

6/24/2021

# SCS 2201

## Data Structures and Algorithms III

String Matching Algorithms

✚ Assignment 01(Code Explanation)

**V.D.L.W Vithanage**  
**2019/CS/180**  
**19001802**

## Selected String Matching Algorithm – Naïve string Matching Algorithm

### Brief explanation of Naïve String Matching Algorithm

```

void find(char* Pattern, char* Text)
{
    int x = strlen(Pattern);
    int y = strlen(Text);

    for ( int i = 0; i <= y - x; i++) {
        int j;

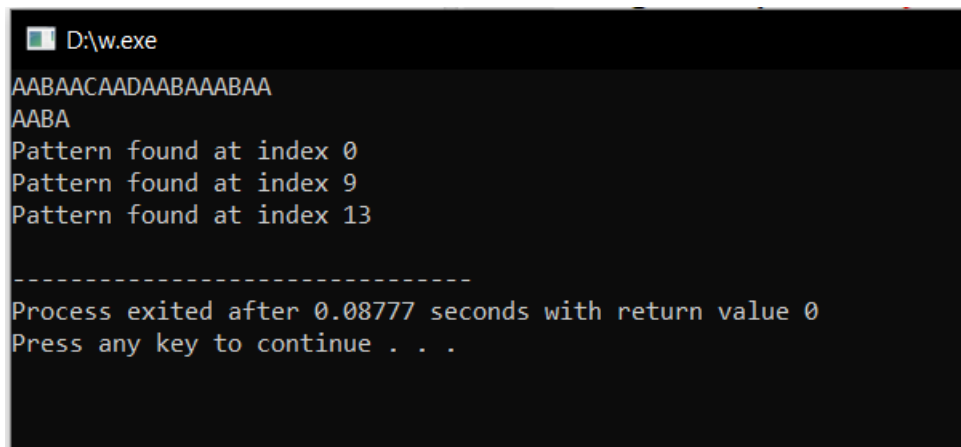
        for (j = 0; j < x; j++)
            if (Text[i + j] != Pattern[j] )
                break;

        if (j == x)
            cout << "Pattern found at index "<< i << endl;
    }
}

```

This takes two inputs called pattern and the text. X is the length of the pattern and the y is the length of the text. The outer loop runs Y-X times and inner loop runs X times. The complexity of the code becomes  $O(XY)$ . Our pattern size is x so the latest position the pattern could find is at Y-X position. That's why we run our outer loop Y-X times.

This checks whether the first character of the text matches with the first character of the pattern. If not, it breaks from there and goes to the next character of the text and again compares with the first character of the pattern and so on. If the first character of both text and the pattern are matched, then it moves forward to the next character of both text and pattern and compares. If it is also matched, it goes ahead. If a mismatch is found, it breaks from there and moves to the next character of the text and the beginning of the pattern.



```

D:\w.exe
AABAACAADAABAAABAA
AABA
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

-----
Process exited after 0.08777 seconds with return value 0
Press any key to continue . . .

```

A	A	B	A	A	C	A	A	D	A	A	B	A	A	A	B	A	A
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

A	A	B	A
0	1	2	3

If we consider this example,

A	A	B	A
0	1	2	3

It first match 0th index of both. It match. Then go to the 1<sup>st</sup> index of both. It also match. Then check 2nd index. It matches. Then check last index. It also match. Then the code ask whether the length of the pattern and the Jth value are equal or not. It equal mean there is a pattern match. If not equal mean there is no pattern. In here with the loop Jth value become 4 and already pattern length is 4. It mean a pattern. Then it print on the console.

```
if (j == x)
    cout << "Pattern found at index "<< i << endl;
```

A	B	A	A
1	2	3	4
A	A	B	A

Then it moved to the next index of the text and to the beginning of the pattern. Both first character are matching but the second characters are not match. A!=B. So the break is done and move to the next character of the text and to the beginning of the string.

Likewise the inner loop and the outer loop will be executed and the relevant pattern matching indexes will identify and write on the console.

B	A	A	C
2	3	4	5
A	A	B	A

## How I modify Naïve String Matching Algorithm to identify wildcard pattern matches

Wildcard = It is a character which Means to any character. In this assignment we use underscore “\_” sign as a wildcard.

What we have to do is identify wildcard character matching's in our text and show their indexes.

Though the wildcard matches to any character, we can ignore wildcard when we found it in our pattern with the character of the text. This is the only improvement I have done for my naïve algorithm.

```
for (j = 0; j < x; j++)
    if (Text[i + j] != Pattern[j] && Pattern[j] != '_')
        break;
```

Earlier what we did was when the character of the text and the pattern mismatch, then we move to the break option. Here we ensure the character of the pattern is not a wildcard. If it is a wildcard

we ignore it and move forward. To do break option, the character of the text and the character of the pattern should be mismatch and the character of the pattern should not be a wildcard character.

This is the only change I have done for naïve algorithm to convert it to find wildcard characters.

Let's look some examples.

