针对 QSP 算法的研究与分析①

李 莉, 江育娥, 林 劼

(福建师范大学 软件学院, 福州 350108)

摘 要: BM 算法是经典的单模式匹配算法, QS 算法是基于 BM 算法的改进算法, 由于 QS 算法仅仅分析下一字符 T[j+m]计算右移量, 整体的匹配效率并不高, 因此在 QS 算法的基础上提出一种改进算法(QSP). QSP 算法在预处理阶段从左向右找出模式串中出现 1 次以上的单字符, 计算出这些字符的跳转期望值差, 得到最大差值和相对应的字符位置 *maxPos*, 并修改 skipp2 数组的值; 在匹配阶段, 首先比较 P[*maxPos*]与 T[*j*+*maxPos*]是否相等, 然后再利用两个数组 skipp1 和 skipp2 进行右移, 保证每次右移的距离达到最大. 通过实验证明, 该算法总的比较次数和运行时间都低于 QS 算法, 匹配效率得到明显的提高.

关键词: 模式匹配; QS 算法; QSP 算法; 跳转期望值差

Research and Analysis of QSP Algorithm

LI Li, JIANG Yu-E, LIN Jie

(School of Faculty of Software, Fujian Normal University, Fuzhou 350108, China)

Abstract: BM algorithm is a classical single pattern matching algorithm. QS algorithm is an improved algorithm of BM algorithm. Because computing the moving distance to right position only by analyzing the character T[*j*+*m*], the overall matching efficiency of QS is not high. Therefore, an improved algorithm (QSP) accordingly to the QS algorithm is proposed. The core idea of QSP algorithm is finding all characters that appear more than once in the pattern string from left to right, calculating the jumping expectation difference value of these characters, getting the highest expectation difference value and *maxPos* value that is the position of it in the preprocessing phase, and changing the value of skipp2 array. During the matching phase, in order to move the pointer farthest at each time, it firstly considers the relationship between P[*maxPos*] and T[*j*+*maxPos*], then moves to right by using the skipp1 and skipp2 arrays. The experimental result shows that the comparison number and matching time of QSP are less than QS. Its efficiency has been improved obviously.

Key words: pattern matching; QS algorithm; QSP algorithm; jumping expectation difference value

1 引言

模式匹配算法不仅在模式识别领域中起到很大的作用, 也广泛应用于图形处理, 文本挖掘, 网络入侵检测系统等领域. 由此可见, 寻找精确而有效的模式匹配算法成为当务之急.

模式匹配算法分为单模式匹配算法和多模式匹配算法. 经典的单模式算法主要包括 BM(Boyer-Moore)算法[1]、KMP(Knuth-Morris-Pratt)算法[2]、BMH 算法[3]和 QS 算法[4]等[8]. 本文论述的算法是基于字符比较的单模式匹配算法, 在 QS 算法的基础上提出一种新的改进算法 QSP(QS Plus)算法, 并将 QSP 算法与 BMH 算法、QS 算法、Im_Sunday 算法[5]和 I_BMH2C 算法[6]的性能进行比较, 理论与实验均证明 QSP 算法的运行效率高于 BMH、QS、Im_Sunday 和 I_BMH2C 算法.

2 相关算法分析

本文中的文本用 T 表示, 即 T=T[0.....n-1],长度为 n; 模式串用 P 表示, 即 P=P[0.....m-1],长度为 m; 并

① 基金项目:国家自然基金(61472082);福建省自然基金(2014J01220)
收稿时间: 2015-06-19; 收到修改稿时间: 2015-08-17

且满足条件n>>m. 如何快速有效得找出文本T中所有匹配的模式串 P 的起始位置 j: T=P[0], T[j+1]=P[1].....T[j+m-1]=P[m-1]是单模式匹配算法研究的重点.

