

太原科技大学

硕士学位论文

对串匹配技术中的Wu-Manber算法的研究

姓名：莫德敏

申请学位级别：硕士

专业：计算机软件与理论

指导教师：刘耀军

20080701

对串匹配技术中的 Wu-Manber 算法的研究

中 文 摘 要

字符串匹配是计算机研究领域中的一个古老、经典而且被广泛研究的课题，是信息检索领域和计算机生物学领域等的关键技术之一。在当今的互联网时代，对匹配算法的需求日新月异，对处理的实时性的需求越来越高。这些都对原有的字符串匹配技术提出了新的挑战，有必要对原字符串匹配技术进行改进和优化。

本文主要是对高效的多字符串匹配算法——Wu-Manber 算法进行研究。本文针对 Wu-Manber 算法的散列表 HASH 的表项：HASH（）链表的不足，提出了基于非空公共子后缀模式的处理算法；而后综合前人对 Wu-Manber 算法的 SHIFT 表的改进，提出更加高效的字符串匹配算法；进而提出一种在大规模串匹配时对 Wu-Manber 算法的改进。

本文的研究成果具体如下：

1. 基于非空公共子后缀的 Wu-Manber 算法的改进：针对 Wu-Manber 算法在处理公共子后缀模式情况下的不足，本文提出了一种基于非空公共子后缀模式的处理算法。该算法把同一 next 链表中有非空公共子后缀的模式汇集在一起，进一步减小了 next 链表的平均长度。在匹配过程中减少了字符比较的次数，从而提高算法的运行效率。本文对搜狗实验室给出的相关文档进行全文检索实验，并和原 Wu-Manber 算法、前人提出的改进算法进行比较。实验结果表明，本文提出的改进算法有效地减少了匹配过程中字符比较的次数，从而提高匹配的速度和效率。
2. 对 Wu-Manber 算法的综合改进：对 1 中提出的算法进行了改进，在对 next 链表进行分类的同时把含有非空公共子后缀的结点提到链表的前部；并整合了前人提出的“精确的不良字符转移和弱化的良好后缀转移的改进”方法。新改进的算法充分利用以上两种算法的优点，使匹配过程中字符比较的次数得到了进一步减少。新改进的 Wu-Manber 匹配算法在实验中取得了更高的效率，比原算法提高到 4.6% 以上。
3. Wu-Manber 算法在大规模模式串下的改进：对 1 提出的算法进行了改进，把原算法中 next 链表中结点的 Same_Suffix 域分裂成两个子域，使得在大规模模式串的情况下，搜索过程中字符比较的次数进一步减少，新算法

的效率比原算法有进一步的提高。实验结果表明，当模式串较少时，新算法效率与原算法相比有一定的损失。而随着模式串的增加，新算法具有更高的效率。因此，新算法比原算法具有更大的适用范围。

关键词：字符串匹配；精确字符串匹配；Wu-Manber；匹配算法；信息检索

Research on Wu-Manber Multiple String Matching Algorithm

Mo De Min(Computer Software and Theory)

Directed by Prof. Liu Yao Jun

ABSTRACT

String matching is an old classic and being widely studied research area in computer science. It's one of the crucial techniques in inform retrieve area and computer biology area. Nowadays the requirements of pattern algorithms change with each passing day at network era. The needs of real time process are growing. All of these mentioned above bring up new challenge to the original algorithms. It's necessary to make some improvements to the existing algorithms.

This thesis concentrates on the efficient multiple patterns algorithm——Wu-Manber algorithm. The thesis produce an algorithm based on ϵ -free subsuffix pattern; and then combined it with the existing technique to produce a much more efficient one; and finally produce an improved algorithm in large scale patterns.

The main results are listed below:

1. An improvement based on the ϵ -free subsuffix of patterns: Realizing there are rooms to improve when some patterns have common subsuffix, the paper produces an improved Wu-Manber multiple patterns matching algorithm, based on the ϵ -free subsuffix patterns. The algorithm collects patterns with common ϵ -free subsuffix and then reduces the amount of comparisons during the matching process. The paper's experiments are based on documents from <http://www.sogou.com/lab/>. Compare the new algorithm with the origin Wu-Manber algorithm and the algorithm introduced by^[36], the results are that the new algorithm can effectively reduce the amount of comparisons and then work more efficiently.

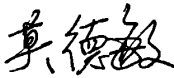
2. An improvement based on combination techniques: Produce a newly modified Wu-Manber multiple patterns matching algorithm which make an improvement of algorithm produced by 1. When we classify patterns by the ϵ -free subsuffix of the patterns in the next links , we drag those patterns having common ϵ -free subsuffix to the front of the links. And make full use of merits of algorithm produced by ^[35] and the technique mentioned above. Tests show that the new algorithm can further reduce the amount of words matching to at least 4.6 %.

3. An improvement algorithm in large scale patterns: Produces a modified Wu-Manber multiple patterns matching algorithm, based on the idea of the Wu-Manber algorithm. The algorithm replaces the Same_Subsuffix link used in 1 with two links: Left_Subsuffix and Right_Subsuffix. When it come to compare the characters in the Same_Subsuffix link, what we need to do is to compare them in one of the links mentioned above. And then reduce the amount of characters matched. The new algorithm works particularly well when the patterns scale is large.

Key words: Wu-Manber Algorithm; Multiple Pattern Matching; Pattern Matching; String Matching; Information Retrieval

承 诺 书

本人郑重声明：所呈交的学位论文，是在导师指导下独立完成的，学位论文的知识产权属于太原科技大学。如果今后以其他单位名义发表与在读期间学位论文相关的内容，将承担法律责任。除文中已经注明引用的文献资料外，本学位论文不包括任何其他个人或集体已经发表或撰写过的成果。

学位论文作者（签章）：
2008年6月3日

第一章 引言

1.1 研究字符串匹配技术的目的和意义

字符串匹配问题是计算机科学中最基本的问题之一，字符串匹配就是在给定文本中查找相关的字符序列。字符串匹配技术在当今的各个领域都有广泛的应用。这些领域有：信息检索、信息过滤、入侵检测及计算生物学等。社会、科学的发展内在要求与其发展相适应的字符串匹配技术。Wu-Manber 算法是字符串匹配算法中效率很高的一种，对其进行改进和优化很有必要。

1.1.1 字符串匹配技术的应用

在当今信息时代，计算机被广泛应用于各个领域、互联网正处于兴旺发展中，成为了我们学习、交流、生活和参政议政的重要场所。各种网络资源，如万维网中的新闻平台、电子邮件、BBS、博客、流媒体等给我们带来无限机会的同时，也让我们陷入遭遇恶意攻击、个人资源受到恶意破坏、公共机能随时有瘫痪的危险。怎样才能在网络中找到自己希望的知识和服务，同时让自己的个人资源免于遭受破坏和窃取？这是我们每个人乃至整个社会共同关注的问题。

网络信息正以指数级的速度增长，在这个信息大爆炸的时代，在无边无际的信息海洋中充满了各式各样的信息。许多黄色信息、恐怖主义者的言论及有关国家安全和机密等等都处于互联网之中。如何对这些信息进行监管、过滤是各个网络监管过滤系统的重要任务。而查找相关的信息就是字符串匹配技术大施拳脚之处。面对海量的信息和处理实时性的要求，字符串匹配算法效率的高低决定了这些网络监管过滤系统的性能的好坏。

入侵检测技术是为保证计算机系统的安全而设计与配置的一种能够及时发现并报告系统中未授权或异常现象的技术，是一种用于检测计算机网络中违反安全策略行为的技术。入侵检测的第一步是信息收集，内容包括系统、网络、数据及用户活动的状态和行为，通过分析这些信息来判断用户的活动是否是合法的。其采用的技术有特征收集和异常检测。特征检测（Signature-based detection）又称 Misuse detection，这一检测假设入侵者活动可以用一种模式来表示，系统的目标是检测主体活动是否符合这些模式。它可以将已有的入侵方法检查出来^[6]。在这

些入侵检测系统中,字符串匹配技术得到广泛的应用^{[2][3][4][1]},并成为影响系统性能的决定性因素^[5]。被广泛研究的入侵检测系统 Snort^[6]在字符串匹配方面主要采用 Boyer-Moore 字符串匹配算法,同时在 2.0 以后的版本中有 Wu-Manber 字符串匹配算法和 Aho-Croshik 字符串匹配算法可供选择^[7]。

病毒检测技术也是字符串匹配算法的重要应用领域之一。病毒检测就是在文件中查找病毒特征的过程。数据显示,2007 年上半年瑞星公司共截获新病毒 133717 个,其中木马病毒 83119 个,后门病毒 31204 个,两者之和超过 11 万,相当于去年同期截获的新病毒总和。这两类病毒都以侵入用户电脑,窃取个人资料、银行账号等信息为目的,带有直接的经济利益特征。从统计数据看来,2007 年上半年的病毒数量比 2006 年同期增加 11.9%^[9]。面对如此海量的病毒,没有高效率的字符串匹配算法是不可想象的。

随着生命科学的发展,人们对生命物质的微观结构有了越来越清晰的认识。人类基因组序列的绘制工作早已完成,Prosite,Genbank 等的大型序列库也已建立^[10]。计算生物学的一个基本问题是:寻找一个或一组基因片断在一个基因序列中的出现位置^[11],以比较基因的相似性和遗传关系;或者根据已知的蛋白质序列来确定未知的序列所属的蛋白质种类和功能。由于蛋白质和基因都可以用建立在一定字符集上的符号序列来表示,所以传统的模式串技术又有了新的用武之地^[12]。

因此,字符串匹配技术在许多的领域中发挥着重大的作用,对字符串匹配算法进行改进和优化有着重大的现实意义。

1.1.2 字符串匹配技术面临的挑战

互联网的应用越来越普遍,网络带宽和流量正飞速增长,我国互联网的规模正迅速扩张。中国互联网络信息中心(CNNIC)2008 年 1 月的统计数字显示^[13],中国国际出口带宽的总容量已达到 368927Mbps,年增长率为 43.7%;IP 地址数已达到 1.35 亿个,年增长率为 38%。在如此海量而且还在迅速增长的网络流量面前,要让相关的网络安全系统能高效运作,需要对现有的字符串匹配算法进行改进优化。

1.2 本文的主要内容

本文主要是在前人研究的基础上对串匹配技术中的高效多模式串匹配算法——

—Wu-Manber 算法进行进一步的改进和优化。主要内容如下：

第一章主要介绍研究字符串匹配技术的目的和意义。

第二章对现有的串匹配算法的综述。

第三章首先介绍前人对 Wu-Manber 算法的改进优化，然后重点论述了在前人基础上提出的三个新的改进算法。

第四章对第三章提出的改进算法的实验验证和分析。

最后对本文进行总结，并对将来的工作进行了展望。

第二章 字符串匹配算法综述

字符串匹配的定义^[1]: 所谓字符串匹配 (string matching, or pattern matching), 就是给定一组特定的字符串集合 P , 对于任意的一个字符串 t , 找出 P 中的字符串在 t 中的所有出现位置。我们称 P 为模式串集合, 称 P 中的元素为模式串 (或关键词), 称 t 为文本。字符串中的字符都取自一个有限的字符集合 Σ , 简称字母表或字符集合^[1]。

根据文献^[14]的分类方法, 可以将字符串匹配分成 4 类: 精确字符串匹配 (exact string matching)、扩展字符串匹配 (extend string matching)、正则表达式匹配 (regular expression matching) 和近似字符串匹配 (approximate string matching)。精确字符串匹配要求文本中被匹配的字符串与模式串严格相等。近似字符串匹配则允许被匹配的字符串与模式串之间有一定的误差 (error): 两个字符串之间的误差可以用距离来度量, 常用的距离有: 编辑距离 (edit distance)、汉明距离 (Hamming distance) 等。而在正则表达式匹配中, 待匹配的不是简单的字符串, 而是正则表达式, 这使得正则表达式匹配比精确串匹配要困难得多。扩展字符串匹配介于精确字符串匹配和正则表达式匹配之间, 他的模式串通常是些特殊的正则表达式, 因而常可以通过特殊的手段来解决。

如果按模式串集合的固定与否, 则字符串匹配又可以分为 2 类: 静态字符串匹配 (static string matching, or fixed string matching) 和动态字符串匹配 (dynamic string matching)。所谓静态字符串匹配, 就是模式串集合是静态不变的; 而动态字符串匹配是模式串集合 P 是动态的, 集合 P 随着时间的变化而增减。

本文主要研究的是精确的、静态的 Wu-manber 字符串匹配算法, 文献^[1]给出精确的静态字符串匹配的严格定义:

已知: 有限字符集合 Σ ; 模式串集合 P ; $(\forall p \in P)(p \in \Sigma^+)$; 文本: $t \in \Sigma^+$ 。

求解: $occur(P, t) = \{(p, |x|) | (\exists x)(\exists y)(t = xpy), p \in P\}$, 即模式串集合中的模式串在文本 t 中出现的所有出现位置

