基于有序二叉树的多模式匹配算法*)

A Multiple Pattern Matching Algorithm Based on Sequential Binary Tree

胡佩华　王永成　刘功申

(上海交通大学电子信息学院　上海200030)

Abstract By analyzing the multiple pattern matching algorithm based on tree structure, a multiple pattern matching algorithm based on sequential binary tree is proposed in this paper. It is proved by experiment that the algorithm has three features: its constructing process is quick. Its cost of memory is small. At the same time, its searching process is as quickly as the traditional algorithm. The algorithm proposed in this paper is suit for the application whose pattern set is changing dynamically, that is to say, it is suit for the application whose automata must be constructed dynamically. So, the algorithm has a good application prospect.

Keywords Multiple pattern matching, DFSA, Sequential binary tree

一、简介

在一个文本串中查找用户指定的模式串在信息抽取和文本编辑中有着广泛的应用。当前,有限状态自动机(DFSA)算法是解决多模式匹配问题的常用方法[1,2]。DFSA 算法在匹配前对模式串集合进行预处理,转换成树型有限状态自动机,然后只需对文本串进行一次扫描就可找出所有模式串,其查找时间复杂度是 $O(n)$。后来,在这个算法的基础上又有一些改进,实现了跳跃式查找[3]。基于树型结构的有限自动机特别适用于模式串集合相对稳定的情况。例如,词典是一个相对稳定的模式串集合。只需在应用程序中构造一次,就可以随时在应用程序中使用。

在实际应用中,有一些新的问题逐渐受到重视:在线更新的模式串集合查找的需求(实现编辑软件中多字符串的查找和替换功能);内存节约的需求(使算法方便应用于 PDA、潜入式系统软件等);同时要求匹配速度不能降低。然而基于树型结构的有限自动机由于构造速度慢、内存空间耗费大的缺点,不能很好地满足这些要求。

本文首次提出有序二叉树(sequential binary tree)的概念(详见定义4),并用有序二叉树来代替树型结构实现了一种新的多模式匹配算法——基于有序二叉树的多模式匹配算法(以下简称 SMA 算法)。实验证明,采用本文方法有下列好处:构造速度快;便于动态增删节点;不用额外的转向、失败和输出表;同时查找效率和传统算法一样高。

二、传统算法的缺点

传统算法的描述详见文[1]。为描述方便举例如下:给定模式串集合{he, hers, his, hour, she, our },其对应的树型结构如图1所示。

在树结构中,如果一个节点有 n 个子树,那么该节点就应有 n 个指针分别指向这 n 个子树。在树的构造过程中,由于不能预先知道每个节点有几个子节点,因此只能采取两种处理方法。第一种方法是每次增加子节点时,重新分配父节点的内存空间。这种方法造成频繁的内存释放和分配操作,使树的构造速度大大降低。第二种方法是每个节点都分配 m(m 是所有节点子树个数的最大值)个指针空间。这种方法会造成内存的极大浪费。

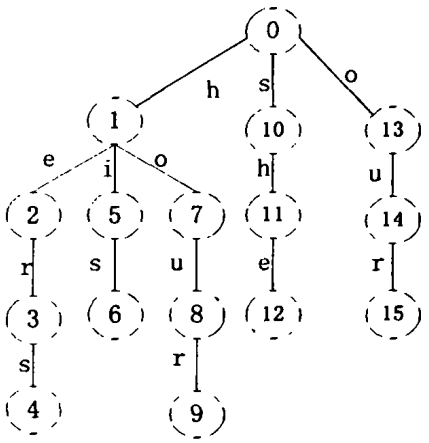

图1　树型结构的有限自动机

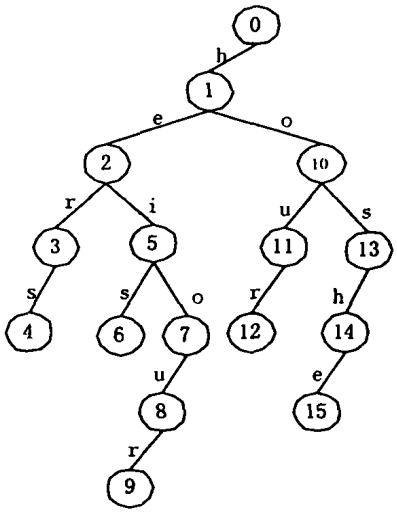

图2　基于有序二叉树结构的有限自动机

*)本课题由国家自然科学基金(60082003)和科技部中小企业创新基金资助。胡佩华　硕士研究生,研究方向:中文信息处理和网络信息智能处理。

传统算法的转向、失败和输出表也需要大量的空间。它们的大小都与树中节点个数成正比例。

传统算法在构造过程中，没有对模式串进行排序。如果给定的模式串集合中的模式串不是按字典序排列的，构造出的树也不会排序。不排序的情况会影响查找阶段的速度[5]。给定例子中，she 和 our 的顺序不是按字典序升序排列的。

