**Boyer-Moore Algorithm（BM算法解析）**

**一、本文内容**

网上有很多论文和blog介绍BM算法，但是论文太理论，blog讲解复杂且不清楚，我花了一天多的时间看懂这个算法，并发现 wikipedia中BM算法条目中C代码实现的两个bug，修改、调试并测试过后才正常运行。我希望能用简单明了的方式来看这个算法。只有这样，外在的 知识经过加工才能内化为自己的东西。

**二、算法特点**

字符串匹配算法中有三个单模式算法，naive算法、KMP算法和BM算法。本文介绍BM算法，据说BM比KMP快3~5倍，还有文本处理软件中的查找(CTRL+F)和替换(CTRL+H)命令用的就是BM算法。

BM算法有三个特点：1、从右向左扫描；2、坏字符规则（Bad character shift rule，以下简称Bc）；3、好后缀规则（Good suffix shift rule，以下简称Gs）。

注：以一个sample说明什么是Bc和Gs

主串：     xyz b Gsxyz

模式串： bGsd Gs

从右至左扫描，模式串的后缀Gs与主串匹配，则称 **模式串** 的 Gs 为好后缀；

模式串与主串出现第一次不匹配，称 **主串** 的不匹配字符 b 为坏字符。

**三、算法描述**

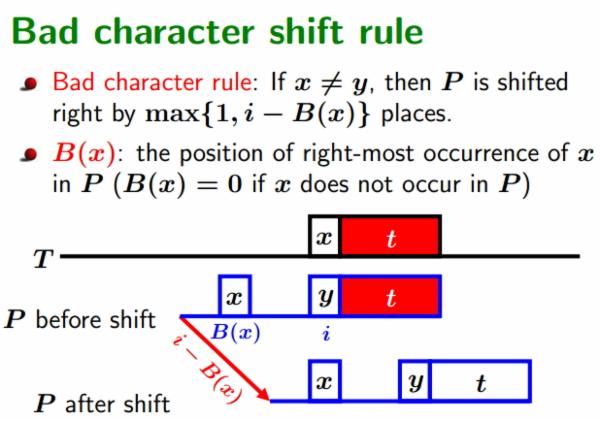
参数：模式串（以下称为P），长度为m；

         主串（以下称为T），长度为n；

算法在对P与T进行匹配之前，先对P进行预处理，计算出Bc算法对应的Bc表和Gs算法对应的Gs表（Bc表表示P、T第一次不匹配时， T 对应字符向右移动的shift值；Gs表表示 P 已匹配的suffix向右移动的shift值）。其中Bc算法和Gs算法同时进行，并取两者之中的大值。

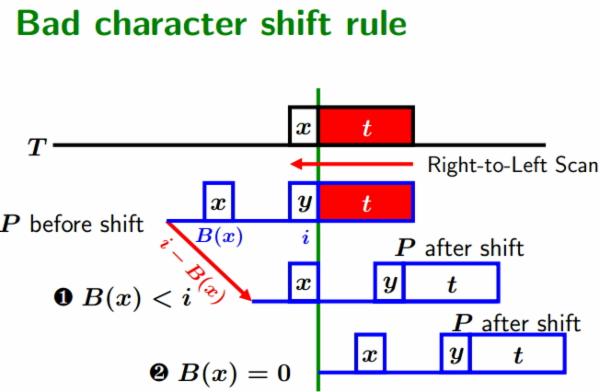
1、Bc规则

为了更好掌握算法特点，直接上图。Bc规则有三种情况，分别见图二、图三。



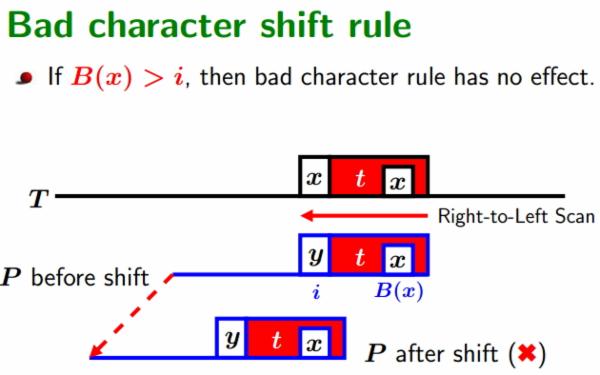
图一. Bc规则的定义

图一说明：B(x)表示P中=T中的Bc字符即x，且最靠右的位置



图二. Bc规则的两种情况

图二说明：case 1： B(x)<i 即正常情况； case 2：B(x)=0表示P中不存在等于Bc的字符即x，则将P整个移至与T中Bc的后一个字符对齐(蓝色横线)。图中有点小bug(x应该是不存在的)。



图三. Bc规则的异常情况

图三说明：case 3：与Bc相等的字符出现在P中已匹配的suffix中，意味着i-B(x)<0需要往左移，自然Bc规则在这种情况下将失效。

2、构造Bc表

参数说明：delta1数组即Bc表，索引为字符表（比如ascii码表）对应的字符，值为T失配字符的shift值；pat即P，patlen即m.

