

```
void insert(string word,int sentenceNum){
    if(word.length() == 1 && word.length() == 2 ){
        return;
    }
    else if (word[0]<123 && 96<word[0] == false){
        return;
    }
    else if (word[1]<123 && 96<word[1] == false){
        return;
    }
    wordArr[word[0]-97][word[1]-97] = insertWord(wordArr[word[0]-97][word[1]-97],
word,sentenceNum);
    }
```

In **second inserting function** which is called by first one. Then words will be added to their trees with regards to their third characters ASCII value. If their value is bigger they will be added to right otherwise left. Insertion operations are done recursively within the child nodes. If the words are found to be same the “sentence number” of the word will be inserted to the “sentence list” of the node. If the third characters are same and words are not same too the word will be added to the right of the node if right of the node is filled the insertion operation recursively continues with the same logic.

```
Node* insertWord(Node* node, string word, int sentenceNum)
{
   if (node == NULL){
      return(CreateNode(word,sentenceNum));
   }
   else if (word[2] < node->word[2])
      node->left = insertWord(node->left, word, sentenceNum);
   else if (word[2] > node->word[2])
      node->right = insertWord(node->right, word , sentenceNum);
   else if ( word[2] == node->word[2] && word != node->word)
      node->right = insertWord(node->right,word , sentenceNum);
   else if ( word == node->word){
      node->sentenceList.insert(node->sentenceList.end(), sentenceNum);
   }

   return node;
}
```

# SEARCHING

Searching logic is same with insertion but this tame if we find the word we return the sentenceList property of the node. sentenceList will be used for finding the correct sentence.

First searching function:

```
   list<int> search(string word){
      if(word.length() == 1 && word.length() == 2){
         list<int> a;
         return a;
      }
      else if (word.length() > 1 && word[0]<123 && 96<word[1] == false){
         list<int> a;
         return a;
      }

      return searchWord(wordArr[word[0]-97][word[1]-97], word);
   }
```

Second searching function:

```
list<int> searchWord(Node* node,string word){
   if(node == NULL){
      list<int> list;
      return list;
   }
   else if(word[2] < node->word[2])
      return searchWord(node->left, word);
   else if(word[2] > node->word[2])
      return searchWord(node->right, word);
   else if ( word[2] == node->word[2] && word != node->word)
      return searchWord(node->right, word);
   else if (word == node->word)
      return node->sentenceList;
   else{
      list<int> list;
      return list;
   } }
```

# GET QUESTION FUNCTION

After the data structure is created and filled with the words. The “questions.txt” will be read line and getQuestion function will be called , our data structure and question string will be passed . Because the function consists of more then 90 line , lets go through it part by part. The function iterates through every char of the function and stops at the question mark.

By finding searching every word , searchs and finds the word in structure and stores the sentence list as sentence nodes and every time it finds the same sentence it increments the point property of **sentenceNode** so that we find the **correct sentence** by looking at sentenceNode’s point property. In for loop we also store the stemmed version of words in question in list named questionWords.

Firt part of the function:

```
void getQuestion(string question,WordStruct *w){

   string qWord = "";
   list<sentenceNode> sentenceList;
   list<string> questionWords;
   for ( char ch : question){
      if (ch == ' ' || ch == '?'){
         // make lower case
         for(int i = 0;i< int(qWord.length());i++){
            if(qWord[i] > 64 && qWord[i] < 91){
               qWord[i] = qWord[i] + 32;
            }
         }
```

```
        // if it is not stop word get the list of sentences that it was used
        if (!check(qWord)){
            qWord = stem(qWord);
            list<int> liste = w->search(qWord);
            questionWords.insert(questionWords.end(), qWord);
            list<int>::iterator it;
            for (it = liste.begin(); it != liste.end(); it++) {
                list<sentenceNode>::iterator i;
                bool exists = false;
                //checck whether exists
                for(i = sentenceList.begin(); i != sentenceList.end(); i++){
                    if(i->num == *it){
                        exists = true;
                        i->point++;
                    }
                }
                if(!exists){
                    sentenceList.insert(sentenceList.end(), createSentence(*it));
                }
            }
        }
        qWord = "";
    }
    else{
        qWord = qWord + ch;        }
```

**Getting the related sentence number:**

```
    list<sentenceNode>::iterator it;
    int biggestPoint = 0;
    int relatedSentence = 0;
    for(it = sentenceList.begin(); it != sentenceList.end(); it++){
        if (it->point > biggestPoint) {
            biggestPoint = it->point;
            relatedSentence = it->num;
        }
    }
```

**Getting the related sentence by searching:**

The find sentence method returns the list of sentence words. Important point is that findSentence method returns the sentence words with stemmed versions but also it stores the original words in a variable called originalSentence

