

6 信息处理技术

2019年4月24日 16:32

1 信息处理技术背景

- [1.1 字符串匹配与信息处理](#)
- [1.2 串匹配](#)

2 串匹配算法概念

- [2.1 串匹配概念](#)
- [2.2 模式匹配的分类](#)
- [2.3 单模匹配vs多模匹配](#)

3 模式匹配算法分类

- [3.1 单模式匹配算法](#)
 - [3.1.1 BF算法](#)
 - [3.1.2 KMP](#)
 - [3.1.3 BM算法](#)
- [3.2 多模式匹配算法](#)
 - [3.2.1 AC算法——有限状态机](#)
 - [3.2.2 agrep算法-快速的多模式匹配](#)

1 信息处理技术背景

- [1.1 字符串匹配与信息处理](#)
- [1.2 串匹配](#)

1.1 字符串匹配与信息处理

- 数据经过高性能数据捕包和协议分析与还原之后,就可以还原成相应的内容。
 - 例如,如果捕获到的数据经过http协议的还原,成为一个完整的网页; 经过smtp协议还原后成为一封完整的邮件;
 - 之后, 根据需要进行相应的分析,即使用字符串匹配技术

1.2 串匹配

- 应用背景
 - 串匹配 (stringmatching) , 也叫模式匹配 (patternmatching) , 可以简单地定义为在给定的字符流中查找出满足某些指定属性的字符串。
 - 这是计算机科学中最古老也最普遍的问题之一, 计算机科学中串匹配的应用可以说随处可见。近年来, 随着计算机技术 (各种应用) 的蓬勃发展, 尤其是信息检索和生物计算领域中的许多共同需求, 极大激发了人们对串匹配算法的研究兴趣。
 - 大概我们最熟悉的应用是文本编辑中所使用的查找替换, 这是一种最简单的串匹配问题。
- 应用领域
 - 在生物计算领域
 - 在一个DNA链上查找某些特定特征, 或者比较两个基因序列有多大差异, 可以简单地归结为在“文本”中查找特定的“模式”的串匹配问题
 - 在信号处理领域
 - 语音识别的一般情形可以大致描述为确定一个语音信号是否符合某些特征
 - 只要事先把语音信号转化为特定形式的数字信息, 就可以应用串匹配算法来解决识别问题。
 - 在自然语言处理方面
 - 信息检索
 - 在网络安全方面
 - 发现具有某些特征码的有害信息
 - 病毒和入侵检测NID (NetworkIntrusionDetection)
 - 串匹配问题不仅在各种实际应用中有着广泛的需要, 而且在计算机理论研究中也占有着十分重要的地位, 它可以不断地提出非常具有挑战性的理论问题。

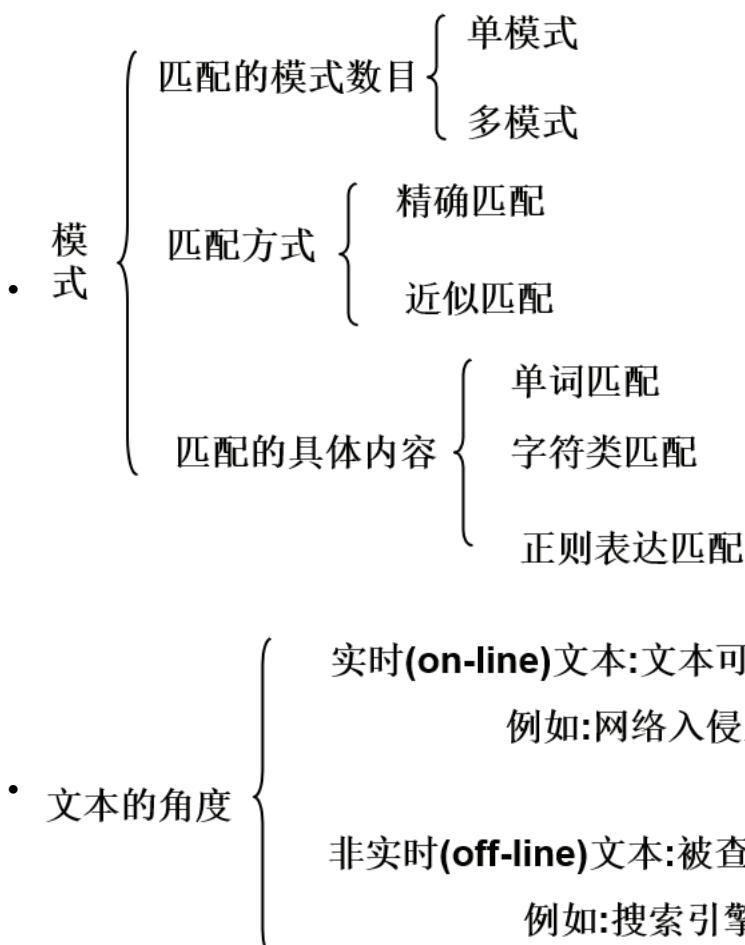
2 串匹配算法概念

- [2.1 串匹配概念](#)
- [2.2 模式匹配的分类](#)
- [2.3 单模匹配vs多模匹配](#)

