**字符串匹配算法的优化与实现**

苗君文

（同济大学软件学院，上海 200000）

**摘要：** 字符串匹配技术作为数据分析的基础和核心，已经被广泛应用于各个领域。本文主要介绍了简要介绍了FM、KMP、BM等三种经典的算法及其改进优化，进行了各算法的性能分析，对FM算法和KMP算法进行了改进。对FM、KMP、BM三种算法在不同长度模式串条件下进行了实验，测试各自的匹配运行时间，进行数据分析和比较，，得出BM算法运行速度快高效高的结论。针对BM算法不能进行超大数据测试的缺陷进行了改进和优化，得到一新的算法BMQ算法，将BM和BMQ算法进行实验数据的分析比较，根据不同的模式串长度下字符匹配的次数、窗口移动的次数及模式匹配的时间和速度，找到一种高效字符串匹配的算法。

**关键词：** 字符串匹配，FM算法，KMP算法，BM算法，改进优化，BMQ算法

**中图分类号：**TP311.12

**文献标志码：**J

算法是始终是计算机学科中具有核心地位的基础课程。对于计算机学生而言，基本的要求是如何读懂算法、设计算法，然而如何设计优化算法则是越来越多的行业需要从海量数据中挖掘蕴藏的内在联系，大数据分析服务。字符串模式匹配技术作为大数据分析的基础和核心，其高效性直接决定大数据分析的效率。

# 1 概述

模式匹配算法作为计算机科学领域中的基础问题，已经有很长的研究历史从1974 年 KMP 算法的提出算起，它经过了近五十年的发展。模式匹配算法根据参与匹配的模式串的个数，分为单模式匹配算法和多模式匹配算法。

最早研究的模式匹配算法是单模式匹配算法，即朴素模式匹配算法 BF(Brute Force)算法。其匹配过程中，从左到右逐字符比较模式串与文本串，模式串只向右移动一个字符，因此BF算法的匹配性能比较低。直到1974 年由 D.E.Knuth与V.R.Pratt和J.H.Morris 共同提出的KMP 算法。与BF算法不同，KMP 算法利用模式串前缀的包含关系，构造了一个 next 数组。如果发生不匹配，根据 next 数组中的数值进行跳转，这样就避免了每次匹配结束都要回溯的问题，提高了算法的性能。 随后，Boyer和Moore于1977 年提出了BM算法，其创新点在于引入两个全新的的概念——好后缀启发式策略（Good Suffix）和坏字符启发式策略（Bad Character）在匹配过程中跳过那些不需要比较的字符，减少匹配次数，被认为是单模式匹配中最好的算法之一。随后各路专家学者在此基础上，对模式匹配算法提出了一系列的改进和创新，引入各种因素如概率、组合、多模式匹配、动态、扩展、内存、双数组字典树、过滤器……不断提升模式匹配的性能，模式匹配算法的研究层出不穷。

本文用C++代码来研究基于字符串匹配的算法，通过优化达到高效匹配的目的。

# 2 Brute Force（BF）算法

## 2.1 BF算法简介

BF算法的思想就是将目标串*T*的第一个字符与模式串*Ｐ*的第一个字符进行匹配，若相等，则继续比较Ｔ的第二个字符和*Ｐ*的第二个字符；若不相等，则比较*Ｔ*的第二个字符和*Ｐ*的第一个字符，依次比较下去，直到得出最后的匹配结果。

BF算法易于理解，但对于要比较多的字符，模式匹配的时间代价主要用于比较字符，所以时间效率不高。BF算法最好状态是：第一次匹配即成功，模式串刚好与目标串的开头长度为*m*的字符串*Ｔ0Ｔ1 ……Ｔm-1*匹配，此时比较次数为模式串长度*m*，时间复杂度为*O(m)*；最坏情况下要进行*ｍ*×*(ｎ-ｍ+１)*次比较，时间复杂度为*O(m×n)*。

## 2.2 Brute Force算法的改进

由于BF算法从Ｔ中的第一个字符开始进行比较，没有考虑已取得的部分匹配的情况，为了减少比较次数，提高匹配效率，《数据结构》教材对BF算法进行了改进，新改进的算法称为EBF算法。EBF算法中抽象模式串中的字符是从整个模式串中选出来的特殊的字符，它的内容体现了整个模式串的内容特征，其字符的位置贯穿整个模式串，相对于EBF算法从头开始一一进行匹配，能加快字符匹配失败的速度，其匹配效率可能就是 EBF算法一一匹配时的几倍或更高。虽然《数据结构》教材中的EBF算法优于BF算法，但仍存在着缺陷，没在考虑模式串本身的特点在文本串的位置，可以从这个角度对EBF算法进行改进优化。