直至今, 相关的算法已经提出了有几百种之多^[15], 在此只分析其中一些比较高效的算法。要分析的算法可以归结为 3 大类^[16]。

2.1 基于前缀搜索的算法

即在搜索窗口内从前向后（沿着文本的方向）逐个读入文本字符，搜索窗口中文本和模式串的最长公共前缀。

2.1.1 KMP 算法和 AC 算法^[1]

KMP 算法^[18]由 Morris、Pratt 和 Knuth 提出。其原理是：在当前文本窗口内从左向右开始匹配，如果在识别到模式串的第 $j+1$ 个字符 p_{j+1} 时发现不匹配，则说明前 j 个字符 $p_1p_2\dots p_j$ 与已经扫描的文本是相等的。如果我们知道 $p_1p_2\dots p_k$ ($0 \leq k < j$) 是 $p_1p_2\dots p_j$ 的后缀，并且 $p_1p_2\dots p_k$ 是满足这个条件的最长前缀串，那么匹配窗口就可以安全地向右移动 $j-k$ 个字符，同时从新的匹配窗口的第 $k+1$ 个字符开始比较。定义 $next[j] = \max\{k | (p_1p_2\dots p_k = p_{j-k+1}\dots p_j) \wedge (j \neq k)\}$ ，那么在位置 $j+1$ 不匹配时，匹配窗口可以安全地移动 $j - next[j]$ 个字符。

Aho-Corasick（以下简称 AC）算法^[19]是 KMP 算法向多模式串情形的自然扩展。AC 算法使用 Trie 数据结构来组织所有的模式串，然后从 Trie 的根结点开始层次遍历，构造结点间的后缀链（supply link）^[17]。在匹配时，当当前状态无法继续识别字符时，就沿着后缀链向上回溯，直到找到一个能够识别该字符的状态为止。Advanced AC 算法是在 AC 算法的基础上，将后缀链进行修改，直接指向当前状态的后继状态，构成的一个完全循环自动机。这样，在匹配时无需沿着后缀链回溯，而是直接跳转到它的后继状态。因此 Advanced AC 比 AC 快得多。（图 2.1-图 2.11 通过 AC 自动机搜索文本 $t=rajrjba$ 的过程来演示 AC 算法的运行机制）

从本质上看，KMP 和 AC 都是对识别模式串最长前缀的确定自动机的模拟。AC 算法不受模式串长度和文本特征的影响，具有抗攻击的优点，是工程应用的首选算法之一。

2.1.2 Shift-And/Shift-Or 算法^[16]

Shift-And 算法维护一个字符串的集合，集合中的每个字符串既是模式串 p 的前缀，同时也是已读入文本的后缀。对每个新读入的文本字符，该算法即用位并行的方法更新该集合。该集合用一个位掩码 $D = d_m \dots d_1$ 来表示。 D 的第 j 位被置为 1（称 D 的第 j 位是活动的）当且仅当 $p_1 \dots p_j$ 是 $t_1 \dots t_i$ 的后缀。如果 p 的长度不超过机器字长 w ，那么 D 可以存入一个机器字中。当 d_m 是活动的时候，就表示有一个成功匹配。

当读入下一个字符 t_{i+1} 时, 需要计算新的位掩码 D' 。 D' 的第 $j+1$ 位是活动的当且仅当 D 的第 j 位是活动的 (即 $p_1 \dots p_j$ 是 $t_1 \dots t_i$ 的后缀) 并且 t_{i+1} 与 p_{j+1} 相等。利用位并行, D' 的计算很容易在常数时间内完成。

Shift-And 算法首先构造一个表 B , 记录字母表中每个字符的位掩码 $b_m \dots b_1$ 。如果 $p_j = c$, 掩码 $B[c]$ 的第 j 位被置为 1, 否则为 0。

首先置 $D = 0^m$, 对于每个新读入的文本字符 t_{i+1} , 可以用如下公式对 D 进行更新: $D' \leftarrow ((D \ll 1) | 0^{m-1}) \& B[t_{i+1}]$ (1)

左移操作 \ll 将 D' 的第 $i+1$ 位的值置为 D 的第 i 位。因为空字符串 ϵ 也是文本的后缀, 所以 $D \ll 1$ 需要在最低位与 0^{m-1} 进行位或操作。因为要找到那些满足 $t_{i+1} = p_{j+1}$ 的位置, 所以需要再将上面的结果与 $B[t_{i+1}]$ 进行位与操作。

Shift-Or 算法的主要思想是通过对比位取反去掉公式 (1) 中的掩码 0^{m-1} , 从而加速 D' 的计算。

以上两种算法可以看作是使用一个非确定性自动机对扫描文本的过程进行模拟。它们只适合于字母表较小并且模式串长度较短的情况。

2.2 基于后缀搜索的算法^[10]

即在搜索窗口内从后向前 (沿着文本的反向) 逐个读入文本字符, 搜索窗口中文本和模式串的最长公共后缀。使用这种搜索方法的算法可以跳过一些文本字符, 从而具有亚线性的平均时间复杂度。

2.2.1 BM 算法、Horspool 算法

Boyer-Moore (简称 BM 算法) 算法^[20] 的思想: BM 算法预先计算三个 shift 函数 d_1 , d_2 和 d_3 , 它们分别对应于以下的三种情况。这三种情况都假设已经读入了一个既是搜索窗口中文本的后缀, 同时也是模式串后缀的字符串 u , 并且读入的下一个文本字符 σ 与模式串的下一个字符 α 不相等。

第一种情况下, 后缀 u 在模式串 p 中的另一个位置出现。设最右出现位置 (不包括在模式串末尾的出现) 为 $j(u = p_{j-|u|+1} \dots p_j)$ 。这种情况下安全的窗口移动方法是, 将窗口移动 $m - j$ 字符, 使得文本中的 u 与模式串中下一个 u 出现的位置相对齐。此时, 对于模式串的每一个后缀, 需要计算它到它的下一个出现之间的距离, 这个距

离用 $shift$ 函数 d_1 来表示。如果 p 的后缀 u 不在 p 中重复出现, 那么 $d_1(u)$ 被置为整个模式串的长度 m 。

第二种情况下, 后缀 u 不出现在 p 中的任何其他位置。此时需要对模式串的所有后缀计算第二个函数 d_2 。对于 p 的每一个后缀 u , $d_2(u)$ 表示既是 p 的前缀, 同时也是 u 的后缀的最长字符串 v 的长度。

第三种情况是在搜索窗口中从后向前搜索时, 文本字符 σ 处不能成功匹配。此时, 如果用函数 d_1 进行窗口移动, 并且对应的模式串字符不是 σ , 那么将会对新的搜索窗口进行一次不必要的验证, $shift$ 函数 d_3 就是用来保证下一次验证时文本字符 σ 一定与模式串中的一个字符 σ 相对应。对于字母表中的每一个字符 σ , $d_3(\sigma)$ 表示 σ 在模式串中的最右出现位置到模式串末尾的距离。如果 σ 不出现在 p 中, $d_3(\sigma)$ 被设置为 m 。

当读入了文本字符串 u 并不在字符 σ 上匹配时, BM 算法对这三种 $shift$ 函数进行两次比较, 以决定窗口的移动距离。

- 第一次比较: 取 $d_1(u)$ 和 $d_3(\sigma)$ 中的大者。
- 第二次比较: 取上面的比较结果和 $m - d_2(u)$ 中的较小者。

经过上面两次比较得到的就是 BM 的移动距离。

但如果抵达了窗口的起始位置, 说明已经发现一个成功的匹配, 此时只使用函数 d_2 来计算窗口的移动距离。

由于 BM 要计算的函数很复杂, 许多情况下使用 BM 的简化算法比 BM 有效。

Horspool 算法^[21]是 BM 算法的简化算法, 它只使用 d_3 来计算窗口的移动距离。对于每个搜索窗口, 该算法将窗口内文本的最后一个字符(设为 β)和模式串的最后一个字符进行比较。如果相等, 则需要一个验证过程。该验证过程在搜索窗口中从后向前对文本和模式串进行比较, 直到完全相等或者在某个字符处不匹配。然后无论匹配与否, 都将根据 β 在模式串中的下一个出现位置将窗口向右移动。

2.2.2 BM 算法、Horspool 算法在多模式串下的扩展^[16]

Commentz-Walter 算法^[22]是 BM 算法的自然扩展。它是第一个具有亚线性平均时间复杂度的多模式串匹配算法, 它的基本思想是: 用 trie 表示 $P = \{p^1, p^2, \dots, p^r\}$ 的反转 $P^r = \{(p^1)^r, \dots, (p^r)^r\}$, 用它识别文本字符。为了保证不遗漏可能的成功匹配, 当前位置 pos 指向 $lmin$ 。对于 pos 的每个新位置, 从 pos 开始, 从后向前识别

文本 $t_1 \dots t_{pos}$ 的最长后缀 u ，使得 u 也是某个模式串的后缀。如果找到了一个出现，就报告一个成功的匹配，然后根据在多模式串集合上扩展的 BM 的三个函数 d_1, d_2 和 d_3 ，将当前位置向右移动。对于 trie 的每个状态，都需要计算前两个函数。当识别了最长后缀 u 并抵达状态 q 时，根据这两个函数进行移动。

- $d_1(q)$ 是使得 $u = L(q)$ 与某个模式串的子串对齐的最小移动距离 $p' \in P$ 。
- $d_2(q)$ 是使得 $u = L(q)$ 的一个后缀与某个模式串的前缀相匹配的最小移动距离 $p' \in P$ 。

对于字母表中的每个字符 α 和每个位置 $0 \leq k < lmax$, $d_3[\alpha, k]$ 是使得位置 $pos - k$ 处的字符与模式串中的某个字符相匹配的最小移动距离 $p' \in P$ 。

如下是移动距离的计算公式：

$$s[q, pos, k] = \min_{d_2[q]}^{\max(d_1[q], d_3[t_{pos-k}, k])}$$

Set Horspool 算法是 Horspool 的扩展，也是 Commentz-Walter 的简化。它只适用于模式串集合很小并且字母表很大的场合。

2.2.3 Wu-Manber 算法^{[23][16]}

扩展的 Horspool 算法在多模式集合上性能很差，这是因为字母表中的每个字符通常都以高概率出现在某个模式串中，从而导致移动距离下降。

Wu 和 Manber 的算法克服了该问题。它通过读入一块字符，以降低字符块在某个模式串中出现的概率。这里设块的长度为 B 。使用散列函数 h_1 将所有可能的块散列到一个有限的表 $SHIFT$ 上。两个不同的块可能散列到 $SHIFT$ 中的同一个位置，设读入一块字符 Bl ，首先算法在 $SHIFT(j)$ 中存入满足 $j = h_1(Bl)$ 的所有块的移动距离的最小值。构建 $SHIFT$ 表的详细描述如下：

- 如果块字符 Bl 不出现在 P 中的任何一个模式串中，则可以向右安全地移动 $lmin - B + 1$ 个字符。因此，将表的每一项都初始化为 $lmin - B + 1$ 。
- 如果块字符 Bl 出现在 P 中的某一个模式串 p' 中，则需要找出 Bl 在 p' 中最右边出现的末尾位置 j ，然后将 $SHIFT(h_1(Bl))$ 置为 $m_i - j$ 。为了计算 $SHIFT$ 表的所有值，对每个模式串 $p' = p'_1 \dots p'_m$ 的每一块 $B = p'_{j-B+1} \dots p'_j$ ，都需要在 $SHIFT$ 表中找出对应的表项 $h_1(B)$ ，并将 $SHIFT(h_1(B))$ 置为它的当前值和 $m_i - j$ 中的较小值。

块长 B 取决于最短关键词的长度 $lmin$ 、模式串集合的大小以及字母表的大小。

Wu 和 Manber 的研究表明, 取 $B = \log_{\lfloor 2 \rfloor}(2 \times lmin \times r)$ 能产生最好的实验结果。
SHIFT 表的大小也可以随着可用空间的大小而变化。

只要移动距离是大于 0 的, 就可以安全地将当前位置向右移动。当移动距离为 0 时, 当前位置的左边可能就是一个成功匹配的模式串。对于此种情况, Wu 和 Manber 使用了另一个散列表 HASH (如图 2.12 为 HASH 的数据结构) 和散列函数 h_2 , 它的每个表项 $HASH(j)$ 是一个链表, 记录了最后一个块在 h_2 下映射到 j 的所有模式串。这样就可以找出那些最后一块的散列值与当前读入的文本块 Bl 散列值相同的所有模式串。

搜索阶段与 Set Horspool 算法类似, 将当前位置 pos 初始化为 $lmin$ 。对于每个当前位置 pos , 从后向前读入 B 个字符的块 Bl 。如果 $j = SHIFT(h_1(Bl)) > 0$, 那么将窗口移动到位置 $pos + j$, 并继续搜索; 如果 $SHIFT(h_1(Bl)) = 0$, 则用 HASH 找出文本块对应的一组模式串 $HASH(h_2(Bl))$, 并逐个与文本进行比较。下面是该算法的伪代码^[6]:

```

Wu-Manber ( $P = \{p^1, p^2, \dots, p^r\}, T = t_1 t_2 \dots t_n$ )
1.  Preprocessing
2.      Computation of B
3.      Construction of the hash tables SHIFT and HASH
4.  Searching
5.       $pos \leftarrow lmin$ 
6.      While  $pos \leq n$  Do
7.           $i \leftarrow h_1(t_{pos-B+1} \dots t_{pos})$ 
8.          If  $SHIFT[i] = 0$  Then
9.              list  $HASH[h_2(t_{pos-B+1} \dots t_{pos})]$ 
10.             Verify all the patterns in list one by one
                against the text
11.              $pos \leftarrow pos + 1$ 
12.          Else  $pos \leftarrow pos + SHIFT[i]$ 
13.          End of if
14.      End of while

