Matrix67: The Aha Moments

Detailed explanation of KMP algorithm

If the computer room is closing soon, or you're in a hurry to date MM, skip to paragraph 6.

The KMP we are talking about here is not used to play movies (although I like this software very much), but an algorithm. The KMP algorithm is used to handle string matching. In other words, given two strings, you need to answer, whether string B is a substring of string A (whether string A contains string B). For example, the string A="I'm matrix67" and the string B="matrix", we say that B is a substring of A. You can politely ask your MM: "If you want to confess to the person you like, is my name a substring in your confession?"

To solve this kind of problem, usually our method is to enumerate from A Where in the string starts to match B, and then verify that it matches. If the length of the A string is n and the length of the B string is m, then the complexity of this method is O (mn). Although many times the complexity is not up to mn (just look at the first one or two letters and find the mismatch), we have many "worst cases", for example, A = "aaaaaaaaaaaaaaaaaaaaaaaaab", B = "aaaaaaaab". What we will introduce is a worst-case O(n) algorithm (assuming m<=n here), the legendary KMP algorithm.

The reason why it is called KMP is because this algorithm was proposed by Knuth, Morris, and Pratt, and took the first letter of the names of these three people. At this time, maybe you suddenly understand why the AVL tree is called AVL, or why Bellman-Ford is a bar instead of a dot in the middle. Sometimes seven or eight people have studied a thing, so how do you name it? Usually this thing is simply not named after people, so as to avoid disputes, such as "3x+1 problem". Pull away.

Personally, I think KMP is the most unnecessary thing to talk about, because a lot of information can be found on the Internet. But the online lectures basically involve concepts such as "shift" and "Next function", which are very easy to misunderstand (at least a year and a half ago, I didn't understand when I read these materials to learn KMP). Here, I explain the KMP algorithm in a different way.

Suppose, A="abababaababacb", B="ababacb", let's see how KMP works. We use two pointers i and j to represent respectively, A[i-j+ 1..i] is exactly equal to B[1..j]. That is to say, i is constantly increasing, and j changes accordingly as i increases, and j satisfies that the string of length j ending with A[i] just matches the first j characters of the B string (of course, the larger the j, the more OK), now we need to check the relationship between A[i+1] and B[j+1]. When A[i+1]=B[j+1], i and j are each added by one; when j=m, we say that B is a substring of A (the B string has been completed), and can be based on The i value at this time calculates the matching position. When A[i+1]<>B[j+1], KMP's strategy is to adjust the position of j (reduce the value of j) so that A[i-j+1..i] and B[1..j] Keep matching and the new B[j+1] exactly matches A[i+1] (so that i and j can continue to increase). Let's take a look at the situation when i=j=5.

```
i = 1 2 3 4 5 6 7 8 9 …
A = abab a baabab …
B = abab a cb
j = 1 2 3 4 5 6 7
```

At this time, A[6]<>B[6]. This shows that j cannot be equal to 5 at this time, and we have to change j to a smaller value j'. How much might j' be? Thinking about it carefully, we find that j' must make the first j' letters and the last j' letters in B[1..j] exactly equal (so that after j becomes j', the i and j can continue to be maintained. nature). Of course, the larger j', the better. Here, B[1..5]="ababa", the first 3 letters and the last 3 letters are "aba". And when the new j is 3, A[6] is exactly equal to B[4]. Thus, i becomes 6, and j becomes 4:

```
i = 1 2 3 4 5 6 7 8 9 ...
A = ababa b aabab ...
B = aba b acb
j = 1 2 3 4 5 6 7
```

From the above example, we can see that how much the new j can take has nothing to do with i, but only with the B string. We can completely preprocess such an array P[j], indicating the maximum new j when the jth letter of the B array is matched and the j+1th letter cannot be matched. P[j] should be the maximum of all satisfying B[1..P[j]]=B[jP[j]+1..j].

Later, A[7]=B[5], and i and j are increased by 1. At this time, the situation of A[i+1]<>B[j+1] appeared again:

```
i = 1 2 3 4 5 6 7 8 9 …
A = ababab a abab …
B = abab a cb
j = 1 2 3 4 5 6 7
```

Since P[5]=3, the new j=3:

i = 1 2 3 4 5 6 7 8 9 …

A = ababab a abab …

B = ab a bacb

j = 1 2 3 4 5 6 7

At this time, the new j=3 still cannot satisfy A[i+1]=B[j+1]. At this time, we reduce the value of j again and update j to P[3] again:

i = 1 2 3 4 5 6 7 8 9 …

A = ababab a abab …

B =   a babacb

j =   1 2 3 4 5 6 7

Now, i is still 7, and j has become 1. At this time, A[8] is still not equal to B[j+1]. Thus, j must be reduced to P[1], which is 0:

i = 1 2 3 4 5 6 7 8 9 …

A = ababab a abab …

B = ababacb

j =   0 1 2 3 4 5 6 7

Finally, A[8]=B[1], i becomes 8, j becomes 1. In fact, it is possible that when j reaches 0, A[i+1]=B[j+1] still cannot be satisfied (for example, when A[8]="d"). So, it is accurate to say that when j=0, we increase the value of i but ignore j until A[i]=B[1] occurs.

