

计算机应用编程实验

多模式字符串匹配

熊永平@网络技术研究院

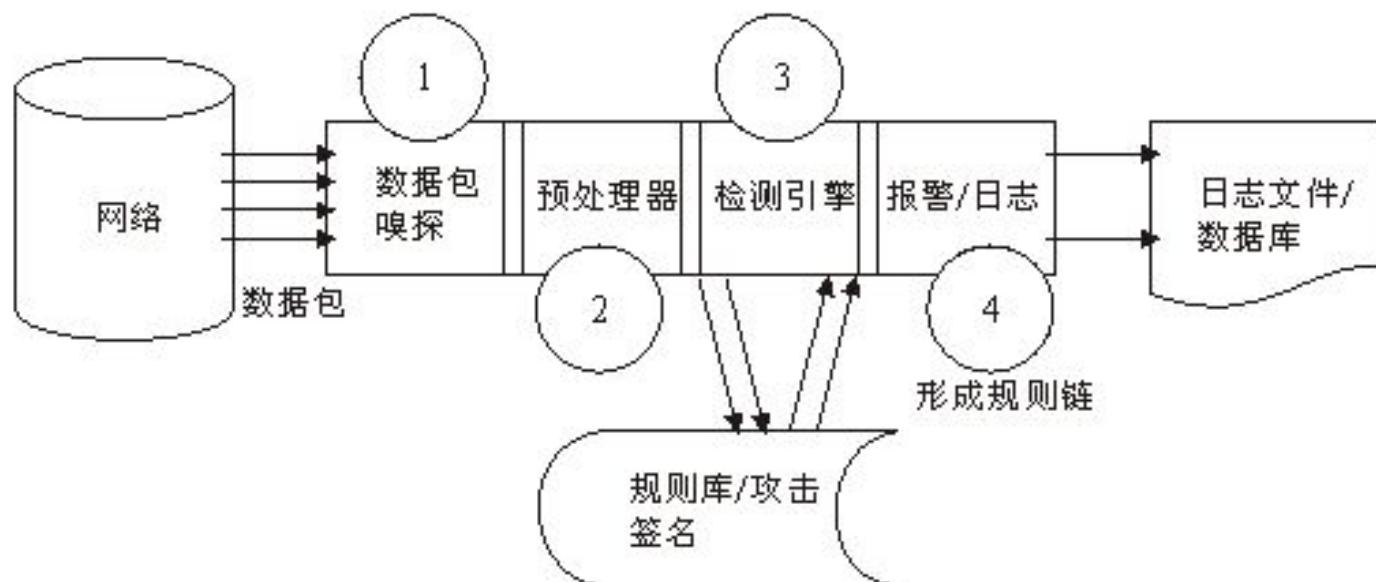
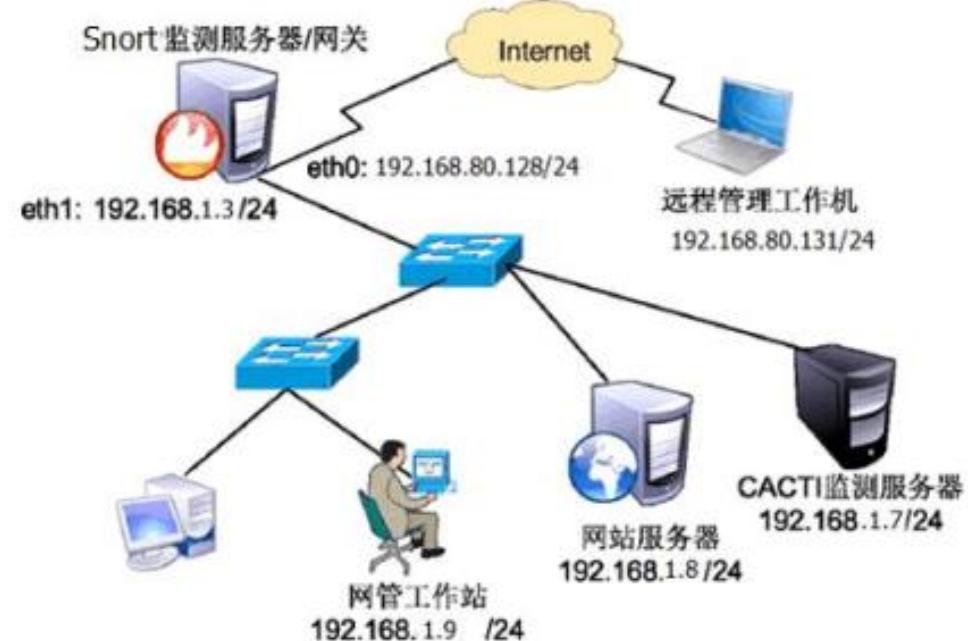
ypxiong@bupt.edu.cn

教三楼233

2019.11

实验背景

- 检测引擎中包含几千万条特征字符串规则，如何在10G网络流量环境下检测网络流中的恶意软件特征？



实验意义

- **高性能串匹配**
 - 在计算机科学领域，串的模式匹配算法一直都是技术热点之一。
 - 在拼写检查、语言翻译、数据压缩、搜索引擎、网络入侵检测、计算机病毒特征码匹配以及DNA序列匹配等应用中，都需要进行串匹配。
 - 串匹配就是在主串中查找模式串的一个或所有出现。
- **关键概念**
 - SQL: LIKE关键词
 - 单模匹配
 - 多模匹配