## 2.3 Brute Force算法的优化（NBF）

在匹配阶段，需要定义如下几个局部变量：

整型变量*k*：用来记录目标串text的下标值，初值为０，终值为*Tlen-1*。

整型变量*i*：用来记录文本串中出现firstchar的位置，如果与模式串pattern的*ｊ*位置字符匹配，则自增。

整型变量*j*：用来记录文本串pattern的下标值,初值为０,如果与字符串text的*ｉ*位置字符匹配，则自增。

优化后BF算法（NBF）的匹配过程是通过判断文本串text的当前下标的值是否与模式串pattern的的首字符相同，若相同则进行匹配；若不相同则不进行匹配。算法的匹配过程描述为：

１）初始化相关变量：*k* ＝０；

２）当*k* ＜ *Tlen*时执行３）；否则执行９）；

３）若text［*k*］= firstchar，则执行４）；否则执行８）；

４）进行变量*i*的赋值，即*i* ＝*ｋ*；

５）进行变量*j*的赋值，即*j* ＝０；

６）当*i*，*ｊ*均小于text、pattern的长度时，执行６）；

７）text的子串text´开始与 pattern进行左匹配，若匹配成功，变量*i*、*ｊ*均自增，执行６）；若不匹配，跳出匹配窗口执行８）；

８）*ｋ*++，回到２）；

９）匹配结束。

算法的程序段为：

*ｋ*＝０；

while（*ｋ*＜*Tlen*）｛

if（text［*k*］= = firstchar)｛

int *i* ＝ *ｋ*;

int *j* ＝ ０;

while ( *i* ＜ *Tlen*＆＆*j* ＜ *Plen* )｛

if（text［*i*］= = pattern［*j*］）｛

*i* ++；

*j* ++；

｝

else

break;

｝

｝

*k* ++;

｝

## 2.4 NBF算法性能分析

NBF算法即优化后的BF算法，在匹配阶段的跳跃次数完全取决于firstchar在*T*中出现的次数，算法最好情况是*T*中没有出现firstchar，此时时间复杂度是*O*(1)；最坏情况*T*中所有字符都是firstchar，即出现*ｎ*次，则此阶段的时间复杂度是*O(m＊n)*。

综上分析，NBF算法性能比BF算法优秀。

# 3 Knuth-Morris-Pratt（KMP）算法

## 3.1 KMP算法的简介

KMP算法是一种改进的字符串匹配算法，由于BF算法效率低在于每轮主串从pos位置开始比较过程中，若字符不相等，主串会回退到pos+1位置，然后与匹配串第一个位置字符开始比较。KMP算法首先算出一个next 数组，匹配串每轮匹配在 *j*位置失配时，匹配串向右滑动的距离为 *j* - next[ *j* ]。next 数组的计算规则如下。其中*j*为匹配串索引：*j* = 0 时，next[ *j* ] = -1；*j* > 0，匹配串0至 *j*-1 位置的子串前缀和后缀数组中最长相同元素的长度为*maxL*，则next[ *j* ] = *maxL*。特殊的，当 *j* =1时，前缀和后缀都为空，所以*maxL*=0，next[1] = 0。

对于KMP 算法关键是next表，next表都是基于模式串，即子串来建立的。next值决定了字符失配回退的位置，它是由前一位字符在整个前子字符串中最长公共前后缀的长度值所决定。

KMP 算法中前缀数组有 next 数组和 nextval 数组之分，通常情况下可以混用。唯一不同的是，next数组在一些情况下有些缺陷，而nextval的创建是为了弥补这个缺陷。其优点在于：每当一趟匹配过程中出现字符不等时，不须回溯目标串，而是由已经得到的“部分匹配”的结果将模式串向右“滑动”尽可能远的一段距离后，继续进行比较，一定程度上能提高KMP算法的匹配效率。

## 3.2 KMP算法代码

int KMP ( char s1[], char s2[ ], int next[] )

