Analysis of string searching algorithms

Naive algorithm, KMP algorithm & Boyer Moore algorithm

Yuhang Zhou (yz853)
Rutgers University
yuhang.zhou@rutgers.edu
4/5/2019

Introduction

String searching algorithm is a kind of searching algorithm which aims to search one or more strings in a long string or article. There are lots of searching algorithms, and the different algorithm has its own pros and cons. In this paper, I will talk about 3 string searching algorithms which are the naïve algorithm, the KMP algorithm, and the Boyer-Moore algorithm. The naïve algorithm is the most basic algorithm. The KMP algorithm is one of the most famous algorithms in this world. And the Boyer-Moore algorithm is highly efficient and widely used in the searching function of editor programs.
In this paper, I will introduce these algorithms, and explain it with the pseudocode. Then analysis this algorithm about its performance with my test result. At last, put them together, compare with each other and make a conclusion.

Naive Algorithm

Introduction

The naive algorithm is the most basic one in the string searching algorithms. In this algorithm, the string will be compared with all the article. Every character of the string will be checked in the article. It's similar to the sequence searching algorithm. Firstly, the first character of the string will be sequenced searched in the article, if the first character is found, other characters will also be compared in sequence until all the characters are matched.

Pseudocode

```
string find is being searched
string data is the dataset
for i in data.length
  for j in find.length
    if find[j] != find[i+j]
      break
  if j==find.length
    string is found!
```

Analysis

It's obvious that characters of the string and the article are compared with each other. Let's say the length of the string is m, the length of the article is n, so the run time is O(mn). But it depends on the characters in the string and the article.
In the best case, the first character of the string never exists in the article, so every time the first character is found not match, the string will move forward, which means other characters of the string is not needed to be compared. In this case, the run time will be order n. On the other hand, the worst case is that every character needs to be compared with each other so that the run time will be order m*n.
But I don't think the m*n run time will happen except in the situation that all the characters in the string and article are all the same. I think in the most case, the run time will be closer to order n because once the character doesn't match, this loop will stop and the string will be moved forward. If the loop won't stop even mismatch happened until the last character is compared, in this case, the run time will always be order m*n, but this will be a really naive algorithm.

KMP Algorithm

Introduction

The full name of the KMP Algorithm is Knuth–Morris–Pratt algorithm, because this algorithm is published by these three people.
The main idea of this algorithm is that, in the string searching process, if part of the string is already matched, then the next character mismatch. In this case, if we are using the naive algorithm, the string just moves 1 character space forward then compare from the first character again. But they think some characters of the article is already compared in the previous "partly matched case", so the information of these characters should be already known. So how to take advantage of these "partly matched characters" is the key to the KMP algorithm.
For a simple example:
The string is "ABCDABD"
The article contains "ABCDABCDABD"
i is the indicator

A	B	C	D	A	B	C	D	A	B	D
A	B	C	D	A	B	D				
						i				

In the first comparison, the indicator has come to the last character, all the characters are matched except the last character. So in the naive algorithm, the string will move 1 character space forward and the indicator will compare from the first character again.

A	B	C	D	A	B	C	D	A	B	D
	A	B	C	D	A	B	D			
	i									

It looks like low efficient because the “BCDAB” are already compared in the previous loop, and they are matched. So how to take advantage of that?
Let’s look at the string itself, inside the string, it looks like it’s partly repeated, or the “AB” is repeated

A	B	C	D	A	B	D

So we can take advantage of this information. If we already know that there are also two “AB”s in the article and they are all matched, that means we can move the first “AB” to the next “AB” directly, which will save us many steps

A	B	C	D	A	B	C	D	A	B	D
				A	B	C	D	A	B	D
				i						

And it looks more efficient, the next step is how to get the inside information of the string like the 2 “AB”s in this one, and how to use that. So that’s an important part of the KMP algorithm call LPS array.
The information of the string will be saved inside an array called LPS, AKA the longest proper prefix of the string which is also a suffix of the string.
Firstly, let’s figure out what’s the prefix and what’s the suffix.
For example, the word “apple”
Prefix: a, ap, app, appl
Suffix: pple, ple, le, e
In conclusion, the prefix the substring which contains the first character, while the suffix is the substring which contains the last character. So the LPS the longest string these two shares.
Back to the previous example “ABCDABD”

A
0

The “A” has no prefix and suffix, so the LPS is 0.

