Data Structures and Algorithms III

SCS2201 | Assignment 01

University of Colombo School of Computing

A.M.S.U. Karunarathne

2020/CS/092

20000928

Table of Contents

Github repository: https://github.com/senithkay/DSA3-assignment1-stringMatcher

1. String/pattern matching

1.1 Preprocessing the file.

This is achieved by dividing the string into lines and search each line one after the other. First this program will open the file and process it from top to bottom. This process will read each line one after the other and store that line in an array data structure (Arrays are not resizable due to the contiguous memory allocation. In c++ there is a collection type called vector which is basically a dynamically resizable array. Implementation of a vector may not be exactly similar to an array but the functionality is the same.)

```
textFile:: textFile(string file_to_open = ""){
            fstream file;
            line_count=0;
            file.open(file_to_open, ios::in | ios::out);
            if (!file.is_open())
                    cout<<"Could not open the file"<<endl;
            else {
                //slicing the file line by line and stores it in a vector
                while (getline(file,line)){
                lines.push_back(line);
                line_count++;
                }
            }
            file.close();

}
```

1.2 Searching for a string

After the preprocessing part this program has a copy of all the string lines that was contained in the file in its own array(vector) called *lines.* When the user wants to find a string, he can enter the desired string into the program and it will call the searchLines() method of the textFile class. This method drives the searching process by passing each line to the searchMatchingLine() method of the textMatcher class.

```
textMatcher textFile:: searchLines(string pattern){
            int line_number;
            textMatcher matcher1(pattern, lines);
            for(line_number=0;line_number<line_count;line_number++){
                if(matcher1.searchMatchingLine(lines[line_number])){
                    matcher1.setMatchedLines(line_number+1);
                }
```

```
            }
            return matcher1;
}
```

This text matcher class constructor will get executed before the searchMatchingLine() method because in order to call the searchMatching(), an object of the class textMatcher should be created. And when creating that object it will call the constructor of the textMatcher class and it will run the create_bad_character() method which is the preprocessing function of the Boyer-Moore-Horspool algorithm.

```
textMatcher:: textMatcher(string p, vector<string> &lines):textLines(lines) {
            number_of_matches = 0;
            pattern = p;
            p_length = p.length();
            create_bad_character();
}
```

```
void textMatcher:: create_bad_character(){
        int i;
        for (i = 0; i < 256; i++)
            bad_c_table[i] = p_length;

        for (i =0; i<p_length - 1; i++){
            if(isupper(pattern[i]))
                bad_c_table[int(pattern[i])+32] = p_length - i - 1;
            else
                bad_c_table[int(pattern[i])] = p_length - i - 1;
        }

}
```

After all this, searchMatchingLine() function will get executed for each element in the array(vector) which are lines of the text file. This seachMatchingLine() method then set the ‘text’ and ‘text length’ of the boyer moore horspool algorithm, and call that algorithm(pattern is set in the beginning of the creation of the object of the class textMatcher so there is no need to again set these values over and over again for each text line. This will reduces the time complexity of the whole process of searching the string).

```
bool textMatcher:: searchMatchingLine(string t){
            text = t;
            t_length = t.length();
            return boyer_moore();
}
```