2.1 Boyer-Moore(BM)算法

1977年Boyer和Moore提出著名算法—BM算法[1], BM 算法的基本思想为字符是从右向左进行逆向匹配, 在匹配过程中利用坏字符跳跃和好后缀两个规则进行右移, 取两者中的最大值为右移量, 跳过尽量多的字符. BM 算法的最坏时间复杂度为 O(n*m), 最好时间复杂度为 O(n/m). 坏字符跳转和好后缀规则定义如下, 其中 c 为文本 T 与模式 P 比较时出现的不匹配文本字符:

$$BCp=\begin{cases}1)m; P[j]\neq c(0\le j\le m-1);\\ 2)m-j-1; j=\max\left\{\begin{matrix}j\mid P[j]=c,\\ 0\le j\le m-1\end{matrix}\right\},\end{cases} \quad (1)$$

$$GSp=\min\left\{\begin{matrix}s\mid (P[j+1,...m-1]=P[j-s+1,...,\\ m-1-s])\&\&P[j]\neq P[j-s](j>s),\\ P[s+1,...,m-1]=P[1,...,m-s](j\le s)\end{matrix}\right\} \quad (2)$$

2.2 BMH 算法与 QS 算法

1980年Horspool提出改进与简化BM算法的论文—BMH算法[3], BMH 算法预处理部分仅使用 BM 算法的坏字符规则, 当 P[0…m-1]与 T[0…n-1]进行字符比较时, 可以采用自左向右或者自右向左方向, 若出现不匹配情况, BMH 算法仅分析 T[j+m-1]字符来决定模式串右移的距离, 当 T[j+m-1]不出现在模式串中, 右移量最大等于 m. 由于 BMH 算法简化了预处理部分, 在匹配阶段也省去比较步骤, 所以运行效率高于 BM 算法.

1990 年 Sunday 在 BMH 算法的基础上提出另一种改进算法—QS 算法[4], QS 算法与 BMH 算法的区别是: 在匹配过程中, 若出现不匹配的情况, 指针 j 至少移动一个位置, 那么在下一次的比较窗口中, T[j+m]就是待处理对象, 因而只使用 T[j+m]决定右移量. 当 T[j+m]字符不在模式 P 中时, 它的右移量为 m+1, 大于 BMH 算法的最大右移量 m, 故匹配效率高于 BMH 算法. QS 算法的坏字符跳跃规则略有改进, 规则如下:

$$QSBc=\begin{cases}1)m+1; P[j]\neq c(0\le j\le m-1);\\ 2)m-j; j=\max\left\{\begin{matrix}j\mid P[j]=c,\\ 0\le j\le m-1\end{matrix}\right\}\end{cases} \quad (3)$$

以 T="ACGATCGCACACCTACCGAA TCAC"和模式串 P="CGAATCAC"为例, 预处理部分得到的 qsBc 数组值为: qsBc[A,C,G,T]=[2,1,7,4], 匹配过程如表:

表 1 QS 算法的匹配过程

	A	C	G	A	T	C	G	C	A	C	A	C	C	T	A	C	C	G	A	A	T	C	A	C
1	C	G	A	A	T	C	A	C																
2			C	G	A	A	T	C	A	C														
3					C	G	A	A	T	C	A	C												
4						C	G	A	A	T	C	A	C											
5										C	G	A	A	T	C	A	C							
6																	C	G	A	A	T	C	A	C
移动窗口次数为: 6 次;																								

2.3 Im_Sunday 算法与 I_BMH2C 算法

Im_Sunday 算法[5]是 2011 年吴旭东在 QS 算法的基础上提出的改进算法, Im_Sunday 算法使用 T[next]和 T[nnext]两个字符进行右移, T[next]字符是模式串最右端对应的文本串字符的下一个字符 T[j+m], T[nnext]字符是 T[next]后面 m 长度位置的字符, 即 nnext=next+m. 当 T[next]字符在模式串中出现时, 匹配方法与 QS 算法一致; 当 T[next]字符不在模式串中出现时, 再次判断 T[nnext]字符是否在模式串中出现, 若出现, 将模式串最右端第一个出现的字符与 T[nnext]对齐, 否则将整个模式串移动到 T[nnext]的右侧.