A	B
0	0

The prefix of AB is A, the suffix is B, no common string, so the LPS is 0

A	B	C
0	0	0

Prefix: A, AB
Suffix BC, C
LPS=0

A	B	C	D
0	0	0	0

The same

A	B	C	D	A
0	0	0	0	1

Prefix: A, AB, ABC, ABCD
Suffix: BCDA, CDA, DA, A
They both have “A”, the length is 1, so the LPS is 1

A	B	C	D	A	B
0	0	0	0	1	2

Prefix: A, AB, ABC, ABCD, ABCDA
Suffix: BCDAB, CDAB, DAB, AB, B
They both have “AB”, LPS is 2

A	B	C	D	A	B	D
0	0	0	0	1	2	0

Prefix:A, AB, ABC, ABCD, ABCDA, ABCDAB
Suffix: BCDABD, CDABD, DABD, ABD, BD, D
LPS is 0
So that the LPS of “ABCDABD” is 0000120.
How to use that? For example, if “ABCDAB” is already matched. The indicator comes to the last “D”, and this “D” doesn’t match. That means 6 characters are matched, the LPS of the already matched part is 2. So the string will move forward for 6-2=4 character spaces, which is also the distance between these two “B”s. So that’s how the KMP algorithm works.

Pseudocode

```
m=find.length
n=data.length
LPS(find, lps[])
{
lps[0]=0
for (int i = 1; i < m; i++)
   {
      int len = 0;
      lps[i] = len;
      for (len = 1; len <= i; len++)
      {
         if (find.substr(0, len) == find.substr(i - len + 1, len))
         {
            lps[i] = len;
}}}}
KMP(find, data, lps)
{
i=j=0
while (i < n)
   {
      if (data[i] == find[j])
      {
         i++
         j++}
      if (j == m)
      {
         string is found!
         j = lps[j - 1]
       }
      else if (i < n && data[i] != find[j])
      {
         if (j == 0)
         {
            i++;
         }
         else
         {
            j = lps[j - 1];
}}}}
```

Analysis

The run time of KMP is O(m+n). The n is the run time to create the LPS array, the m is the run time to go over the article.
But it depends on different situation, the run time of the KMP algorithm is complex. Let's say if the string is a word that no character repeats which means the LPS array is all 0s. So that it's almost the same as the naive algorithm. Or in another case, this string is partly repeated, the LPS array is not all zero, but in this case, there is a very strict requirement for the article. This means there should be lots of similar strings in the article, these strings are highly similar to the searched string, so that the KMP algorithm can come to the ideal situation as I said before. As this ideal situation should exist a lot because recall the LPS array also waste time.
But it's hard to deny that the KMP algorithm is a great algorithm. It's really smart to take advantage of the information of the string itself, do some pretreatments before start searching. But it also makes this algorithm much complex.

Boyer-Moore Algorithm

Introduction

The Boyer-Moore algorithm, as its name, it's designed by Bob Boyer and J Strother Moore.
It's different from the KMP algorithm, the main idea of the KMP algorithm is to take advantage of the string itself. But we can find that, in the ideal situation, the KMP situation can skip some characters so that it can be faster.
So the main idea of the Boyer-Moore algorithm is trying to skip as more characters as possible, but the matched ones will be ignored.
Unlike the naive algorithm and the KMP algorithm, the Boyer-Moore algorithm starts from the end of the string. For example

A	B	C	D	E
A	B	D		
		i		

i is the pointer
If the last character doesn't match, also we can see that the "C" doesn't exist in the searched string so that the whole string can skip the "C", so the next step can skip 3 character space

A	B	C	D	E	
			A	B	D
			i		

In this case, the speed can be very fast, and we can make sure no matched one is missed.

But in other cases, it will be a little more complex. The mismatched character may exist in another place of the string. For example

A	B	A	B	D
A	B	D		
		i		

The “A” mismatched, but A exists in the first character of the string, so the string can move forward for 2 character space. So it will be

A	B	A	B	D
		A	B	D
		i		

In this case, we still can skip 1 comparison.

So that means we need to do some pretreatment for the string. We will record the position of every character in the string. It’s actually an array called “Badchar” array. The badchar array is an int[256] array. It follows the ASCII standard. The ASCII can transfer every character to an integer which is the index in this Badcharr array. Any character doesn’t exist in the string will be -1 in this array. Then every character's last position will be the number in this array.

For example, in the string “ABD”, the A exits in the first position, and A is 65 in the ASCII chart. So that means Badchar[65]=1.

At last, the distance we will skip when the mismatch happens is 3-1=2 in this case. The 3 is the length of the string, the 1 is the number is the array.