```

SHIFT[]表的计算方法^[24]:

$$\text{SHIFT}[X] = \begin{cases} m-B+1 & (\text{如果 } X \text{ 不在任何模式中出现}) \\ \min\{m-j \mid X[k]=P_j[j-B+k], 1 \leq k \leq B\} & \end{cases}$$

Wu-Manber 算法的时间复杂度平均情况是 $O(\frac{B \times n}{m})$ [24]。

其中 B 是块字符的长度, n 是文本长度, m 是模式的最小长度。(图 2.13、图 2.14 和图 2.15 演示了在三种情况下 Wu-Manber 算法的跳跃过程)

2.3 基于子串搜索的算法 [16]

基于子串搜索的方法, 搜索窗口的移动很简单。假设在搜索窗口中已经从后向前识别了模式串的子串 u , 并且无法继续识别下一个字符 σ 。也就是说 σu 不再是 p 的子串, p 在文本中的出现不可能覆盖 σu 。所以, 此时可以安全将搜索窗口移到 σ 之后。该方法的难点是如何识别模式串中的所有子串。

2.3.1 BDM 算法

Backward Dawg Matching (BDM) 算法 [25] 使用后缀自动机搜索子串, 并且对基于基本的子串搜索方法进行了改进。

后缀自动机的目的是判别字符串 u 是否是模式串 p 的一个子串。它有如下 3 个性质:

性质 1: 可以在 $O(|u|)$ 的时间内确定一个字符串 u 是否为模式串 p 的子串。字符串 u 是 p 的一个子串当且仅当 p 的后缀自动机中存在一条从初始状态开始的标号为 u 的路径。

性质 2: 可以识别模式串的所有后缀。从初始状态到某个终止状态的路径上的字符组成的字符串是模式串 p 的一个后缀。

性质 3: 模式串 $p = p_1 p_2 \dots p_m$ 对应的后缀自动机 [用 $SA(p = p_1 p_2 \dots p_m)$ 表示] 可以通过在线的方法在 $O(m)$ 时间内构建完成, 即依次将字符 p_j 添加到 $SA(p = p_1 p_2 \dots p_{j-1})$ 以构造 $SA(p = p_1 p_2 \dots p_j)$ 。

为了在文本 $T = t_1 t_2 \dots t_n$ 中搜索模式串 $p = p_1 p_2 \dots p_m$, 需要构建模式串 p 的反转 $p^r = p_m p_{m-1} \dots p_1$ 对应的后缀自动机。算法使用后缀自动机在搜索窗口中从后向前搜索模式串的子串。在搜索过程中, 如果达到一个终止状态, 并且对应的不是整个模式串, 那么它在窗口中的当前位置被保存在变量 $last$ 中。根据性质 2, 这对应于找

到了一个模式串 p 的前缀，并且是当前所识别的最长前缀。它从位置 $last$ 开始，到窗口末端结束。这个反向搜索过程以两种可能的方式结束：

i) 在识别一个子串时失败了，即读入了一个字符 σ ，而在 p^n 的后缀自动机的当前状态没有 σ 转移。此时将窗口向右移动，使得它的起始位置与 $last$ 对齐。这样移动窗口不会遗漏任何可能的匹配，否则将会在窗口中发现模式串的一个更长的前缀。

ii) 抵达了窗口的起始位置，这意味着整个模式串 p 被匹配成功。报告一个成功的匹配，然后象 i) 一样移动窗口。

2.3.2 SBDM 算法^[16]

SBDM 是 BDM 在多模式下的扩展，其构造和搜索类似于 BDM。在实际应用中，SBDM 的后缀自动机构建代价很大，对于很大的模式串集合，构建的时间很难被搜索阶段的时间分摊抵消。而且，随着模式串规模的增大，后缀自动机耗用的存储空间也急剧增大。

2.3.3 BNDM 算法^[16]

Backward Nondeterministic Dawg Matching (BNDM) 算法的搜索方法与 BDM 算法相同，但它使用了位并行来识别子串。与原始的 BDM 相比，BNDM 更简单，内存用量更少，具有更好的引用局部性，并且易于扩展到更复杂的模式串情形。

在当前搜索窗口中，设已读入的字符串为 u ，BNDM 算法维护一个集合，记录 u 在 p^n 中的所有出现位置。同 *Shift-And* 算法一样，该集合可以用一个位向量 D 来表示。如果子串 $p_j \dots p_{j+m-1}$ 等于 u ，那么 D 的第 $m-j+1$ 位是 1，表示 p 的位置 j 是一个活动状态。

当读入一个新的文本字符 σ 时，要从 D 更新到 D' 。 D' 的一个活动状态 j 对应于 σu 在模式串中的一个起始位置，也就是说： u 从现在模式串的位置 $j+1$ ，即 D 的第 $j+1$ 位是活动的； σ 在模式串的位置 j 处出现。

同 *Shift-And* 算法一样，BNDM 算法预先计算一张表 B ，表 B 用一个位掩码记录了该字符在 p 中的出现位置。那么利用如下公式，就可以从 D 更新到 D' ：

$$D' \leftarrow (D \ll 1) \& B[\sigma]$$

BNDM 与 BDM 使用的搜索方法是一样的, 区别在于前者用位并行技术进行子串搜索, 而后者使用后缀自动机。Multiple BNDM 算法是 BNDM 算法在多模式串下的扩展其搜索过程跟 BNDM 算法类似。

2.3.4 BOM 算法^[16]

当字符串长度大于机器字长 w 时, 虽然 BDM 算法可以满足使用之需, 但是后缀自动机的构造过于复杂, 使得它并不实用。文献^[37]指出, 在基于子串搜索的方法中, 要安全地移动搜索窗口, 并不需要知道 u 是否为模式串 p 的一个子串, 而只需要知道 σu 不是 p 的子串就足够了。解决方法是在字符串 p 上构建 Factor Oracle, 它能识别的字符串集合是 p 的子串集合的超集。所以, 在窗口中从后向前识别时可能会多读入字符, 而带来一定的性能损失, 但 Factor Oracle 易于理解和实现, 并且结构简洁紧凑, 从而弥补性能上的损失。Set Backward Oracle Matching (SBOM) 算法是 BOM 算法在多模式串下的扩展, 其构建和搜索与 BOM 算法类似 (图 2.16、2.17、2.18 展示了 BOM 算法的运行机制)。

2.4 近段的进展^[1]

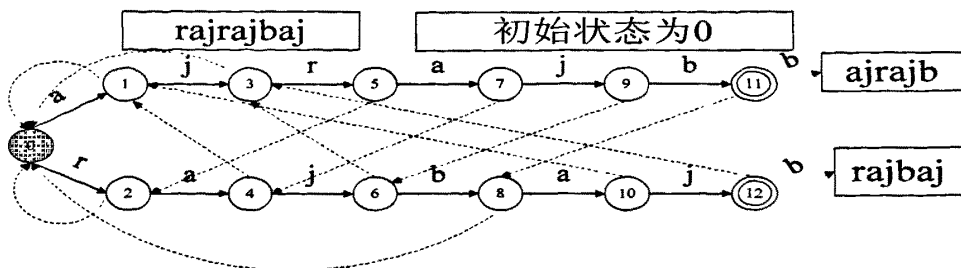
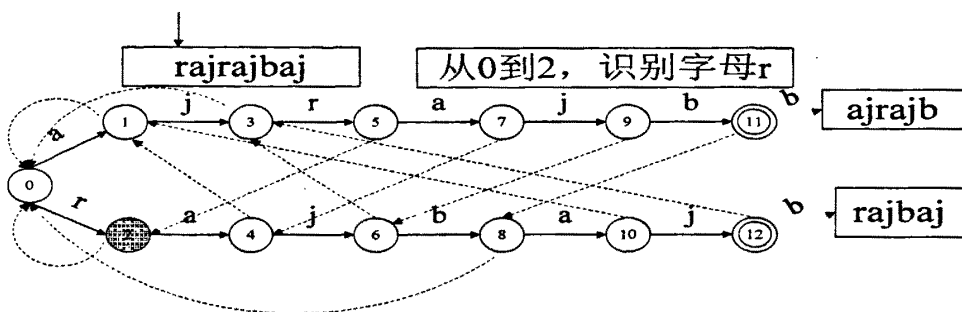
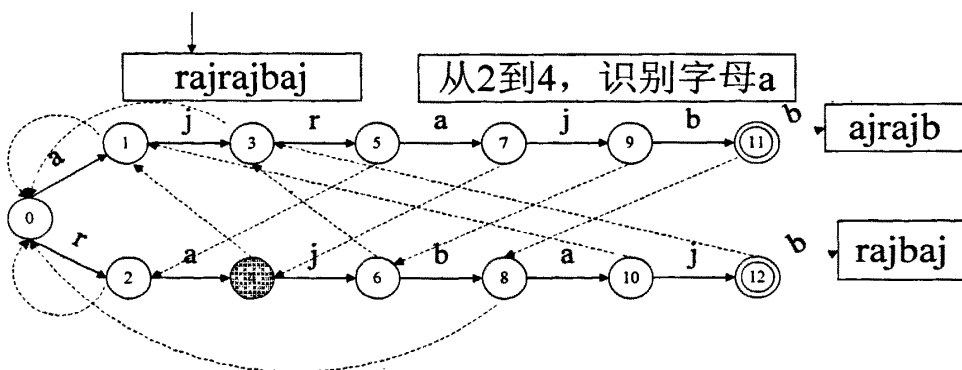
近年来, 串匹配又有如下一些进展:

文献^[26]提出了一种基于分组的串匹配算法。文中证明了最优分组定理, 并提出了基于最短路径和动态规划的两种最优分组策略。分组算法适合模式串长度变化幅度非常大的情形, 是解决大规模串匹配的一种有效方法。

文献^[27]提出了一种时间复杂度最优的精确匹配算法。该算法将文本分成 $\lfloor \frac{n}{m} \rfloor$ 个长度为 $2m-1$ 的相互重叠的窗口。在每个窗口内, 综合使用后缀自动机 *DAWG* 和 *AC* 自动机进行扫描, 保证了 $O(n)$ 的最坏时间复杂度和 $O(\frac{n}{m} \log_{\square} rm)$ 的平均时间复杂度。

文献^[28]使用 q -Grams 对 *Horspool*、*Shift-OR*、*BNDM* 算法进行了扩展, 进行 1 万-10 万级规模的串匹配, 取到了比较好的效果。

文献^[29]使用 Band-Row 的方法来压缩 *SNORT* 中使用的 *AC* 算法, 速度比原算法提高了 17%。这种表示法能够获得 $O(1)$ 的状态节点转换速度, 只需要两次额外的边界检查, 但是当一行中的非空元素个数大于 3 时, 空间压缩的效果不明显。此外, 两

图2.2 AC自动机搜索文本 $t=rajrajbaj$ 的过程图2.3 AC自动机搜索文本 $t=rajrajbaj$ 的过程图2.4 AC自动机搜索文本 $t=rajrajbaj$ 的过程