2.1 串匹配概念

1. **文本:** 由若干个字符组成的有限序列, 设为 $y=\{y_1, y_2, y_3, \dots, y_n\}$, 其中 n 为文本长度, 即文本中总的字符个数。
2. **模式:** 也称为关键字, 由若干个字符组成的有限序列 $k=\{k_1, k_2, k_3, \dots, k_m\}$, 其中 m 为模式长度, 即模式中字符总数。
3. **模式集:** 指所有需要匹配的模式形成的集合, 记为 $P=\{p_1, p_2, p_3, \dots, p_r\}$, 其中 p_i 是模式集中第 i 个模式。记模式集中所有模式长度的总和为 $|P|$ 。
4. **最小模式长度:** 假设模式集中各个模式长度分别为 l_1, l_2, \dots, l_r , 那么最小模式长度是指所有模式长度的最小值, 即 $\text{minlen} = \min\{l_1, l_2, \dots, l_r\}$ 。
5. **前缀:** 两个字符串 p 和 x , 若存在字符串 v (v 可为空串), 使得 $p=xv$ 成立, 称 x 为 p 的前缀。
6. **后缀:** 两个字符串 p 和 x , 若存在字符串 u (u 可为空串), 使得 $p=ux$ 成立, 称 x 为 p 的后缀。
7. **子串:** 两个字符串 p 和 x , 若存在字符串 u, v (u, v 可以为空串), 使得 $p=uxv$ 成立, 称 x 为 p 的子串。
8. **字符集:** 在模式或文本中所有可能出现的字符形成的集合, 记为 Σ , 其大小记为 $|\Sigma|$ 。

2.2 模式匹配的分类



2.3 单模匹配vs多模匹配

- 单模式匹配
 - 单模式匹配可定义为：在一个文本text(设长度为n)中查找某个特定的子串pattern(设长度为m)。如果在text中找到等于pattern的子串，则称匹配成功，函数返回pattern在text中出现的位置(或序号)，否则匹配失败。
- 多模式匹配
 - 多模式匹配可定义为：在一个文本text(设长度为n)中查找某些特定的子串patterns(设长度为m)。如果在text中找到等于patterns中的某些子串，则称匹配成功，函数返回pattern在text中出现的位置(或序号)，否则匹配失败。

3 模式匹配算法分类

- [3.1 单模式匹配算法](#)
 - [3.1.1 BF算法](#)
 - [3.1.2 KMP](#)
 - [3.1.3 BM算法](#)
- [3.2 多模式匹配算法](#)
 - [3.2.1 AC算法——有限状态机](#)
 - [3.2.2 agrep算法-快速的多模式匹配](#)

3.1 单模式匹配算法

- [3.1.1 BF算法](#)
- [3.1.2 KMP](#)
- [3.1.3 BM算法](#)

3.1.1 BF算法

- BF (BruteForce) 算法(又称Naive算法)是最早出现的单关键字匹配算法
 - 思想简单
 - 理论上的时间复杂度很差
 - 实际使用效果尚可
 - ANSI C中提供的strstr函数就是使用这种算法的改进版本。
- 由于BF算法扫描字符串时常常需要回溯，这样当文本难于随机访问时，就显得特别不方便。
- 主要思想：
 - BF算法是出现最早的一种算法，其思想非常简单：从左向右，依次比较，每次移动一个字符位置。比较方向可以任意选定。无预处理阶段。
- 原理：
 - 在主串s中从第i(i的初值为start)个字符起，并且长度和t串相等的子串和t比较，若相等，则求得函数值为i,否则i增1，直至串s中不存在从i开始和t相等的子串为止。

BF算法举例：

文本： **GCATCGCAGAGAGTATAACAGTACG**

模式： **GCAGAGAG**

First attempt

GCATCGCAGAGAGTATAACAGTACG
1 2 3 4

GCAGAGAG

Second attempt

GCATCGCAGAGAGTATAACAGTACG
1
GCAGAGAG

Third attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Fourth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Fifth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Sixth attempt

GCATCGCAGAGAGTATACAGTACG
1 2 3 4 5 6 7 8
GCAGAGAG

Seventh attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Eighth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Ninth attempt

GCATCGCAGAGAGTATACAGTACG
1 2
GCAGAGAG

Tenth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Eleventh attempt

GCATCGCAGAGAGTATACAGTACG
1 2
GCAGAGAG

Twelfth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Thirteenth attempt

GCATCGCAGAGAGTATACAGTACG
1 2
GCAGAGAG

Fourteenth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Fifteenth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Sixteenth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