{ int *i* = 0 ; int *j* = 0 ;

int *len1* = strlen ( *s1* ) ;

int *len2* = strlen ( *s2* );

while ( *i* < *len1* && *j* < *len2* ){

if ( *j* = = - 1 || s1[ *i* ] ==s2[ *j* ] ) {

*i* ++;

*j* ++; ｝

else

*j* = next [ *j* ];

}

if ( *j* > = *len2* )

return *i* - *len2* + 1;

else

return -1;

}

## 3.3 KMP算法的改进

KMP改进算法的优化思路可简单理解为：在匹配阶段出现不匹配的字符时，结合BM算法和KMP算法的优点，利用已知部分，令模式串最大程度地后移。 匹配阶段在遇到 *T*、*P* 不匹配时进行两个部分操作。第一部分是计算模式串移动 nextval[ *j* ]后末字符在文本中的位置；第二部分是比较目标串和模式串在该位置的字符是否匹配，如果两字符不匹配，则文本该位置的字符应进行坏字符跳转，否则按KMP的匹配过程继续比较。 KMP改进算法的流程图见图1。

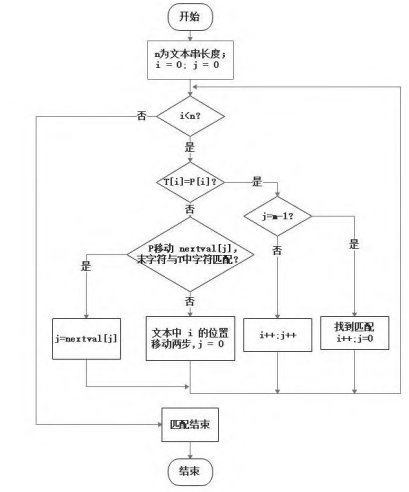


图1 KMP改进算法的流程图

## 3.4 程序代码

while ( *i* < length ){

if ( *j* = = -1 || *T* [ *i* ] = = *P* [ *j* ] ) {

if ( *j* = = m - 1 ) {

*i* + + ；*j*=0;//找到一个匹配 }

else {

*i* + +;

*j* + +;}

}

else{

//计算模式串移动 nextval [ *j* ]后末字符在文本中的位置

tLast = *T* [ *i* + pLen － nextval [ *j* ] －1 ];

//判断该位置的文本字符与末字符是否匹配

if ( tLast ! = pLast ) { *j* =0;

//进行坏字符跳转 *i* = *i* + bmBc [ tLast ] － nextval [ *j* ];

}

else

*j* = nextval [ *j* ];

} }

## 3.5 KMP算法性能分析

改进后的KMP算法在理论上可以弥补KMP算法从左开始对比的局限性，当左边的判断移动效率不高时，改为从右开始对比的BM算法坏字符跳转，结合两者的优点，减少比较次数。但是如果工作量较大，文本不充足时会导致匹配不准确，主要是因数KMP 算法匹配时有限制，可以对KMP算法进行改进：添与随机字符串相似的字符串，可以有效地提高效率。

在KMP算法的基础上加入变异的概念，以KMP为基础的相似字符串搜索匹配模型，以提高匹配的准确度。当匹配失败但在设定容错范围内时，忽略错误，继续进行比较。这样，KMP算法的准确度会大大提高。

# 4 Boyer-Moore（BM）算法

## 4.1 BM算法的简介

KMP算法在进行模式匹配时是基于前缀搜索的算法。与之相反，BM算法模式匹配时从右往左逐字符比较的算法，被称为基于后缀搜索的算法。BM算法是目前运行效率较高、匹配速度较快、使用最多的单模式匹配算法。BM算法的基本思想有三种，即从右到左字符比较法、坏字符启发式和好后缀启发式。关键是在匹配不成功时，利用坏字符规则（Bad Char）和好后缀规则(Good Suffix) 来确定跳转的距离。可以跳过很多的无用的字符，极大的提高了算法的性能，运行速度大幅提高。

## 4.2 BM算法规则

假设文本串为*T*，模式串为*P*，文本串和模式串的长度分别为*n*和*m*，*T i* （1 ≤ *i* ≤ *n* ) 和 *Pj* （1 ≤ *j* ≤ *m*) 分别表示文本串和模式串中的某个字符。

1. 坏字符规则

坏字符规则根据算法匹配失败时，匹配不成功的字符来计算模式串向右移动的距离。在计算移动距离时有下面两种情况：

1. 文本串和模式串当前参与匹配的字符分别为 *x* 和 *y，x ≠ y* ， *y* 是模式串第 *i* 个字符，在模式串的前 *i -* 1个字符中找到第 *j* 个字符为 *x* ，则模式串向右移*i - j* 位。