#define   ALPHABET\_LEN 256

#define   NOT\_FOUND patlen

void make\_delta1(int \*delta1, uint8\_t \*pat, int32\_t patlen) {

    int i;

    /\*初始化整个字符表的shift值为模式串P的长度(即case 2：出现坏字符时，P中无相同的字符)\*/

    for (i=0; i < ALPHABET\_LEN; i++) {

        delta1[i] = NOT\_FOUND;

    }

    /\*从左至右更新相同字符离失配位置(即patlen-1)的最近距离(case 1)\*/

    for (i=0; i < patlen-1; i++) {

        delta1[pat[i]] = patlen-1 - i;

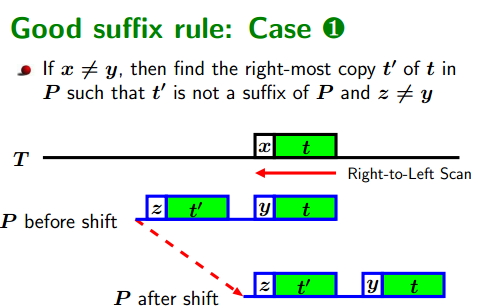
    }

}

注：case 3直接忽略就OK。

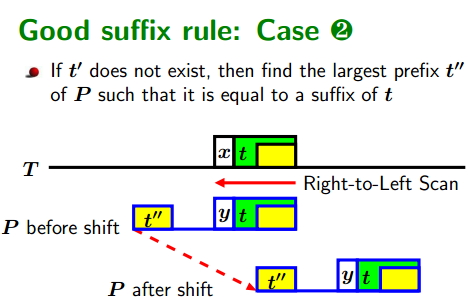
3、Gs规则

BM算法的难点和精髓也就是Gs表的构建上。Gs规则同样有三种情况。



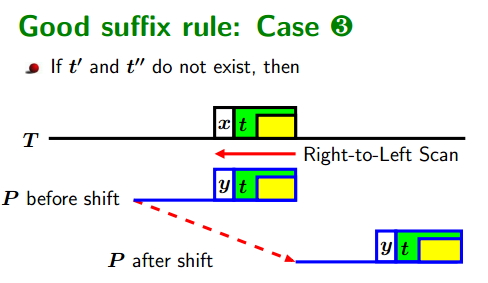
图四. Gs规则的case 1

图四说明：case 1：P中失配字符y之前存在已匹配后缀的副本即t'（限制条件：1、副本最靠近失配字符，2、前导字符不相等z≠y），



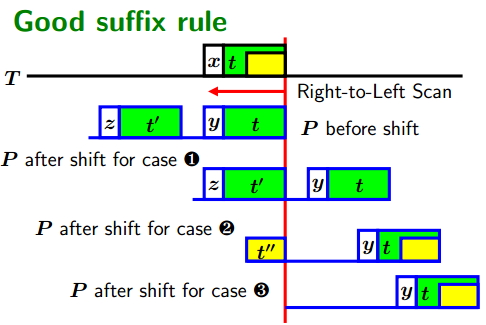
图五. Gs规则的case 2

图五说明：case 2：case 1条件不满足，若P存在前缀t''=已匹配后缀t的后缀，则找到最长的t''，失配时，直接移至suffix对 对齐。



图六. Gs规则的case 3

图六说明：case 3：case 1、2都不存在，则失配时，可以直接将P移至首部与T已匹配后缀的尾部对齐（即shift值为P自身长度）



图七. Gs的三种case总览

4、构造Gs表

/\*Gs规则case 2：suffix-prefix对，从已匹配后缀[pos, wordlen)判断word是否存在前缀，

   即word[0, suffixlen) == word[pos, wordlen)

\*/

int is\_prefix(uint8\_t \*word, int wordlen, int pos) {

    int i;

    int suffixlen = wordlen - pos;

    // could also use the strncmp() library function here

    for (i = 0; i < suffixlen; i++) {

        if (word[i] != word[pos+i]) {

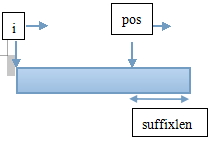
            return 0;

        }

    }

    return 1;