实验任务

- 问题

- 给定测试大规模中文文本串，数据量约800M，目标测试中文文本数据约3G
- 待查找的key大约220万个词作为模式串，从搜狗新闻语料训练
- GB2312编码格式
- 需要从大文本串中查找出所有模式串出现的位置和次数

- 挑战

- 模式串的规模较大，需要构造一个高效数据结构来处理

主串

BBC ABCDAB ABCDABCDABDE

模式串

ABCDABD



strstr():朴素的串匹配算法

首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。



BBC ABCDAB ABCDABCDABDE

ABCDABD

因为B与A不匹配，搜索词再往后移。

首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。



BBC ABCDAB ABCDABCDABDE

ABCDABD

就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。
接着比较字符串和搜索词的下一个字符。

BBC ABCDAB ABCDABCDABDE

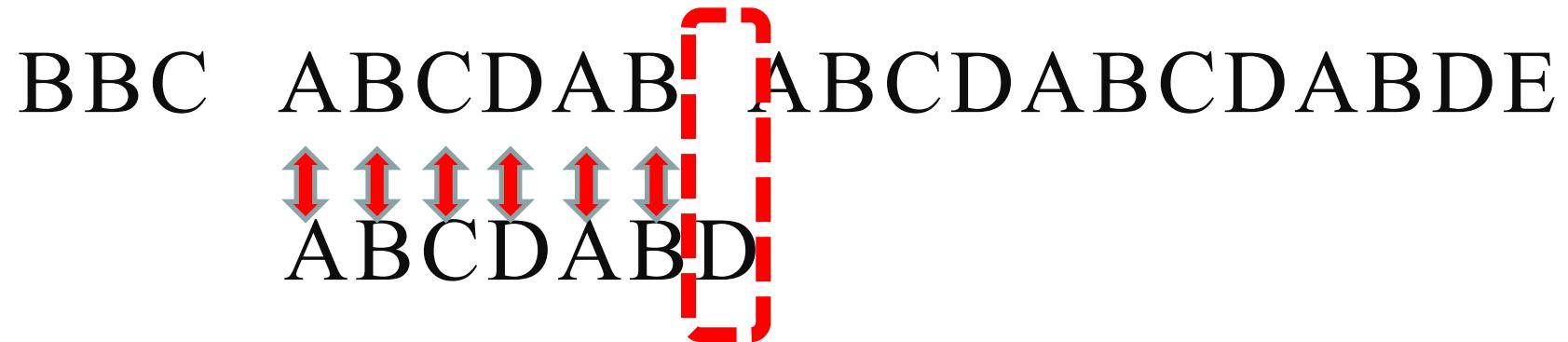


ABC ABCDAB

还是相同。

BBC ABCDAB ABCDABCDABDE
ABCDA
ABCDA

直到字符串有一个字符，与搜索词对应的字符不相同为止。



这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。

BBC ABCDAB ABCDABCDABDE



ABCDABD

这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

BBC ABCDAB ABCDABCDABDE



ABC

ABC

版本（1）：朴素串匹配

- 朴素串匹配
 - 假如串的长度为m,子串的长度为n的话，那么这个算法在最坏的情况下时间复杂度为 $O(m*n)$
- 计算指标
 - 比较次数
 - 内存占用量

一个基本事实是，当空格与D不匹配时，你其实知道前面六个字符是 "ABCDAB"。KMP算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

BBC ABCDAB  ABCDABCDABDE

ABCDABD

怎么利用这个已知信息呢？

可以针对搜索词，算出一张《NEXT值表》，即失败指针。
这张表是如何产生的，等下再介绍，这里只要会用就可以了。

搜索词	A	B	C	D	A	B	D
Next 值	-1	0	0	0	0	1	2

已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，字符D对应的“NEXT值”为2，因此按照下面的公式算出向后移动的位数：

