一种改进的 BMH 模式匹配算法*)

刘胜飞[1] 张云泉[1,2]
(中国科学院软件研究所并行计算实验室 北京 100080)[1]
(中国科学院软件研究所计算机科学国家重点实验室 北京 100080)[2]

摘 要 分析了目前网络上最流行的 BM 算法及其改进算法 BMH,在此基础上提出了 BMH 算法的改进算法 BMH2。考虑了模式串自身的特征,在原有移动距离数组的基础上增加一个新的移动数组,从而充分利用模式串特征进行更大距离的移动,使算法获得更高的效率。实验证明,改进后的算法能够增加"坏字符"方法的右移量,有效地提高匹配速率。

关键词 模式匹配,BM 算法,BMH 算法,时间复杂度

Improved Pattern Matching Algorithm of BMH

LIU Sheng-fei[1] ZHANG Yun-quan[1,2]
(Lab of Parallel Computing, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China)[1]
(State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China)[2]

Abstract Based on the discussion of the most popular BM and BMH algorithms for pattern matching on the network, an improved algorithm BMH2 was presented. Considering the feature of pattern itself, a new skip distance array was added based on the old one. It can increase the skip distance by known information. Experimental data shows that the improved algorithm can increase the right shift of "bad characters" and enhance the matching speed effectively.

Keywords Pattern matching, BM algorithm, BMH algorithm, Time complexity

1 引言

模式匹配是字符串的基本运算之一。在给定的主串 $text=t_1t_2t_3\cdots t_n$ 中查找模式串 $pattern=p_1p_2p_3\cdots p_m$,如果 *text* 中存在等于 *pattern* 的子串,则匹配成功,返回模式串在主串中出现的第一个位置,否则称为匹配失败。模式匹配在很多领域都发挥着重要的作用,比如在网络搜索、入侵检测、情报检索、DNA 序列检索等各个方面都有着重要的应用。

目前关于模式匹配的算法很多,其中最著名的两个是 KMP 算法和 BM 算法。两个算法在最坏情况下均具有线性的搜索时间。但在实际使用中,KMP 算法并不比最简单的 C 语言库函数 *strstr*()快多少,而 BM 算法则往往比 KMP 算法快上 3~5 倍[1]。在 BM 的发展过程中,Horspool 发表了改进与简化 BM 算法的论文,即 BMH 算法。在一般情况下,BMH 算法比 BM 算法有更好的匹配性能[2]。

2 Boyer_Moore 算法

1977 年,Boyer 和 Moore 提出了著名的 Boyer_Moore (BM)模式匹配算法[3]。下面简要说明 BM 算法的基本原理。假设有长度为 n 的文本字符串 $text[n]$ 和长度为 m 的模式字符串 $pattern[m]$($m\leqslant n$)。BM 算法是从右向左地把模式串同文本做比较。为了确定匹配失效时扫描文本的指针向前移动的距离,BM 算法同时采用"坏字符"和"好尾缀"的启发式策略进行计算,取其大者作为文本指针的移动距离。

"坏字符"思想:定义 *skip* 数组,如果字符 *ch* 没有在 *pattern* 中出现过,则 $skip[ch]=m$;如果字符 *ch* 在 *pattern* 出现过,记 *e* 表示 *ch* 在 *pattern* 中出现的最后位置(下标从 0 开始),则 $skip[ch]=m-e-1$。从右向左地把模式串同文本做比较,设匹配进行到比较 $text[k-m+1\cdots k]$ 和 $pattern[0\cdots m-1]$,从右至左依次检查 $text[k]$, $text[k-1]$ … $text[k-m+1]$。当匹配失败发生在 $text[i]$! $=pattern[j]$($i\in[k-m+1\cdots k]$, $j\in[0\cdots m-1]$)时,采用 $skip[text[i]]$ 表示扫描文本的指针向前移动的位数。"坏字符"思想实际上是将 $text[i]$ 在 *pattern* 中最后一次出现的位置与文本中的 $text[i]$ 对齐后开始新的一轮匹配。