I_BMH2C 算法[6]是 2014 年王艳霞在 BMH2C 算法的基础上提出的改进算法, BMH2C 算法结合了 BMH 算法和 QS 算法的优点, 同时使用 T[j+m-1]和 T[j+m]两个字符, 并将两个字符作为子串来决定模式串的右移量[7], 但 BMH2C 算法并没有考虑到双字符串在模式串中出现一次及以上的情况. I_BMH2C 算法考虑双字符串在模式串中出现的次数, 以及该双字符串在模式串中对应位置的后继字符与字符 T[k+2]的相等关系. 该改进算法分为双字符在模式串中出现 0 次、1 次和 1 次以上

三种情况进行分析. 利用两个右移数组和一个模式串预处理数组, 在匹配过程中通过判断字符 T[k+2]与模式串预处理数组中相应字符是否相等, 再选择右移数组之一的对应值作为当前窗口的右移量[6].

上述两种改进算法仅利用 T[*j*+m-1]或 T[*j*+m]等字符信息进行右移, 从而进行很多次不必要的比较和跳跃, 造成匹配速度变慢. 当模式串长度较长且字符集较小时, T[next]在模式串中的出现概率偏大, Im_Sunday 算法的最大跳跃距离的出现概率偏小, 匹配效率并没有得到很大的提高. 在一般情况下, 双字符在模式串中出现 1 次以上的情况是属于较少部分, 当双字符在模式串中出现 1 次以上的概率较小且 pre[t[k]][t[k+1]]= T[k+2]的出现概率偏大时, 每次向右移动的距离几乎与 BMH2C 算法相等, I_BMH2C 算法在预处理和匹配阶段的计算复杂度比 BMH2C 和 QS 算法的大, 此时 I_BMH2C 算法的运行效率并没有提高很大. 综上所述两种改进算法都有待进一步提升.

3 QSP算法

3.1 QSP 算法思路

通过研究单模式匹配算法的多种改进方法[9-11], 并结合 Im_Sunday 算法和 I_BMH2C 算法的不足之处, 考虑单字符在模式串中出现 1 次以上的概率比双字符的大很多, 所以本文从单字符的角度进行分析, 提出一种改进算法 QSP 算法.

QSP 算法的思路如下:

预处理阶段根据公式(3)得到 skipp1 数组和初步的 skipp2 数组; 从左向右寻找模式串中出现 1 次以上的单字符, 并计算这些字符的跳转期望值差, 得到最大的差值和相应字符的位置—*maxPos*; 改变 skipp2[P[*j*+m-*maxPos*]]的值为 *maxPos-j*, 并将 skipp2 数组中小于 *maxPos-j* 的值都改为 maxPos-j. 从左往右遍历模式串 P 中的每个字符, 分为两种情况进行分析:

1) 当满足条件 P= P[*i*](0≤*j*<*i*<m)时, P[i]的跳转期望值差的计算公式如下:

$$ES_i=\begin{Bmatrix} i-j-skipp1[P[j+(m-i)]], \\ (P[i]=P[j],0\le j<i<m) \end{Bmatrix} \quad (4)$$

单字符的跳转期望值差的计算原理是: 由于 P= P[*i*], 若 P[*j*]=T[*i*], 则右移 *i-j* 后, P=T[*i*](此处假设 P[*i*]字符对应的文本字符为 T[*i*], 末字符的下一字符为 T), 若出现 P[*j*+(m-*i*)]=T 的情况, 右移 *i-j* 后, 已有两字符相匹配, 而只考虑下一字符 T, 右移距离为 skipp1[T], 两者的差就是 P[i]字符的跳转期望值差; 如果出现 P[*j*+(m-*i*)] ≠T 的情况时, 比较 skipp1[T]与 maxPos-*j* 的大小, 选择较大者为右移量, 保证每次右移量达到最大. 当 *i-j* 越大, ESi 就越大, 每次窗口的右移量就越大, 从而提高算法的匹配效率.

2) 当模式串中不存在 P= P[*i*](0≤*j*<*i*<m)的情况时, *maxPos* 值默认为 m-1.