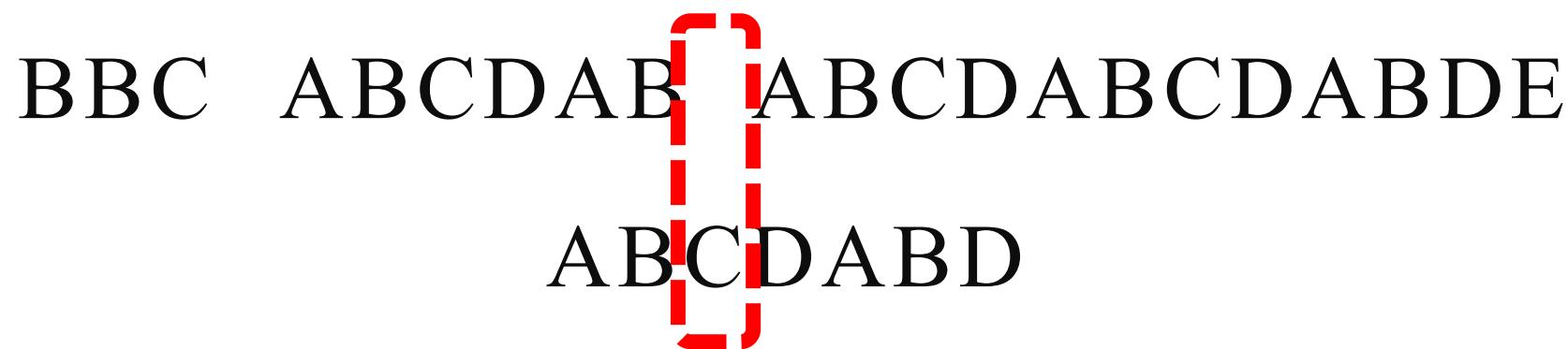
搜索词	A	B	C	D	A	B	D
Next 值	-1	0	0	0	0	1	2

移动位数 = 已匹配的字符数 - 对应的NEXT值

因为 $6 - 2$ 等于4，所以将搜索词向后移动4位。

BBC ABCDAB  ABCDABCDABDE
ABCDABD

因为空格与 C 不匹配，搜索词还要继续往后移。这时，已匹配的字符数为 2 ("AB")，C 对应的“NEXT 值”为 0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移 2 位。



搜索词	A	B	C	D	A	B	D
Next 值	-1	0	0	0	0	1	2

因为空格与A不匹配， $0-(-1)=1$ ，所以继续后移一位。

BBC ABCDAB  ABCDABCDABDE
 ABCDABD

逐位比较，直到发现C与D不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动4位。

BBC ABCDAB ABCDABCDABDE

ABCDABD



逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动7位，这里就不再重复了。

BBC ABCDAB ABCDABCDABDE

ABCDABD



下面介绍《NEXT值表》是如何产生的。

搜索词	A	B	C	D	A	B	D
Next 值	-1	0	0	0	0	1	2

搜索词	A	B	C	D	A	B	D
Next 值	-1	0	0	0	0	1	2

第一位的next值必定为-1；

计算第n个字符的NEXT值：

- 1.查看第n-1个字符对应NEXT值，设为a；
- 2.判断a是否为-1，若为-1，则第n个字符next值为0
- 3.若不为-1，将第n-1个字符与第a个字符比较
- 4.如果相同，第n个字符对应的NEXT值为 $a+1$
- 5.如果不同，令a等于第a个字符的NEXT值，执行第2步。

KMP算法

- 基本思想

- 每当匹配过程中出现字符串比较不等时，不需回溯主串，而是充分利用已经得到的“部分匹配”结果，过滤掉那些多余的比较，将模式串向右“滑动”尽可能远的一段距离后，继续进行比较，从而提高模式匹配的效率。
- 核心next函数：
 - 若 $p[k] == p[j]$ ，则 $next[j + 1] = next[j] + 1 = k + 1$ ；若 $p[k] \neq p[j]$ ，如果此时 $p[next[k]] == p[j]$ ，则 $next[j + 1] = next[k] + 1$ ，否则继续递归前缀索引 $k = next[k]$ ，而后重复此过程。
- 该算法的时间复杂度为 $O(m+n)$ 。

模式串	A	B	C	D	A	B	D
next	-1	0	0	0	0	1	2

NEXT值表：

```
void get_next(char *t, int *next)
{
    int i=0, j=-1;
    next[0]=-1;
    while (i<(int)strlen(t))
    {
        if (j == -1 || t[i] == t[j])
        {
            i++;
            j++;
            next[i] = j;
        }
        else
            j = next[j];
    }
}
```

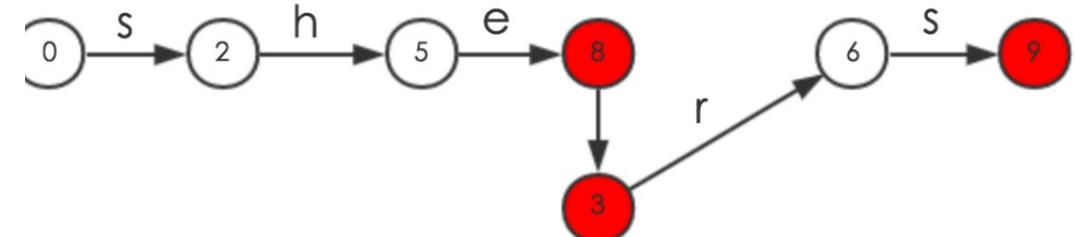
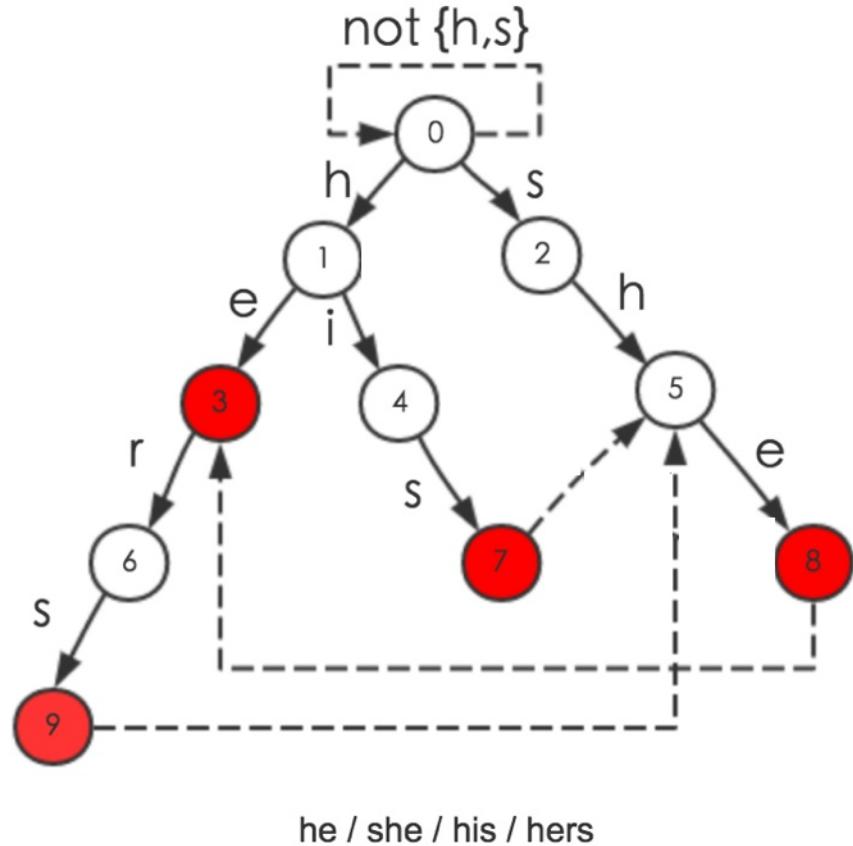
KMP部分：

```
int index_KMP (char *s, char *t, int pos)
{
    int len1=strlen(s), len2=strlen(t);
    int i=pos, j=0;
    while(i<len1 && j<len2)
    {
        if(j == -1 || s[i] == t[j]) //继续进行后续字符串比较
        {
            i++;
            j++;
        }
        else
            j=next[j]; //模式串右移
    }
    if(j >= len2) //匹配成功
        return i - len2 + 1;
    else //匹配不成功
        return -1;
}
```

