**1、概述**

在用于查找子字符串的算法当中，BM（Boyer-Moore）算法是目前相当有效又容易理解的一种，一般情况下，比KMP算法快3-5倍。

BM算法在移动模式串的时候是从左到右，而进行比较的时候是从右到左的。

常规的匹配算法移动模式串的时候是从左到右，而进行比较的时候也是是从左到右的，基本框架是：

j = 0；

while（j <= strlen(主串)- strlen(模式串)）{

for (i = 0;i < strlen(模式串) && 模式串[i] == 主串[i + j]; ++i)

；

if (i == strlen(模式串))

Match；

else

++j；

}

而BM算法在移动模式串的时候是从左到右，而进行比较的时候是从右到左的，基本框架是：

j = 0；

while (j <= strlen(主串) - strlen(模式串)) {

for (i = strlen(模式串) - 1; i >= 0 && 模式串[i] ==主串[i + j]; --i)

if (i < 0)

match；

else

++j；

}

显然BM算法并不是上面那个样子，BM算法的精华就在于++j

**2、BM算法思想**

BM算法实际上包含两个并行的算法，坏字符算法和好后缀算法。这两种算法的目的就是让模式串每次向右移动尽可能大的距离（j+=x，x尽可能的大）。

几个定义：

例主串和模式串如下：

主串  :  mahtavaatalomaisema omalomailuun

模式串: maisemaomaloma

好后缀：模式串中的aloma为“好后缀”。

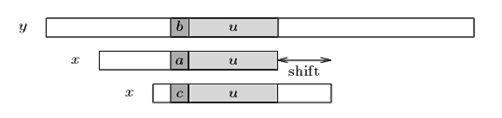
坏字符：主串中的“t”为坏字符。

**好后缀算法**

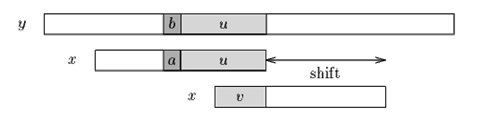
如果程序匹配了一个好后缀, 并且在模式中还有另外一个相同的后缀, 那

把下一个后缀移动到当前后缀位置。好后缀算法有两种情况：

Case1：模式串中有子串和好后缀安全匹配，则将最靠右的那个子串移动到好后缀的位置。继续进行匹配。

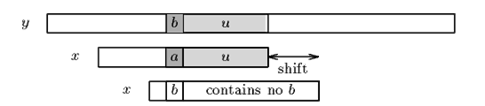
[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-979_2.png)

Case2：如果不存在和好后缀完全匹配的子串，则在好后缀中找到具有如下特征的最长子串,使得P[m-s…m]=P[0…s]。说不清楚的看图。

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-1152_2.png)

**坏字符算法**

当出现一个坏字符时, BM算法向右移动模式串, 让模式串中最靠右的对应字符与坏字符相对，然后继续匹配。坏字符算法也有两种情况。

Case1：模式串中有对应的坏字符时，见图。   
[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-1349_2.png)

Case2：模式串中不存在坏字符。见图。

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-1472_2.png)

**移动规则**

BM算法的移动规则是：

将概述中的++j，换成j+=MAX（shift（好后缀），shift（坏字符）），即

BM算法是每次向右移动模式串的距离是，按照好后缀算法和坏字符算法计算得到的最大值。

shift（好后缀）和shift（坏字符）通过模式串的预处理数组的简单计算得到。好后缀算法的预处理数组是bmGs[]，坏字符算法的预处理数组是BmBc[]。

**3、代码分析**

**定义**

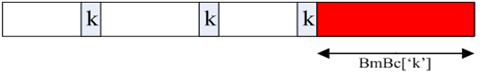
BM算法子串比较失配时，按坏字符算法计算模式串需要向右移动的距离，要借助BmBc数组。

注意BmBc数组的下标是字符，而不是数字。

BmBc数组的定义，分两种情况。

1、 字符在模式串中有出现。如下图，BmBc[‘k’]表示字符k在模式串中最后一次出现的位置，距离模式串串尾的长度。

2、 字符在模式串中没有出现：，如模式串中没有字符p，则BmBc[‘p’] = strlen(模式串)。

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-1885_2.png)

BM算法子串比较失配时，按好后缀算法计算模式串需要向右移动的距离，要借助BmGs数组。

BmGs数组的下标是数字，表示字符在模式串中位置。

BmGs数组的定义，分三种情况。

1、 对应好后缀算法case1：如下图：i是好后缀之前的那个位置。

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-2036_2.png)

2、 对应好后缀算法case2：如下图所示：

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-2086_2.png)