匹配阶段分为两部分: 首先比较 P[*maxPos*]与 T[*j*+*maxPos*]是否相等, 若不等, 直接利用 skipp1 数组分析 T[*j*+m]字符计算右移量; 若相等, 则继续寻找全匹配的模式串, 再引用 skipp2 数组进行跳转.

算法流程图如图 1 所示.

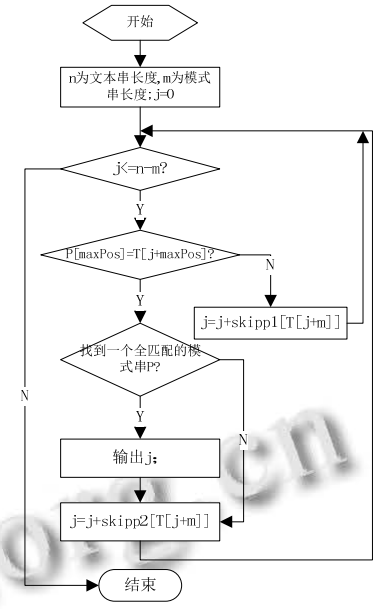

图 1　QSP 算法流程图

3.2 预处理阶段

QSP 算法的预处理部分包含两个部分: skipp1 数组在 QSP 算法中的作用是计算匹配阶段第一部分的右移量, 当 P[*maxPos*]!=T[*j*+*maxPos*]时,引用 skipp1 数组决定模式串的右移距离; 第二部分是 QSP 算法预处理部分的改进部分, skipp2 数组的作用是当 P[*maxPos*]=T[*j*+*maxPos*]时, 调用 skipp2 数组进行右移. 预处理阶段的时间复杂度为 O(m*m).

预处理部分的核心代码如下:

```
for(i =1;i<m-1;i++){
//从左向右寻找符合条件的单字符, 并计算 ESi;
```

```
    for(j =0; j<i ;j++){
      if (P=P[i]){
        ESi=(i-j)- skipp1 [P[j+m-i]];
        }
    }
  //寻找最大的跳转期望值差和 maxPos
    if (ESi >=maxESi){
      maxPos=i;
      maxj=j;
      maxESi = ESi;
      flag=1;
    }
  }
  //找到最大 ESi,并修改 skipp2 数组的值
  if (flag==1){
     maxvalue= maxPos- maxj;
     skipp2[T[maxj+m-maxPos]]= maxvalue;
       for (i = 0; i <ASIZE; i++){
                if(skipp2[i]<maxvalue)
                    skipp2 [i]=maxvalue;
       }
  }
```

以模式串 P="CGAATCAC"为例, skipp1 数组的值为: skipp1 [A,C,G,T]=[2,1,7,4]. 当 i=3 时, 对应字符为'A', 位置 j=2 处的字符'A'与它一致, i=3 与 m=8 的相差距离为 5, 模式串中位置 7(j+m-i=2+8-3=7)处的字符为'C', 当 T[j+m]='C'时, 右移 i-j=3-2=1 步后, 在下一轮的比较窗口中, 模式串中已有两个字符与文本中对应字符相匹配. P[3]的跳转期望值差为 ES_3=(i-j)-skipp1 =(3-2)-1=0, 如表 2 所示. 同理当 i=5 时, ES_5=(i-j)- skipp1 [A]= (5-0)-2=3; 当 i=6 时, 位置 j=2 和 j=3 的字符'A'与 P[6]相等, 应取最右边的字符 P[3] (避免漏掉), 即 ES_6=(i-j)-skipp1= (6-3)-1=2; 同理ES_7=0; 所以该模式串的*maxPos*=5, 如表 3 所示. 最后修改值: skipp2[P[j+m-*maxPos*]]=skipp2 [P[0+8-5]]=skipp2[A]=*maxPos*-j=5-0=5, skipp2=5, skipp2[T]=5.