三、SMA 算法的优点

为了克服传统算法的缺点，SMA 算法采用有序二叉树来代替树型结构，实现了多模式匹配算法。给定模式串集合{he, hers, his, hour, she, our }，SMA 算法对应的有序二叉树结构如图2所示。采用本文方法有下列好处：

1. 提高构造速度　二叉树只有左右两棵子树，实现时不用预测有几个子节点，避免了内存的浪费和频繁的分配释放操作。

2. 提高查找速度　由于有序二叉树结构体现了模式串的字典序，状态转向的速度得到了提高。

3. 便于动态增删模式串　采用二叉树结构使节点的增删更加方便。所以，本文算法易于实现模式串的增删。

4. 不用额外的转向、失败和输出表　本文算法用指针代替了额外的转向、失败和输出表，节省了内存开销。

四、有序二叉树的构造

4.1　相关定义

图2是根据模式集合{he, hers, his, hour, she, our }构造的有序二叉树。根据一定的规则访问二叉树，从根到某一个叶子节点的过程就可以得到一个相应的模式串。访问规则描述如下：

定义1(访问规则)　栈 s 用来存放模式串。指针 p 用来跟踪节点。当 p 指向当前节点的右子树时，s 的栈顶元素出栈，右树枝上的元素入栈。当 p 指向当前节点的左子树时，左树枝上的元素直接入栈。重复上述过程直到 p 指向叶子节点。此时，栈内的元素就是某一个模式串。例如，访问模式串 our 的过程如下：

第一步：s=Φ; p=0;(根节点)

第二步：s={h}; p=1; (左子树)

第三步：h 出栈，o 入栈，s={o}; p=10; (右子树)

第四步：u 入栈，s={o, u}; p=11; (左子树)

第五步：r 入栈，s={o, u, r}; p=12; (左子树)

算法遇叶子节点结束。此时栈 s 中的字符就对应模式 our。

定义2(状态深度)　节点的状态深度和二叉树的节点深度的定义不同。状态深度反映了字符在相应的模式串中的位置。状态深度可以递规定义如下：

根节点的状态深度为0;

如果某节点的状态深度是 h，则它的左儿子节点的状态深度是 h+1；右儿子节点的状态深度为 h。

依次类推可以确定所有节点的状态深度。

根据定义，可以计算出本文例子的每个节点的状态深度。图3示意了各节点的状态深度。圆圈内的阿拉伯数字表示该节点的状态深度。

定义3(父、子状态节点)　如果节点 l 是节点 f 的左子树，节点集合 R={r|r 是 l 的右子树上的节点并且具有和 l 相同的状态深度}，则 f 是 l 和 R 中所有节点的父状态节点，l 和 R 中所有节点是 f 的子状态节点。图3示意了父子状态节点的关系。带箭头的点线描述了状态节点的父子关系，箭头端的节点是末端节点的父状态节点，末端节点是箭头端节点的子状态节点。

定义4(有序二叉树)　对二叉树按 NLR 方式遍历，应用本文访问规则可以得到一系列的模式串。如果这些模式串得到的先后顺序和它们的字典序相同，那么这棵二叉树就是有序二叉树。图2就是一棵有序二叉树。