}

画了个图，大概就是这个意思   

/\*Gs规则case 1：suffix-suffix对，从pos向←查找与从P末尾（即已匹配后缀）向←查找相等的最长后缀，

   并返回最长后缀的长度

 \*/

int suffix\_length(uint8\_t \*word, int wordlen, int pos) {

    int i;

    // increment suffix length i to the first mismatch or beginning of the word

    //比较范围[1, pos]与[patlen-pos, patlen-1],  注意 :串word[0..pos]的后缀不包含自身

    for (i = 0; (word[pos-i] == word[wordlen-1-i]) && (i < pos); i++);

    return i;

}

参数说明：delta2即Gs表，

/\*假设P中p对应的字符失配，p之后全部匹配\*/

void make\_delta2(int \*delta2, uint8\_t \*pat, int32\_t patlen) {

    int p;

    int last\_prefix\_index = patlen-1;

    /\*first loop：Gs规则case 2\*/

    for (p=patlen-1; p>=0; p--) {

        if (is\_prefix(pat, patlen, p+1)) { //从p+1开始的后缀是否存在前缀(p失配)

            last\_prefix\_index = p+1;      //last\_prefix\_index记录从右至左最后一个匹配字符的index(即p的右边)

        }

 //若存在前缀，保存最后一个匹配字符的index；否则，保存上次已匹配字符的index

        delta2[p] = last\_prefix\_index;

    }

    /\*second loop：Gs规则case 1，因为case 2是前缀，而中间的子串(可以看做[0,p]的suffix)也可能=P的suffix，

       且有可能不止一个中间子串，故p从左向后进行处理，保存最靠近P的suffix的对应子串前一个字符的shift长度

     \*/

    for (p=0; p < patlen-1; p++) {

        int slen = suffix\_length(pat, patlen, p);         //末尾向左对应的从p向左的最长后缀的长度

        /\*若已匹配suffix-suffix对的前导字符不匹配，保存向左的第一个失配字符的shift长度(即suffix-suffix对的起始位置之差)

     若匹配，则为前缀即case 2，无需改变shift值

         \*/

        if (slen > 0 && pat[p - slen] != pat[patlen-1 - slen]) {   //slen=0, 即case 3：delta2[patlen-1-slen]=delta2[patlen-1]=patlen

            delta2[patlen-1 - slen] = patlen-1 - p ;

        }

    }

}

**四、算法实现**

**源代码来源于wikipedia的BM算法条目，但是发现两个bug，分别位于make\_delta2中的@bug1和@bug2，修改后运行正确。**

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#define ALPHABET\_LEN 256

uint32\_t patlen;

#define NOT\_FOUND patlen

#define max(a, b) ((a < b) ? b : a)

/\*构造Bc表\*/

void make\_delta1(int \*delta1, uint8\_t \*pat, int32\_t patlen) {

    int i;

    /\*初始化整个字符表的shift值为模式串P的长度(即case 2：出现坏字符时，P中无相同的字符)\*/

    for (i=0; i < ALPHABET\_LEN; i++) {

        delta1[i] = NOT\_FOUND;

    }

    /\*从左至右更新相同字符离失配位置(即patlen-1)的最近距离(case 1)\*/

    for (i=0; i < patlen-1; i++) {

        delta1[pat[i]] = patlen-1 - i;

    }

}

/\*Gs规则case 2：suffix-prefix对，从已匹配后缀[pos, wordlen)判断word是否存在前缀，

   即word[0, suffixlen) == word[pos, wordlen)

\*/

int is\_prefix(uint8\_t \*word, int wordlen, int pos) {

    int i;

    int suffixlen = wordlen - pos;

    // could also use the strncmp() library function here

    for (i = 0; i < suffixlen; i++) {

        if (word[i] != word[pos+i]) {

            return 0;

        }

    }

    return 1;

}

/\*Gs规则case 1：suffix-suffix对，从pos向←查找与从P末尾（即已匹配后缀）向←查找相等的最长后缀，

   并返回最长后缀的长度

 \*/

int suffix\_length(uint8\_t \*word, int wordlen, int pos) {

    int i;

    // increment suffix length i to the first mismatch or beginning of the word

    //比较范围[1, pos]与[patlen-pos, patlen-1], 注意:串word[0..pos]的后缀不包含自身

    for (i = 0; (word[pos-i] == word[wordlen-1-i]) && (i < pos); i++);

    return i;

}

/\*构造Gs表\*/

void make\_delta2(int \*delta2, uint8\_t \*pat, int32\_t patlen) {

    int p;

    int last\_prefix\_index = patlen-1;