3、 当都不匹配时，BmGs[i] = strlen（模式串）

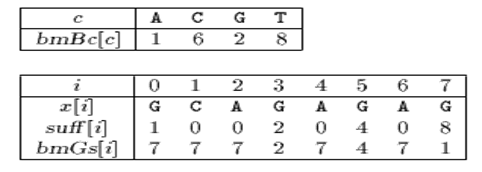
[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-2145_2.png)

在计算BmGc数组时，为提高效率，先计算辅助数组Suff。

Suff数组的定义：suff[i] = 以i为边界, 与模式串后缀匹配的最大长度，即P[i-s...i]=P[m-s…m]如下图：

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-2272_2.png)

举例如下：

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-2380_2.png)

**分析**

用Suff[]计算BmGs的方法。

1） BmGs[0…m-1] = m；（第三种情况）

2） 计算第二种情况下的BmGs[]值：

for（i=0；i

if（-1==i || Suff[i] == i+1）

for（；j < m-1-i；++j）

if（suff[j] == m）

BmGs[j] = m-1-i；

3） 计算第三种情况下BmGs[]值，可以覆盖前两种情况下的BmGs[]值：

for（i=0；i

BmGs[m-1-suff[i]] = m-1-i；

如下图所示：

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-2697_2.png)

Suff[]数组的计算方法。

常规的方法：如下，很裸很暴力。

Suff[m-1]=m；

for（i=m-2；i>=0；--i）{

q=i；

while（q>=0&&P[q]==P[m-1-i+q]）

--q；

Suff[i]=i-q；

}

有聪明人想出一种方法，对常规方法进行改进。基本的扫描都是从右向左。改进的地方就是利用了已经计算得到的suff[]值，计算现在正在计算的suff[]值。

如下图所示：

i是当前正准备计算的suff[]值得那个位置。

f是上一个成功进行匹配的起始位置（不是每个位置都能进行成功匹配的，  实际上能够进行成功匹配的位置并不多）。

q是上一次进行成功匹配的失配位置。

如果i在q和f之间，那么一定有P[i]=P[m-1-f+i]；并且如果suff[m-1-f+i]=i-q, suff[i]和suff[m-1-f+i]就没有直接关系了。

[](http://p.blog.csdn.net/images/p_blog_csdn_net/sealyao/594039/o_wps_clip_image-3177_2.png)

**代码**

[**view plain**](http://blog.csdn.net/sealyao/article/details/4568167)

1. **void preBmBc(char \*x, int m, int bmBc[]) {**
3. **int i;**
5. **for (i = 0; i < ASIZE; ++i)**
7. **bmBc[i] = m;**
9. **for (i = 0; i < m - 1; ++i)**
11. **bmBc[x[i]] = m - i - 1;**
13. **}**
15. **void suffixes(char \*x, int m, int \*suff) {**
17. **int f, g, i;**
19. **f = 0；**
21. **suff[m - 1] = m;**
23. **g = m - 1;**
25. **for (i = m - 2; i >= 0; --i) {**
27. **if (i > g && suff[i + m - 1 - f] < i - g)**
29. **suff[i] = suff[i + m - 1 - f];**
31. **else {**
33. **if (i < g)**
35. **g = i;**
37. **f = i;**
39. **while (g >= 0 && x[g] == x[g + m - 1 - f])**
41. **--g;**
43. **suff[i] = f - g;**
45. **}**
47. **}**
49. **}**
51. **void preBmGs(char \*x, int m, int bmGs[]) {**
53. **int i, j, suff[XSIZE];**
55. **suffixes(x, m, suff);**
57. **for (i = 0; i < m; ++i)**
59. **bmGs[i] = m;**
61. **j = 0;**
63. **for (i = m - 1; i >= 0; --i)**
65. **if (suff[i] == i + 1)**
67. **for (; j < m - 1 - i; ++j)**
69. **if (bmGs[j] == m)**
71. **bmGs[j] = m - 1 - i;**
73. **for (i = 0; i <= m - 2; ++i)**
75. **bmGs[m - 1 - suff[i]] = m - 1 - i;**
77. **}**
79. **void BM(char \*x, int m, char \*y, int n) {**
81. **int i, j, bmGs[XSIZE], bmBc[ASIZE];**
83. **/\* Preprocessing \*/**
85. **preBmGs(x, m, bmGs);**
87. **preBmBc(x, m, bmBc);**
89. **/\* Searching \*/**
91. **j = 0;**
93. **while (j <= n - m) {**
95. **for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);**
97. **if (i < 0) {**
99. **OUTPUT(j);**
101. **j += bmGs[0];**
103. **}**
105. **else**
107. **j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);**
109. **}**
111. **}**