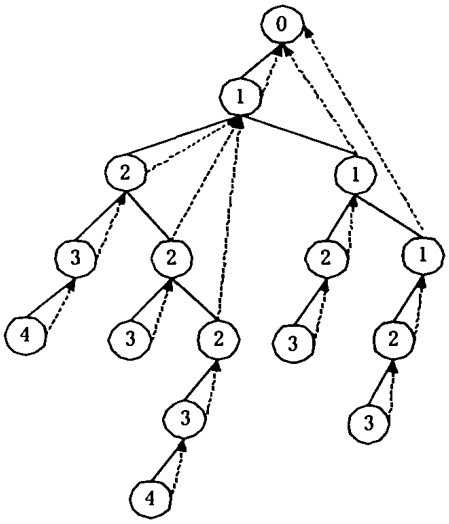

图3　状态深度和它们父子关系

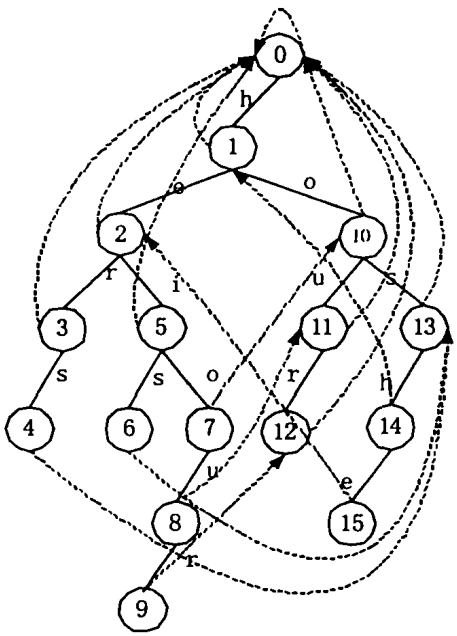

图4　带有失败指针的有序二叉树

4.2　有序二叉树的构造

给定一个模式串集合(各模式一般不是按字典序排列的)，本文首要的任务就是构造一个有序二叉树。在构造的过程中完成以下工作：构造整个有序二叉树；确定输出节点和确定每个节点父状态节点指针；在构造中对模式串进行排序，以提高查找时的速度。算法如下：

节点的数据结构：

```
Structure Node{
    Node * Lchild;
    Char Lchar;
```

```
    Node * Rchild;
    Char Rchar;
    Node * fatherstate;
    Node * failstate;
    Boolean output;
};
```

算法1　有序二叉树的构造

```
输入:模式串集合
输出:有序二叉树、输出节点和父状态节点指针
Begin
  for each pattern do{
    p=root; i=0;
    while( (p=goto(p, pattern[i]))!=NULL ) i++;
    在 p 处插入相应的 pattern[i:strlen(pattern)];
  }
End.
```

定义5(goto(state,character)函数)　根据访问规则从状态节点 state 出发到其任一子状态节点的过程中,如果到达了子状态节点 child 并且此时栈中有且仅有字符 char,则 goto(state, char)=child,否则 goto(state, char)=NULL。算法的描述如下:

算法2　goto(p, char)

```
输入:节点 p, 字符 char
输出:相应的子状态
Begin
  if ((char<p. Lchar) || (p. Lchild==NULL) ){
    return NULL;
  }else if (char==p. Lchar){
          return p. Lchild
    }else{
      p=p. Lchild;
       while ((pattern[i]>p. Rchar) && (p! = NULL) ) p =
            p. Rchild;
      if (pattern[i]==p. Rchar)
          return p. Rchild;
      else
          return NULL;
    }
End.
```

4.3　失败指针的确定

失败指针表示:在状态 state 时,如果 goto(state, char) = NULL(也就是说,从状态 state 经过字符 char 不可能到达 state 的任一个子状态节点),当前状态应改为 state. failstate。下面是失败指针的处理原则:

·根节点的失败指针指向根节点;

·状态深度为1的节点的失败指针指向根节点;

·对于状态深度大于等于2的节点 s,若其父状态为 r 且 goto(r, a)=s。经过下列运算:

```
while (goto((s. fatherstate). failstate, a)==NULL) s
        = s. fatherstate;
```

则 s 的失败指针指向状态节点 s. fatherstate。

如图4所示,失败指针用点线标出。每一个状态节点都有一个指针指向一个失败状态节点。例如,状态节点2的失败指针指向状态节点0。标注失败指针的算法见算法3。

算法3　失败指针的确定

```
输入:有序二叉树,根节点是 root;
输出:标注了失败指针的右序二叉树
Build_Fail_Func(Structure Node s)
Begin
    按上述原则标注节点 s 的失败指针;
    Build_Fail_Func(s. Lchild);
    Build_Fail_Func(s. Rchild);
End
```

4.4　节点的增、删处理

在实践中发现,有时需要对模式串集合进行少量的变化,增加某些模式串到集合中或者从集合中删除某些模式串。这些轻微的变化不需要重新对整个有序二叉树进行构造,因此,节点的增删操作是非常必要的。节点增删后,失败指针也应作相应的变化。