Seventeenth attempt

GCATCGCAGAGAGTATACAGTACG
1
GCAGAGAG

BF代码

```
void BF(char *x, int m, char *y, int n) {  
    int i, j;  
    /* Searching */  
    for (j = 0; j <= n - m; ++j) {  
        for (i = 0; i < m && x[i] == y[i + j]; ++i);  
        if (i >= m)  
            OUTPUT(j);  
    }  
}
```

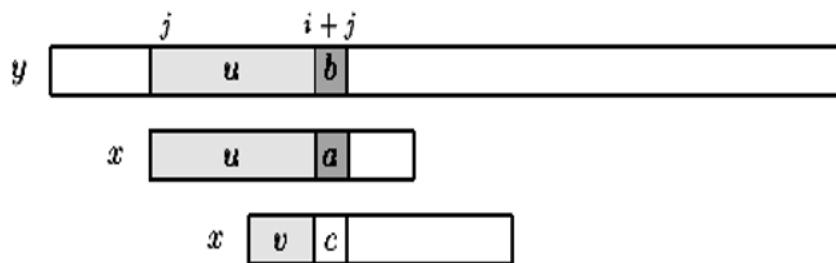
3.1.2 KMP

- KMP (DEKnuth,JHMorris, and VRPratt) 主要是基于对BF算法的改进：
- BF算法只是简单的每次移动一个字符位置，并没有考虑已匹配成功部分的信息
 - 其实这些信息是可以利用的，此即所谓的“前缀模式”——模式中不同部分存在的相同子串。
- KMP算法根据前缀模式可以使模式向前推进若干字符位置（依前缀模式长度而定），而不只是一个字符，避免了重复比较，同时也实现了文本指针的无回溯

KMP原理

假定试探第 j 个位置时，匹配失败发生在模式字符 $x[i]=a$ 和文本字符 $y[i+j]=b$ ，即 $x[i+1 \dots m-1]=y[i+j+1 \dots j+m-1]=u$ 且 $x[i] \neq y[i+j]$ 。

当转移的时候，模式集的 V 能够与文本中的 u 的一些后缀相匹配，最长的 V 我们定义为 u 的边界标记（tagged border）。 $kmpNext[i]$ 定义为 $x[0 \dots i-1]$ 的最长边界， $kmpNext[0]$ 定义为-1。



举例：

- 文本：GCATCGCAGAGAGTATAACAGTACG
- 模式：GCAGAGAG

计算 $kmpNext[i]$

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	0	1	0	1	0	1

$kmpNext$ 表

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

First attempt

GCATCGCAGAGAGTATAACAGTACG

1 2 3 4

GCAGAGAG

Shift by: 4 ($i-kmpNext[i]=3- -1$)

Second attempt

GCATCGCAGAGAGTATAACAGTACG

1
G C A G A G A G

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

Second attempt

G C A T C G C A G A G A G T A T A C A G T A C G
 1
G C A G A G A G

Shift by: 1 ($i - kmpNext[i] = 0 - -1$)

Third attempt

G C A T C G C A G A G A G T A T A C A G T A C G
 1 2 3 4 5 6 7 8
G C A G A G A G

Shift by: 7 ($i - kmpNext[i] = 8 - 1$)

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

Fourth attempt

G C A T C G C A G A G A G T A T A C A G T A C G
 1
G C A G A G A G

Shift by: 1 ($i - kmpNext[i] = 1 - 0$)

Fifth attempt

G C A T C G C A G A G A G T A T A C A G T A C G
 1
G C A G A G A G

Shift by: 1 ($i - kmpNext[i] = 0 - -1$)

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

Sixth attempt

G C A T C G C A G A G A G T A T A C A G T A C G
 1
G C A G A G A G

Shift by: 1 ($i - kmpNext[i] = 0 - -1$)

Seventh attempt

G C A T C G C A G A G A G T A T A C A G T A C G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

Eighth attempt

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

KMPNext表生成

```
void preKmp(char *x, int m, int kmpNext[ ]) {
    int i, j;
    i = 0;
    j = kmpNext[0] = -1;
    while (i < m) {
        while (j > -1 && x[i] != x[j])
            j = kmpNext[j];
        i++;
        j++;
        if (x[i] == x[j])
            kmpNext[i] = kmpNext[j];
        else kmpNext[i] = j;
    }
}
```

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$j=kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