Pseudocode

```
n = data.length
m = find.length
badchar[256]
for (int i = 0; i < 256; i++)
       badchar[i] = -1;
for (int i = 0; i < m; i++)
        badchar[int(find[i])] = i;
```

```
//the int(char) can transfer the character to the ASCII number
void boyer_moore(string data, string find)
{
    int count = 0;
    int i = 0;
    int j;
    while (i <= n - m)
    {
        j = m - 1;
        while (j >= 0 && data[i + j] == find[j])
        {
            j--;
        }
        if (j < 0)
        {
            String is found!
            if (i < n - m)
            {
                i = i + m - badchar[int(data[i + m])];
            }
            else
            {
                i++;
            }
        }
        else
        {
            i = i + max(1, j - badchar[int(data[i + j])]);
        } } }
```

Analysis

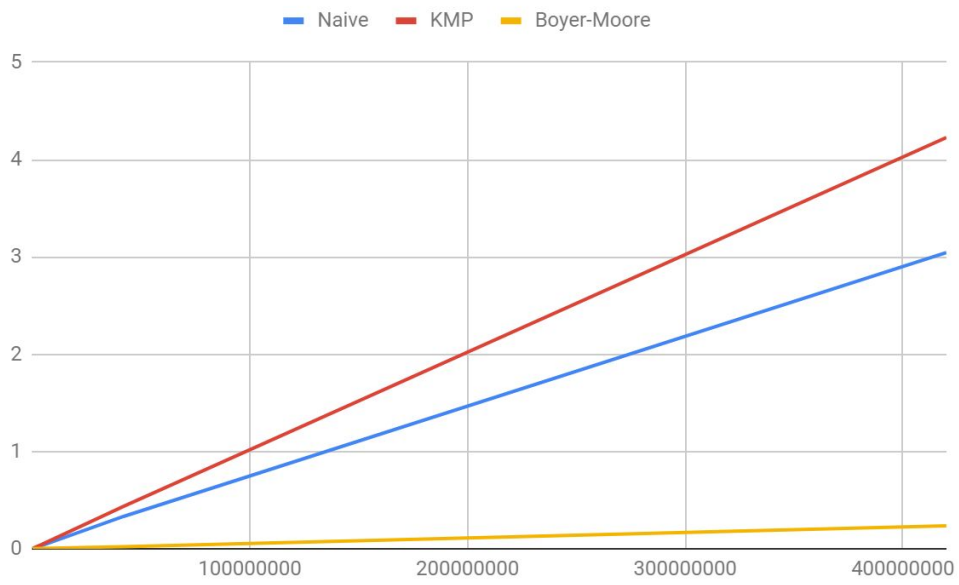
In the best case of this algorithm, the run time will be order n/m, which means in every comparasion loop, the last character never be contains in the string, so that every time it can skip m character spaces.
The worst case is order m*n, which means all the article and the string has the same character.
The average run time of this algorithm should be great because the main idea of this algorithm is trying to skip as more steps as possible so that it can save more run time.

Experiment

In the experiment, I will use 5 different size txt files as datasets, the content of the txt file is some algorithm articles from Wikipedia. And the searched string is “algorithm”. Sizes of 5 text files are 42k, 420k, 4.2m, 42m, 420m.

Results

Run time	42000	420000	4200000	42000000	420000000
Naive	0	0.003	0.031	0.332	3.044
KMP	0.002	0.004	0.042	0.436	4.228
Boyer-Moore	0.001	0.001	0.003	0.022	0.235



The run time is proportional to the size of the dataset because we didn’t change the searched string.
The Boyer-Moore algorithm is much faster than the Naive algorithm and the KMP algorithm. The KMP is the slowest.

Conclusion

The result is similar to my analysis. The KMP is the worst because it wastes lots of time in the pretreatment of the string, and every time when it does a comparison, the program should recall the LPS array, but that won't help a lot in most cases.
While the Boyer-Moore algorithm is designed to skip as more comparisons as possible so that the run time can be very short compared with another two. But it's still a little longer than the naive when the article is very short. I think the reason is the pretreatment and recalling the Badchar array.
I cannot deny that the KMP algorithm is a smart and advanced algorithm. But its performance is not good in most cases. But I think it's still useful in some specific cases, for example, the words in the article are highly similar to each other, and the string we are searching is partly repeated so that it can always come to the ideal case of this algorithm.
This experiment inspires me a lot about algorithms. I think there is no better or worse algorithm, every algorithm has its own ideal case that this algorithm's performance can be very well in this case. Even the simplest naive algorithm, it's still faster than the Boyer-Moore algorithm in the small size datasets.

Reference

String-searching algorithm From Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/String-searching_algorithm

Boyer–Moore string-search algorithm From Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm

Knuth–Morris–Pratt algorithm From Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

KMP Algorithm for Pattern Searching GeeksforGeeks A computer science portal for geeks
https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

Boyer Moore Algorithm for Pattern Searching GeeksforGeeks A computer science portal for geeks
https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/