表 2 寻找在 P 中出现 1 次以上的单字符

	0	1	2	3	4	5	6	7	8
P	C	G	**A**	**A**	T	C	A	**C**	
			j=2	**i=3**					
	若 T[3]=P[3]且下一个字符为 C，右移 1 步，可保证 P 中位置 2 与 7 的两字符 A…C 与文本相对应字符相等								

表 3 每个单字符的跳转期望值差

	0	1	2	3	4	5	6	7	8
P	C	G	A	**A**	T	**C**	**A**	**C**	
				i=3, ES_3=0					
						i=5, ES_5=3			
							i=6, ES6=2		
								i=7,ES_7=0	
其他单字符不存在跳转期望值差									

当 P[5]=T[j+5]且 T[j+m]字符为'A'时, 右移 5 个位置, 不仅保证下次窗口中已有两个字符匹配, 而且右移量也大于原先的跳转值 2; 若 T[j+m]字符为'T'时(即不等于'A'时), 右移 4 步, P[1] ≠'C', 存在不必要的比较, 应右移 5 步, P[0] ='C', 所以将 skipp2[T]的值改为 5. 每次右移距离大于或者等于 QS 算法的右移量, 因此算法的匹配效率得到一定的提高.

3.3 匹配阶段

QSP 算法的匹配阶段分为两个步骤, 具体核心代码如下:

```
  while(j<=n-m){
  //若 P[maxPos] ≠T[j+maxPos], 引用 skipp1 数组;
    while(P[maxPos]≠T[j+maxPos]){
      j =j+ skipp1 [T[j+m]];
      if(j>n-m ){
        return
      }
    }
    //若相等, 引用 skipp2 数组;
    compare P[0…,m-1] and T[j,…j+m-1]
    if all matched then do
      output j
    end if
    j =j+ skipp2 [T[j+m]];
  }
```

同样以文本 T="ACGATCGCACACCTACCGAATCAC"和模式串 P="CGAATCAC"为例, 预处理部分得到的 maxPos 和两个数组值分别为: maxPos=5,skipp1[A,C,G,T]=[2,1,7,4],skipp2=[A,C,G,T]=[5,5,7,5], 匹配过程如表 4 所示.

表 4 中可见, 每次窗口移动后, 先比较 P[5] 与 T[j+5]是否相等, 在第一次窗口中 P[5]=T[5], 且没有找到全匹配模式串, 右移 skipp2 [A]=5 步, j=0+5=5; 在

第二次的窗口比较中由于 P[5] ≠T[10],故直接右移 skipp1[T]=4 步,*j*=5+4=9; 同理第三次窗口的右移距离为 7, *j*=9+7=16; 最后在第四次窗口中找到全匹配模式串, 输出 *j* =16. QSP 算法的右移次数为 4 次, 低于 QS 算法的 6 次, 匹配效率高于 QS 算法.

表 4　QSP 算法的匹配过程

	A	C	G	A	T	C	G	C	A	C	A	C	C	T	A	C	C	G	A	A	T	C	A	C
1	C	G	A	A	T	C	A	C																
2						C	G	A	A	T	C	A	C											
3										C	G	A	A	T	C	A	C							
4																	C	G	A	A	T	C	A	C
移动窗口次数为: 4 次;																								

4　算法的分析

QSP 算法的目的是尽量使每次窗口移动的距离达到最远, 从而减少不必要的比较和跳跃次数, 降低匹配时间, 提高运行效率. 算法的最坏时间复杂度为 O(m*n),最好时间复杂度为 O(m/n).

