文章编号:1001 - 9081(2007)06 - 1415 - 03

改进的多模式字符串匹配算法

蔡晓妍,戴冠中,杨黎斌
(西北工业大学 自动化学院,陕西 西安 710072)
(caixiaoyan1982@hotmail.com)

摘 要:在经典的 AC 多模式字符串匹配算法的基础上,结合 BMH 算法的优点,提出了一种快速的多模式字符串匹配算法。一般情况下,该算法不需要匹配目标文本串中的每个字符,而是在实际比较之前跳过尽可能多的字符,以减少字符比较的操作,实现快速匹配。在模式串较长和较短的情况下,算法都有很好的性能。实验表明,在模式串较短时,本算法所需的时间仅为 AC 算法的 50% ~ 30%;在模式串较长时,所需时间为 AC 算法的 26.7% ~ 15.2%。

关键词:字符串匹配;AC 算法;BMH 算法;多模式匹配;算法复杂度
中图分类号: TP18 **文献标识码**:A

Improved multiple patterns string matching algorithm

CAI Xiao-yan, DAI Guan-zhong, YANG Li-bin
(*College of Automation, Northwestern Polytechnical University, Xi' an Shaanxi* 710072, *China*)

Abstract: Combined with the advantage of the Boyer-Moore-Hoospool (BMH) algorithm, a faster algorithm for performing multiple patterns matching in a string was proposed on the basis of Aho-Corasick (AC) algorithm. In general, it does not need to inspect every character of the string. It skips as many characters as possible to decrease pattern match operations before matching patterns. The proposed algorithm achieves excellent performance in the cases of both short patterns and long patterns. Experimental results show that in case of short patterns the time it takes for the proposed algorithm to search a string is only 50% ~30% that of the AC algorithm, while in case of long patterns the ratio is 26.7% ~15.2%.

Key words: string matching; AC algorithm; BMH algorithm; multiple patterns matching; computational complexity

0 引言

多模式匹配技术在很多领域都有重要的应用,如计算机病毒扫描、内容检索、内容过滤、网络入侵检测等,研究高效的多模式匹配算法具有非常重要的理论和现实意义[1]。所谓多模式匹配,就是在文本串 $T[1\cdots n]$ 中一次匹配多个模式串 $P_1, P_2, \cdots P_q$,其中 q 为模式串的个数,$q = 1$ 时,多模式匹配蜕化为单模式匹配,模式串 P_k 的长度是 m_k,即 $P_k[1\cdots m_k]$($1 \leqslant k \leqslant q$),$minlen$ 是最短模式串的长度,即 $minlen = \min\{m_k \mid (1 \leqslant k \leqslant q)\}$。多模式匹配比多个模式串逐个进行普通的单模式匹配的速度要快许多。

经典的单模式匹配算法是 KMP (Knuth-Morris-Patt)算法[2]和 BM (Boyer-Moore)算法[3]。KMP 算法实现了无回溯匹配,字符串中的每个字符只匹配一次,时间复杂度为 $O(n + m)$; BM 算法采用跳跃方式,匹配时跳过不需匹配的字符,最优情况下的时间复杂度为 $O(n/m)$, 平均情况下也大大优于 KMP 算法;BMH(Boyer-Moore-Horspool)算法[4]则是 BM 算法的简化实现,由于它只使用了坏字符规则,有效地减少了字符比较次数,因此在实际应用中效率比 BM 算法高。

AC 算法[5]是同时搜索多个模式的经典算法,该算法将待匹配的多个字符串转换为树型有限自动机,然后只需对文本串进行一次扫描就可找出所有模式串,其时间复杂度为 $O(n)$。文献[6]把 AC 和 BM 算法相结合得到的 FS 算法,在平均情况下比 AC 算法速度更快。

本文提出了改进的多模式匹配算法,在 AC 算法的基础上,吸收 BMH 算法的思想,只使用坏字符跳转,并充分利用匹配过程中匹配失败的信息,以期在每一次跳跃中跳跃尽量大的距离,实现了更快的匹配过程。

1 BMH 算法和 AC 算法

1.1 BMH 算法[4]

BMH 算法是 BM 算法的简化实现。算法将失配与文本指针的右移量独立,文本指针右移时并不关心正文中哪个字符造成了失配,而是简单地使用正文中与模式最右端字符对齐的那个字符来决定右移量,即只使用坏字符移动数组。预处理阶段的时间复杂度是 $O(m + n)$,空间复杂度为 $O(n)$,搜索阶段的时间复杂度为 $O(mn)$。