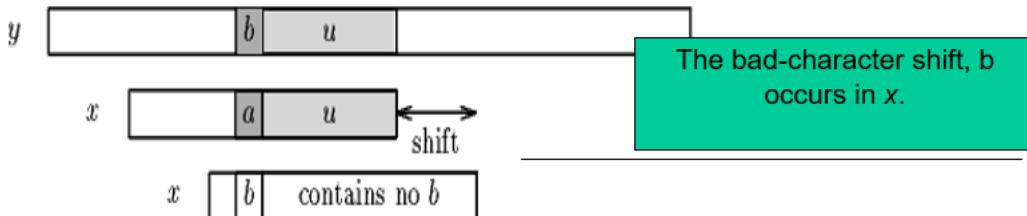
KMP函数实现代码

<pre>void KMP(char *x, int m, char *y, int n) { int i, j, kmpNext[XSIZE]; /* Preprocessing */ preKmp(x, m, kmpNext); /* Searching */ i = j = 0; while (j < n) { while (i > -1 && x[i] !=</pre>	<pre>y[j]) { i = kmpNext[i]; i++; j++; if (i >= m) { OUTPUT(j - i); i = kmpNext[i]; } } }</pre>
--	--

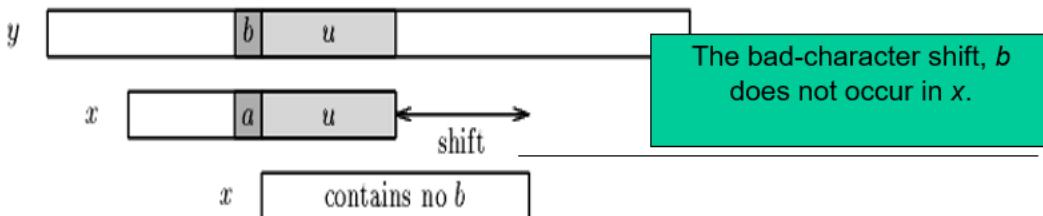
3.1.3 BM算法

- BM算法被誉为速度最快的算法。
- 主要思想：

- 算法对模式从最右端自右向左扫描。在不匹配（或完全匹配）时，用两个预先计算的函数bad character和good suffix来确定指针在正文中移动的距离。
- 原理
 - BM算法的关键是找出不必要的比较。进行比较时从字符串的右端而不是左端开始比较，当比较不匹配时，可直接向右移若干个位置；当被比较的字符相等时，则比较其前面的字符。如果成功次数等于模式串长度，则匹配成功。
 - 如果正在比较的主串中的字符在模式串中不存在，并且也不存在部分匹配，那么应右移的位置数等于模式的长度（如第一种情况）
 - 如果正在比较的主串中的字符在模式串不存在，但存在部分匹配，则右移的位置数等于模式串长度减去部分匹配的字符个数（如第二种情况）
 - 如果正在比较的主串中的字符出现在模式串中，则右移的位置数应为从模式串的最右端到模式串中该字符的距离（如第三种情况）
- BM算法实现
 - 具体实现该算法时，可以设计一张表，表中包含每一个可能比较的字符元素，表中的每个纪录存贮的是相应字符的移动计数。对于那些未出现在模式串中的字符，移动的计数等于模式串的长度。对于那些出现在模式串中的字符，移动的计数等于从模式串中最右端到该字符在模式串中出现的最右位置之间的距离，在模式匹配的过程中，当出现字符不匹配的情况下，只需查一下该表，就可以知道右移的位置数。

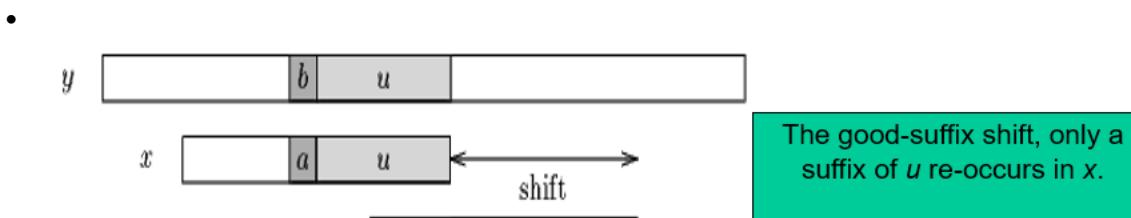
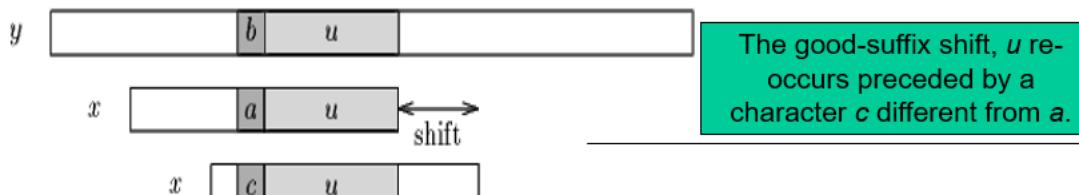


○



Badcharacter

- 计算出字符集中每个字符的偏移值**bmBC[i]**,对于未在模式中出现的字符，偏移为m,否则取m-i-1,(其中i为字符在模式中的位置),即取此字符在模式中出现的最右位置和文本中此字符位置对齐，若字符未在模式中出现，则可将模式向前推m个字符位置。但是，在某些情况下这种偏移可能是负的。实际的偏移值取得是两者中值较大者。