C	O	G	W	R	G	A	C	C	A	G
0	1	2	3	4	5	6	7	8	9	10

C	_	G
---	---	---

C == C. Then we check 'O' with "\_". These two do not match. But we do not move to break option. Because now our code ignore "\_" wildcard and move forward. Then we check G == G. After the value of J and the length of the pattern will be equal and it show that there is a pattern match at index 0.

```

D:\2nd Year\1st semester\scs 2201 - DSA III\Assignments\wild_card_string_matching.exe
****Text1****
Text :- cogwrgaccag
Pattern :- c_g

Pattern found at index 0
Pattern found at index 8

```

### text.txt files, Pattern.txt files, patternmatch.output files

We have to provide some test data in order to make examiner easy to mark our code. So I have created three text files namely

- text.txt – where our text is stored.
- Pattern.txt – where our wildcard pattern is stored.
- Patternmatch.output – where the output is stored (index positions of wildcard pattern matches).

First we stored the locations of these files inside our code.

```

#include <iostream>
#include<string.h>
#include<fstream>

using namespace std;

//stote the locations of the text.txt,pattern.txt and patternmatch.output files
char *textarray[10] = {"text1.txt", "text2.txt", "text3.txt", "text4.txt", "text5.txt", "text6.txt", "text7.txt", "text8.txt", "text9.txt", "text10.txt"};
char *patternarray[10] = {"pattern1.txt", "pattern2.txt", "pattern3.txt", "pattern4.txt", "pattern5.txt", "pattern6.txt", "pattern7.txt", "pattern8.txt", "pattern9.txt", "pattern10.txt"};
char *patternmatcharray[10] = {"patternmatch1.output", "patternmatch2.output", "patternmatch3.output", "patternmatch4.output", "patternmatch5.output", "patternmatch6.output", "patternmatch7.output", "patternmatch8.output", "patternmatch9.output", "patternmatch10.output"};

```

Then we open text.txt file and pattern.txt file and get the data stored on them. This is done with the use of file handling options in C++ . Read operation has been used.

```
int main()
{
    char Text[100];
    char Pattern[20];

    int i=0,j=0;

    for(i;i<10;i++){

        char Text[100]={ " "};
        char Pattern[20]={ " "};

        cout<<"****Text"<<i+1<<"****"<<endl;           //open text.txt file and get the data
        ifstream text;
        text.open(textarray[i]);
        text>>Text;                                       //read from the text file
        cout<<"Text :- "<<Text<<endl;
        text.close();

        ifstream pattern;                               //open pattern.txt file and get the data
        pattern.open(patternarray[i]);
        pattern>>Pattern;                                //read from the pattern file
        cout<<"Pattern :- "<<Pattern<<endl<<endl;
        pattern.close();
    }
}
```

Then we call our modified naïve string matching algorithm.

```
find(i,Pattern, Text);           //call the modified naive string matching algorithm
cout<<endl;

}

return 0;
}
```

Now our modified naïve algorithm is called.

In there we open our output file (patternmatch.output). We find the starting indexes of the wildcard matching and stored these locations on the output file. Write operation has been used.

```
ofstream patternmatch;           //open output file to store data
patternmatch.open(patternmatcharray[z]);
```

Modified Naïve string matching algorithm:

```
//modified naive sting matching algorithm
void find(int z,char* Pattern, char* Text)
{
    int x = strlen(Pattern);
    int y = strlen(Text);

    ofstream patternmatch; //open output file to store data
    patternmatch.open(patternmatcharray[z]);

    for ( int i = 0; i <= y - x; i++) {
        int j;

        for (j = 0; j < x; j++)
            if (Text[i + j] != Pattern[j] && Pattern[j] != '_') //*****the only change done to the
                                                                    //naive string matching algorithm
                break;

        if (j == x) {
            cout << "Pattern found at index "<< i << endl;
            patternmatch<<"Pattern found at index "<< i << endl; //write into the output file
        }
    }
    patternmatch.close();
}
```

The change I have done for naïve algorithm is explained earlier. To justify what I done was, I make the inner for loop of the code to ignore wildcards. Earlier before modify, it check the characters of the text and the pattern. If the characters do not match it goes to the break option. Then again begin to compare the next character of the text with the beginning of the pattern.

Here after the modification, inner loop ignore the wildcard character of the pattern and move to the next character of both text and the pattern and do the comparing. This was successful change to find wildcard matches using naïve string matching algorithm.

### **Why I do not select any other string matching algorithms for this**

There are many string matching algorithms such as KMP, Rabin Krap, and Boyer-Moore likewise. Their internal implementation is much complex than naïve string matching algorithm. Actually naïve algorithm is very simple and easy to understand. Here in our problem we have to modify our algorithm to identify the wildcards. There is only one change I have to done in naïve algorithm and it was make the inner for loop of the code to ignore wildcard and move to the next character.

To do this thing with other algorithms is not much easy than naïve algorithm. There are many internal logical changes have to done to modify the code. Because of that reason I have chosen naïve string matching algorithm for this wildcard matching problem.