1.2 AC 算法[5]

AC 算法把所有的关键字合并到一个集合中,利用有限自动机的原理同时接受集合中的所有串。由于自动机是结构化的,因此每个前缀都可用唯一的状态来标识,甚至是那些多个模式的前缀。当文本中的下一个字符不是模式期待接受的下一个字符时,转到一个模式的最长前缀所代表的那个状态,这个状态也就是当前状态的相应后缀。AC 算法模式匹配的时

收稿日期:2006 - 12 - 29;**修订日期**:2007 - 03 - 16
基金项目:国防基础科研项目(C2720061361);国家 863 计划项目(2005AA147030)
作者简介:蔡晓妍(1982 -),女,山东淄博人,博士研究生,主要研究方向:智能信息处理、网络与信息安全; 戴冠中(1937 -),男,上海人,教授,博士生导师,主要研究领域:自动控制、信息安全; 杨黎斌(1981 -),男,江西吉安人, 博士研究生,主要研究方向:网络与信息安全、嵌入式系统.

间复杂度为 $O(n)$，包括预处理时间在内，AC 算法总时间复杂度是 $O(M+n)$，其中 M 为所有模式串的长度总和。

2 改进的多模式匹配算法

进一步提高模式匹配算法效率的主要途径是利用模式串匹配失败时可以获取的信息以进一步增大跳跃距离。受文献[6]所提到想法的启发，我们综合了 AC 算法和 BMH 算法的优点，提出了一种改进的多模式匹配算法。该算法的核心思想是：首先在文本串中查找任一模式串的尾字符，若不匹配则文本指针一直向右移动。即在找到任一模式串尾字符的情况下，再从右向左比较模式串中的其他字符，此后无论相等与否都将使文本指针向右移动该尾字符所对应的偏移个位置。算法分为两个阶段：模式集的预处理阶段和文本的匹配阶段。

2.1 预处理阶段

预处理阶段首先把模式串集合转换成反向有限自动机，根据模式集合中字符的分布，得到 *goto* 函数和 *output* 函数；然后构造 $skip_1$ 和 $skip_2$ 函数。

$skip_1(char)$ 的值为 *char* 在任一模式串中最右出现的位置到该模式串尾的距离，但是 $skip_1$ 的值不能超过 *minlen*，否则最短的模式串有可能被漏掉。$skip_1(char)$ 定义如下：

$$skip_1(char)=\begin{cases}\min\{m_k-j \mid P_k[j]=char, m_k-minlen\leqslant j<m_k, 1\leqslant k\leqslant q\}, & char\text{ 出现在}\{P\}\text{ 中}\\ minlen, & char\text{ 不出现在}\{P\}\text{ 中}\end{cases}$$

$skip_1(char)$ 函数值分两步计算：首先对字符集中的每个字符 *char*，令 $skip_1(char)=minlen$，然后，对模式串 P_k 中的字符 $P_k[j](0\leqslant j\leqslant m_k-1)$，令 $skip_1(P_k[j])=\min\{skip_1(P_k[j]), m_k-j-1\}$。只有当 $char=P_k[m_k-1]$ 时，$skip_2(char)$ 函数才起作用，$skip_2(char)=skip_1(P_k[m_k-1])$。

2.2 匹配阶段

本文提出的算法是从模式串集右端向左逆向匹配。匹配开始前，首先令 $skip_1(P_k[m_k-1])=0$。在匹配过程中，匹配从 $s=0, i=minlen-1, T[i]$ 开始，匹配任一模式串的尾字符，若匹配失败，则文本指针右移 $skip_1[T[i]]$，即 $i=i+skip_1[T[i]]$，直到找到某一模式串的尾字符，然后根据 *goto* 函数向左进行匹配，若匹配成功，则令 $j=i, j=j-1$，匹配下一个 $T[j]$；若匹配失败，则文本指针右移 $skip_2[T[i]]$，即 $i=i+skip_2[T[i]]$，重新开始一次匹配。在匹配过程中，如果成功匹配到某个模式串，则输出模式串在文本中的位置。以上述方式处理文本至文本末尾，可以找出所有模式的出现位置。整个匹配算法可描述如下：

算法 改进的多模式匹配算法

```
输入:goto(),output(),skip1(),skip2(),T;
输出:模式串在 T 中出现的位置
i = minlen - 1;
for(k = 0;k < q;k ++)
{
 skip1(pk[mk - 1]) = 0;
}
while(i < n)
{
 s = 0;
 d = skip1[T[i]];
 while(d ≠ 0)
 {
  i = i + d;
  d = skip1[T[i]];
 }
 j = i;
 s = goto(s,T[j]);
 while(s ≠ fail)
 {
  if (output(s) ≠ NULL)
    找到模式的一个匹配,输出;
  else
  {
   j = j - 1;
   s = goto(s,T[j]);
  }
 }
 i = i + skip2(T[i]);
}
```