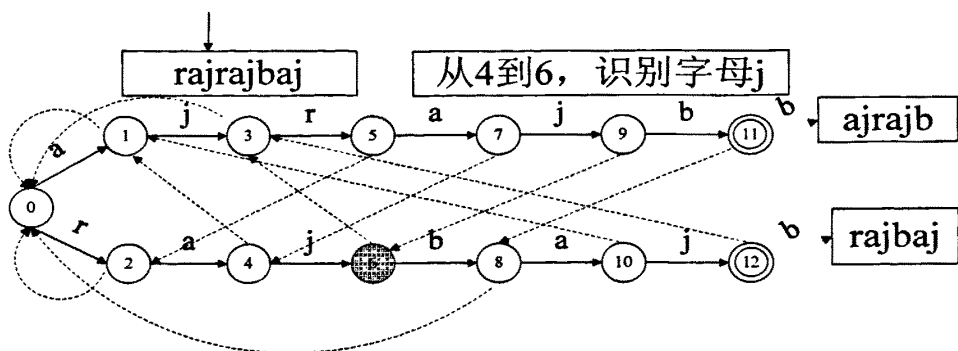


图2.5 AC自动机搜索文本 $t=rajrajbaj$ 的过程

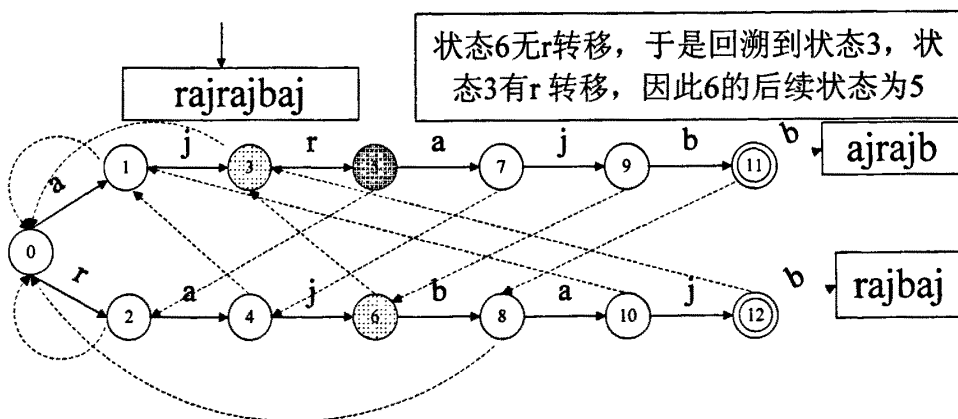


图2.6 AC自动机搜索文本 $t=rajrajbaj$ 的过程

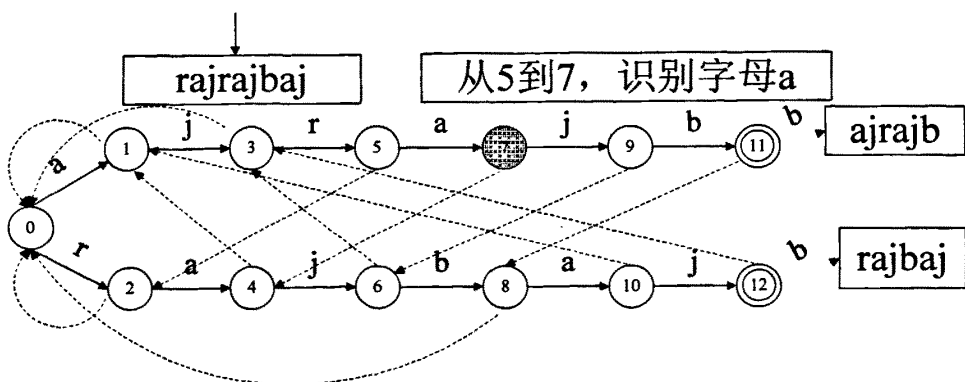
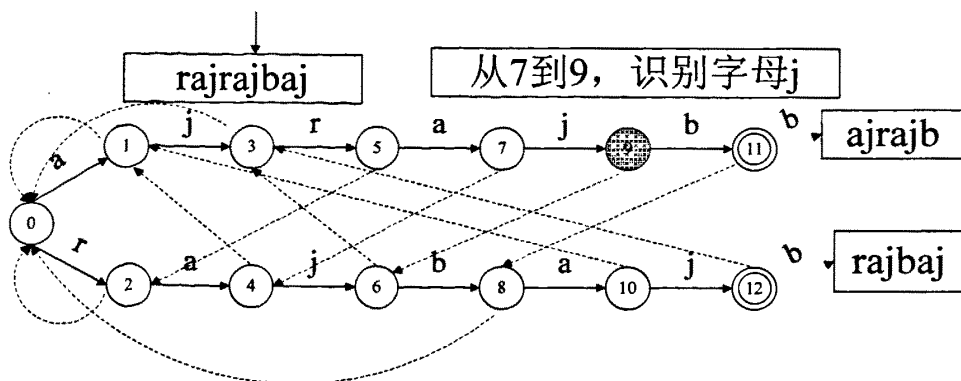
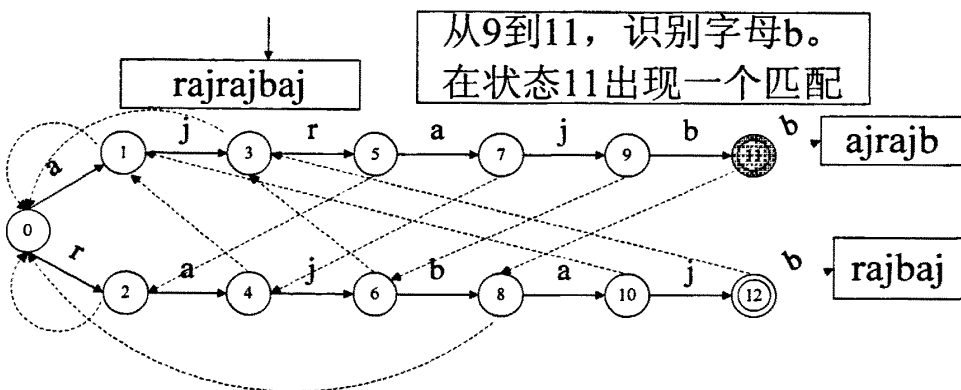
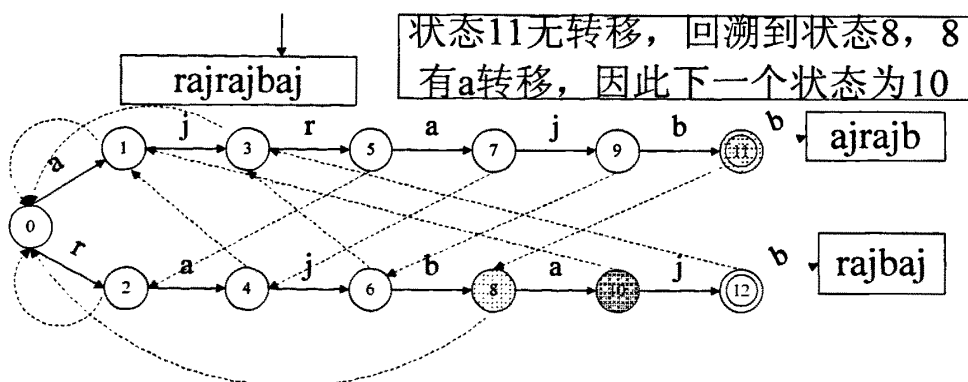


图2.7 AC自动机搜索文本 $t=rajrajbaj$ 的过程

图2.8 AC自动机搜索文本 $t=rajrajbj$ 的过程图2.9 AC自动机搜索文本 $t=rajrajbj$ 的过程图2.10 AC自动机搜索文本 $t=rajrajbj$ 的过程

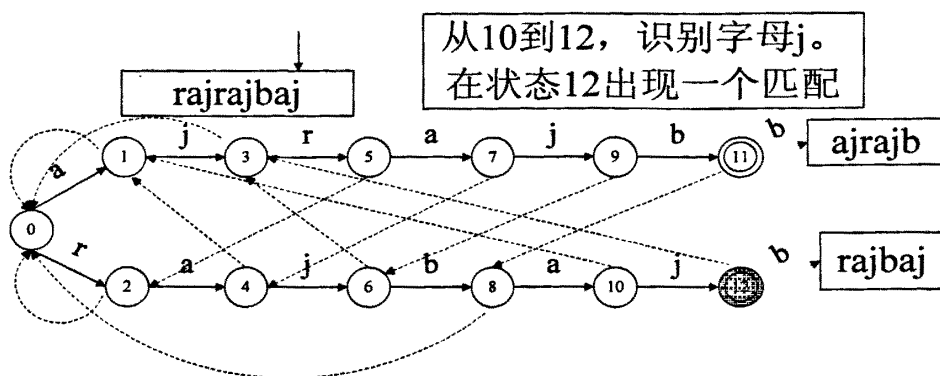


图2.11 AC自动机搜索文本 $t=rajrajbj$ 的过程

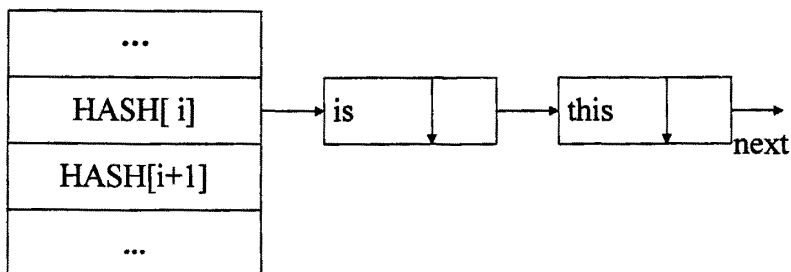


图2.12 Wu-Manber算法中的HASH数据结构

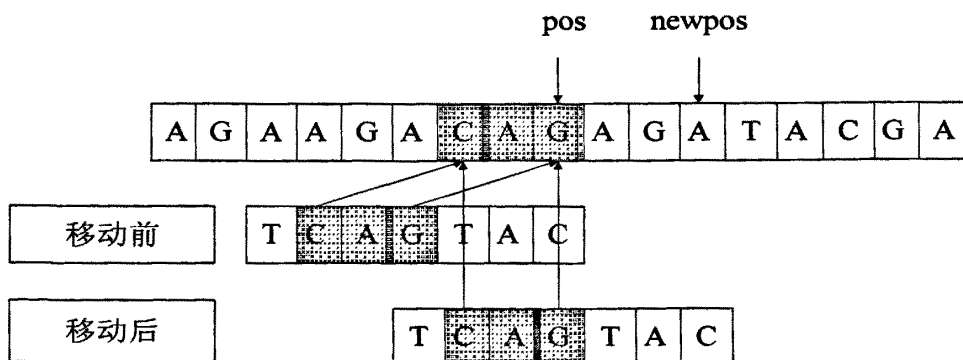


图2.13 Wu-Manber的跳跃过程：块B出现在模式串中

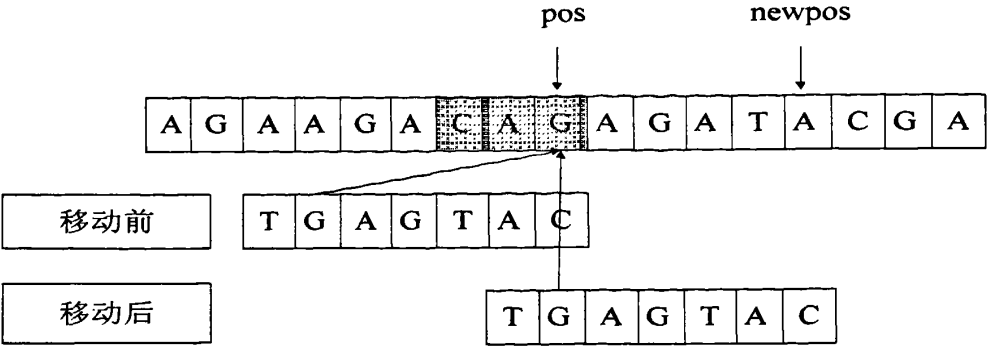


图2.14 Wu-Manber的跳跃过程：块B没有出现在模式串中

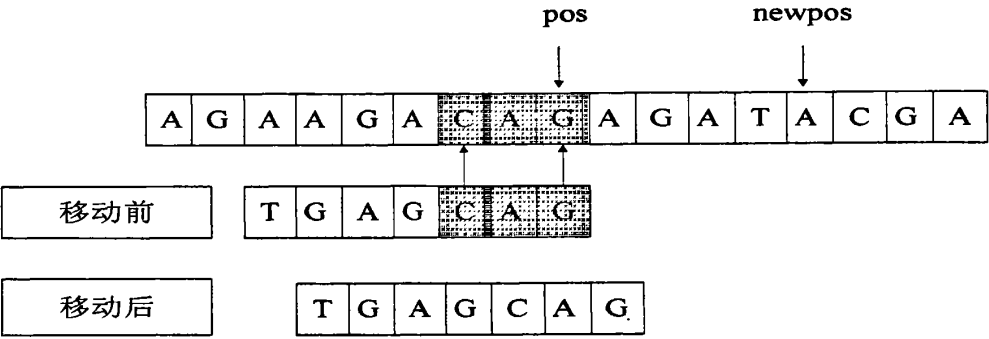
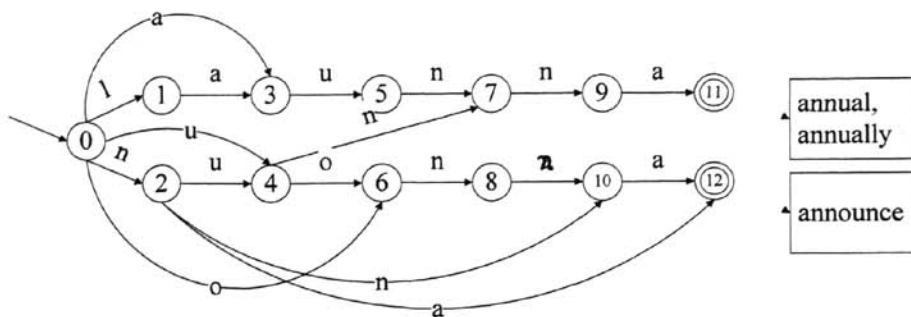