QSP 算法不仅考虑 T[*j*+m-1]或 T[*j*+m]等字符信息, 还对整个模式串进行分析, 找出最大的期望值差和相应字符的位置 *maxPos*, 在匹配阶段同时调用 skipp1 和 skipp2 数组进行跳跃, 保证每次的右移距离达到最大. 一般情况下, 单字符在模式串中出现 1 次以上的概率比双字符的大, 找到最大期望值差的概率更大, QSP 算法在预处理阶段考虑单字符在模式串中出现 1 次以上的情况比 I_BMH2C 算法的更有优势, 且计算复杂度也比 I_BMH2C 算法的低, QSP 算法得到的 ES_i 越大, 每次的右移量就越大, 且右移后存在两字符已匹配的概率, 找到全匹配模式串的速度就更快. QSP 算法在匹配阶段也存在一定的优势, 当 P[*maxPos*] 的值不等于 T[*j*+*maxPos*]时, 说明 P[0…m-1]与 T[*j*...*j*+m-1]不匹配, 直接引用 skipp1 数组, 分析下一字符 T[*j*+m]决定跳转距离, 只有当 P[*maxPos*] =T[*j*+*maxPos*]时, 才进行一一比较; QS、I_BMH2C 和 Im_Sunday 算法的第一步都是将 P[0…m-1]与 T[*j*...*j*+m-1]从左往右或者从右往左进行一一比较, 出现不匹配情况, 才进行模式串右移; 例如出现模式串格式为“baaaabaaaa”和文本格式以 a 为主“aaaaaaaa…..”的情况, QSP 算法的字符比较次数将得到很大的降低, 而在一般情况下, 平均比较次数也会得到一定的减少[12-13].

通过以下的实验证明, QSP 算法在“10M.txt”文本中的平均字符比较次数比 QS、I_BMH2C 和 Im_Sunday 算法的分别降低 73%、28%和 33%, 在“bible.txt”文本中的平均字符比较次数比 QS、I_BMH2C 和 Im_Sunday 算法的分别降低 147%、36%和 52%; QSP 算法在“10M.txt”文本中的平均运行时间比 QS、I_BMH2C 和 Im_Sunday 算法的分别减少 62%、35%和 34%, 在“bible.txt”文本中的平均运行时间比 QS、I_BMH2C 和 Im_Sunday 算法的分别减少 93%、36%和 43%. 综上所述, QSP 算法的匹配效率得到了一定的提高.

5　实验结果

本文针对 BMH 算法、QS 算法、I_BMH2C 算法、Im_Sunday 算法和 QSP 算法进行对比实验分析. 算法的实验是在 VC6.0 编译器上实现, 并运行在 CPU 是 Intel(R)2.30GHz,内存为 4GB 的计算机上, 算法采用 C++语言实现.

实验的文本是“10M.txt”的文本 和“bible.txt”文本. “10M.txt”是随机生成的 10MB 大小的文本, 字符集大小为 128; “bible.txt”文本来源于坎特伯雷语料(http://corpus.canterbury.ac.nz/), 字符集大小为 63. 实验随机构建长度为 m(5<m<50)的模式串,每组测试的次数为 10 次, 执行上述算法程序, 统计出当模式串长度不同时五种算法的比较次数和运行时间.

表 5　五种算法的比较次数(10M)

10M					
m	BMH	QS	I_BMH2C	Im_Sunday	QSP
5	2425918	2126950	2055199	2037999	2032801
10	1572479	1505285	1152478	1212955	1091996
15	1394391	1283519	1189438	1166160	1105455
20	1231518	1239842	819839	951680	737596
25	1273178	1108797	759358	745759	515841

30	1048320	1055035	1585920	1320481	1051679
35	903838	887040	534240	551353	315841
40	856796	866880	651839	640479	510721
45	863522	833274	567841	685827	392321
50	967672	917264	490558	657442	413289

表 6 五种算法的比较次数(bible)

bible					
m	BMH	QS	I_BMH2C	Im_Sunday	QSP
5	4623636	4291501	3221970	3172593	3104618
10	2608379	2565438	1718166	1557300	1694463
15	1880527	1936604	1165197	1146103	1109838
20	1831669	1675669	892171	1170599	830821
25	1461369	1356931	898933	821442	795046
30	1409999	1405119	606043	780165	322593
35	992047	955387	682353	746143	362501
40	1095053	1145456	539600	675395	373934
45	890876	890372	422853	487526	289755
50	771927	778674	413730	450776	240741