2.3 示例

我们采用文献[7]中各匹配算法所使用的文本串 T = “gcabcgcagagagbabaca”，并选取含有 3 个模式串的模式串集合 {P} = {gca, gacb, gagag}，用本文算法在文本串 T 中查找模式串集合 P 中的任一模式串。

模式串集合转换成的反向有限自动机如图 1 所示，*minlen* = 3。图中的箭头及箭头上的字符表示 *goto* 函数，如果 $goto(s,char)$ 函数不存在，则指定 $goto(s,char)=fail$。图中双圈状态表示有输出；输出状态所对应的模式串用方框表示。$skip_1$ 和 $skip_2$ 的值如表 1 所示，匹配过程如表 2 所示，表中每一行表示从状态 0 开始，按照 *goto* 函数从后向前比较，直到 $goto(s,T[j])=fail$。

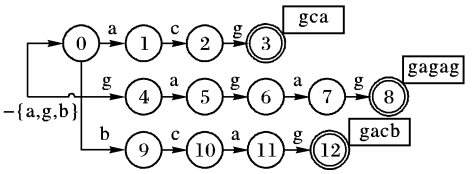

图 1 自动机的 *goto* 函数和 *output* 函数

表 1 $skip_1$ 与 $skip_2$ 的值

char	a	c	g	其他
$skip1(char)$	0	1	0	3
$skip2(char)$	1	–	2	–

表 2 匹配过程

字符比较	i	$T[i]$	$skip_1$	$skip_2$	输出
gcabcgcagagagbabaca	2	a	0	1	gca
gcabcgcagagagbabaca	3	b	3	–	–
gcabcgcagagagbabaca	6	c	1	–	–
gcabcgcagagagbabaca	7	a	0	1	gca
gcabcgcagagagbabaca	8	g	0	2	–
gcabcgcagagagbabaca	10	g	0	2	–
gcabcgcagagagbabaca	12	g	0	2	gagag
gcabcgcagagagbabaca	14	a	0	1	–
gcabcgcagagagbabaca	15	b	3	–	–
gcabcgcagagagbabaca	18	a	0	1	–

3　算法时间复杂度分析

设字母表的大小为 σ,所有模式串的模式长度之和为 $\sum_{k=1}^{q} m_k$。模式集合中,模式串的最小长度为 $minlen$,最大长度为 $maxlen$。

3.1　预处理阶段的时间复杂度

根据 AC 算法,构造 *goto* 函数的时间复杂度为 $O(\sum_{k=1}^{q} m_k)$。由于 $skip_1$ 的计算分两步,第一步的计算量与 σ 成线性关系,第二步的计算量与 $\sum_{k=1}^{q} m_k$ 成线性关系,所以 $skip_1$ 的时间复杂度为 $O(\sigma + \sum_{k=1}^{q} m_k)$。因此,在预处理阶段,本文算法的时间复杂度为 $O(\sigma + \sum_{k=1}^{q} m_k)$。

3.2　匹配阶段的时间复杂度

匹配阶段中,最优情况下的时间复杂度为 $O(n/minlen)$,最坏情况下的时间复杂度为 $O(n \cdot maxlen)$。平均情况下,时间复杂度的分析比较复杂,并且与字符的出现概率有关,需要通过概率模型进行计算。文献[3]中提出,平均情况下算法的性能可通过发现不匹配字符所花的代价和发现该不匹配后能够移动的距离之比的概率平均值来考察。根据上述考察方法,本文提出的算法平均性能如下:

$$P = \frac{\sum_{j=0}^{maxlen} prob(j) \cdot cost_1(j)}{skip_1(j)} + \frac{\sum_{j=0}^{maxlen} (1 - prob(j)) \cdot cost_2(j)}{skip_2(j)} \tag{2}$$

上式等号右边第一部分说明实际模式比较前文本的跳跃情况;$prob(j)$ 为 $skip_1$ 函数中不为 0 的字符(即不是模式串的尾字符)出现的概率;$cost_1(j)$ 为查找 $skip_1$ 函数所花的代价;$skip_1(j)$ 为发现该字符不匹配后平均能够跳过的字符,且 $skip_1(j) = \sum_{k=1}^{minlen} k \cdot P_{skip_1}(j,k)$,$P_{skip_1}(j,k)$ 为查找到文本第 j 个字符时,发现不匹配后能够跳过 k 个字符的概率。