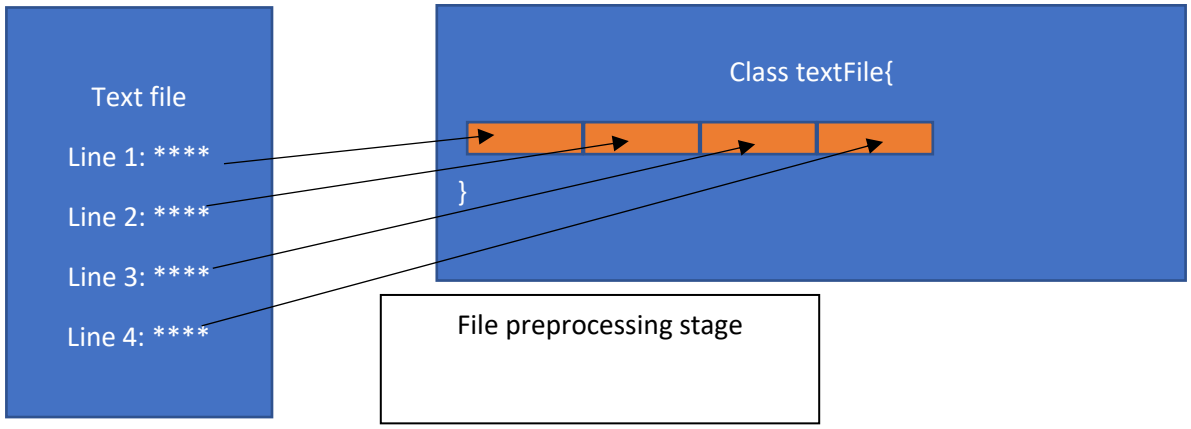
```
bool textMatcher:: boyer_moore(){
            int shift_value = 0;
            int j;
            while (shift_value <= (t_length - p_length))
            {
                for (j = p_length - 1; j >= 0; j--){
                    if(isupper(pattern[j]) && !isupper(text[shift_value + j])){
                        if (int(pattern[j]) +32 != int(text[shift_value + j]))
                            break;
                    }

                    else if(!isupper(pattern[j]) && isupper(text[shift_value +
j])){
                        if (int(pattern[j]) != int(text[shift_value + j])+32)
                            break;
                    }
                    else{
                        if (pattern[j] != text[shift_value + j])
                        break;
                    }
                }
                if (j < 0){
                    number_of_matches++;
                    return true;
                    if (shift_value + p_length < t_length){
                        if(isupper(text[j + p_length + 1]))
                            shift_value += bad_c_table[int(text[j + p_length +
1])+32];
                        else
                            shift_value +=
bad_c_table[int(text[shift_value+p_length])];

                    }
                    else
                        shift_value++;
                }
                else{
                    if(isupper(text[shift_value + (p_length - 1)]))
                        shift_value += bad_c_table[int(text[shift_value +
(p_length - 1)])+32];
                    else
                        shift_value += bad_c_table[int(text[shift_value +
(p_length - 1)])];
                }
```

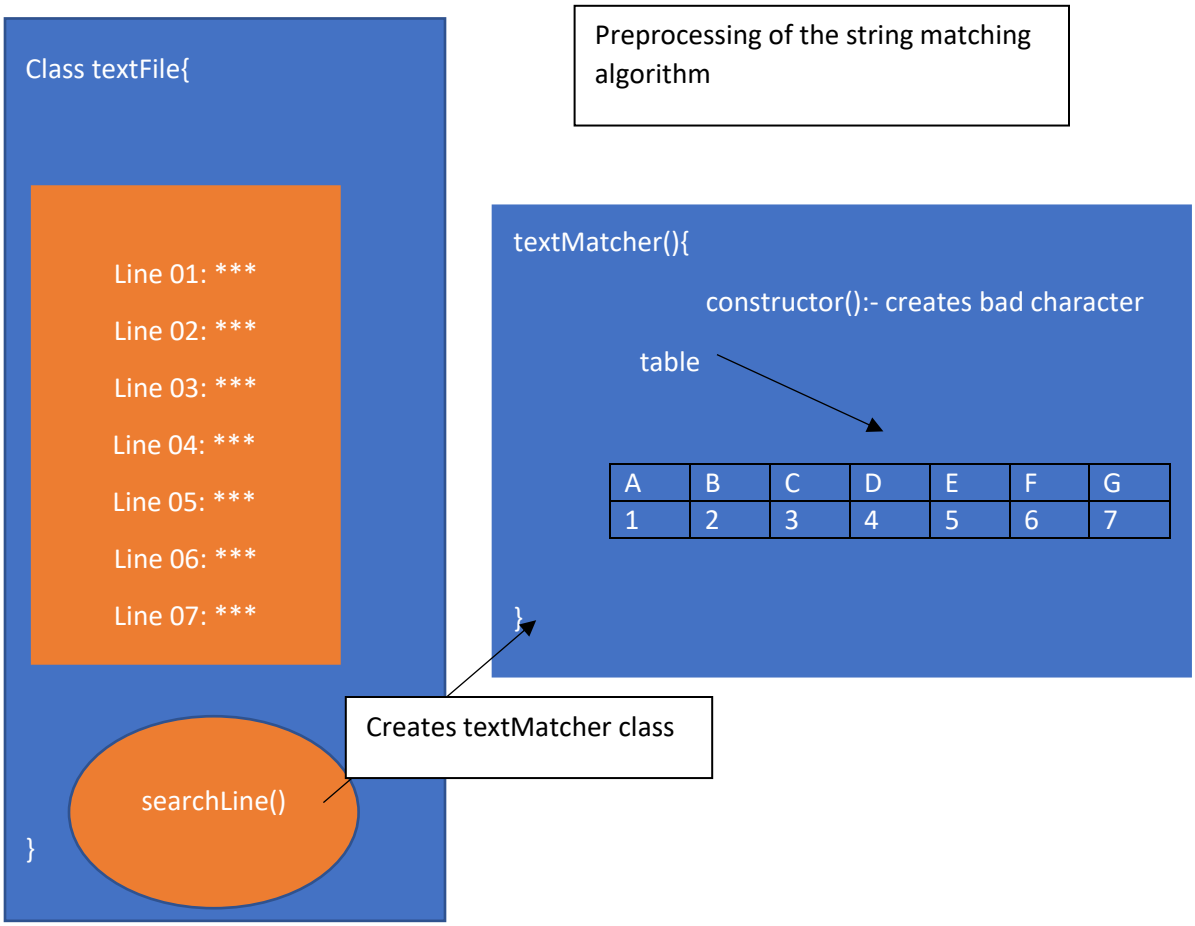
```
            }
            return false;
}
```