图2.15 Wu-Manber的跳跃过程：块B出现在模式串中且SHIFT[B]=0



$P=\{\text{announce, annual, annually}\}$ 的反转对应的Factor Oracle自动机

图2.16

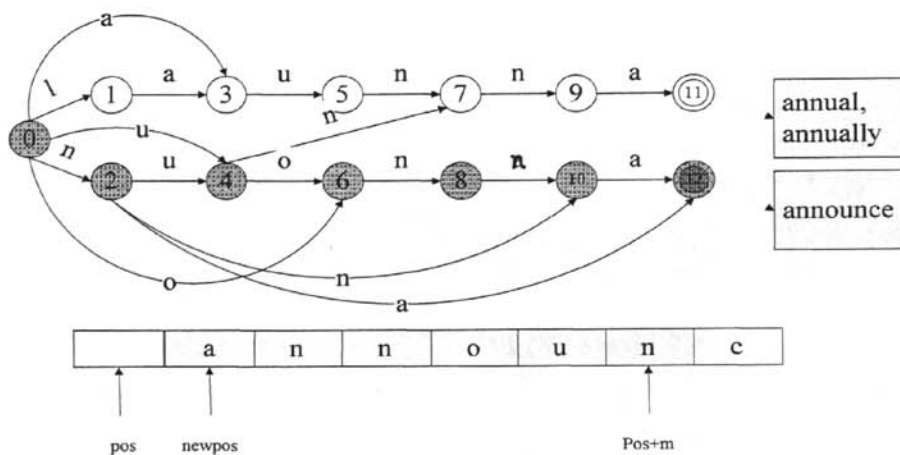


图2.17 Factor Oracle 完全识别匹配窗口

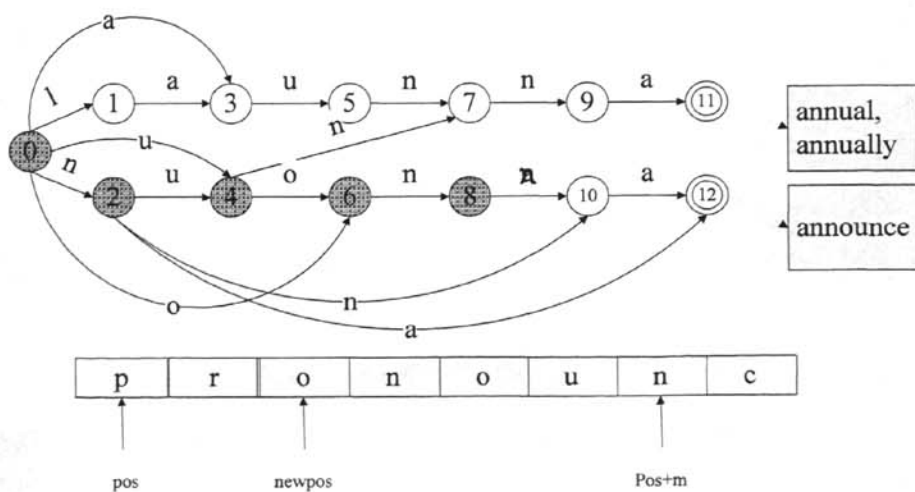


图2.18 Factor Oracle 部分识别匹配窗口

第三章 对 Wu-Manber 算法的改进

2.2.3 已经对 Wu-Manber 算法进行了描述。同时，在后缀的散列函数相等的概率很大时，原 Wu-Manber 算法增加一个 PREF 散列表来减少访问 HASH 链表的概率。但它的 SHIFT 跳跃距离表和 HASH 散列表都有需要改进的地方。

3.1 精确的不良字符转移和弱化的良好后缀转移^[35]

本节叙述文献^[35]提出的改进算法，该算法提出“精确的不良字符转移和弱化的良好后缀转移”增加了有关匹配跳跃的距离，而且在跳跃距离为 0 时，原算法在找出可能匹配的字符串后不是简单的向前移动一位，而是查找相关散列表继续进行跳跃。

3.1.1 精确的不良字符转移

Wu-Manber 算法在选择匹配入口点的问题中使用的 SHIFT[] 表记录了基于块字符的不良字符移动距离。在相关 SHIFT[] 的计算公式中，最大值只能达到 $m-B+1$ ，但很多时候，该最大值还可以更大，甚至达到 m 。

该算法对 SHIFT[] 表的改进是如下过程计算

- 1) 用 m 填写 SHIFT[] 表;
- 2) for($i=0; i < B; i++$)
 - {对所有 $B_c \in \{\text{suffix}(B_c, i) = \text{prefix}(\text{pattern}, i)\}$
 - $\text{shift}[B_c] = m - i;$ }
- 3) for(每一个关键词)
 - for(当前关键词中的每一个块字符
 - 计算 $\text{shift}[B_c];$)

3.1.2 弱化的良好后缀转移

当 $\text{shift}[B_c] = 0$ 时，原算法在进入匹配计算模块后，不论匹配结果如何，下一个匹配开始位置都固定的向右移动一位。该算法提出的改进办法是引入一个

GBSshift[]表, 该表记录了每一个关键词的长度为 B 的后缀 (最后一块字符) 在所有关键词中的所有非后缀出现位置与相应关键词词尾的距离的最小值。这样, 在 shift[Bc]=0 进入匹配阶段后, 就可以用 GBSshift[Bc]来确定下一个可能的匹配的位置与当前匹配位置的距离。由于 GBSshift[]表和 SHIFT[]表的计算上的近似性, 不需要为其作额外的计算工作, 只需要复制计算出的 SHIFT[]表的中间结果, 所以它所需的额外处理时间与 SHIFT[]相同。

3.1.3 结论

通过实验证明, 该算法在附加微小的预处理时间后, 改进算法能有效降低比较次数, 从而减少了对整个文本数据的扫描时间。该算法对原算法中 SHIFT[]表的计算方法进行改进, 提出精确的不良字符转移距离和弱化的良好后缀转移距离, 进而减少匹配中字符串的比较次数。

3.2 一种基于后缀模式处理的改进算法^[36]

本节叙述文献^[36]改进算法, 该算法针对 Wu-Manber 算法在处理后缀模式情况下的不足, 给出一种改进的后缀模式处理算法, 提出对 Wu-Manber 算法中后缀模式处理算法和匹配过程的改进。

3.2.1 后缀模式处理的改进思路

算法的思想是 P 中有的模式串是其他模式串的后缀 (如 this、his、is, is 是 his 和 this 字符串的后缀), 则通过修改 HASH 数据结构, 增加一个 Same_Suffix 指针域 (图 3.1), 以后如果有匹配的就只需查找到其 Same_Suffix 域, 而无须再访问 next 域所指向链表的所有其他结点, 从而减少了匹配过程中字符比较的次数, 提高了算法的运行效率。

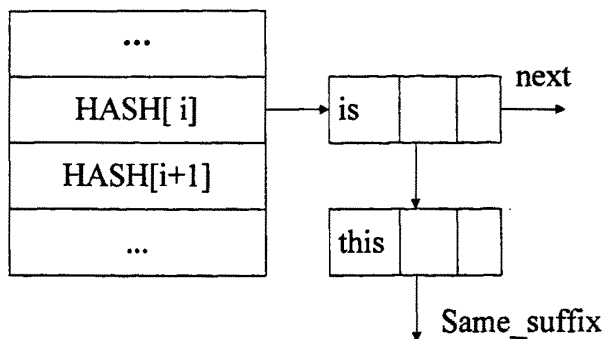


图3.1 增加Same_suffix指针的HASH数据结构

3.2.2 后缀模式处理的改进代码

Input: patterns[1..q], 原 HASH[] 表

Output: 新 HASH[] 表

function m_preProcess(patterns[1..q], HASH[])

for P=1 to q do

 h=hash(patterns[P], B);

 p=HASH[h];

 for each p in the list of HASH[h] do

 if(p.pattern.id==P) then

 q=p;

 else if((p.pattern 是 pattern[P] 的后缀) (sp 为空或者
 p.pattern.length<sp.pattern.length)) then

 sp=p;

 end if

 end for

 if(sp 不为空) then

 把 q 从 HASH[h] 的 next 链表中取出;

 把 q 链接到 sp.same_suffix 链表;

 end if

end for

匹配算法部分: Input:textbegin、textend、HASH[]、SHIFT[]

Output:所有遇到 textbegin...textend 中的匹配

```
function match_process(textbegin, textend, HASH[], SHIFT[])
text:=textbegin+m-1;
while text<=textend do
begin
  h=hash(text, B);
  shift=SHIFT[h];
  if(shift==0) then
    p=HASH[h];
    for each p in the list of HASH[h] do
      begin
        px=p.pattern;
        qx=text-p.pattern.length+1;
        if(p.pattern 与 [qx...text]匹配) then
          报告发现匹配;
          for each q in the list p.same_suffix do
            begin
              q.pattern 中 p.pattern 前面的字符与 text 进行匹配并且报告发现的
              匹配;
            end for
          end if
          if(发现一个或者多个模式) then
            break; //结束匹配
          end if
        end for
        shift=1;
      end if
      text+=shift;
    end while
```

3.2.3 结论

该改进算法提出的后缀模式处理算法，在匹配入口处，不必对 HASH[] 的 next 链表的每一个模式与正文 text 进行匹配，通过引入 same_suffix 链表来处理后缀模式，减少了匹配过程串字符比较的次数。

3.3 基于非空公共子后缀的 Wu-Manber 算法的改进

3.2 节的改进算法是对关键字互为后缀时的改进，在实际应用中，关键字互为后缀时，它们就有非空公共子后缀；也就是说关键字互为后缀是关键字具有非空公共子后缀的子集。基于这种情况，本节提出了 Wu-Manber 算法的一种基于非空公共子后缀的改进。

3.3.1 非空公共子后缀的定义

在字符串集合 $P = \{P_1, \dots, P_r\}$ 中，称 P_i 和 P_j 有非空公共子后缀，如果分别删除 P_i 和 P_j 长度为 B 的后缀之后，他们的子串有相同的非空后缀（B 为一个模式串中用于计算 HASH 值的后缀字符个数）。如我们取 $B=2$ ，则“method”和“other”的子串“meth”和“oth”有相同的非空后缀“th”。

3.3.2 基于非空公共子后缀的改进思路

该改进算法利用 HASH 的 next 链表中模式串具有非空公共子后缀的关系给 next 链表中的模式串进行分类，把其中具有非空公共子后缀的模式串通过一个指针链接起来。搜索时只需找到相应的 next 链表中与文本 T 中的相应字符串相等的非空公共子后缀的入口，而不用逐一比较原 next 链表中的全部模式串。

首先对 HASH 的数据结构进行修改，修改后的 HASH 数据结构增加了一个 Same_Subsuffix 域，然后修改算法的预处理过程，把具有相同 HASH 值并且具有非空公共子后缀的模式串通过 Same_Subsuffix 链接起来，这样相同 HASH 值并且具有非空公共子后缀的模式串在 next 链表中就只存在一个结点，如图 3.2。

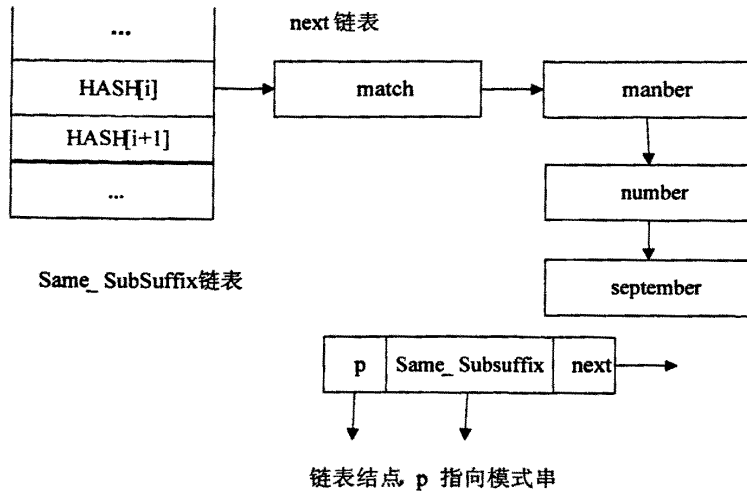


图3.2 增加Same_Subsuffix指针的HASH数据结构

3.3.3 基于非空公共子后缀的改进代码

以下是对 Wu-manber 算法中预处理算法的改进（伪代码）：

Input: patterns[1..q], 空的 HASH[] 表

Output: HASH[] 表

function m_preProcess(patterns[1..q], HASH[])

for P=1 to q do

begin

h=hash(patterns[P], B);

p=HASH[h];

while(p 不为空) do

begin

if(patterns[P]与 p.pattern 有非空公共子后缀) then

把 patterns[P] 链接到 p.same_subsuffix 链表

Break; //退出 while 循环

else

p=p.next;

end if

end while

```

if(p 为空, 即链表 p 中的模式和 patterns[P] 没有公共子后缀) then
    (把 patterns[P] 加入 HASH[h] 的 next 链表的链尾;)
end if

```

以下为预处理算法的其他部分, 跟原 Wu-Manber 算法中的算法一样 (略)

```

end for

```

匹配算法部分:

Input: textbegin、textend、HASH[]、SHIFT[]

Output: 所有遇到 textbegin...textend 中的匹配

```

function match_process(textbegin, textend, HASH[], SHIFT[])

```

```

    text:=textbegin+m-1;

```

```

    while text<=textend do

```

```

        begin

```

```

            h=hash(text, B);

```

```

            shift=SHIFT[h];

```

```

            if(shift==0) then

```

```

                p=HASH[h];

```

```

                for each p in the list of HASH[h] do

```

```

                    begin

```

```

                        px=p.pattern;

```

```

                        qx=text-p.pattern.length+1;