版本 (2) : multi-KMP版本

- 每个模式串做一个next表
 - 多个模式串循环处理
-
- 指标:
 - 比较次数
 - 内存占用量

多模式匹配：Trie+KMP？



版本 (3) : Ac自动机

- Aho-Corasick自动机算法（简称AC自动机）1975年产生于贝尔实验室。
- 该算法应用有限自动机巧妙地将字符比较转化为了状态转移。
- 基本思想
 - 在预处理阶段，AC自动机算法建立了三个函数，转向函数`goto`，失效函数`failure`和输出函数`output`，由此构造了一个树型有限自动机。
 - 在搜索查找阶段，则通过这三个函数的交叉使用扫描文本，定位出关键字在文本中的所有出现位置。
- 此算法有两个特点，一个是扫描文本时完全不需要回溯，另一个是时间复杂度为 $O(n)$ ，时间复杂度与关键字的数目和长度无关。

树形有限自动机

- 树型有限自动机
 - ▣ 树型有限自动机包含一组状态，每个状态用一个数字代表。
 - ▣ 状态机读入文本串y中的字符，然后通过产生状态转移或者偶尔发送输出的方式来处理文本。
 - ▣ 树型有限自动机的行为通过三个函数来指示：
 - 转向函数g
 - 失效函数f
 - 输出函数output

例如：对应模式集 $\{he, she, his, hers\}$ 的树型有限自动机

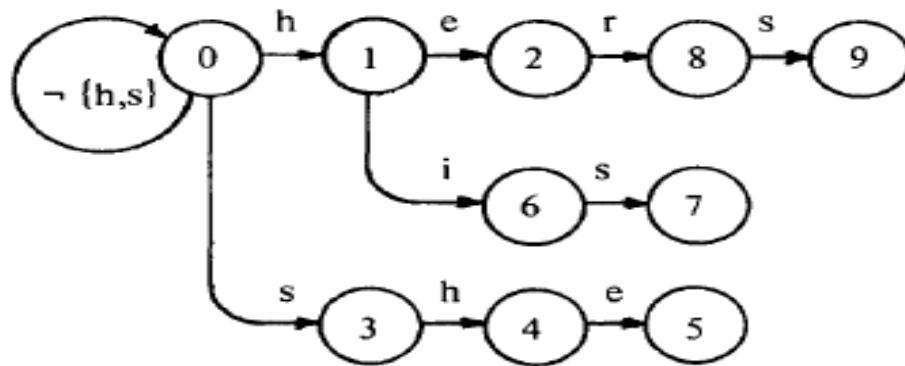


图1 a) 转向函数 g

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

图1 b) 失效函数 f

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

图1 c) 输出函数 $output$

自动机构建

- 转向，失效和输出函数的构建
 - ▣ 根据一个关键字集建立正确的转向、失效和输出函数。
 - ▣ 整个构建包含两个部分：
 - 在第一部分确定状态和转向函数
 - 在第二部分我们计算失效函数。
 - 输出函数的计算则是穿插在第一部分和第二部分中完成。
 - ▣ 为了构建转向函数，需要建立一个状态转移图。
 - 开始，这个图只包含一个代表状态0。
 - 然后，通过添加一条从起始状态出发的路径，依次向图中输入每个关键字p。
 - 新的顶点和边被加入到图表中，产生了一条能拼写出关键字p的路径。
 - 关键字p会被添加到这条路径的终止状态的输出函数中。
 - 当然只有必要时才会在图表中增加新的边。