在状态深度为 h 的节点上增加或删除子节点后,状态深度大于 h 的节点的失败函数都有改变的可能。也就是说,节点增删后需要对部分节点的失败指针进行重新计算。

五、查找阶段

5.1　输出匹配结果

在搜索过程中,当遇到 output 为真值的节点 s 时,按访问规则从根到状态节点 s 访问,并输出栈中的字符作为结果。令状态节点 f 是状态节点 s 的失败指针指向的节点,如果 f. output 为真值时,同时按访问规则从根到状态节点 f 访问,并输出栈中的字符作为结果。由于输出需要一步回溯,对查找速度有一定的影响。

5.2　搜索算法的描述

有序二叉树构造完成以后,就可以方便地从字符串中经过一次扫描查出一切与给定模式串集中任何模式串相同的子字符串。查找的过程如下:从有序二叉树的根节点出发,逐个取出文本串中的字符,根据 goto 函数和失败指针确定下一个状态节点。当某状态节点的 output 域为真值时,输出结果。以本文构成的有序二叉树为例,查找文本串"ushers"的过程如下:

从根 root 开始,goto(root, u)=0; goto(0, s)=13; goto(13, h)=14; goto(14, e)=15。此时输出节点15,也就是输出 she,同时输出节点2,也就是输出 he。

接下来,状态节点15的失败指针指向状态节点2。goto(2, r)=3; goto(3, s)=4。此时输出状态节点4,也就是输出 hers。此时查找过程结束。

上述过程可以形式地用算法4描述。

算法4　模式匹配算法

```
输入:文本串 string=a1a2…an. 根为 root 的有序二叉树。
输出:各个模式以及它们在文本 string 中出现的位置
p ← root;
for i ← 1 until n {
    while ( (p=goto(p, ai )) == NULL ) do p <-- p. failstate;
    if (p. output){
        print i; print p;
        if(p. failstate. output)
            print p. failstate
    }
}
```

六、算法的分析

6.1　空间复杂度

与传统的多模式匹配算法相比,由于 SMA 算法采用了有序二叉树,不需要转换、失败和输出表需要的额外的存储空间。本文算法所需的空间仅仅是二叉树节点所占空间。有序二叉树的节点总数跟两个因素有关:一个是模式串集合中所有模式长度的总和,另一个是各个模式之间的共同前缀的多少。因此,有序二叉树最终的节点总数不能通过形式分析求出。

6.2　时间复杂度

时间复杂度的分析分构造和查找两个阶段进行。为了描述方便,设模式串集合的模式个数为 k。有序二叉树的节点个数为 m。待搜索文本串的长度为 n。

·构造阶段

算法1的时间复杂度主要由外层循环确定，所以时间复杂度为O(k)。

确定各个节点失败指针的算法参见算法3。算法3的主要过程是对整个有序二叉树进行NLR遍历，所以其时间复杂度为O(m)。

总之，构造阶段的时间复杂度为O(m+k)。

·查找阶段

查找阶段和传统的多模式匹配基本相同。

算法4的外层循环主要由文本串的长度n确定。每一次循环的时间主要消耗在算法2上。下面分析算法2的时间复杂度。

从父状态节点到某一子状态节点的过程主要由子状态节点的个数决定。例如，从状态节点0到各个子状态节点(1,10,13)需要的搜索的路径数分别为(1,2,3)。设各个状态节点的平均子状态节点数为h，由于搜索路径上的字符是从小到大排序的，因此算法2时间复杂度为O(h/2)[4]。h的上限是语言的字母表的大小，英文在忽略大小写的情况下h<=26。

综上所述，查找阶段的时间复杂度为O(hn/2)。

七、对比试验

实验选择的样本包括模式串集合和文本两部分。模式串集合包括NAME词典(含8843个英文人名)、DNS词典(含1871个互联网网址后缀)和ABBR词典(含1470个英文缩写词)。待搜索文本是从互联网下载的英语新闻，包括10k字节、20k字节、40k字节和80k字节四个样本。表1是两种算法自动机构造时间对照表，如表1所示本文算法的构造时间比普通算法快一倍，非常适用于模式串集合动态变化的情况。另外，本文分别用三种词典对四个文本进行搜索，取平均时间作为查找时间。如表2所示，两种算法的搜索效率是基本相同的，传统算法比本文算法稍微慢一点。可见本文算法并没有因为存储结构的改变而影响搜索效率。