Good suffix的思想来源于KMP算法，

- 即一个模式中在不同部分可能又存在相同的子串的性质。可以利用已经成功完成的字符匹配，把已匹配部分看作整个模式的子模式，考虑模式前面一段中是否有与此子模式相匹配的片断。这样，就有可能把模式串向前推更远的距离（在算法中是向前移动文本指针）。由于BM算法是从右向左比较的，所以构造出的不是前缀数组，而是后缀数组。

举例：

- 文本：**
GCATCGCAGAGAGTATACAGTACG
- 模式：****GCAGAGAG**

计算**bmBc[c]**

c	A	C	G	T
$bmBc[c]$	1	6	2	8

计算**bmGs[i]**

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$bmGs[i]$	7	7	7	2	7	4	7	1

First attempt

G	C	A	G	A	G	T	A	C	G	T	A	C	G
1													
G	C	A	G	A	G	T	A	C	A	G	T	A	C

Second attempt

G	C	A	G	A	G	T	A	C	A	G	T	A	C	G
3 2 1														
G	C	A	G	A	G	T	A	C	A	G	T	A	C	G

Third attempt

G	C	A	G	A	G	A	G	T	A	C	A	G	T	A	C	G
8 7 6 5 4 3 2 1																
G	C	A	G	A	G	A	G	T	A	C	A	G	T	A	C	G

Fourth attempt

G	C	A	G	A	G	A	G	T	A	C	A	G	T	A	C	G
3 2 1																
G	C	A	G	A	G	A	G	T	A	C	A	G	T	A	C	G

Fifth attempt

GCATCGCAGAGAGTATAACAGTAG
 21
 GCAGAGAG

算法比较

- BF 算法
String length: 24
Pattern length: 8
Attempts: 17
Character comparisons: 30
 - KMP 算法
Attempts: 8
Character comparisons: 18
 - BM 算法
Attempts: 5
Character comparisons: 17

3.2 多模式匹配算法

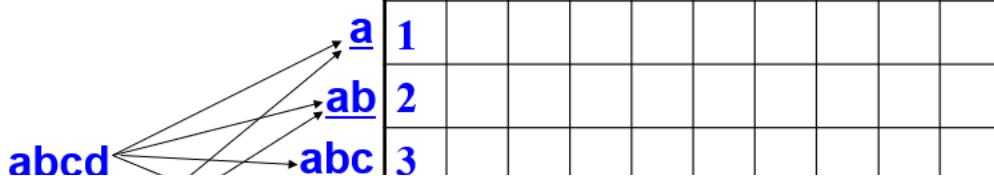
- [3.2.1 AC算法—有限状态机](#)
 - [3.2.2 agrep算法-快速的多模式匹配](#)

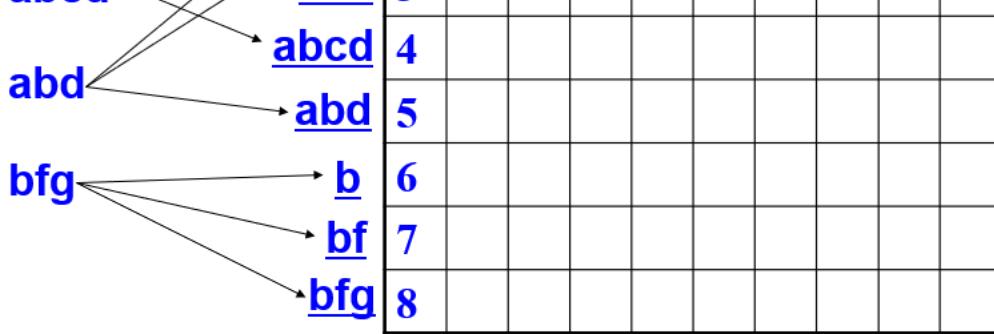
3.2.1 AC算法——有限状态机

- 问题：
 - 已知文本abfabcd.....
 - 已知关键字abcd、abd、bfg
 - 确定关键字在文本中位置
 - 自动机 (*Automata*)：确定型有限自动机
DFA(Deterministic finite automata) 是一个五元组 $M = \{S, \Sigma, \delta, s_0, S_f\}$, 包括：
 - 状态集 S
 - 输入的字符集 Σ
 - 状态转换函数 δ
 - 起始状态 s_0
 - 终止状态集 S_f