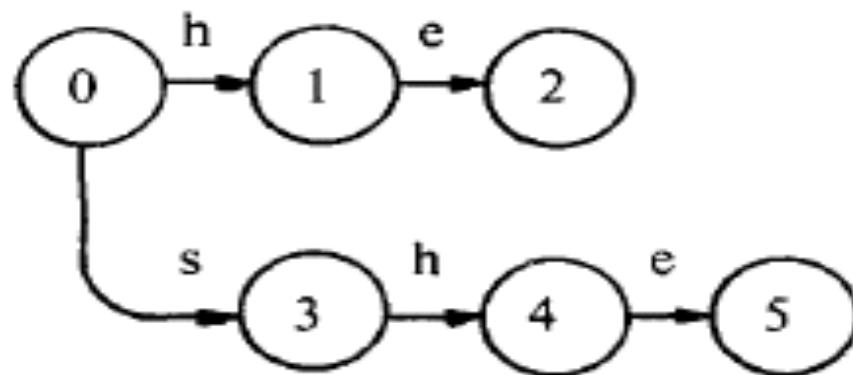
例如：对关键字集 $\{he, she, his, hers\}$ 建立转向函数。

➤ 向图表中添加第一个关键字，得到：



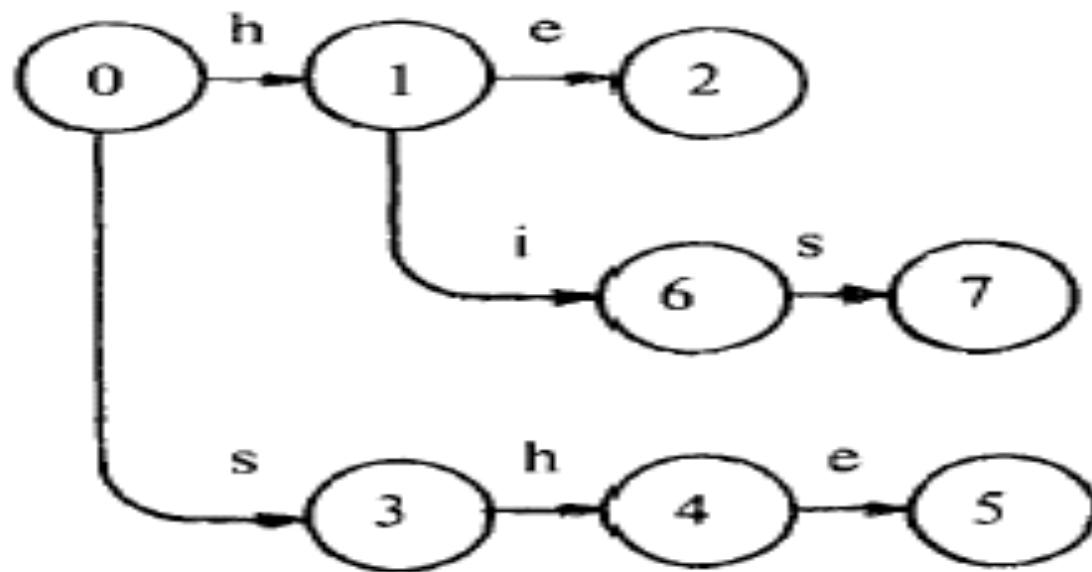
从状态0到状态2的路径拼写出了关键字“*he*”，我们把输出“*he*”和状态2相关联。

➤ 添加第二个关键字“*she*”，得到：



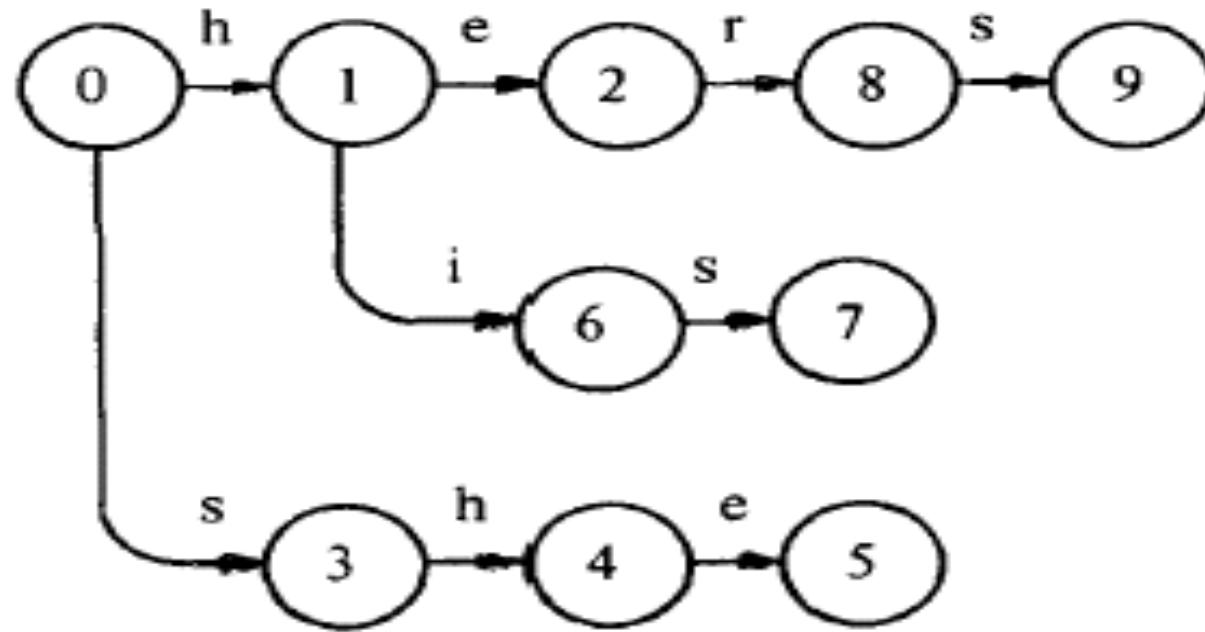
输出“*she*”和状态5相关联。

➤ 增加第三个关键字 “*his*” , 我们得到了下面这个图。注意到当我们增加关键字 “*his*” 时, 已经存在一条从状态0到状态1标记着*h*的边了, 所以我们不必另外添加一条同样的边。