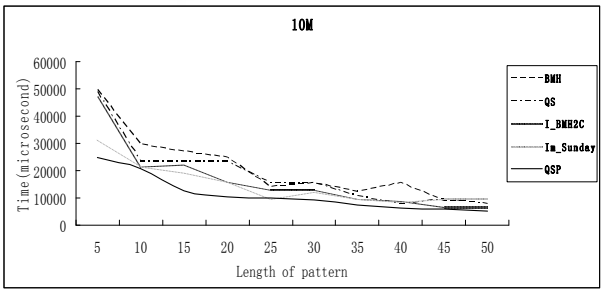

图 2 五种算法的运行时间(10M)

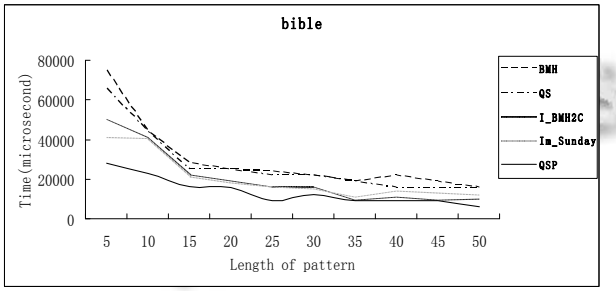

图 3 五种算法的运行时间(bible)

从以上的比较次数表和运行时间图可以看出, QSP 算法的运行效率明显高于 BMH 算法、QS 算法、I_BMH2C 算法和 Im_Sunday 算法, 说明在一般情况下, QSP 算法在预处理和匹配阶段的改进之处比 I_BMH2C 和 Im_Sunday 算法的更有优势.

6 结语

本文通过分析 BMH 算法、QS 算法、I_BMH2C 算法和 Im_Sunday 算法,并在 QS 算法的基础上提出一种改进算法—QSP 算法, 该改进算法在预处理部分计算出现在模式串中 1 次以上的字符的跳转期望值差, 并得到差值最大的字符位置 maxPos, 在匹配阶段针对不同的情况进行比较, 分别调用 skipp1 和 skipp2 数组进行右移, 使得每次右移距离达到最大. 通过实验证明, QSP 算法总的比较次数和运行时间都低于 BMH 算法、QS 算法、I_BMH2C 算法和 Im_Sunday 算法, QSP 算法的运行效率得到明显的提升.

参考文献

1 Boyer RS, Moore JS. A fast string searching algorithm. Communications of the ACM, 1977, 20(10): 762–772.

2 Knuth DE, Morris JH, Pratt VR. Fast Pattern matching in string. SIAM Journal on Computing, 1977, 20(6): 323–350.

3 Horspool RN. Practical fast searching in strings. Software-Practice and Experience, 1980, 10(6): 501–506.

4 Sunday DM. A very fast substring search algorithm. Communications of the ACM, 1990, 33(1): 132–142.

5 武旭东.Snort 入侵检测系统研究与应用[学位论文].长春:吉林大学,2011.

6 王艳霞,江艳霞,王亚刚,李烨.BMH2C 单模匹配算法的研究与改进.计算机工程,2014,40(3):298–302.

7 钱屹,侯义斌.一种快速的字符串匹配算法.小型微型计算机系统,2004,25(3):410–413

8 Faro S, Lecroq T. The exact online string matching problem: a review of the most recent results. Acm Computing Surveys, 2013, 45(2): 94–111.

9 韩光辉,曾诚.BM 算法中函数 shift 的研究.计算机应用,2013, 33(8):2379–2382.

10 周燕,侯整风,何玲.基于有序二叉树的快速多模式字符串匹配算法.计算机工程,2010,36(17):42–44.

11 韩光辉,曾诚.Boyer-Moore 串匹配算法的改进.计算机应用,2014,34(3):865–868.

12 莱维汀,潘彦.算法设计与分析基础.第 3 版.北京:清华大学出版社,2015.

13 Cormen TH, Leiserson CE, Rivest RL, Stein C, 殷建平,徐云等,译.算法导论.第 3 版.北京:机械工业出版社,2013.