AC算法实现

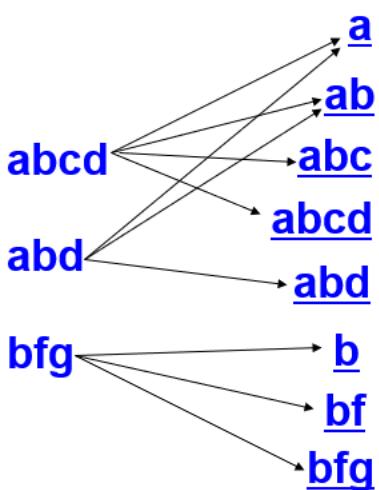
- 拆分关键字
 - 生成状态图





AC算法第一步：初始化状态机

- 拆分关键字
 - 生成状态图



	0	1	...	a	b	c	d..	g	...
0	0	0	...	1	6	0	...	0	...
1	0	0	...	1	2	0	...	0	...
2	0	0	...	0	0	3	E..	0	...
3	0	0	...	0	0	0	E..	0	...
4	0	0	...	1	6	0	...	0	...
5	0	0	...	1	6	0	...	0	...
6	0	0	...	0	6	0	..7	0	...
7	0	0	...	0	0	0	...	E	...
8	0	0	...	1	6	0	...	0	...

AC算法第一步：使用状态机

问题：

已知文本**abfabcd**a，

确定关键字**abcd**、
abd、**bfg**在文本中
位置。



	0	1	...	a	b	c	d..	g	...
0	0	0	...	1	6	0	...	0	...
1	0	0	...	1	2	0	...	0	...
2	0	0	...	0	0	3	E..	0	...
3	0	0	...	0	0	0	E..	0	...
4	0	0	...	1	6	0	...	0	...
5	0	0	...	1	6	0	...	0	...
6	0	0	...	0	6	0	..7	0	...
7	0	0	...	0	0	0	...	E	...
8	0	0	...	1	6	0	...	0	...

```

Char * txt="sdfasdfsadfsadf";
Int txtlen=strlen(txt);
Int statnum = 9;
Int stat[9][257];
Int match(char* txt)
{
    int i,j;
    i = 0;
    j = 0;
    int k = 0;
    while(k < txtlen)
    {
        j=txt[k];
    }
}

```

```

    i=stat[i][j];
    if(stat[i][256] != -1)
        return stat[i][256];
    else
        k++;
}
}

```

多模式匹配算法

- 多模式匹配问题在生物计算、信息检索及信号处理领域有着非常广泛的应用。
- 最早也是最著名的以线性时间复杂度解决这个问题的算法是1975年Aho-Corasick提出的AC75；
- 对于单模式还有非常好高效的算法BM77,由BM中的跳跃思想衍生出了许多变种，应用于单模式和多模式匹配中。
- Commentz-Walter就是在结合AC和BM的思想产生的一种多模式匹配算法，
- 另外，1992年Wu.Sum和Udi.manber提出的agrep是最好的多模算法之一。
- 近年来，由于后缀树（suffix tree）和后缀自动机(suffix auromaton)的引入，又出现了DAWG—MATCH和MultiBDM。

有限自动机多模式匹配——AC算法

- AhoCorasick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。
- 该算法的基本思想是这样的：
 - 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和输出函数output。

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

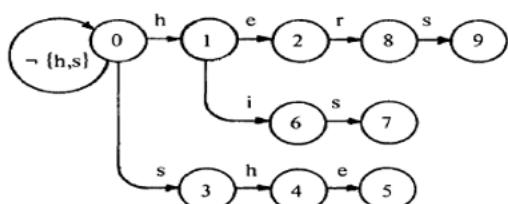


图1 a) 转向函数g

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

3. 转向，失效和输出函数的构建

现在说明如何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，我们需要建立一个状态转移图。开始

为了构建转向函数，我们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的方式，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，以致于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。

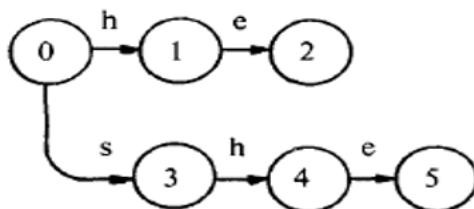
例如：对关键字集{**he**, **she**, **his**, **hers**}建立转向函数。

➤ 向图表中添加第一个关键字，得到：



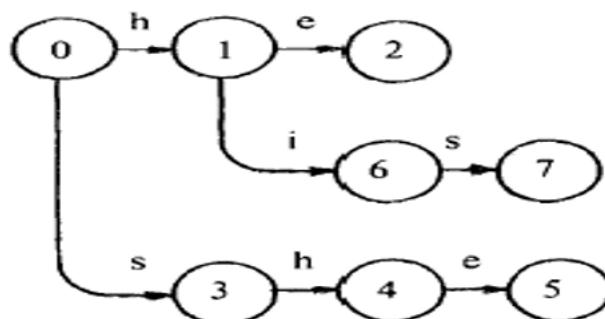
从状态**0**到状态**2**的路径拼写出了关键字“**he**”，我们把输出“**he**”和状态**2**相关联。