```

```

                        if(p.pattern 与 [qx...text] 有非空公共子后缀) then

```

```

                            进行完全匹配, 如果成功, 报告发现匹配;

```

```

                            for each q in the list p.same_subsuffix do

```

```

                                begin

```

```

                                    px=q.pattern;

```

```

                                    qx=text-q.pattern.length+1;

```

```

                                    if(q.pattern 与 [qx...text] 匹配) then

```

```

                                        报告发现匹配;

```

```

                                    end if

```

```

                                end for

```



```

    end if
    if(已经进入过 same_subsuffix 域) then
        break; //结束匹配
    end if
end for
shift=1;
end if
text+=shift;
end while

```

非空公共子后缀改进算法的预处理过程的时间复杂度最坏时是 $O(|\Sigma| \times r)$, Σ 为相关的字符表; 最坏情况是所有模式的 HASH 值都相等, 且除了最初的 $|\Sigma|$ 个模式串, 其他模式串都处于 next 链表末端的 Same_Subsuffix 链表中, 显然这种情况不大可能发生。最好情况是 $O(r)$ (其中 r 为模式串的数量)。最好情况是模式串集合中 HASH 值相等的模式串都具有相同的非空公共子后缀。一般情况下, 当选取适合的 hash 函数时, 模式串在 HASH 表中有较均匀的分布, 则预处理算法的时间复杂度是可以接受的。

改进后的匹配算法与原 Wu-Manber 算法的时间复杂度相同, 平均情况下可达到 $O(\frac{B \times n}{m})$ 。

3.4 对 Wu-Manber 算法的综合改进

分析图 3.3 (a) 和图 3.3 (b) 的情况, 可以发现在等概率的情况下, 在图 3.3 (a) 中查找相关模式串时的平均字符比较次数将大于在图 3.3 (b) 的情况。

也就是说, 在等概率的条件下, 当出现图 3.3(a) 的情形时, 我们通过把图 3.3(a) 转换成 3.3(b) 的情形, 就可以缩短查找模式串的平均查找长度。

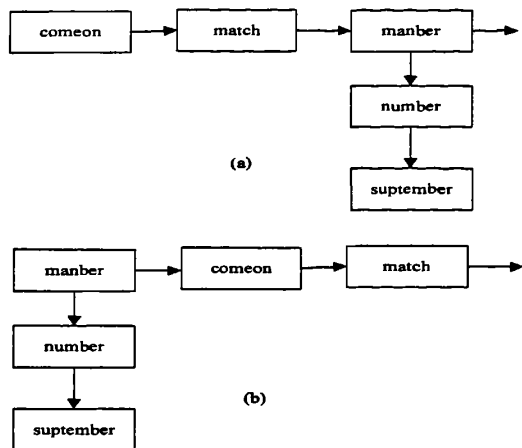


图3.3

3.4.1 Wu-Manber 算法综合改进思路

经过分析,发现 3.1 节的改进算法与 3.3 节的改进算法时从不同的角度来对 Wu-Manber 算法进行改进,也就是说,这两种改进方法是互补的,因此,有必要把这两种改进算法的优势综合起来,以期获得更高效率的算法。同时 3.3 节的预处理过程中找到有公共子后缀的结点时,只是简单地把要加入 next 链表中的结点链接到 next 链表中相关结点的 Same_Subsuffix 域,因此会获得类似于图 3.3(a)的 next 链表,我们可以把该 next 链表改进为类似于图 3.3(b)的 next 链表,即把有非空公共子后缀的结点移到 next 链表的头部;这样,我们就有可能在一定程度上减少 next 链表中字符串的平均查找长度,从而获得比原来的改进算法更高效率的算法。

3.4.2 Wu-Manber 算法综合改进的改进代码

以下是对 3.1 节提出的算法中预处理算法的改进部分(伪代码):

```

m_preProcess(.....) // (只增加 3.1 节中的改进代码,并把有非空公共子后缀的结点移动到 next 链表的首部的代码<斜体部分>)
{

```

.....

//以上省略的是把字符串 P_i 加入到 HASH[] 表之前的代码, 以下修改的代码把 P_i 加入 HASH[] 表中相应的 Same_Subsuffix 链中

// $|sp \rightarrow p|$ 为 $sp \rightarrow p$ 指针所指向模式串的长度, $|P_i|$ 为模式串 P_i 长度, B 为用于计算 HASH 值的块字符的字符个数

```

    spmid=NULL;
    ptn=new (新的结点);
    ptn->next=NULL;
    ptn->Same_Subsuffix=NULL;
    ptn->p= $P_i$ ; //ptn 为指向  $P_i$  字符串

```

//的将加入 HASH[] 表中的结点指针

```

    h=hash( $P_i$ , B);
    sp=HASH[h];

```

//sp 指向相应的 HASH[] 表

```

    spmid= sp;
    while(sp)
    {

```

```

        if(*( $sp \rightarrow p + |sp \rightarrow p| - (B+1)$ ))==*( $P_i + |P_i| - (B+1)$ ))

```

//如果具有非空公共子后缀 (即判断 $sp \rightarrow p$ 所指向的字符串倒数第 B+1 个字符是否和 P_i 的倒数第 B+1 个字符相等), 则加入相关的 Same_Subsuffix 链表中

```

    {
        ptn->Same_Subsuffix= $sp \rightarrow$ Same_Subsuffix;
         $sp \rightarrow$ SameSubsuffix=ptn;

```

//////////以下为本改进新加入的代码

```

        if( $m\_spmid \neq sp$ )//如果sp不指向next链表的首部则移动

```

结点到首部

```

    {
         $spmid \rightarrow next = sp \rightarrow next$  ;
         $sp \rightarrow next = hhh[mhash1]$ ;
         $hhh[mhash1] = sp$ ;
    }

```

```

//////////以上为本改进新加入的代码

        break;//否则直接退出循环
    }
    else //继续查看 next 表
    {
        spmid=sp;
        sp=sp->next;
    }
}

If (sp==NULL)
//sp 为空则直接加入
    spmid=ptn;
//以下为预处理程序的其他部分
    .....
}

```

匹配部分算法的改进部分（增加的部分与 3.1 中的改进算法类似）：

```

Match_Proccess(...)
{
    .....

    shift1=shift(Curtext, B);
    //Curtext 为当前查找文本 T 中指针指向的位置
    .....

//只修改 shift1=0 时的代码段
//当 shift1=0 时对链表的处理：
    h=hash(Curtext, B);
    sp=HASH[h];
    while(sp)
    {

```

```

        If(*(&sp->p+|sp->p|-|B|-1)!=*(Curtext -|B|))
//在 next 链表中查找相关非空公共子后缀
    {
        sp=sp->next;
//不是相关非空公共子后缀，则继续查看下一 next 结点
    else
        break; //否则，跳出循环
    }
    while(sp)
// 在 Same_Subsuffix 链表中查找
    {
//在此对 sp 指向的串跟 T 进行一次全串匹配，并记录成功匹配的串在 T 中的位置
//（代码略）
        sp=sp->Same_Subsuffix;
//逐一访问 Same_Subsuffix 链表中的结点
    }
//以下是原代码的其他部分
    .....
}

```

3.5 Wu-Manber 算法在大规模模式串下的改进

在大规模模式串下，Wu-Manber 算法及其改进算法的匹配效率都有不同程度的下降，因此有必要对算法在该情况下的运行效率进行研究。

3.5.1 Wu-Manber 算法在大规模模式串下的改进思路

本节把 3.1 节所提出的 Same_Subsuffix 域分裂为两个子域 Left_Subsuffix 和 Right_Subsuffix，把原算法中处于 Same_Subsuffix 域中的结点分配到分裂出来

的两个子域中。在匹配查找相应结点时，只需根据要匹配的字符串查找其中的一个子域，从而减少字符比较的次数。改进后的匹配算法与原 Wu-Manber 算法的时间复杂度相同，平均情况下可达到 $O(\frac{B \times n}{m})$ 。

3.5.2 Wu-Manber 算法在大规模模式串下改进的改进代码

以下是对 Wu-manber 算法中预处理算法的改进代码(伪代码):

Input: patterns[1..q], 空的 HASH[] 表, 空的 SHIFT[] 表

Output: HASH[] 表, SHIFT[] 表

function m_preProcess(patterns[1..q], HASH[], SHIFT[])

 for P=1 to q do

 begin

 h=hash(patterns[P], B);

 p=HASH[h];

 while(p 不为空) do

 begin

 if(patterns[P]与 p.pattern 有非空公共子后缀) then

 分情况把 patterns[P]链接到 p. Left _subsufffix 链表或 p. Right _subsufffix 链表

 (即: 一、如果 patterns[P]要链接到 p. Left _subsufffix 链表, 而 1)

 p. Left _subsufffix 链表为空, 则直接把 patterns[P]要链接到 p. Left _subsufffix 域; 2) p. Left _subsufffix 链表非空, 则把 patterns[P]

 要链接到 p. Left _subsufffix 域所指向结点的 Left _subsufffix 域中;

 二、如果 patterns[P]要链接到 p. Right _subsufffix 链表, 而 1) p. Right _subsufffix 链表为空, 则直接把 patterns[P]要链接到 p. Right _subsufffix 域; 2) p. Right _subsufffix 链表非空, 则把 patterns[P]

 要链接到 p. Right _subsufffix 域所指向结点的 Left _subsufffix 域中;)

 Break; //退出 while 循环

 else

```

        p=p. next;
    end if
end while
if(p 为空, 即链表 p 中的模式和 patterns[P] 没有公共子后缀) then
    (把 patterns[P] 加入 HASH[h] 的 next 链表的链尾; )
end if
    在此计算预处理算法的 SHIFT[] 表的值, 跟原 Wu-Manber 算法中的算法一样(略)
end for

```

匹配算法部分: (修改了代码的斜体部分)

Input: textbegin、textend、HASH[]、SHIFT[]

Output: 所有遇到 textbegin...textend 中的匹配

function match_process(textbegin, textend, HASH[], SHIFT[])

text:=textbegin+m-1;

while text<=textend do

begin

h=hash(text, B);

shift=SHIFT[h];

if(shift==0) then

 p=HASH[h];

 for each p in the list of HASH[h] do

 begin

 px=p. pattern;

 qx=text-p. pattern. length+1;

 if(*p. pattern* 与 [*qx...text*] 有非空公共子后缀) then

 进行完全匹配, 如果有相关匹配串, 报告发现匹配;

 判断应该进入 p 的哪个分支, 然后把 p 指向 p 要进入的分支 (Left _
Subsuffix 或 Right _Subsuffix)

 if(*p. pattern* 与 [*qx...text*] 匹配) then

 报告发现匹配;

 for each q in the list p. Left_subsuffix do

```
begin
    px=q.pattern;
    qx=text-q.pattern.length+1;
    if(q.pattern 与[qx...text]匹配) then
        报告发现匹配;
    end if
end for
end if
if(已经进入过 p 的分支域) then
    break; //结束匹配
end if
end for
shift=1;
end if
text+=shift;
end while
```


第四章 验证和分析

第三章重点阐述了对 Wu-Manber 算法的三个新的改进算法的思路和代码，本章将分别对它们进行实验，并对实验数据进行分析，得出算法的相关结论。

4.1 基于非空公共子后缀的 Wu-Manber 算法的改进

4.1.1 实验环境

本实验对原 Wu-Manber 算法、3.2 节的算法和基于非空公共子后缀的 Wu-Manber 改进算法在匹配中字符串比较的次数和运行效率进行比较，以验证基于非空公共子后缀的 Wu-Manber 改进算法的优势。实验中，称原 Wu-Manber 算法为 WM，3.2 节的算法为 OWM，基于非空公共子后缀的 Wu-Manber 改进算法为 MWM。实验使用的文本 T 是从搜狗实验室下载的文档 SogouT2.reduced.txt (40.9M)，作为模式串的词语从搜狗实验室下载的文档 SogouLabDic.dic (2.37M) 中随机提取。

实验平台为 CPU: P4 3GHz，内存: 192M，操作系统: windowsXP，编译器: Microsoft Visual studio 2005 试用版中的 Visual C++编译器。

4.1.2 实验数据及分析

表 4.1 给出三种算法的实验数据，分别给出了算法匹配过程的字符比较次数、跳跃计算次数、字符比较次数和跳跃计算次数的比例以及 MWM 相对于 WM 和 OWM 的字符比较次数减少的程度。

表 4.1 比较 WM 算法和 MWM 算法在匹配过程中字符比较的次数，分别给出了 MWM 算法相对于 WM、OWM 算法的字符比较次数减少的程度。

该表显示模式从 1000 到 5000，三种算法都能找出全部的匹配。相对于三种算法，都是字符比较次数比跳跃计算次数增长速度快。比如 OWM，模式的规模为 1000 时，字符比较次数和跳跃计算次数的比值为 2.82；而模式的规模为 2000 时，它们的比值上升到 4.28；模式的规模为 3000 时，它们的比值上升到 5.00；模式的规模为 4000 时，它们的比值上升到 5.49；当规模达到 5000 时，他们的比值是 5.69。所以随着模式的增加，字符比较带来的开销所占的比重也随着增大。从上表可以看