式(2)等号右边的第二部分表示当第 j 个字符为模式串的尾字符时,进行模式比较后文本的跳跃情况;$1 - prob(j)$ 为模式串尾字符出现的概率,$cost_2(j)$ 为模式比较过程中发现不匹配字符所花的代价,$skip_2(j)$ 是在查找到第 $j+1$ 个字符时发现不匹配后,平均能够跳过的字符,且:

$$cost_2(j) = \sum_{j=0}^{maxlen} P(j) \cdot cost(j)$$

$$skip_2(j) = \sum_{j=0}^{maxlen} P(j) \cdot (\sum_{k=0}^{minlen} k \cdot P_{skip_2}(j,k))$$

$cost(j)$ 为查找 $j+1$ 个字符所花的代价,$P(j)$ 为查找到第 $(j+1)$ 个字符时才发现不匹配的概率。

类似,FS 算法的平均性能可通过下式考察:

$$P' = \frac{\sum_{j=0}^{maxlen} P(j) \cdot cost(j)}{\sum_{j=0}^{maxlen} P(j) \cdot (\sum_{k=0}^{minlen} k \cdot P_{skip_2}(j,k))} = \frac{cost_2(j)}{skip_2(j)} \tag{3}$$

由于 $cost_1(j) \ll cost_2(j)$,而 $skip_1(j) > skip_2(j)$,因此,当 $prob(j) > 0$ 时,(2)式的值必小于(3)式的值。所以本文算法比 FS 算法有更好的平均性能。

4　实验结果及分析

为测试本文提出算法的性能,并与 AC 算法和 FS 算法进行比较,选取 3.58Mbit 的英文文本,在其中查找多个模式串,找出所有出现的位置。从文本中选取如下 3 组不同长度的模式串。为测试在不同模式串长度下算法的性能,这 3 组模式串中每组内模式串的长度是一致的。在实验中,每组的模式串数目从 1 到 19 以 3 为单位递增。测试环境为 Pentium 2.8 GHz、512MB 内存,WindowsXP。编译环境为 Microsoft Visual C++ 6.0。测试结果如表 3 所示(1 为模式串长)。

表 3　查找时间(单位:ms)

	算法	模式串数目						
		1	4	7	10	13	16	19
$l=2$	AC	692	723	741	765	778	794	811
	FS	269	321	368	385	403	428	457
	本文算法	226	278	337	379	391	415	432
$l=9$	AC	701	746	779	792	810	829	845
	FS	265	297	306	312	343	372	387
	本文算法	218	251	286	301	338	369	381
$l=16$	AC	716	741	780	789	823	852	889
	FS	204	256	260	268	275	291	320
	本文算法	179	198	235	259	267	283	312

从上面的测试数据可以看出,本文算法和 FS 算法在模式串较长且模式串数目较少时,都比 AC 算法有更好的性能。与 FS 算法相比,本文算法在模式串较短、模式串数目较少时,性能相对更好。因为当模式串较短、模式串数目较小时,模式串集合中出现的字符较少,而 $skip_1$ 值较大的概率也会增加,本文的算法比 FS 算法性能优越;反之,当模式数目较多时,模式结尾字符出现的概率较多,模式比较之前跳过的字符较少,本文算法优势不明显。

5　结语

本文把单模式匹配算法中的 BMH 算法与经典的多模式匹配算法 AC 算法相结合,提出了一个快速的多模式匹配算法。该算法在匹配过程中能够尽可能多地跳过待查文本串字符,并且在模式串较长和较短的情况下,均有相当好的性能。由于查找问题的普遍性,故该方法具有广阔的应用前景。

参考文献:

[1] KURI J, NAVARRO G, ME L. Fast Multipattern Search Algorithms for Intrusion Detection[J]. Fundamenta Informaticae, 2003, 56(1-2): 23-49.

[2] KNUTH D, MORRIS J, PRATT V. Fast pattern matching in strings [J]. SIAM Journal on Computing, 1977, 6(1): 323-350.

[3] BOYER RS, MOORE JS. A fast string searching algorithm [J]. Communications of ACM, 1977, 20(10): 762-772.

[4] NIGEL HR. Practical fast searching in strings[J]. Software Practice and Experience, 1980, 10(6): 501-506.

[5] AHO AV, CORASICK MJ. Efficient string matching: an aid to bibliographic search[J]. Communications of ACM, 1975, 18(6): 333-340.

[6] FAN JJ, SU KH. An Efficient Algorithm for Matching Multiple Patterns[J]. IEEE Transaction on Knowledge and Data Engineering, 1993, 5(2): 339-351.

[7] CHARRAS C, LECROQ TT. Handbook of Exact String Matching Algorithms[M]. London: King's College London Publicatons, 2004.