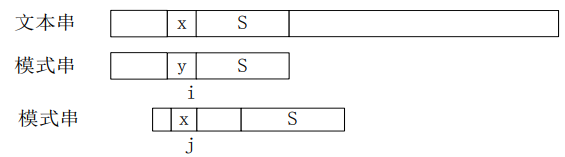


图2 BM算法坏字符规则情况 a)

1. 文本串和模式串当前参与匹配的字符分别为*x* 和 *y，x ≠ y* ， *y* 是模式串第 *i* 个字符，在模式串的前 *i -* 1个字符不存在字符*x* ，则模式串向右移*i* 位。

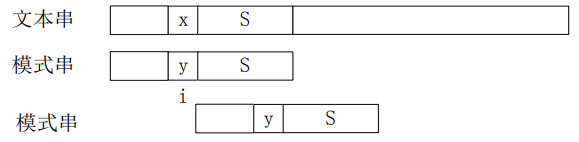


图3 BM算法坏字符规则情况b)

1. 好后缀规则

好后缀规则根据算法匹配失败时，已经匹配过的子串的来计算模式串向右移动的距离。

1. 文本串和模式串当前参与匹配的字符分别为*x* 和 *y，x ≠ y*， 已经匹配的子串为*S*， 在模式串*P*中还存在一个子串*P*[ *j - s* + 1*...j* ]与子串*S*相等，并且 *y ≠ z* ， 则模式串向右移动 *m - j* 位。

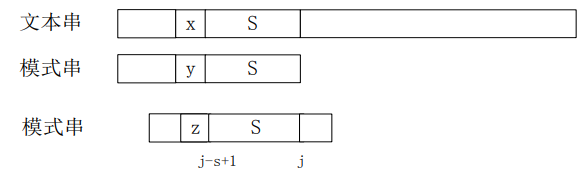


图4 BM算法好字符规则情况 a)

1. 文本串和模式串当前参与匹配的字符分别为*x* 和 *y，x ≠ y*，已经匹配的子串为*S* ，在模式串*P* 中存在子串*S*的最大后缀 *S*1，其中*S*1 为 *P* [0...  *j* ]，则模式串向右移动 *m - j* 位。

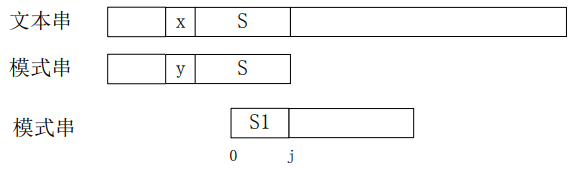


图5 BM算法好字符规则情况 b)

1. 如果 a)和 b)这两种情况都不满足，那么模式串向右移动 1 位。

## 4.3 BM算法的代码

＃define SET－DIMENSION 50 //C 的大小

chat ∗text； //文本 长度为 *n*

char ∗w； //模式 长度为 *m*

int d［SET－DIMENSION］;

int charno（char x）;

//给出字符 x 在 C 中的序号

void dvalue（int d［］, char ∗w）

｛

for （x in C） d［charno（*x*）］＝*m*;

for （*j*＝*m*－1;*j*＞0;*j*－）

if ( d［w［*j*］］＝*m* )

d［w［*j*］］＝*m*－*j*;//计算每个字符的 d 值

int BM（char ∗text, char ∗w） ｛

*i*＝*m*;

do｛

*j*＝*m*; *k*＝*i*;

while （*j*＞0）＆（w［*j*］＝text［*k*］）｛

*j*－; *k*－;

｝

*i*＋＝d［text［*i*］ ］;

｝

while（（*j*＝＝0）｜｜（*i*＞*n*））

if *j*＝＝0 return 1;

else return 0;

｝

## 4.4 BM算法的改进优化

BM算法很好地利用了后缀的匹配方式，是很优秀的单模匹配算法，但是BM算法构建好后缀表需要花大量的时间，不能满足大吞吐量数据的匹配需求。这是因数BM算法存在不必要的比较，当模式串在文本串中的初次匹配失败后，根据对BM算法分析，结合了BM算法的坏字符规则思想以及参考其它改进算法的优点，考虑了模式串末字符对应的文本字符与文本字符下一个字符的独特性和组合性，提出了一个改进算法—BMQ算法。