➤ 添加第二个关键字“**she**”，得到：



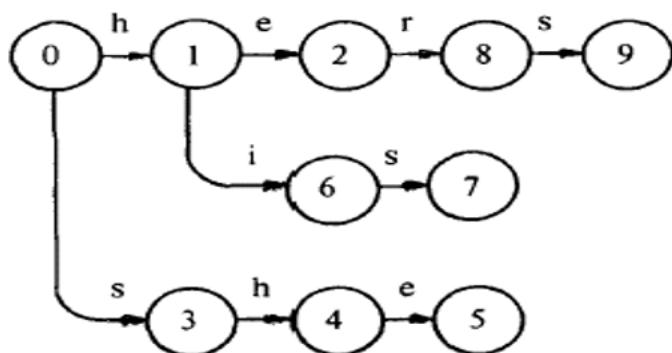
输出“**she**”和状态**5**相关联。

➤ 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：



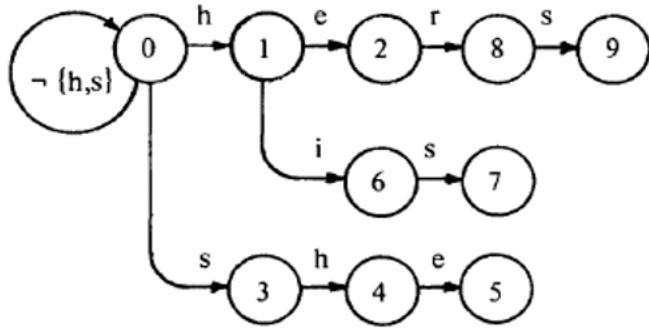
输出“**hers**”和状态**9**相关联。

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。

标记着 h 的边，——表示从状态 0 到 h 标记着 s 的边。

这样，图已经成为一棵带根的树。为了完成转向函数的构建，我们对除了 h 和 s 外的其他每个字符，都增加一个从状态 0 到状态 s 的循环。这样，我们得到了如图1 a) 所示的状态转移图，这个图就代表转向函数。



算法1：建立转向函数 g 。

输入：关键字集 $P = \{p^1, p^2, p^3, \dots, p^r\}$ 。

输出：转向函数 g 和部分的 $output$ 函数。

方法：
begin
 newstate $\leftarrow 0$
 for $i \leftarrow 1$ until k do enter(y_i)
 for all a such that $g(0, a) = \text{fail}$ do $g(0, a) \leftarrow 0$
 end

 procedure enter($a_1 a_2 \dots a_m$):
 begin
 state $\leftarrow 0$; $j \leftarrow 1$
 while $g(state, a_j) \neq \text{fail}$ do
 begin
 state $\leftarrow g(state, a_j)$
 $j \leftarrow j + 1$
 end
 for $p \leftarrow j$ until m do
 begin
 newstate $\leftarrow newstate + 1$
 $g(state, a_p) \leftarrow newstate$
 state $\leftarrow newstate$
 end
 output(state) $\leftarrow \{a_1 a_2 \dots a_m\}$
 end
end

图2 建立转向函数 g 的伪代码

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。例如图1 a) 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 ，等等。

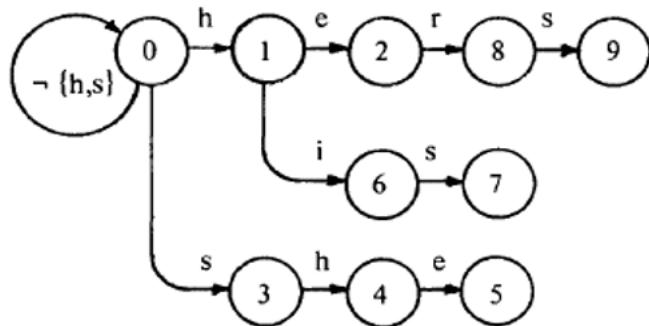


图1 a)

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

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

➤ 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。

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**state = f(r)**。

零次或者多次令**state = f(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）

➤ Step2：记 $f(s) = g(\text{state}, a)$

以图1 a)为例说明计算的失效函数f；

①先令 $f(1) = f(3) = 0$ ，因为1和3是深度为1的状态。

②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 $f(4) = 1$ 。

③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。

算法2：建立失效函数f。

输入：转向函数g和部分的输出函数output。

输出：失效函数和完整的输出函数output。

方法：begin

```
queue ← empty
for each a such that g(0, a) = s ≠ 0 do
begin
    queue ← queue ∪ {s}
    f(s) ← 0
end
while queue ≠ empty do
begin
    let r be the next state in queue
    queue ← queue - {r}
    for each a such that g(r, a) = s ≠ fail do
begin
    queue ← queue ∪ {s}
    state ← f(r)
    f(s) ← state
end
end
```

```

while g(state, a) = fail do state ← f(state)
    f(s) ← g(state, a)
    output(s) ← output(s) ∪ output(f(s))
end
end