出相对于 WM，MWM 在模式的各规模上降低了比值，而且增长速度比 WM 的都低。

图 4.1 是对三种算法在各个规模下匹配时字符比较次数，从图 4.1 可以发现在三种算法中 MWM 算法的字符比较次数是最少的，MWM 的字符比较次数比 OWM 的少的主要原因是由于对公共子串进行汇总后，MWM 的 HASH[h] 的 next 链表的平均长度比 OWM 的小，从而在匹配成功之前遇到的模式比 OWM 更少，因而减少了总的模式比较次数，也就减少了字符比较的次数。

4.1.3 结论

通过上述的实验数据和分析，可得出如下结论：改进后的算法由于采用模式中的非空公共子后缀对 next 链表中的结点进行了较好的分类，减少了 next 链表的平均长度，且在匹配过程中不再需要比较原 WM 算法中相应 next 链表中的所有结点，能比 WM、OWM 更早地结束循环，所以提高了匹配的效率。

模的数目		1000	2000	3000	4000	5000
WM	字符比较次数	115847690	245922178	369779275	500330129	634966767
	跳跃计算次数	29689192	31819496	32518991	32640387	32670340
	字符比较/跳跃计算	3.90	7.73	11.37	15.33	19.44
	得到的模式数目	1744369	3486806	5328190	7168166	8886666
OWM	字符比较次数	83585039	136225146	162627542	179046326	185909995
	跳跃计算次数	29689192	31819496	32518991	32640387	32670340
	字符比较/跳跃计算	2.82	4.28	5.00	5.49	5.69
	得到的模式数目	1744369	3486806	5328190	7168166	8886666
MWM	字符比较次数	51980722	88219866	111764896	131458245	145982078
	跳跃计算次数	29689192	31819496	32518991	32640387	32670340
	字符比较/跳跃计算	1.75	2.77	3.44	4.03	4.47
	得到的模式数目	1744369	3486806	5328190	7168166	8886666
字符比较次数 1-MWM/WM		55.13%	64.13%	69.78%	73.73%	77.01%
字符比较次数 1-MWM/OWM		37.81%	35.24%	31.28%	26.58%	21.48%

表 4.1

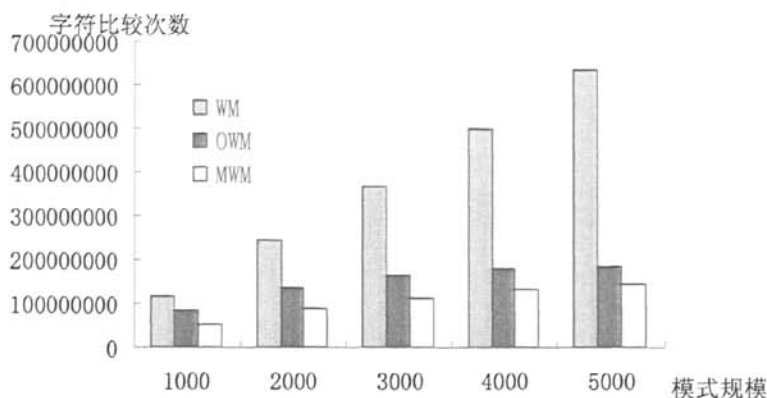


图 4.1

4.2 Wu-Manber 算法的综合改进

4.2.1 实验环境

本实验对原 3.1 节的算法、3.3 节的算法和 Wu-Manber 算法的综合改进算法在匹配中字符串比较的次数和运行效率进行比较, 验证 Wu-Manber 算法的综合改进算法的优势。实验中, 称 3.1 节给出的算法为 LWM, 3.3 节给出的算法为 NWM, Wu-Manber 算法的综合改进算法为 MWM。实验使用的文本 T 是随机生成的一个 36.2M 的文档, 模式串从该文档中随机提取。本实验取 $B=2$, 为了简化算法, 每次实验所提取的字符串都具有相同的长度。

为了实现各个算法的真实模拟, 实验中, NWM 算法和 MWM 算法在找到某一匹配的字符串后, 依然要比较 Same_Suffix 链表中的其他结点; LWM 算法要历遍整个 next 链表。

实验平台为 CPU: P4 3GHz, 内存: 192M, 操作系统: windowsXP, 编译器: Microsoft Visual studio 2005 试用版中的 Visual C++编译器。

4.2.2 实验结果及分析

表 4.2 给出了 NWM 算法、LWM 算法和 MWM 算法在匹配过程中字符比较的次数,

并分别给出了 MWM 算法相对于 LWM、NWM 算法的字符比较次数减少的程度。

模式串数目			50	100	500	1000
NWM	模式串长度 为 5 个字符	字符比较次数	689062	1563697	15073088	40803936
LWM			759990	1657962	13892048	39123133
MWM			657041	1431517	11821221	32926677
(NWM-MWM)/NWM			0.0465	0.0845	0.2157	0.1931
(LWM-MWM)/LWM			0.1355	0.1366	0.1491	0.1584
NWM	模式串长度 为 10 个字符	字符比较次数	355725	925226	14280715	41196442
LWM			390659	976256	12682831	38221777
MWM			337541	844230	10853304	32274923
(NWM-MWM)/NWM			0.0511	0.0875	0.2400	0.2166
(LWM-MWM)/LWM			0.1360	0.1352	0.1443	0.1556
NWM	模式串长度 为 20 个字符	字符比较次数	227764	688781	14095747	39721933
LWM			248765	717735	12848171	39218205
MWM			215901	616334	11032887	33084165
(NWM-MWM)/NWM			0.0521	0.1052	0.2173	0.1671
(LWM-MWM)/LWM			0.1321	0.1413	0.1413	0.1564
NWM	模式串长度 为 30 个字符	字符比较次数	189067	630248	13684148	39676656
LWM			205951	649106	12339179	36984121
MWM			178789	562049	10467648	31139904
(NWM-MWM)/NWM			0.0544	0.1082	0.2351	0.2152
(LWM-MWM)/LWM			0.1319	0.1341	0.1517	0.1580
NWM	模式串长度 为 40 个字符	字符比较次数	158082	574775	13695747	39905030
LWM			176768	604300	12398829	38900244
MWM			149956	516849	10642881	32780645
(NWM-MWM)/NWM			0.0514	0.1008	0.2230	0.1785
(LWM-MWM)/LWM			0.1517	0.1447	0.1416	0.1573

表 4.2

从表 4.2 的数据可以看出 MWM 算法相比 LWM 算法和 NWM 算法减少了匹配过程中字符比较的次数, 减少的程度达到 4.6% 以上, 因此可以认为 MWM 算法的效率有明显的提高。

图 4.2 是从表 4.2 中取模式串长度为 20 个字符的数据作为代表生成的字符比较图。从该图可以清楚地看出 MWM 算法对比前两种算法, 在各个规模的模式串上, 比较次数都有显著的减少。

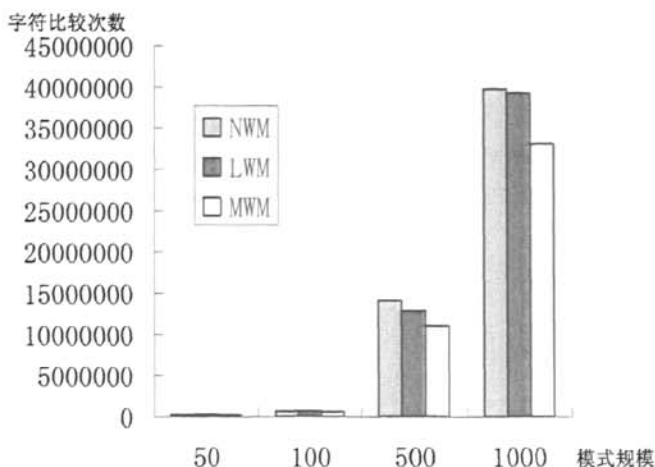


图 4.2

4.2.3 结论

通过上述的实验数据和分析, 可得出如下结论: 改进后的算法由于把 NWM 算法的 next 链表中 Same_Subsuffix 域不为空的结点提到链首, 同时把 LWM 算法和 NWM 算法的相关改进结合起来, 进一步优化了算法, 从而提高了匹配的效率。

4.3 Wu-Manber 算法在大规模模式串下的改进

4.3.1 实验环境

本实验对 3.1 节的算法和 Wu-Manber 算法在大规模模式串下的改进算法在匹配中字符串比较的次数和运行效率进行比较, 验证改进后算法的优势。实验中, 称 3.1

节的算法为 MWM, Wu-Manber 算法在大规模模式串下的改进算法为 LWM。实验使用的文本 T 是随机生成的一个 36.2M 的文档, 模式串从该文档中随机提取。本实验取 $B=2$, 为了简化算法, 每次实验所提取的字符串都具有相同的长度。

实验平台为 CPU: P4 3GHz, 内存: 192M, 操作系统: WindowsXP, 编译器: Microsoft Visual studio 2005 试用版中的 Visual C++编译器。

4.3.2 实验结果及分析

表 4.3 比较 WM 算法和 MWM 算法在匹配过程中字符比较的次数, 分别给出了 LWM 算法相对 MWM 算法的字符比较次数减少的程度。

从表 4.3 的数据可以看出, 新算法 (LWM) 在模式串比较少时匹配效率比原算法 (MWM) 有一定的下降, 但幅度很小, 而随着模式串数目的增加, 新算法表现出比原算法更好的效率。这是因为新的算法对原算法的 Same_Suffix 域分裂出 Left_Suffix 和 Right_Suffix 两个子域, 当模式串小时原算法中的 Same_Suffix 域很小, 新算法需要判断应该进入哪个子域, 效率有一定的损失, 最大时达 0.05%, 但当模式串规模比较大时, 原算法需比较整个 Same_Suffix 域, 而新算法只需比较两个子域中的一个, 所以减少了比较的次数, 提高了算法的效率。

图 4.3 给出了两种算法在各个规模下字符比较次数的柱状图, 从图中可以看出, 在规模较小时两种算法的字符比较次数几乎一样, 而当规模很大时, LWM 算法比 MWM 算法具有更好高的效率。

模式串数目(条)			1000	3000	5000	8000	10000	20000	30000
LWM	模式串长(50 个字符)	比较次数(单位:千次)	39603	129561	207013	307070	366869	593889	737043
MWM			39600	129532	206954	306938	366698	593716	737360
提高			-0.01%	-0.02%	-0.03%	-0.04%	-0.05%	-0.03%	0.04%
模式串数目(条)			40000	50000	60000	70000	80000	90000	100000
LWM	模式串长(50 个字符)	比较次数(单位:千次)	836642	905653	954198	990390	1018273	1038686	1054987
MWM			837861	908281	958591	996869	1027068	1050051	1069095
提高			0.15%	0.29%	0.46%	0.65%	0.86%	1.08%	1.32%

表 4.3

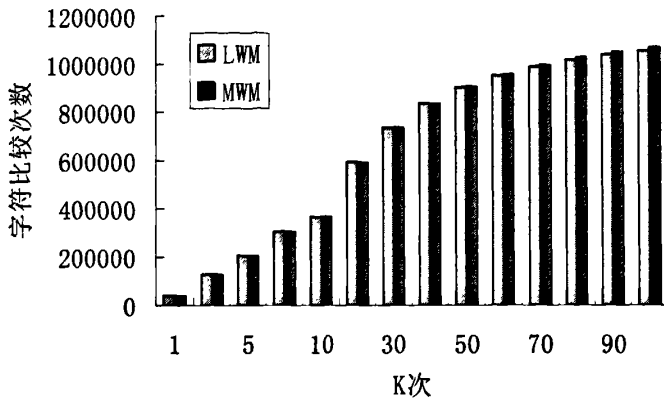


图 4.3

4.3.3 结论

通过上述的实验数据和分析,可得出如下结论: Wu-Manber 算法在大规模模式串下的改进算法由原算法的 Same_Suffix 域分裂出 Left_Suffix 和 Right_Suffix 两个子域,当模式串规模比较小时,跟原算法的效率相比,新算法的效率有一定的损失,但在可以接受的范围之内,当模式串规模比较大时,匹配不再需要比较原 WM 算法中相应 Same_Suffix 域中的所有结点。新算法的效率随着模式串数量增加而不断得到提高,从而新算法比原算法有更大的适用范围。

总结和展望

字符串匹配是计算机领域的一个古老而经典的问题。对字符串匹配算法的研究有积极的现实意义。

本文主要对其中的高效算法——Wu-Manber 算法进行研究。在前人研究成果的基础上, 本文取得如下的研究成果:

- 1、基于非空公共子后缀的 Wu-Manber 算法的改进: 针对 Wu-Manber 算法在处理公共子后缀模式情况下的不足, 本文提出了一种基于非空公共子后缀模式的处理算法。该算法把同一 next 链表有非空公共子后缀的模式串汇集在一起, 进一步减小了 next 链表的平均长度。在匹配过程中减少了字符比较的次数, 从而提高算法的运行效率。本文对搜狗实验室给出的相关文档进行全文检索实验, 并和原 Wu-Manber 算法、前人提出的改进算法进行比较。实验结果表明, 本文提出的改进算法有效地减少了匹配过程中字符比较的次数, 从而提高匹配的速度和效率。
- 2、对 Wu-Manber 算法的综合改进: 对 1 中提出的算法进行了改进, 在对 next 链表进行分类的同时把含有非空公共子后缀的结点提到链表的前部; 并整合了前人提出的“精确的不良字符转移和弱化的良好后缀转移的改进”方法。新改进的算法充分利用以上两种算法的优点, 使匹配过程中字符比较的次数得到了进一步减少。新改进的 Wu-Manber 匹配算法在实验中取得了更高的效率, 比原算法提高到 4.6% 以上。
- 3、Wu-Manber 算法在大规模模式串下的改进: 对 1 中提出的算法进行了改进, 把原算法中 next 链表中结点的 Same_Suffix 域中分裂成两个子域, 使得搜索过程中字符比较的次数进一步减少, 从而提高算法的效率。特别是在大规模模式串的情况下新算法的效率比原算法有进一步的提高。实验结果表明, 当模式串较少时, 新算法效率与原算法相比有一定的损失。而随着模式串的增加, 新算法具有更高的效率。因此, 新的算法比原算法具有更大的适用范围。

由于时间和精力有限, 还有如下相关的问题有待解决:

- 1、在多核 CPU 情况下对 Wu-Manber 算法的研究;
- 2、把相关改进的 Wu-Manber 算法应用于 snort 系统;
- 3、把 Wu-Manber 算法与其他匹配算法相结合的研究; 等等。

参考文献

- [1] 刘燕兵. 串匹配算法的优化, 硕士毕业论文, 中国科学院计算技术研究所, 2006. 6
- [2] L. Me L., Heye J. Kuri, G. Navarro. A pattern matching based filter for audit reduction and fast detection of potential intrusions. In proceedings of the 3rd International Workshop on the Recent Advances in Intrusion Detection(RAID 2000), 2000.
- [3] g. Navarro J. kuri. Fast multipattern search algorithms for intrusion detection. Technical Report TR/DCC-99-11. Dept. of Computer Science, Univ. of Chile, 1999.
- [4] G. Varghese M. Fst. Applying fast string matching to intrusion detection. 2002.
- [5] J. Mcalderney C. J. Coit, S. Staniford. Towards faster string matching for intrusion detection or exceeding the speed of snort. In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II), 2002.
- [6] <http://www.snort.org>.
- [7] <http://www.chinaunix.net/>.
- [8] <http://baike.baidu.com/>.
- [9] <http://tech.sina.com.cn/>.
- [10] L. Falquet A. Bairoch K. Hofmann, P. Bucher. The prosite database. Nucleic Acids Research, 27(1):215-219, 1999.
- [11] R. Giancarlo G. F. Italiano D. Eppstein, Z. Galil. Efficient algorithms for sequence analysis. Proc. Sequences II: Combinatorics, Compression, Security, Transmission, pages 225-244, 1991.
- [12] M. Raffinot G. Navarro. Fast and simple character classes and bounded gaps pattern matching with application to protein searching. Proceedings of the fifth annual international conference on Computational biology, pages 231-240, 2001.
- [13] 中国互联网信息中心(CNNIC). 中国互联网络带宽调查报告。

<http://www.cnnic.net.cn/>.

- [14] G. Navarro and M. Raffinot. Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences. 2002.
- [15] T. Lecroq C. Charras. Handbook of Exact String Matching Algorithms. Feb. 2004.
- [16] Gonzalo Navarro and Mathieu Raffinot. Flexible Pattern Matching in strings. (柔性字符串匹配[M], 中科院计算所网络信息安全研究组译. 北京电子工业出版社 2007.3 ISBN 978-7-121-03858-7).
- [17] G. Navarro and M. Raffinot. Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences. 2002.
- [18] J. H. Morris D. E. Knuth and V. R. Pratt. Fast pattern mathing in strings. SIAM J. Comput. 6(1):322-350, 1977.
- [19] V. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18:333-340, 1975.
- [20] R. S. Boyer and J. S. Moore. A fast string searching algorithm. Communications of the ACM, 20:762-772, 1977.
- [21] R. N. Horspool. Practical fast searching in strings. Software Practice and Experience, 10(6):501-506, 1980.
- [22] C. Walter. A string matching algorithm fast on the average. Proc. 6th International Colloquium on Automata, Languages, and Programming, pages 118-132, 1979.
- [23] W. Sun. A fast algorithm for multi-pattern searching. Technical Report Department of Computer Science Chung-Cheng University.
- [24] S. Wu and V. Manber. A fast algorithm for multi_pattern searching[R]. Report TR-94-17, Department of Computer Science. University of Arizona, Tucson, AZ, 1994.
- [25] M. Raffinot. On the multi backward dawg matching algorithm(multibdm). Proceedings of the 4th South American Workshop on String Processing, pages 149-165, 1997.

- [26] Tan Jian-long Liu Ping and Liu Yan-bing. A partion-based efficient algorithm for large scale multiple-stringt matching. SPIRE 2005, 2005.
- [27] 贺龙涛, 方滨兴, 余翔湛. 一种时间复杂度最优的精确串匹配算法. 软件学报, 16(5):676-683, 2005.
- [28] J. Tarhio J. Kytojoki, L. Salmela. Tuning string matching for huge pattern sets. Combinatorial Pattern Matching, pages 211-224, 2003.
- [29] Marc Norton. Optimizing pattern matching for intrusion detection. <http://www.idsresearch.org>, 2004.
- [30] B. Calder N. Tuck, T. Sherwood and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. IEEE INFOCOM, 2004.
- [31] S. Fukamachi T. Nishmura and T. Shinohara. Speed-up of aho-corasick pattern matching machines by rearranging states. SPIRE 2001, pages 175-185, 2001.
- [32] R. H. Katz F. Yu and T. V. Laskhman. Gigabit rate packet pattern matching with tcam. UCB technical report, 2004.
- [33] K. ChiaNan L. RongTai, H. NenFu and C. ChihHao. A fast pattern matching algorithm for network processor-based intrusion detection system. International Conference on Information Technology: Coding and Computing, 271-275, 2004.
- [34] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. Proc. 32nd Annual International Symposium on Computer Architecture (LISA), 2005.
- [35] 张鑫, 等. 一种改进的 Wu-Manber 多关键词算法[J]. 计算机应用, 2003, 23 (7): 29-31
- [36] 孙晓山, 等. 一种改进的 Wu-Manber 多模式匹配算法及应用[J]. 中文信息学报, 2006. 20 (2): 47-52.
- [37] C. Allauzen, M. Crochemore , and M. Raffinot. Efficient experimental string matching by weak factor recognition. In Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, number 2089 in Lecture Notes in Computer Science, pages 51-72. Springer-Verlag, 2001.

- [40] 陈瑜, 陈国龙. Wu-Manber 算法性能分析及其改进. 计算机科学, 2006 Vol. 33(6)
- [41] 杨东红, 等. 改进的 Wu-Manber 多模式串匹配算法. 清华大学学报 (自然科学版) 2006. 46(4).
- [42] 刘燕兵, 等. 可动态增删关键词的串匹配算法. 计算机工程与应用. 2005. 35
- [43] 唐谦, 张大方. 入侵检测中模式匹配算法的性能分析. 计算机工程与应用. 2005. 17
- [44] 李雪莹, 刘宝旭, 许榕生. 字符串匹配技术研究. 计算机工程. 2004. 22
- [45] 甘学士, 孙力娟. 改进的模式匹配算法及在入侵检测中的应用. 计算机技术与发展. 2006. 16(7)
- [46] 宋明秋, 等. 入侵检测多模式匹配算法. 计算机工程. 2006. 32(5)
- [47] 李红霞, 等. 一种应用于入侵检测的基于排除的模式匹配算法. 电子技术应用. 2006. 1.
- [48] 李晓秋, 等. 入侵检测系统中的快速多模式匹配算法. 计算机应用于软件. 2004. 21(2)
- [49] 伊静, 刘培玉. 入侵检测中模式匹配算法的研究. 计算机应用与软件. 2005. 22(1)
- [50] 王成, 刘金刚. 一种改进的字符串匹配算法. 计算工程. 2006. 32(2)
- [51] James Allen 著, 刘群等译. 自然语言理解, 电子工业出版社, 2005: 31-45
- [52] 傅清祥, 王晓东. 算法与数据结构, 北京: 电子工业出版社 1998
- [53] 陈开渠, 等. 快速中文字符模糊匹配算法, 中文信息学报, 2004, 18 (2) 58-65。
- [54] 王素琴, 邹旭楷. 一种优化的并行汉字/字符串匹配算法, 中文信息学报, 1995, 19 (1): 49-53
- [55] Franklin F, Carver D, Hutchings B .Assisting network intrusion detection with reconfiguration hardware. Ln: Pocek KL, ed. Proc. Of the IEEE Symp. On Field Programmable Custom Computing Machines, Los Alamitos: IEEE computer Society, 2002, 111 – 120.
- [56] Moscola J , Lockwood J , Loui RP, Pachos M , Implementation of a content Scanning module for an Internet file wall, In: Pocek KL, ed . Proc. Of the 11th Annual IEEE Symp. On Field Programmable custom computing Machines. Los Alamitos: IEEE Computer Society, 2003 31-38.
- [57] L. Me L., Heye J. Kuri, G. Navarro. A pattern matching based filter for audit reduction and fast detection of potential intrusions. In proceedings of the 3rd International Workshop on the Recent Advances in

Intrusion Detection(RAID 2000),2000.

- [58] G.Navarro J.kuri. Fast multipattern search algorithms for intrusion detection. Technical Report TR/DCC-99-11. Dept. of Computer Science, Univ. of Chile,1999.
- [59] Muth R, Manber U. Approximate Multiple String Search, In: Proc, 7th Combinatorial Pattern Matching (CPM, 96). LNCS 1075, 1996, 75 - 86.
- [60]Crochemore M, Czumaj A , Gasieniec L , Lecroq T, Plandowski W , Rytter W .Faster Practical multi-pattern matching Inf. Process. Leu, 1999, 71 (3-4): 107 - 113.
- [61] Wu Sun Manber U GLIMPSE: A Tool to Search Through Entire Filesystem. Usenix Winter Technical Conference, San Francisco, 1994.
- [62] Aho A V , Corasick M J , Efficient string matching, an aid to bibliographic search , Communication of ACM,1975, 18(6): 333 - 340.
- [63] Frank Stomp. Correctness of substring preprocessing in Boyer Moore pattern matching algorithm [J], Theoretical Computer Science, 2005, 29(1): 59 - 78 .
- [64] Tarhio J. Peltora H. String matching in the DNA alphabet [J]. Software - Practice and Experience, 1997, 27(7): 851 - 861.
- [65] Cole R. Tight bounds on the complexity of the Boyer Moor pattern matching, angorithm [J], SIAM Journal on Computing, 1994, 23(5): 1075 - 1091.

发表文章目录

- 1、莫德敏，刘耀军，对 Wu-Manber 算法的一种改进，已投修改稿到中文信息学报。
- 2、莫德敏，刘耀军，Wu-Manber 算法的综合改进，太原师范学院学报（刊号 ISSN 1672-2027，CN14-1304/N，季刊），2008 年第 2 期。
- 3、莫德敏，刘耀军，Wu-Manber 算法在大规模模式串下的改进，晋中学院学报（刊号 CN14-1327/Z，ISSN 1673-1808，双月刊），2008 年第 3 期。

致 谢

首先，衷心地感谢我的导师刘耀军教授！本论文得以完成，离不开他的无微不至的关怀和悉心指导。刘老师严谨的治学态度令人钦佩，自由、开放的学术态度让我在艰辛的研究工作中得以充分发挥，真正的获益良多！

非常感谢陈桂芳教授等几位副导师对我的关心和照顾。他们的创新意识和对研究的执着精神无时无刻不激励着我，是我研究路上的导航灯。

感谢同学唐群晖、孙抗震、孟士伟、蒋天刚等。风格各异的他们让我在学习生活中享受着无穷的快乐。

感谢我的师弟、师妹，他们的独特见解常常让我的研究柳暗花明。

最后我要特别感谢我的父母和亲人多年来对我的关心和支持。我能走过最黑暗的时刻完全归功于他们！他们的期望和信任是我研究路上最大的推手；也是我今后工作、生活的最大动力！

个人简况及联系方式

个人简况

姓名：莫德敏

性别：男

籍贯：广西马山县

简历：1995.9-1999.7 中南民族大学 计算机软件与理论 本科生

1999.7-2005.7 马山县邮政局 技术人员

2005.9-2008.6 太原科技大学 计算机软件与理论 研究生

联系方式

E-Mail: modemin1@163.com

对串匹配技术中的Wu-Manber算法的研究

作者: [莫德敏](#)
学位授予单位: [太原科技大学](#)

本文读者也读过(2条)

1. [孙晓山](#), [王强](#), [关毅](#), [王晓龙](#), [SUN Xiao-shan](#), [WANG Qiang](#), [GUAN Yi](#), [WANG Xiao-long](#) 一种改进的Wu-Manber多模式匹配算法及应用[期刊论文]-[中文信息学报](#)2006, 20(2)
2. [王喜聪](#) [入侵检测系统snort下的模式匹配算法研究](#)[学位论文]

本文链接: http://d.wanfangdata.com.cn/Thesis_Y1373606.aspx