"好尾缀"思想:若 *pattern* 后部与 *text* 一致的部分之中,有一部分在 *pattern* 中其它地方出现,则可以将 *pattern* 向右移动,直接使这部分对齐,且要求这一部分尽可能大。其基本算法和 KMP 中的 *Next* 函数比较相似[4]。

BM 算法为了确定在不匹配的情况下最大的可移位距离而使用了两种启发,即"坏字符"启发和"好后缀"启发,两种启发均能导致最大为 m 的移动距离。但是,由于"好后缀"启发的预处理和计算过程都比较复杂,Horspool 于 1980 年发表了改进与简化 BM 算法的论文,即 Boyer_Moore_Horspoo (BMH)算法[2]。BMH 算法在移动模式时仅考虑了"坏字符"策略。它首先比较文本指针所指字符和模式串的最后一个字

*)本文工作受到国家自然科学基金(No. 60303020),国家自然基金重点项目(No. 60533020),国家重点基础研究发展计划(No. 2005CB321702),国家 863(No. 2006AA01A102,No. 2006AA01A125)和北邮网络与交换国家重点实验室开放基金的部分资助。**刘胜飞** 硕士研究生,研究方向为并行计算模型;**张云泉** 博士,研究员,博士生导师,研究方向为高性能计算及并行数值软件、并行计算模型、并行数据库、海量数据并行处理。

符，如果相等再比较其余 $m-1$ 个字符。无论文本中哪个字符造成了匹配失败，都将由文本中和模式串最后一个位置对应的字符来启发模式向右的移动。即当匹配开始比较 $text[k-m+1\cdots k]$ 和 $pattern[0\cdots m-1]$ 时，从右至左依次检查 $text[k]$，$text[k-1]$ … $text[k-m+1]$，一旦发现不匹配，就将文本指针重新赋值为 $k+skip[text[k]]$（k 是一个中间变量，表示文本中每次从右至左开始比较的起始位置）。

关于“坏字符”启发和“好尾缀”启发的对比，孙克雷在文献[5]中的研究表明：“坏字符”启发在匹配过程中占主导地位的概率为 94.03%，远远高于“好尾缀”启发。在一般情况下，BMH 算法比 BM 有更好的性能，它简化了初始化过程，省去了计算“好尾缀”启发的移动距离，并省去了比较“坏字符”和“好尾缀”的过程。

BMH 算法的具体描述如下：

```
Algorithm BMH
for(i=0;i<MAX_LEN;i++) //skip[ch]表示 pattern 中最后出现的 ch 到 pattern 末尾的距离
    skip[i]=m; //当 ch 没有在 pattern 中出现时，skip[ch]等于模式串长度 m
for(i=0;i<m-1;i++)//不考虑 pattern[m-1]，否则 skip[pattern[m-1]]=0，模式串将停止不前
    skip[pattern[i]]=m-i-1;
i=m-1;
while(i<n) {
    k=i;j=m-1; //k 记录 text 中每次从右至左开始比较的起始位置
    while((j>=0)&&(pattern[j]==text[i])){
        i--;j--;
    }
    if(j==-1) return i+1; //在 text[i+1]处匹配成功
    i=k+skip[text[k]]; //采用“坏字符”策略移动文本指针
}
return -1;//匹配失败
```

根据 BMH 算法，以 *Text*=“*abhdgfdabbdbdabdbfd*”，*Pattern*=“*abdbfd*”子串为例，开始匹配时，把模式串与正文自左边对齐，匹配过程如表 1 所示。

表 1　BHM 算法匹配过程

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
文本	a	b	h	d	g	f	d	a	b	b	d	b	d	a	b	d	b	f	d	文本指针移动距离
第 1 次	a	b	d	b	f	d														skip[‘f’]=1
第 2 次		a	b	d	b	f	d													skip[‘d’]=3
第 3 次				a	b	d	b	f	d											skip[‘b’]=2
第 4 次					a	b	d	b	f	d										skip[‘b’]=2
第 5 次						a	b	d	b	f	d									skip[‘a’]=5
第 6 次														a	b	d	b	f	d	