BMQ算法思想：设文本串与模式串的位置对应情况为*T*[*i*]…*T*[*i*+*m*-1]与*P*[1]…*P*[*m*]，当出现失配的时候，考虑文本字符*T*[*m*]的下一字符*T*[*m*+*i*]，且利用坏字符规则，计算Skip[text[*m*+*i*]]的值。然后利用坏字符表，考查模式串末字符对应的文本串字符的更远的字符*T*[*i*+*m*+*x*]（0≤*x*≤*m*），以获得更大右移。与此同时，并考虑模式串首字符*P*[1]，计算移动距离。此种方法有很大概率减少字符比较次数，并能有效缩短字符匹配的过程。BMQ算法的流程图参见图6。

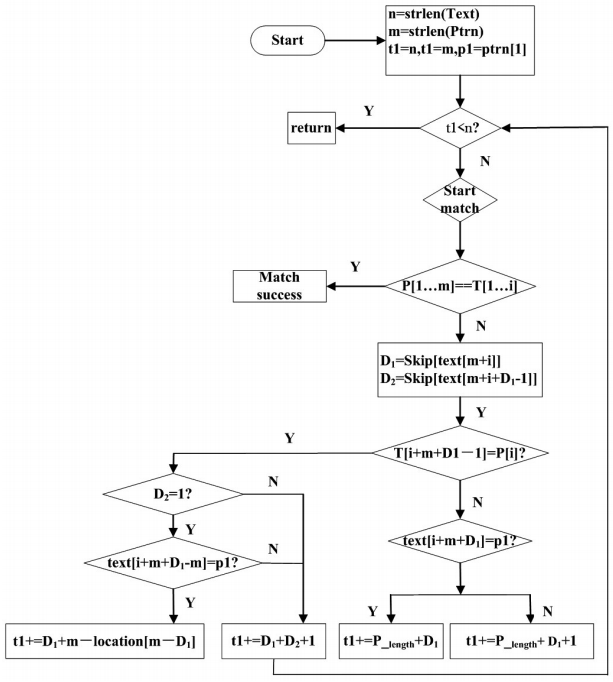


图5 BMQ算法流程图

BMQ 算法的执行过程主要由预处理和模式

匹配两部分组成。

（1）在BMQ算法预处理阶段，需生

成位置函数 location 及坏字符数组函数 Skip 。

Skip数组的预处理过程如下：

while（模式串长度*m*!=0）

skip[(unsigned char）\*pattern++]=*m*--;

return skip;

函数location实现方法如下：

temp=skip[*p*[*i*]];

skip[*p*[*i*]]=*i*;//location 数组中保存相同字符

的前一skip值

location[*i*]=temp;

*i*++;

对模式串进行预处理之后，算法进入模式匹配阶段。当发生字符不匹配时，该算法首先计算模式串*P*最右端字符对应文本串字符*Ti*所确定的下一次右移位置*D*1，同时计算*Ti*+1确定的下一次右移位置*D*2。

*D*1 = Skip [[ *m* + *i* ]]

*D*2 = Skip[text[*m* + *D*1 + *i* - 1]]

## 4.5 BMQ算法的性能分析

BMQ算法是对BM算法的改进，算法性能要优于BM算法。

（1）两种算法利用坏字符规则，在文本串*T*1*T*2…*T*n中，自向左右查找与*P*1*P*2…*P*m完全相同的子串，计算BM 算法坏字符移动值不会大于BMQ算法的移动值。随着模式串增长这两种算法的最大移动量相差就越大，BMQ 算法右移次数、字符比较次数会明显少于BM算法。

（2）对于BMQ算法，考虑了模式串末字符对应的文本字符与文本字符下一个字符的独特性和组合性，即考虑两个字符。在模式串中出现的概率要比一个字符在模式串中出现的概率低很多。因此，BMQ算法获得最大移动距离的概率要比BM算法的概率大很多。

（3）BMQ 算法相对于其它改进算法，在预处理阶段增加位置函数location，生成大小为模式串长度 *m* 的数组 location，用于记录字符所在的位置。为此，BMQ算法在预处理阶段增加额外的辅助空间 *O*(*m*)，而 BM 算法完成搜索的时间复杂度是*O*(*m*+*nr*)。在匹配阶段，算法的时间复杂度为*O*(*mn*)，最好的情况下时间复杂度为*O*(*n*/(2*m*+2))，BM在最好情况下的时间复杂度分别为 *O(n*/*m*)。