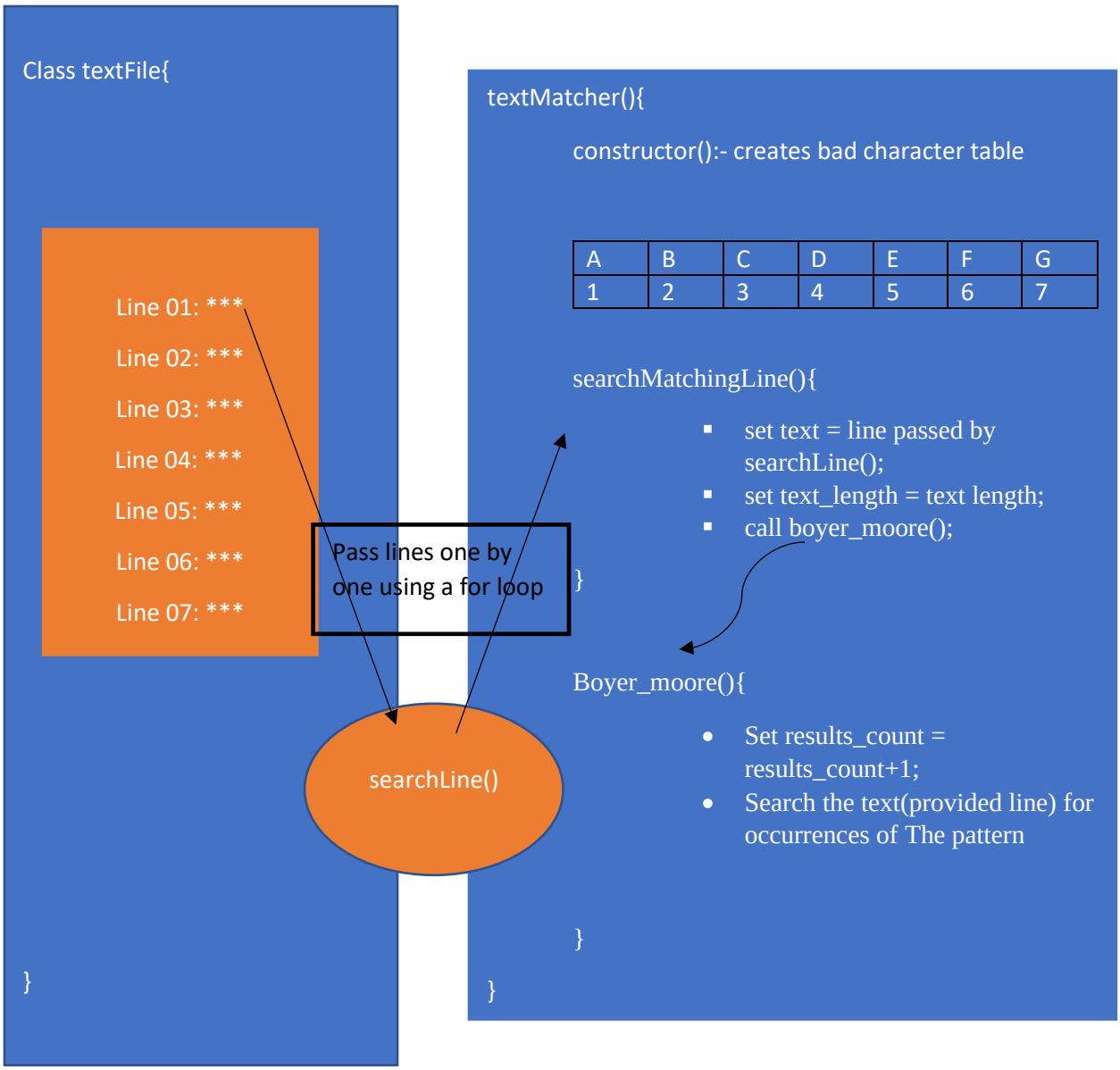
After searching the text, my implementation of the Boyer-Moore-Horspool algorithm will returns a boolean value true or false(if found returns true else returns false) and give the control back to the searchLine() method. If a line contains the pattern this method then save this line number in an array which is declared inside the class textMatcher.