上例使用 BMH 算法经过 6 次循环后匹配成功。当字符集元素个数为 s 时，BMH 算法预处理时间复杂度为 $O(m+s)$，空间复杂度为 $O(s)$。搜索阶段在最坏情况下的时间复杂度为 $O(mn)$，最好情况下的性能是 $O(n/m)$。

在模式匹配中存在两个基本定理：任何字符串匹配算法在最坏情况下必须检查至少 $n-m+1$ 个文本中的字符；任何字符串匹配算法至少检查 n/m 个字符[6]。因此，没有一个算法比 BM 算法有更好的计算复杂度。但是我们可以通过改进来减少比较次数，提高匹配的平均性能。

3　改进后的 BMH2 算法

通过观察模式串，我们注意到：

(1)字符集中的大部分字符在模式串中出现的次数为 0 或 1；

(2)当某字符在模式串中多次出现时，相同字符的间距一般大于 2。

设字符集合为 Σ，以 *pattern*=“*string search*”为例。Σ 中的字符在 *pattern* 中出现一次的字符集合 $A1=\{$‘*t*’, ‘*i*’, ‘*n*’, ‘*g*’, ‘\0’ ‘*e*’, ‘*a*’, ‘*c*’, ‘*h*’$\}$，出现两次或两次以上的字符集合 $A2=\{$‘*s*’, ‘*r*’$\}$，出现 0 次的字符集合 $A0=\Sigma-A1-A2$。若以 $text[k]$ 进行启发，BMH 算法实际上是将 *pattern* 中最后一次出现的 $text[k]$ 与文本中的 $text[k]$ 对齐后重新匹配。因此当 $text[k]\in A0$ 时，扫描文本的指针可以向前移动最大距离 m(模式长度)。本文的基本出发点是让文本指针能够以更高的概率向前移动最大距离 m。假设能够将 *pattern* 中倒数第二次出现的 $text[k]$ 与文本中的 $text[k]$ 对齐后开始新的一轮匹配，则当 $text[k]\in(A0\cup A1)$ 时，文本指针都可以向前移动最大距离 m，而当 $text[k]\in A2$ 时，文本指针的移动距离也可以得到提高。比如模式串中‘*s*’的间距和‘*r*’的间距分别为 7 和 8，相应地，文本指针的移动距离将分别增加 7 和 8。

为此，增加一个 *newSkip* 数组，如果字符 *ch* 在模式串 *pattern* 中出现的次数为 0 或 1，则 $newSkip[ch]=m$；如果字符 *ch* 在 *pattern* 中出现的大于等于 2，记 f 表示 *ch* 在 *pattern* 中倒数第二次出现的位置(下标从 0 开始)，则 $newSkip[ch]=m-f-1$。另外，定义 *preChar* 数组，如果字符 *ch* 在模式串 *pattern* 中最后出现在 $pattern[e]$，则 $preChar[ch]=pattern[e-1]$；如果字符 *ch* 没有在 *pattern* 中出现过，则 $preChar[ch]=-1$。当 $pattern[0]$ 在模式串中仅出现一次时，由于 $pattern[0]$ 前面没有字符，因此将 $newSkip[pattern[0]]$ 单独赋值为 $m-1$。*newSkip* 数组和 *preChar* 数组的长度与 *skip* 数组相同，均为字符集中元素的个数。若为 ASCII 码，则长度为 256。