    /\*first loop：Gs规则case 2\*/

    for (p=patlen-1; p>=0; p--) {

        if (is\_prefix(pat, patlen, p+1)) { //从p+1开始的后缀是否存在前缀(p失配)

            last\_prefix\_index = p+1;      //last\_prefix\_index记录从右至左最后一个匹配字符的index(即p的右边)

        }

        //若存在前缀，保存最后一个匹配字符的index；否则，保存上次已匹配字符的index

        delta2[p] = last\_prefix\_index;                                       //@bug 1: + (patlen-1 - p);

    }

    /\*second loop：Gs规则case 1，因为case 2是前缀，而中间的子串(可以看做[0,p]的suffix)也可能=P的suffix，

       且有可能不止一个中间子串，故p从左向后进行处理，保存最靠近P的suffix的对应子串前一个字符的shift长度

     \*/

    for (p=0; p < patlen-1; p++) {

        int slen = suffix\_length(pat, patlen, p);         //末尾向左对应的从p向左的最长后缀的长度

        /\*若已匹配suffix-suffix对的前导字符不匹配，保存向左的第一个失配字符的shift长度(即suffix-suffix对的起始位置之差)

           若匹配，则为前缀即case 2，无需改变shift值

         \*/

        if (slen > 0 && pat[p - slen] != pat[patlen-1 - slen]) {   //slen=0, 即case 3：delta2[patlen-1-slen]=delta2[patlen-1]=patlen

            delta2[patlen-1 - slen] = patlen-1 - p ;                    //@bug 2: + slen;

        }

    }

}

/\*打印预处理得到的Bc表和Gs表\*/

void print\_pre\_table(int \*delta1, int \*delta2, uint8\_t \*pat, uint32\_t patlen){

      uint32\_t i;

      printf("模式串：%s\n", pat);

      printf("坏字符shift表：\n");

      for (i=0; i < patlen-1; i++) {

          printf("(%c, %d)\n", pat[i], delta1[pat[i]]);

      }

      printf("(其他字符, %d)\n", NOT\_FOUND);

      printf("\n好后缀shift表：\n");

      for (i=0; i < patlen; i++) {

           printf("(%u, %d)\n", i, delta2[i]);

      }

}

/\*BM算法主框架\*/

uint8\_t boyer\_moore (uint8\_t \*string, uint32\_t stringlen, uint8\_t \*pat, uint32\_t patlen) {

    uint32\_t i;

    int delta1[ALPHABET\_LEN];

    int \*delta2 = (int \*)malloc(patlen \* sizeof(int));

    make\_delta1(delta1, pat, patlen);

    make\_delta2(delta2, pat, patlen);

    print\_pre\_table(delta1, delta2, pat, patlen);

    i = patlen-1;

    while (i < stringlen) {

        int j = patlen-1;

        while (j >= 0 && (string[i] == pat[j])) {

            --i;

            --j;

        }

        if (j < 0) {

            free(delta2);

      return i+1;       //返回T中匹配的位置

        }

        i += max(delta1[string[i]], delta2[j]);   //j失配（ [j+1, patlen)已匹配 ），i向右移动的距离取主串T中坏字符delta1[string[i]]与模式串P中好后缀delta2[j]的大者

    }

    free(delta2);

return -1;

}

int main()

{

uint8\_t pat[]="abracadabra";

uint8\_t txt[]="abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra";

patlen = sizeof(pat)/sizeof(pat[0]) - 1;

uint32\_t n = sizeof(txt)/sizeof(txt[0]) - 1;

uint8\_t ans=boyer\_moore(txt, n, pat, patlen);

printf("\n匹配位置:%d\n", ans);

return 0;

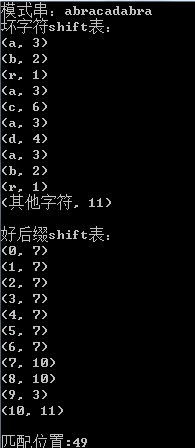
}

**五、算法测试**

T：abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra

P：abracadabra

运行结果



参考：

wiki BM算法条目 <http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm>

BM算法的applet演示(貌似省略了Gs规则) [http://people.cs.pitt.edu/~kirk/cs1501/animations/String.html](http://people.cs.pitt.edu/%7Ekirk/cs1501/animations/String.html)

算法解释 <http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html> （很好的sample解释）

<http://blog.csdn.net/sealyao/article/details/4568167>

<http://stackoverflow.com/questions/6207819/boyer-moore-algorithm-understanding-and-example>

 另外形式的两种实现：

<http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm>

[http://www-igm.univ-mlv.fr/~lecroq/string/node14.html](http://www-igm.univ-mlv.fr/%7Elecroq/string/node14.html)