**KMP算法小结**

Posted on [2011/06/14](http://www.chaoswork.com/blog/2011/06/14/kmp%e7%ae%97%e6%b3%95%e5%b0%8f%e7%bb%93/) by [huangchao](http://www.chaoswork.com/blog/author/huangchao/" \o "View all posts by huangchao)

主要看了[这里](http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm)，感觉讲的十分的不错，总结一下。

首先声明要搜索的串为S,设长度为n，要匹配的串为M,设长度为m.

先考虑暴力的算法，暴力的算法是遍历S的每一个字符，然后从这个字符开始和M串进行匹配。时间复杂度为O(nm).

怎么在此基础上进行优化？假设现在从某个位置(设为s)开始和M串进行匹配，如果匹配不成功，暴力算法是从这个位置的下一个位置(s+1)进行匹配,直观上来说就是匹配的字符串向后“滑动”了一位。

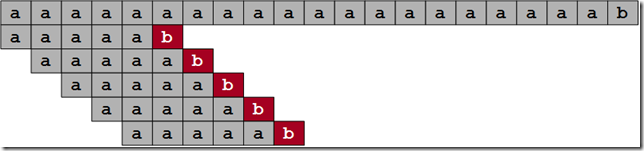
[](http://www.chaoswork.com/blog/wp-content/uploads/2011/06/image.png)

图1

能不能想办法让M向后移动的距离最大化？考虑最好的情况，如果和M匹配的S中的m个字符和M中的字符没有一个相等，那么能向右移动m位；考虑最坏的情况，比如上图，只能移动一位。

而KMP就是在这里做文章，让M串向后“滑动”的距离最大化。

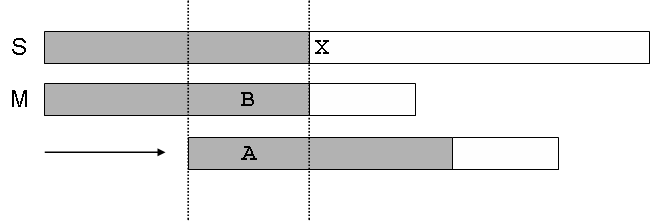
[](http://www.chaoswork.com/blog/wp-content/uploads/2011/06/image22.png)

图2

考虑上面的图，M中灰色部分已经和S的灰色部分匹配上了，而灰色部分后一个字符不匹配，则现在M要向后滑动，假设一直向后滑动，直到如图位置又和S再一次匹配上了，那么从这里我们可以得到如下的结论：

* A段字符串是M的一个前缀。
* B段字符串是M的一个后缀。
* A段字符串和B段字符串相等。

这样，如果暂时不考虑S，只看M的话，假设已经匹配的M的字串(即图中M中灰色部分)为subM，则subM有个【相等】的【前缀】和【后缀】。而且M在遇到不匹配的时候可以直接滑动到使subM的前缀和subM的后缀重合的地方。而M向后滑动的时候，第一次subM的前缀和后缀重合意味着此时这个相等的subM的前缀和后缀的长度是最大的。

我们的任务就是要寻找subM的最长的前缀和后缀相等的串。

知道了这一点，离KMP的真谛也就不远了。现在结合这上面的图模拟一下KMP算法的整个流程：

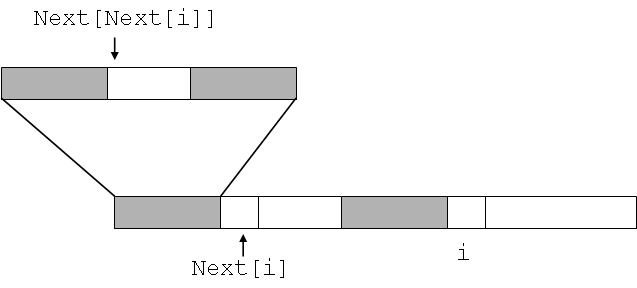
* 将S串和M串从第一个字符开始匹配；
* 如果匹配成功，则subM即灰色部分增加；
* 如果不成功，则M向后滑动使滑动后的subM的前缀和滑动前的subM的后缀重合，再进行匹配，如果还不成功，则再次滑动M，直到匹配成功或者M滑动到X处。如果到了X处，则从M串的起始位置进行匹配。

从上面的步骤可以知道，KMP的关键就是要知道当S串中的字符和M串中的字符不匹配时，S串要和M串中的哪个字符继续进行匹配。这个就是在利用状态机模型来解释KMP算法时的状态转移.