输出 “*his*” 是和状态7相关联的

➤ 添加第四个关键字 “*hers*”，可以得到：



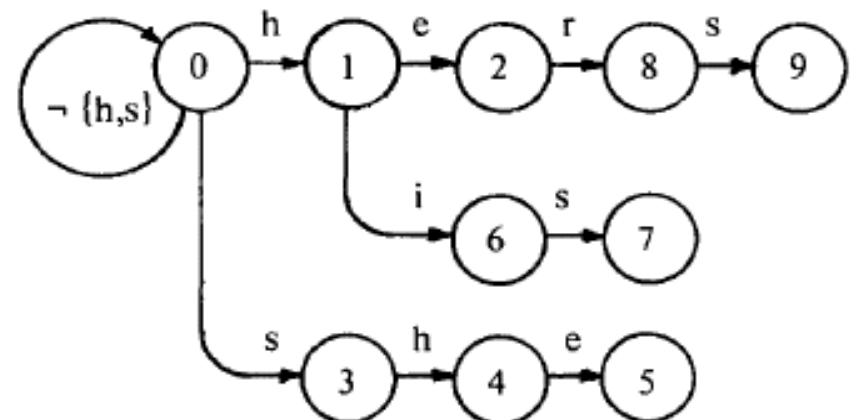
输出 “*hers*” 和状态9相关联。

在这里，我们能够使用已有的两条边：一条是从状态0到1标记着*h*的边；一条是从状态1到2标记着*e*的边。

转向函数构建

- 定义

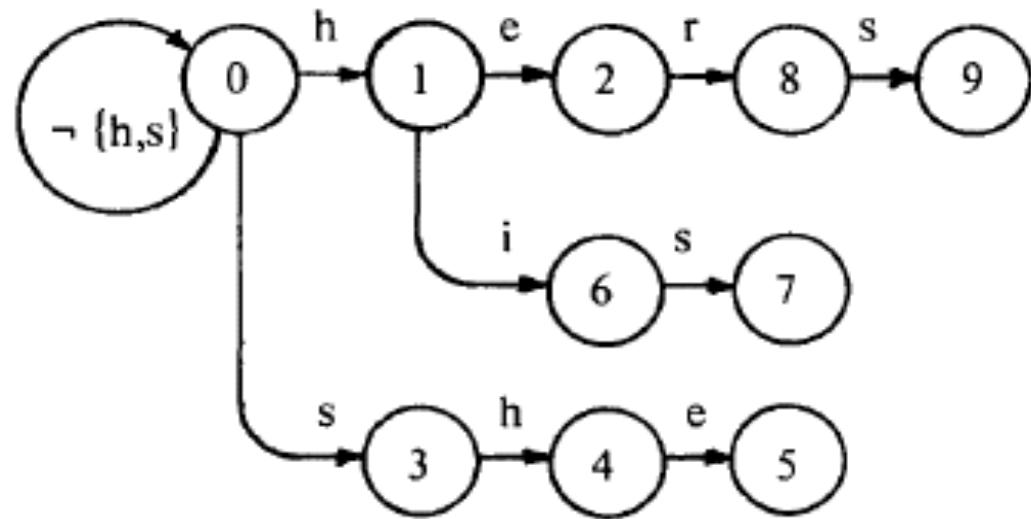
- 转向函数把一个由状态和输入字符组成的二元组映射成另一个状态或者一条失败消息。
- 状态图代表转向函数 g 。
- 比如,从0到1标记着 h 的边表示 $g(0, h) = 1$ ，如果缺省箭头则表示失败。
- 对除 e 和 i 之外的其他输入字符，都有 $g(1, h) = \text{fail}$ 。所有的树型有限自动机都有一个共同的特点，即对任何输入字符 a , 都有 $g(0, a) \neq \text{fail}$ 。
- 转向函数在0状态上的这种性质确保每个输入字符都可以在状态机的一个操作循环内被处理。



失效函数概念

- 概念

- 失效函数是根据转向函数建立的。
- 首先，我们定义状态转移图中状态s的深度为从状态0到状态s的最短路径。
- 如图起始状态的深度是0，状态1和3的深度是1，状态2，4，和6的深度是2



$$d(0) = 0; \quad d(1) = d(3) = 1; \quad d(2) = d(6) = d(4) = 2$$

$$d(8) = d(7) = d(5) = 3; \quad d(9) = 4$$

失效函数计算

- 计算思路
 - 先计算所有深度是1的状态的失效函数值，然后计算所有深度为2的状态，以此类推，直到所有状态（除了状态0，因为它的深度没有定义）的失效函数值都被计算出。
- 计算方法
 - 用于计算某个状态失效函数值的算法在概念上是非常简单的。
 - 首先，令所有深度为1的状态s的函数值为 $f(s) = 0$ 。
 - 假设所有深度小于d的状态的f值都已经被算出了，那么深度为d的状态的失效函数值将根据深度小于d的状态的失效函数值来计算。