Code:

```
    list<string> sentenceWords = findSentence(relatedSentence);
```

**Finding the answer as stemmed version:**
It iterates through every sentenceWord and compares whether it is not same , the answer should not be in the question so it finds the word which is not mentioned in question. Remember it is still in stemmed version.

```
string answerStemmed;
for(z = sentenceWords.begin(); z != sentenceWords.end(); z++){
   if (*z != *find(questionWords.begin(), questionWords.end(), *z)) {
      answerStemmed = *z;
   }
}
```

**Finding the answer as stemmed version:**
After finding the answerStemmed. We iterate through every word in originalText , which is a global variable and filled in findSentence function. In every word we check how many letters does it match with our answerStemmed and makes the word with biggest matchCount our correct answer. Then we print the word which is the answer.

Related code:

```
list<string>::iterator m;
   string answer;
   int biggestMatch = -1;
   int stemmedSize = int(answerStemmed.length());
   for(m = originalSentenceList.begin(); m != originalSentenceList.end(); m++){
      string nonStemmed= *m;
      int matchCount = 0;
      int iterationCount;
      if(stemmedSize < int(nonStemmed.length()) ){
         iterationCount = stemmedSize;
      }
      else{
         iterationCount = int(nonStemmed.length());
      }

      for(int i=0;i<iterationCount;i++){
         if(answerStemmed[i] == nonStemmed[i]){
            matchCount++;
         }
      }
      if (matchCount > biggestMatch){
         answer = nonStemmed;
         biggestMatch = matchCount;
      }
   }
   cout << answer << endl << endl;
```

# SENTENCE NODE

A node stores sentence's number and its points in order to be used in getQuestion method and there is a createSentence function as well.

Related code:

```
struct sentenceNode {
   int num;
   int point;
   };
```

Creating the sentence node:

```
sentenceNode createSentence(int num){
   sentenceNode* a;
   a  = new sentenceNode();
   a->num = num;
   a->point = 1;
   return *a;
}
```

# FIND SENTENCE FUNCTION

This function returns the list of words in a sentence in stemmed form also it updates the originalSentenceList and adds the words to it. It works with the same iteration logic in order to find the correct sentence number , therefore all stop word comparisons are made again.

```
list<string> originalSentenceList;
list<string> findSentence(int num){
   originalSentenceList.clear();
   list<string> sentenceWordsList;
   ifstream file;
   string fileName = "the_truman_show_script.txt";
   file.open(fileName);

   string word;
   int sentenceNum = 0;
   bool newSentence = false;

   if (file.is_open() == true){
      while(file >> word){
         int size = word.length();
         // make all letters lowercase
         for(int i = 0;i<word.length();i++){
            if(word[i] > 64 && word[i] < 91){
               word[i] = word[i] + 32;
```

```
            }
        }
        if (size >1 && (word[size-1] == '.'  || word[size-1] == '!' || word[size-1] == '?')  ){
            string modifiedWord(word.begin(),word.end()-1);
            word = modifiedWord;
            newSentence = true;
        }
        else if(size >1 && word[size-1] == 34 && (word[size-2] == '.' || word[size-2] == '!' ||
word[size-2] == '?') ){
            string modifiedWord(word.begin(),word.end()-2);
            word = modifiedWord;
            newSentence = true;
        }

        string originalWord = word;
        word = stem(word);

        if (check(word)) {
            if(newSentence){
                sentenceNum++;
                newSentence = false;
            }
            continue;
        }

        if (sentenceNum == num) {
            originalSentenceList.push_back(originalWord);
            sentenceWordsList.push_back(word);
        }
        else if (sentenceNum == num+1){
            return sentenceWordsList;
        }

        if (newSentence){
            sentenceNum++;
            newSentence = false;
        }
    }

    file.close();
  }
  else{
    cout << "file not open" << endl;
  }

  return sentenceWordsList; }
```

## Average Speed

Average speed of the algoritihm is 27 milliseconds , this number is computed regarding the average times in different computers.

## Correctness of Algorithm

Algorithm works with the questions that requires one word but if the question requires an answer more than one word the algorithm prints the word which is the most of the sentence. Second defect of the algorithm is about words on related sentence , if there is a word in sentence which is not answer and this word doesn't show in question or in stop words , algorithm can't make the difference between this word and the anwer word , again in this case algorithm prints the word which is the most of the sentence. Other than these flaws and in example questions algorithm prints out the correct answers.

## Libraries

From the standard library of C++ "fstream" and "list" were used.
For evaluating the time from the standard library "chrono" was used.
For stemming Oleander Stemming algorithm is used.
Reference:
https://github.com/OleanderSoftware/OleanderStemmingLibrary