KMP是通过一个定义了一个next数组，这个next数组保存了如果S中的字符和M中的字符不匹配时S要和M中的哪个字符重新进行匹配的坐标值。如图2中所示是例子，S中的X位置和M不匹配了，那么S要和M中A段后面的字符进行比较，从图中来看是M向后滑动了。

换句话说，next[i]总是保存了当M[i]不匹配时要从M[next[i]]处进行匹配，这个M[next[i]]可能会匹配，如果还不匹配？那么可能会在M[next[next[i]]]处匹配了。这里同时隐含着一个信息，就是i之前的一段字符和next[i]之前的一段字符是相同的，也就是M[0…i-1]相等的前缀和后缀。

现在考虑next[0],next[1]…next[i]都已经知道了，那么图示如下：

[](http://www.chaoswork.com/blog/wp-content/uploads/2011/06/image1.png)

设j=next[i]，灰色部分表明这两段字符是相等的，如果i位置的字符和j位置的字符相等，那么next[i+1]=j+1;因为前一段灰色部分和j位置的字符组成的字符串和后一段灰色的与i连接所形成的字符串是相等的。这正是前面对next数组的定义。如果不相等，则要找到从i开始包括i往前的一段字符串与从0开始的一段字符串相等，这样形成相等的前缀和后缀。所幸我们知道next[next[i]]的值，因为next[i]前面的字串也有最长的公共前缀和后缀，而这个公共的前缀与现在i以及往前形成的字串可能相等，这样一直向前找，如果找不到，则说明i位置的字符从来没有在之前出现过。

这样求出来的next数组其实是从下标1开始的，因为下标0之前是个空串，下标1则对应着M串的第0个字符。我们设next[0]=-1，仅仅是个标志而已，没有什么特殊的含义。

那么根据前面所述，可以很容易的写出初始化next数组的代码

1: void kmpGetNext()

2: {

3: int i=0, j=-1;

4: b[i]=j;

5: while (i<m)

6: {

7: while (j>=0 && p[i]!=p[j]) j=b[j];

8: i++; j++;

9: b[i]=j;

10: }

11: }

知道了next数组的值，则和S串进行匹配则相对简单了，因为如果碰到不匹配的时候去查找next数组即可,直到找出和当前字符匹配的那个字符。如果找不到怎么办？找不到则会得到-1，也就是没有字符和他进行匹配，那么跳过这个字符，直接从下一个字符进行匹配即可。

代码如下：

1: void kmpSearch()

2: {

3: int i=0, j=0;

4: while (i<n)

5: {

6: while (j>=0 && t[i]!=p[j]) j=b[j];

7: i++; j++;

8: if (j==m)

9: {

10: report(i-j);

11: j=b[j];

12: }

13: }

14: }

看到上面的代码，两层循环，貌似这个代码并不是线性的，其实不然。外层循环了n次这个没有问题，关键是里面的while循环，这个循环的次数是多少并不好确定，然而考虑单单考虑j的值的变化，会发现第七行j增加1，而第6行j则减少，可能减少1，可能减少2，可能少的更多，但是j<0时循环就终止了，也就是说j有n次增加的机会，会有多少次减少的机会？或者问j最多减少多少次？j减少的次数最多的时候，就是每次减少1，这样最多的会减少n次，也就是说第六行的循环最多会执行n次。平摊到每个循环，则执行次数为O(1)，所以kmpSearch的时间复杂度仍然是线性的O(n)，同理，kmpGetNext的时间复杂度为O(m).详情请参考[matrix67](http://www.matrix67.com/blog/archives/115)大牛的文章,下面有犀利的评论：

*倒数第七段*

*“…每一次执行while循环都会使j减小（但不能减成负的），而另外的改变j值的地方只有第五行。每次执行了这一行，j都只能加1；因此，整个过程中j最多加了n个1。于是，j最多只有n次减小的机会（j值减小的次数当然不能超过n，因为j永远是非负整数）。这告诉我们，while循环总共最多执行了n次。… ”*

*这里不大明白，整个过程中j是在回退然后前进的，假设第一遍比较回退一次，第二遍比较回退两次，于是总共加起来j减小和变大的次数都要大于n，不是吗？*