```

图3 建立失效函数f的伪代码

4. 转向函数的构建

图1中树型自动机的状态有0, 1, ..., 9。某个状态（通常是0状态）被指定为起始状态。

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。

图1 a) 的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。可见，对除e和i之外的其他输入字符，都有 $g(1, \text{char}) = \text{fail}$ 。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 $g(0, a) \neq \text{fail}$ 。我们将看到，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。

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”，图4显示了M在处理文本串时产生的状态转移情况。

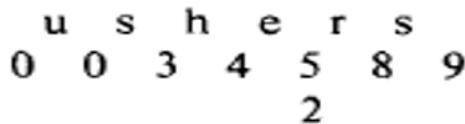


图4 扫描“ushers”时的状态转换序列

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 $g(5, r) = \text{fail}$ ，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。

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：

1. 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 $\text{output}(s)$ 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。
2. 如果 $g(s, a) = \text{fail}$ ，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。

算法3：树型有限状态机。

输入：一个字符串 $y = \{y_1, y_2, y_3, \dots, y_n\}$ （其中 y_i 是一个输入字符）；

一台包含上述转向函数g，失效函数f和输出函数output的树型

有限自动机。
输出：关键字在y中出现的位置。

```
begin
    state ← 0
    for i ← 1 until n do
        begin
            while g(state, ai) = fail do state ← f(state)
            state ← g(state, ai)
            if output(state) ≠ empty then
                begin
                    print i
                    print output(state)
                end
            end
        end
    end
```

图5 建立树型有限自动机的算法伪代码

5. AC自动机

➤ 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。

经典的多模式匹配算法——AC自动机。

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。

1975年，Aho和Corasick提出基于确定性有限自动机(DFA)理论的模式匹配算法，这又是模式匹配问题中一个经典的算法。实际上，多模式匹配算法中，有很大一部分都是基于自动机理论的，而且有不少都是基于对AC75算法的改进。

1979年，Commentz和Walter. B 发明的算法（简称CW79算法）结合了BM算法，在AC75的自动机算法上实现了跳跃扫描文本。

除了自动机这种主流多模式匹配思想外还有一种很有效的想法。这就是哈希 (Hashing) , Hashing方法的串查寻最早是在1971年被Harrison介绍,之后得到了充分地分析。1992年到1996年,台湾人Sun Wu和他的导师Udi Manber发表了一系列的论文,详细地介绍了他们设计的匹配算法,并用此算法实现了一个Unix下类似fgrep的工具: agrep。

3.2.2 agrep算法-快速的多模式匹配

- agrep算法是92年台湾学者吴升发明,是模式中最为著名的快速匹配算法之一,采用了跳跃不可能匹配的字符策略和hash散列的方法,对处理大规模的多关键字匹配问题有很好的效果。
- 该算法的基本思想是这样的:
 - 在预处理阶段,算法主要使用了三个数据结构: SHIFT表、HASH表、和PREFIX表。
 - 在搜索查找阶段,SHIFT表用于在扫描文本串的时候,根据读入字符串决定可以跳过的字符数,如果相应的跳跃值为0,则说明可能产生匹配,就要用到HASH表和PREFIX表进一步判断,以决定有哪些匹配候选模式,并验证究竟是哪个或者哪些候选模式完全匹配

预处理过程

- 假设模式集合 P 中最短的模式长度为 m ,那么,后续讨论仅仅考虑所有模式的前 m 个字符组成的模式串,即要求所有匹配的模式长度相等。为了加快比较速度,对长为 m 的串进行分组,以 B 个长度的字符串为基本单位,每次比较长度为 B 的子串。对于 B 的选取,原文给出了指导公式计算出一个合适的 B 值: $B=\log_2 M$ 。这里, $M=k \times m$, k 是模式的数目;而 c 表示字符集的大小即 $c=|\Sigma|$.

AC和QS结合的反向自动机

- 王永成等人受FW92(一种融合了BM的自动机算法),提出的相类似的结合QS的反向自动机多模式匹配算法,而且是针对纯中文的处理算法。

WuSum和Udimanber的agrep

- 92年台湾学者吴升发明的agrep是多模式中最为著名的快速匹配算法之一,对处理大规模的多关键字匹配问题有很好的效果。

DAWG-MATCH

- DAWG是一种后缀自动机 (Suffix Automaton),是建立在模式集P上,能够辨认出模式集P中所关键字后缀的确定型自动机。这种思想主要是AC和RF的结合结果。

Raffinot的MultiBDM

- 在上述的AC和DAWG两种自动机扫描思想上产生的多模匹配算法。根据匹配过程中使用时刻的不同,作者提出了两种改进。在作者的实验中,处理大规模的多关键字匹配问题中有较好的优势。

SumKim99

- 一种使用HashTable和Bit-Parallel的算法。它的处理过程比较特别,先对模式数值化压缩存储,然后使用HashTable直接定位出当前读入的字符将可能匹配上的关键字的范围,接着再用位运算对可能匹配上的关键字逐个比较,判定文本中是否有关键字出现。