表1 构造时间对照表

算法 \ 词典	NAME	NAME DNS	NAME DNS ABBR
传统算法(T/ms)	270	320	440
SMA算法(T/ms)	110	130	150

表2 查找速度对照度

算法 \ 长度	10K	20K	40K	80K
传统算法(T/ms)	14	30	53	115
SMA算法(T/ms)	15	20	50	100

结论 为了适应动态构造自动机的需要，本文首次提出采用有序二叉树结构来组织自动机。实验证明，该方法具有相当高的构造速度，并且对文本串的查找速度和传统算法基本相当。因此，SMA算法具有很好的应用前景。

SMA算法的查找过程中必须检查每一个字符，如果和Boyer-Moore算法[6]或Quick Search算法[7]等高效单模式匹配算法相结合可以实现跳跃式查找，从而可以进一步提高算法的查找速度。另外，对于大字符集语言(例如汉语)，由于根节点的子状态节点数量很大，因此第一层状态节点可以用hash技术来组织。也就是构造一个由有序二叉树组成的森林。这种组织方法可以进一步改进查找速度。

参考文献

1 Aho A V, Corasick M J. Efficient string matching: An aid to bibliographic search. Commun. ACM, 1975, 18(6): 333～340

2 Lewis H R, Papadimitriou C H. Elements of the theory of computation (Second Edition). Prentice-Hall International, Inc. 1998

3 Fan J-J, Su K-Y. An efficient algorithm for match multiple patterns. IEEE Trans. on Knowledge and Data Engineering, 1993, 5(2): 339～351

4 Shaffer C A. A practice introduction to data structures and algorithm analysis. New York Prentice Hall, 1997

5 Chen Guilin. Some Technology Research in Automatic Abstract: [PH. D Paper]. Shanghai Jiaotong University, Shanghai China, 2000, 4 (In Chinese)

6 Boyer R S. Moore J S. A fast string searching algorithm. Commun. ACM, 1977, 20(10): 762～772

7 Sunday D M. A very fast substring search algorithm. Commun. ACM, 1990, 33(8): 132～142

(上接第127页)

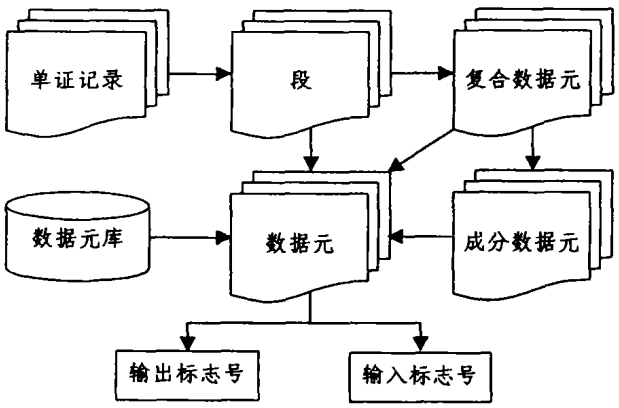

图6

和标准单证数据项之间的对应关系。这一步的关键就是建立对应的数据元库。标准单证的数据元库可以根据国际标准和国家标准建立，企业单证的数据元库则要按照需要建立。

基本思路是根据标准的数据结构检索出转换单证数据项最终数据元(最小元素)的标志号(唯一的)，通过事先建立好的数据元的对应关系(以标志号作关联)，检索到目标单证对应的数据元，从而实现数据项的转换。

结束语 由于XML语言引入，使得建立EDI变得更加容易，其应用也变得更为广泛。显然，XMLEDI的应用和推广必将促进电子商务的进一步实质性发展。因此，建立XMLEDI不是目的，因为它只是推进国民经济信息化和社会信息化的一个基础平台和技术手段。

参考文献

1 杨坚争，杨晨光著. 电子商务基础与应用(第三版). 西安电子科技大学出版社，2001

2 灯芯工作室编著. 用XML轻松开发Web网站(第一版). 北京希望电子出版社，2001

3 杨寅华，蒋忠仁，束剑红，李翔. EDI编译器的研究与开发. 华东师范大学学报(自然科学版)，2001(1)

4 李玲，蒋励. EDI在Internet上的应用. 西安邮电学院学报，2001，6(1)

5 http://www.xmledi.org