The code for this process is short (really short) and we give it here:

```
j:=0;
for i:=1 to n do
begin
   while (j>0) and (B[j+1]<>A[i]) do j:=P[j];
   if B[j+1]=A[i] then j:=j+1;
   if j=m then
   begin
      writeln('Pattern occurs with shift ',i-m);
      j:=P[j];
   end;
end;
```

The j:=P[j] at the end is to keep the program going, since we may find multiple matches.

This program may be simpler than expected, because the code uses a for loop for the increasing value of i

. Therefore, this code can be visually understood as follows: scan string A and update where B can be matched.

Now, we are left with two important questions: one, why this program is linear; two, how to quickly preprocess the P array.

Why is this program O(n)? In fact, the main controversy is that the while loop makes the number of executions uncertain. We will use the main strategy in the amortized analysis of time complexity, which is simply to accumulate scattered, messy, and irregular execution times by observing changes in the value of a variable or function. The time complexity analysis of KMP can be described as a typical example of amortized analysis. We start with the value of j for the above program. Each execution of the while loop will decrease j (but not negative), and the only other place to change the value of j is the fifth line. Each time this line is executed, j can only be incremented by 1; therefore, j is incremented by at most n 1s throughout the process. Therefore, j has at most n opportunities to decrease (of course, the number of times the value of j decreases cannot exceed n, because j is always a non-negative integer). This tells us that the while loop executes at most n times in total. According to the amortization analysis, the complexity of a for loop is O(1) after being amortized into each for loop. The whole process is obviously O(n). Such an analysis is also effective for the subsequent P array preprocessing process, and it can also be obtained that the complexity of the preprocessing process is O(m).

Preprocessing does not need to be written as O(m^2) or even O(m^3) according to the definition of P. We can get the value of P[j] by the values of P[1], P[2],...,P[j-1]. For the previous B="ababacb", if we have found P[1], P[2], P[3] and P[4], let's see how we should find P[5] and P[6]. P[4]=2, then P[5] is obviously equal to P[4]+1, because it can be known from P[4] that B[1,2] is already equal to B[3,4], and now there is B[3]=B[5], so P[5] can be obtained by adding a character after P[4]. Is P[6] also equal to P[5]+1? Obviously not, because B[P[5]+1]<>B[6]. Then, we have to consider "stepping back". We consider whether P[6] can be obtained by the substring contained in the case of P[5], that is, whether P[6]=P[P[5]]+1. If you can't figure it out here, take a closer look:

```
    1 2 3 4 5 6 7
B = a b a b a c b
P = 0 0 1 2 3 ?
```

P[5]=3 because B[1..3] and B[3..5] are both "aba"; and P[3]=1 tells us that B[1], B[3] and B[5] are all "a". Since P[6] cannot be obtained from P[5], it may be obtained from P[3] (if B[2] happens to be equal to B[6], P[6] is equal to P[3]+1) . Obviously, P[6] cannot be obtained by P[3] either, because B[2]<>B[6]. In fact, it is not enough to push all the way to P[1], and finally, we get, P[6]=0.

Why is this preprocessing process so similar to the previous KMP main program? In fact, the preprocessing of KMP itself is a process of "self-matching" of B strings. Its code is similar to the above code:

```
P[1]:=0;
j:=0;
for i:=2 to m do
begin
   while (j>0) and (B[j+1]<>B[i]) do j:=P[j];
   if B[j+1]=B[i] then j:=j+1;
   P[i]:=j;
end;
```

A final point to add: Since the KMP algorithm only preprocesses B strings, this algorithm is very suitable for the problem: given a B string and a group of different A strings, ask which A strings are substrings of B.

String matching is a very valuable research problem. In fact, we also have suffix trees, automata, and many other methods, which cleverly use preprocessing to solve string matching in linear time. We'll talk about it later.

I found a very dizzy thing yesterday, do you know how to remove BitComet ads? Just set the interface language to English.
Also, Kingsoft and Dr.eye can commit suicide, Babylon is king.

Matrix67 original repost
please indicate the source

November 29, 2006 / KMP algorithm , code , complexity , algorithm , proof

282 comments

hot and sour dog

MM said: My confession is "Actually... I don't like matrix67, I like you, Hotdog"

congratulations. . You are the substring in her confession.

Re: Why isn't your English name sour-and-hot dog?

March 29, 2007 21:45 / Reply

dawson sour-spicy hot dog