所以，BMQ 算法平均时间复杂度、空间复杂度比BM算法都要小。

综上表明，BMQ算法在最大移动量、产生最大移动量的概率、右移次数、字符比较次数、时间复杂度、空间复杂度方面都优于BM算法。所以，BMQ算法有效地加快了匹配速度，提高了匹配效率。

# 5实验测试和结果分析

## 5.1 FB算法、KMP算法和BM算法比较

实验采用1000000个英文文本作为正文串，从中截取不同个数的文本串长度，分别3个、5个、10个、100个字符作为文本串，利用随机函数，随机截取，得出实验数据见图7、8、9、10、11、12。

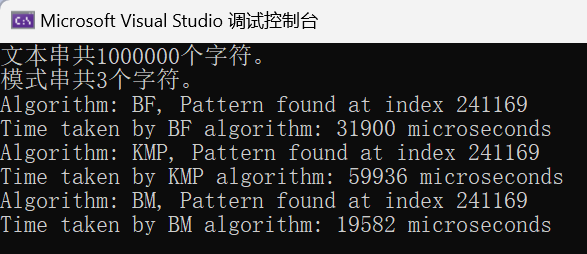


图7 模式串3 字符FB、KMP、BM算法的运行时间



图8 模式串5 字符FB、KMP、BM算法的运行时间



图9 模式串10字符FB、KMP、BM算法的运行时间



图10模式串100字符时FB、KMP、BM算法的运行时间

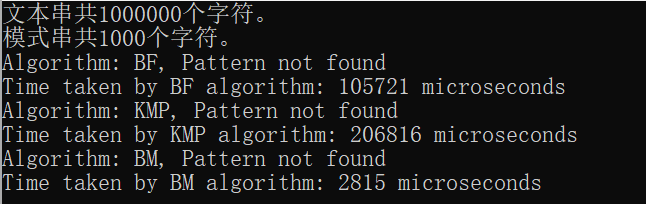


图11 模式串1k字符时FB、KMP、BM算法的运行时间

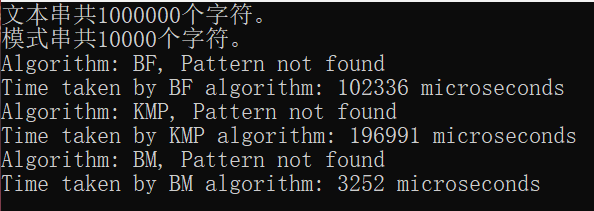


图12模式串10k字符时FB、KMP、BM算法的运行时间

可以得出以下结论：开始时随着模式串长度的增加FB算法、KMP算法和BM算法匹配运行的时间都增加；当模式串长度达到某一值时，匹配运行时间会达到峰值出现拐点，之后匹配运行时间会减少，最终模式串长度达到某一值后匹配运行时间基本趋于相对稳定，见表1.。

表1. FB、KMP、BM算法运行时间统计表

单位：ms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模式串长度 | 匹配情况 | FM算法 | KMP算法 | BM算法 |
| 3字符 | 成功 | 31900 | 59936 | 19582 |
| 5字符 | 失败 | 128601 | 283759 | 50358 |
| 10字符 | 失败 | 129179 | 271707 | 27418 |
| 100字符 | 失败 | 161649 | 337460 | 5885 |
| 1k字符 | 失败 | 105721 | 206816 | 2815 |
| 10k字符 | 失败 | 102336 | 196991 | 3252 |

从表1.统计可知：KMP算法在模式串长度短小的情况下比FM算法匹配运行的时间少、有优势；但当模式串长度过长时反而不如FM算法。而BM算法无论模式串长短，其匹配运行的时间都比FM算法和KMP算法少，对特别长的模式串更具优势，大大减少了匹配时间，提高了效率。

## 5.2BMQ算法与BM算法比较

实验条件：由实验数据可得出以下结论：实验采4.65 MB的英文文本内容作为正文串。从中截取文本串500个，每个文本串长度为2.6万个字符；利用随机函数，随机截取模式串5000个，长度在5~30个字符之间。