失效函数流程

- 为了计算深度为 d 状态的失效函数值，我们考虑每个深度为 $d-1$ 的状态 r ，执行以下步骤：
 - Step1：如果对所有状态 a 的 $g(r, a) = \text{fail}$ ，那么什么都不做
 - Step2：否则，对每个使得 $g(r, a) = s$ 存在的状态 s ，执行以下操作
 - 记 $\text{state} = f(r)$ 。
 - 零次或者多次令 $\text{state} = f(\text{state})$ ，直到出现一个状态使得 $g(\text{state}, a) \neq \text{fail}$ （注意到，因为对任意字符 a ， $g(0, a) \neq \text{fail}$ ，所以这种状态一定能够找到）
 - 记 $f(s) = g(\text{state}, a)$
 - 本质上是BFS先深遍历

失效函数计算示例

- 以图1 a)为例说明计算的失效函数f
 - ▣ 先令 $f(1) = f(3) = 0$ ，因为1和3是深度为1的状态。
 - ▣ 计算深度为2的状态2, 6和4的失效函数。
 - ▣ 计算 $f(2)$ ，令 $state = f(1) = 0$ ；由于 $g(0, a) = 0$ ，得到 $f(2) = 0$ 。
 - ▣ 计算 $f(6)$ ，令 $state = f(1) = 0$ ；由于 $g(0, i) = 0$ ，得到 $f(6) = 0$ 。
 - ▣ 计算 $f(4)$ ，令 $state = f(3) = 0$ ；由于 $g(0, h) = 1$ ，得到 $f(4) = 1$ 。
 - ▣ 按这种方式继续，最终得到了如图1 b) 所示的失效函数f。
- 在计算失效函数的过程中，也更新了输出函数。
 - ▣ 当求出 $f(s) = s$ ，时，我们把状态s的输出和状态 s ，的输出合并到一起。对于上文中的例子，从图1 a) 我们求出 $f(5) = 2$ 。这时，把状态2的输出集，也就是 $\{he\}$ ，增加到状态5的输出集中，这样就得到了新的输出集合 $\{he, she\}$ 。
 - ▣ 最终各状态的非空输出集如图1 c) 所示。