当匹配开始比较 $text[k-m+1\cdots k]$ 和 $pattern[0\cdots m-1]$ 时，从右至左依次检查 $text[k]\cdots text[k-m+1]$。如果发现不匹配，则比较 $text[k-1]$ 和 $preChar[text[k]]$。当 $text[k-1]!=preChar[text[k]]$ 时，将文本指针重新赋值为 $k+newSkip[text[k]]$；否则将文本指针重新赋值为 $k+skip[text[k]]$。事实上，当 $text[k]$ 没有出现在模式串中时，$preChar[text[k]]$ 可以初始化为任何值。因为此时 $skip[text[k]]$ 和 $newSkip[text[k]]$ 的值均为 m，无论 $text[k-1]$ 和 $preChar[text[k]]$ 的值是否相等，文本指针都将重新赋值为 $k+m$。

BMH2 算法描述如下：

```
Algorithm BMH2
for(i=0;i<MAX_LEN;i++){
    skip[i]=m; newSkip[i]=m; preChar[i]=-1; //此处 preChar[i]可以初始化为任何值
}
skip[pattern[0]]=newSkip[pattern[0]]=m-1; //单独处理 pattern[0]的情况
for(i=1;i<m-1;i++) {
    ch=pattern[i]; //newSkip[ch]表示 pattern 中倒数第二次出现的 ch 到 pattern 末尾的距离
```

(下转第 173 页)

[11] Weiss G M. Mining with Rarity - Problems and Solutions: A Unifying Framework [C]. SIGKDD Explorations, 2004, 6(1): 7-19

[12] 肖健华，吴今培. 样本数目不对称时的 SVM 模型[J]. 计算机科学，2003，30(2)：165 - 167

[13] Lin C F, Wang S D. Fuzzy support vector machines [J]. IEEE Transactions on Neural Networks, 2002, 13(2): 464-471

[14] Blake C L, Merz C J. UCI Repository of Machine Learning Database[DB/OL]. 1998. [2007-5-21]. http://www.ics.uci.edu/~mlearn/MLRepository.html

(上接第 165 页)

```
    newSkip[ch]=skip[ch]; //当 ch 在 pattern 中出现 0 次或 1 次时,newSkip[ch]等于模式串长度 m
    skip[ch]=m-i-1;// skip 数组的定义与 BMH 算法相同
    preChar[ch]=pattern[i-1];
}
i=m-1;
while(i<n) {
    k=i; j=m-1; //k 记录 text 中每次从右至左开始比较的起始位置
    while((j>=0)&&(pattern[j]==text[i])){
        i--; j--;
    }
    if(j==-1) return i+1; //在 text[i+1]处匹配成功
    if(text[k-1]! =preChar[text[k]])
        i=k+newSkip[text[k]]; //采用改进后的策略移动文本指针
    else
    i=k+skip[text[k]];
}
return -1; //匹配失败
```

根据 BMH2 算法，上文中的文本与模式串的匹配过程如表 2 所示。

表 2　改进算法匹配过程

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
文本	a	b	h	d	g	f	d	a	b	b	d	b	d	a	b	d	b	f	d	文本指针移动距离
第1次	a	b	d	b	d															newSkip['f']=6
第2次								a	b	d	b	f	d							skip['b']=2
第3次									a	b	d	b	f	d						newSkip['a']=5
第4次															a	b	d	b	f	d

由表 2 可以看出，只需进行 4 次循环就匹配成功。BMH2 算法通过提高模式串的平均移动距离，获得了更高的匹配效率。当模式串中没有相同字符或者相同字符的间距较大时，BMH2 算法可以取得更好的匹配效率。

4　测试结果及结论

实验环境采用曙光服务器作为硬件测试平台，其中 CPU 为 AMD Opteron 2.0GHz(双核)，内存为 1.5GB，操作系统采用 Linux 2.6.9 内核，编译器为 pgcc7.0。测试分别使用 BMH 算法和 BMH2 算法进行模式匹配，比较它们的实际效率。每个算法分别执行 1000 次，运行时间取平均值。