BMQ算法与BM算法实验数据见表2。

表2. BMQ算法与BM算法模式匹配统计表

|  |  |  |  |
| --- | --- | --- | --- |
| 模式串长度 | 次数 | BM算法 | BMQ算法 |
| 50000字符 | 字符匹配（次） | 258974 | 348675 |
| 窗口移动（次） | 198000 | 176896 |
| 70000字符 | 字符匹配（次） | 348675 | 246500 |
| 窗口移动（次） | 288675 | 216547 |
| 150000字符 | 字符匹配（次） |  |  |
| 窗口移动（次） |  |  |

可以得出以下结论：无论是BM算法还是BMQ算法一开始随着模式串长度的增加其字符比较次数、字符移动次数都在增加；当模式串长度达到某一数量级时，两种算法都达到峰值，出现拐点，之后随着模式串长度的继续增加，字符比较次数和字符移动次数呈下降趋势；当模式串长度达到某一数量级时两种算法的字符比较次数和字符移动次数彼此接近，见图13。

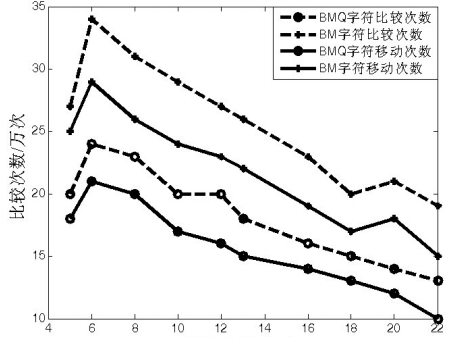


图13 BM算法和BMQ算法字符比较及移动次数图

然而是，BM算法和BMQ算法则是随着长度模式串增加，其匹配所需的运行时间在增加，匹配的速度在见图14。

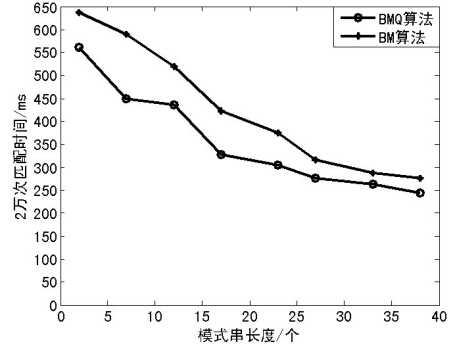


图14 BM算法和BMQ算法运行时间图

总体上，BMQ算法字符比较次数和字符移动次数比BM算法要低很多，匹配所需的运行时间也少，由此可见BMQ算法比BM算法更高效，是对BM算法的改进优化。

# 6 结语

本文首先介绍模式匹配的研究现状及三个经典算法，分别为 BF 算法、KMP 算法、BM 算法等，针对这三个经典算法存在的缺陷做了改进，并通过实验测试结果将三个经典算法查找模式串所消耗的时间，进行分析对比，其结果表明，BM算法比较简单，在一般情况下时间复杂度为 *T*（*n*）＝*O*（*n*）；但是在某些特殊情况下，其时间复杂度为*O* ( *m× n* )，BM 算法较KMP 算法简单，效率更高，实用性也更强。同样，在对三个经典算法的改进后，可以减少匹配次数，提高匹配效率。

BF算法的改进算法NBF算法和KMP算法的改进有着一曲同功之处，都是利用匹配阶段的减少跳跃次数。不同之处在于规则的设置，特别是坏字符。而BM算法的改进算法BMQ算法则是在利用坏字符增大了移动距离，在预处理阶段增加额外的辅助空间，在匹配阶段，减少了算法的时间，达到提高字符串匹配算法的效率。

当然，随着移动互联网的发展，网络数据的规模会越来越大，针对于大规模的模式集，按具体的应用的要求，根据实际情况做出调整，设计出更适针对海量数据的并行模式匹配算法是发展趋势。

**参考文献：**

1. 殷人昆.数据结构（用面向对象方法与C++语言描述）[M]. 3版. 北京：清华大学出版社，2021.
2. 马锐彦.KMP算法的优化与应用[J].电脑知识与技术,2023,19(20):73-75.DOI:10.14004/j.cnki.ckt.2023.1008.
3. 余飞.模式匹配算法的分析与研究[J].电脑知识与技术,2018,14(10):251-252.DOI:10.14004/j.cnki.ckt.2018.1165.
4. 刘泷,高仲合,宋先强等.Boyer-Moore模式匹配算法的一种改进算法[J].电子技术,2016,45(11):72-75.