This textMatcher class has a member called *number_of_matches* which is responsible of keeping track of results count. This value is initially '0' and gets incremented each time when a result is found.

Following diagram shows the complete process of text matching.




Class textFile{
Line 01: ***
Line 02: ***
Line 03: ***
Line 04: ***
Line 05: ***
Line 06: ***
Line 07: ***
searchLine()
}
Preprocessing of the string matching algorithm
textMatcher(){
constructor():- creates bad character
table
A B C D E F G
1 2 3 4 5 6 7
}
Creates textMatcher class



Example execution of Boyer-Moore-Horspool algorithm for text “SPA4283 Level D HE Further Advanced Spanish Translation Exchange Students Semester 2” and pattern “s semester”

NOTE: There are 2 white spaces between word “Translation” and word “Exchange” in the text.

1.3 Filtering and Displaying results

For filtering the result. This program will run textMatcher:: split() function to split each line of results into words and insert one by one into a 2-dimentional array called results(). Then user wants to filer and print only module codes it will print the first element of each sub array. If the user wants to print only the module name it will print all the elements of the sub array starting from the 2nd element of the sub array.

```
vector< vector<string> > results;
                vector<string>:: iterator i=textLines.begin();
                vector<int>:: iterator j;
                for(j= matched_lines.begin();j<matched_lines.end();j++){
                    results.push_back(split(*(i+*j-1)));
                }

                vector< vector<string> > :: iterator k;
                for(k=results.begin();k<results.end();k++)
                    if (filter == 2)
                        printStringVector(*k,0,0);
                    else if(filter==3)
                        printStringVector(*k,1);
```

Below diagram will clarify this process of making the 2 dimentional array.

Eg: APL1001 Alternative Practice Histories.

APL2005 Twentieth Century Architecture.

APL3001 Alternative Practice Coproducing Space.

APL1001	Alternativ	Practice	Histories	
APL2005	Twentieth	Century	Architectu	
APL3001	Alternativ	Practice	Coproduci	Space

1.4 Why Boyer-Moore-horspool algorithm?

I have divided the text(string to be searched for a particular pattern) of this problem into lines. Therefore the text is not very large and it implies that the pattern is large when compared to the Text. As we know, the time complexity of this algorithm is O(nm) for worst case and O(n/m) for best case where n is the length of the text and m is the length of the pattern. Therefore when m gets close to the value of n, best case time complexity goes to O(1) (**Not equal**). Hence the Boyer-Moore algorithm or the Boyer-Moore-Horspool algorithms are the best candidates. Among them, the Boyer-Moore algorithm has very complex preprocessing steps for 'good suffix shift table' and most of the time we may not get the benefit of 'good

suffix shift' rule. Therefore the ideal string matching algorithm for this scenario would be the Boyer-Moore-Horspool algorithm.

Another reason to choose Boyer-Moore-Horsepool over KMP and Rabin Karp algorithms is the most of the lines in the 'module.txt' file does not contains the searching pattern because this file contains total of 2377 lines. Boyer-Moore-Horsepool algorithm performs better in these kinds of scenarios.

Ex:

String: APL2023 The Place of Houses

Pattern: China

In this situation Boyer-Moore-Horspool algorithm works better than KMP algorithm

2. Print a line by providing the line number

Since textFile class contains an array(a vector) of string that contains all the lines, we can directly access each line using array index.

Index = line number-1

there is a function in the textFile class called printLine() to print a line.

```
void textFile:: print_line(int line_number){
            cout<< lines[line_number-1] <<endl;
}
```

3. Slice a file from a given line

We can access lines using array index same as mentioned above. To slice a file, fileSlice() function will create a new file and copy all the lines from the *lines* array, starting from the user provided line number.

4. Split a line into words

We can take any line and split them into words using the textFile :: split() function. Then we can easily print that line as we did in the printLine() function.