失效函数的一种实现

接下来找找失败指针构造我们前缀树的失效指针！

0号节点的所有连出的字

都连向ROOT1号节点

父亲是1号节点，连接字符为A

B,查找父亲的失效指针1号

节点是否通过连接的子。

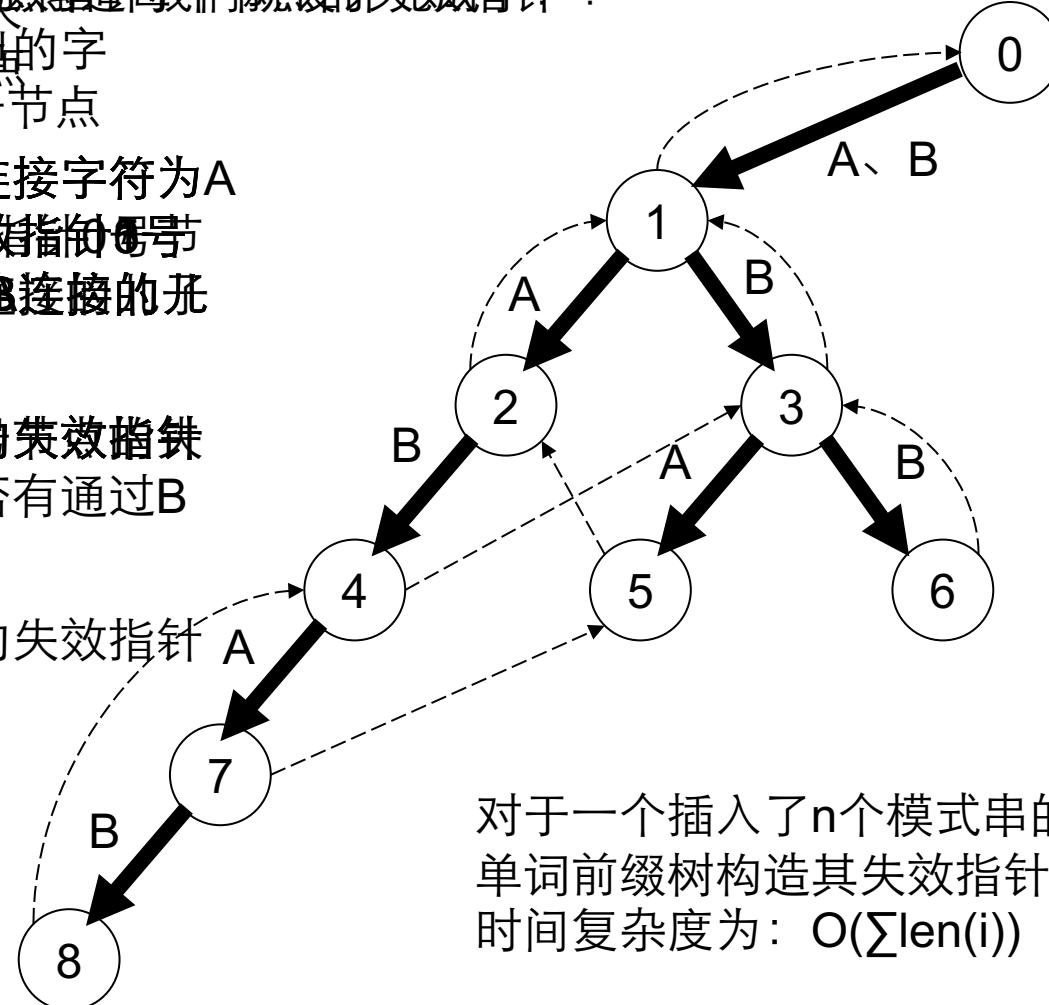
有利于链接查找的失效指针

指向1号节点是否通过B

连接的儿子。

有！于是8号节点的失效指针

指向4号节点

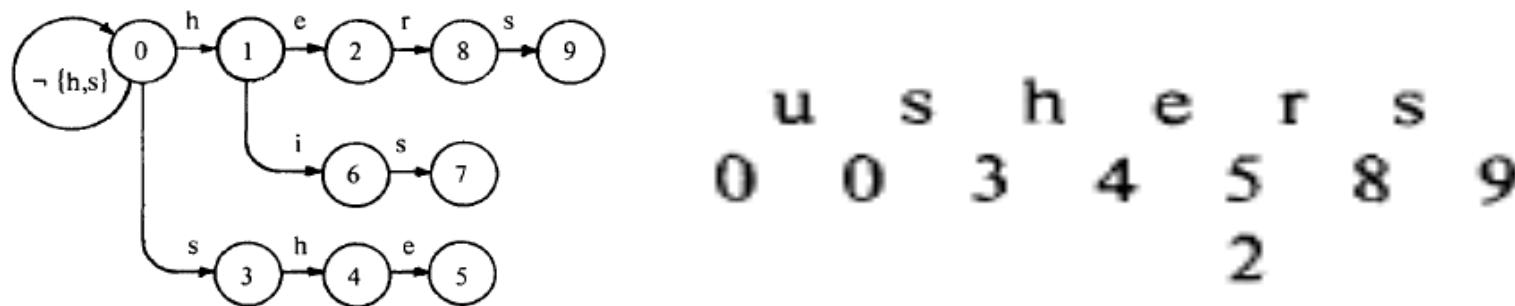


对于一个插入了n个模式串的
单词前缀树构造其失效指针的
时间复杂度为： $O(\sum \text{len}(i))$

自动机上的串匹配

- 匹配流程

- 举个例子，记树型有限自动机为状态机M。状态机M利用图1的函数去处理输入文本“ushers”，M在处理文本串时产生的状态转移。



- 考虑M在状态4，且当前输入字符为e时的操作循环。由于 $g(4, e) = 5$ ，状态机进入状态5，文本指针将前进到下一个输入字符，并且输出 $output(5)$ 。这个输出表明状态机已经发现输入文本的第四个位置是“she”和“he”出现的结束位置。在状态5上输入字符r，状态机M在此次操作循环中将产生两次状态转移。由于 $g(5, r) = fail$ ，M进入状态2 = f(5)。然后因为 $g(2, r) = 8$ ，M进入状态8，同时前进到下一个输入字符。在这次操作循环中没有输出产生。

状态转移流程

- 记 s 为状态机的当前状态, a 为输入文本 y 的当前输入字符。
- 树型有限自动机的一次操作循环可以定义如下:
 - 如果 $g(s, a) = s$, 那么树型有限自动机将做一个转向动作。自动机进入状态 s , 而且 y 的下一个字符变成当前的输入字符。另外, 如果 $output(s,)$ 不为空, 那么状态机将输出与当前输入字符位置相对应的一组关键字。
 - 如果 $g(s, a) = fail$, 状态机将询问失效函数 f 并且进行失效转移。如果 $f(s) = s$, 那么状态机将以 s , 作为当前状态, a 为当前输入字符重复这个操作循环。

AC算法总结

- 预处理阶段
 - ▣ 转向函数把一个由状态和输入字符组成的二元组映射成另一个状态或者一条失败消息。
 - ▣ 失效函数把一个状态映射成另一个状态。当转向函数报告失效时，失效函数就会被询问。
 - ▣ 输出状态，它们表示已经有一组关键字被发现。输出函数通过把一组关键字集（可能是空集）和每个状态相联系的方法，使得这种输出状态的概念形式化。
- 搜索查找阶段
 - ▣ 文本扫描开始时，初始状态置为状态机的当前状态，而输入文本y的首字符作为当前输入字符。然后，树型有限自动机通过对每个文本串的字符都做一次操作循环的方式来处理文本。

程序要求

- 分别实现三个版本，程序分别命名为
 - strstr: 朴素查找
 - Multikmp: 多模式串的kmp
 - ac_auto: ac自动机匹配

程序要求

- **输入数据**
 - 词典串pattern.txt: 约200万个
 - 主串: string.txt
- **实验结果result.txt**
 - 每行一个, 模式串 出现次数, 但出现次数倒序排列
 - Keyword1 1124
 - Keyword2 1098
 - 最后一行输出两个数字, 用空格分割:
 - 字符/字节比较次数
 - 内存开销 (kb)

报告要求

- 实验报告
 - 主要数据结构和流程
 - 实验过程
 - 遇到的问题
 - 结果指标: cpu 内存 准确率等
 - 结论和总结

THE END