测试 1：选取 1M 的纯英文文本，并采用不同长度的英文短语作为模式串，得到的测试数据如表 3 所示。

表 3　纯英文文本的匹配结果对比

	模式串平均每次移动距离(位/次)		检索速率(millions chars/s)		
模式串长度	BMH 算法	BMH2 算法	BMH 算法	BMH2 算法	加速比
5	4.61	4.95	424	431	1.017
6	5.51	5.99	386	422	1.093
7	5.59	6.84	488	535	1.096
8	6.27	7.89	535	602	1.125
9	7.89	8.64	476	552	1.160
10	7.60	9.66	574	694	1.209
15	8.29	12.63	617	806	1.306
20	9.89	14.66	714	870	1.218
25	9.33	17.85	641	847	1.321

注 1：加速比＝BMH2 算法的检索速率/BMH 算法的检索速率。

测试 2：采用 NCBI(美国国家生物情报中心)发布的蛋白质序列文件(4.0M)作为文本，并从中选取不同的氨基酸序列作为模式串，得到的测试数据如表 4 所示。

表 4　蛋白质序列文件的匹配结果对比

	模式串平均每次移动距离(位/次)		检索速率(millions chars/s)		
模式串长度	BMH 算法	BMH2 算法	BMH 算法	BMH2 算法	加速比
5	4.78	4.97	433	446	1.030
10	7.64	9.94	602	676	1.123
15	11.25	14.72	775	862	1.112
20	13.25	19.35	840	935	1.113
25	15.87	22.06	893	971	1.087

在测试 1 中，当模式串长度小于等于 7 时，改进效果不太明显。这是由于模式串较短，任意字符 *ch* 在 *pattern* 中出现的概率都很小，改进前后的平均移动距离都很接近模式串长度，因此检索速率差不多。当模式串长度大于 7 时，BMH2 的模式串平均移动距离明显高于 BMH 算法，因此能够获得更高的匹配效率。从表 3 可以看出，对纯英文文本的检索，BMH2 算法的匹配速率比 BMH 算法平均提高了 15%到 30%。

在测试 2 中，BMH2 算法匹配的加速效果低于纯英文文本的加速效果。根据 BMH2 的算法思想，当模式串中没有相同字符或者相同字符的间距较大时，BMH2 算法可以取得较好的匹配效率。由于组成蛋白质的常见氨基酸只有 20 种，每个氨基酸的缩写为一个字符。氨基酸序列中出现相同字符的概率增加，相同字符间的平均距离减小，使得 BMH2 算法的加速效果减弱。从表 4 可以看出，对蛋白质序列文件的检索，BMH2 算法的匹配速率相比 BMH 算法平均提高了 10%左右。

结束语　模式匹配是当前入侵检测和情报检索系统中普遍采用的策略之一。随着网络带宽的不断增加，网络服务和应用对检索效率要求越来越高。本文对经典的模式匹配算法 BM 以其改进算法 BMH 做了简要的分析，并针对 BMH 算法提出了改进。若 BMH2 算法应用到入侵检测和情报检索等领域，将能够有效提高系统的检索效率。本文的下一步工作可以考虑将 BMH2 思想和其它模式匹配技术相结合，如将 BMH2 思想应用到多模式匹配算法中，使其实际性能更优。

参 考 文 献

[1] 闵联营，赵婷婷. BM 算法的研究与改进[J]. 武汉理工大学学报，2006，30(3)：528-530

[2] Horspool N R. Practical Fast Searching in Strings [J]. Software Practice and Experience, 1980, 10(6): 501-506

[3] Boyer R S, Moore J S. A Fast String Searching Algorithm [J]. Communications of the ACM, 1977, 20(10): 762-772

[4] 巫喜红，凌捷. 单模式匹配算法研究[J]. 网络与通信，2006，22(8)：202-204

[5] 孙克雷. IDS 中一种快速模式匹配算法[J]. 安徽理工大学学报，2006，26(3)：52-55

[6] 李雪莹，刘宝旭，等. 字符串匹配技术研究[J]. 计算机工程，2004，30(22)：24-26