字符串精确匹配问题

- 问题描述：对于模式串集合P={p1,...,pk}，在目标串T[m]中找出现了哪些模式串。

- 设n=|p1|+...+|pk|
- 普通算法的时间复杂度是O(n+km)

- AC自动机算法是解决这种问题的一个经典方法，时间复杂度为O(n+m+z)，其中z是T中出现的模式串的数量。
- AC自动机是基于keyword tree的，并对其进行一些补充。

- KMP算法所解决的问题与这个类似，区别是KMP解决的问题是在目标串T中找一个模式串p出现的位置。其实两个算法的精髓也一样：当匹配失败时可以最大程度的利用之前已匹配成功的串以避免重复匹配。

Keyword tree

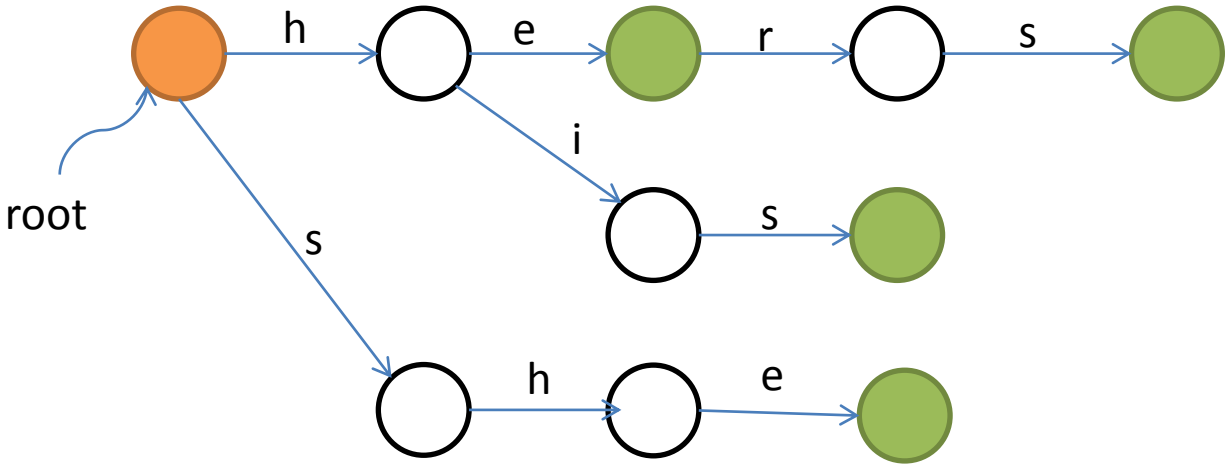
◆ keyword tree又叫“单词树”、“字典树”、“trie树”

- 对于模式串集合P={p1…pk}的keyword tree是具有以下特性的有根树：

1. 每条边的值都是一个字符
2. 从一个点出发的任意两条边的值都不同
3. 结点v的值被定义为L(v)={从根结点到结点v的路径上的所有边的值的序列}
4. 对任意模式串pi，都能找到一个结点v使得L(v)==pi
5. 对任意的叶子结点v，都能找到一个pi使得L(v)==pi

Keyword tree

- A keyword tree for P={"he","she","his","hers"}



keyword tree：构建

对于模式串集合P={p1…pk}，构建keyword tree：

1．从仅有的一个根结点开始

2．依次插入模式串pi：从根结点出发，随着值为pi中当前字符的边一直走下去，

- 如果在pi中的字符结束之前，路径就终止了，那么为pi中剩余的所有字符创建新的边和结点
- 在pi结束的时候，记录一下当前的结点为模式串pi的终止结点

- 构建的时间复杂度为O(|p1|+...+|pk|)=O(n)

keyword tree：查找

- 在keyword tree中查找一个字符串p，也就是判断p是否出现在模式串集合P中：从根结点出发，随着值为p中当前字符的边一直往下走。
  - 如果在p结束时，当前的结点是一个被记录的终止结点，那么查找成功，p在模式串集中
  - 如果在p结束前，路径就终止了，则查找失败，p不在模式串集中。

- 查找时间复杂度为O(|p|)——一个高效的查找方法
- 一个普通的模式串匹配方法的时间复杂度将是O(nm)，模式串的数量为n，平均长度为m。

- 下面将把keyword tree扩展为一个自动机（Automaton），它可以在线性的时间内进行匹配。

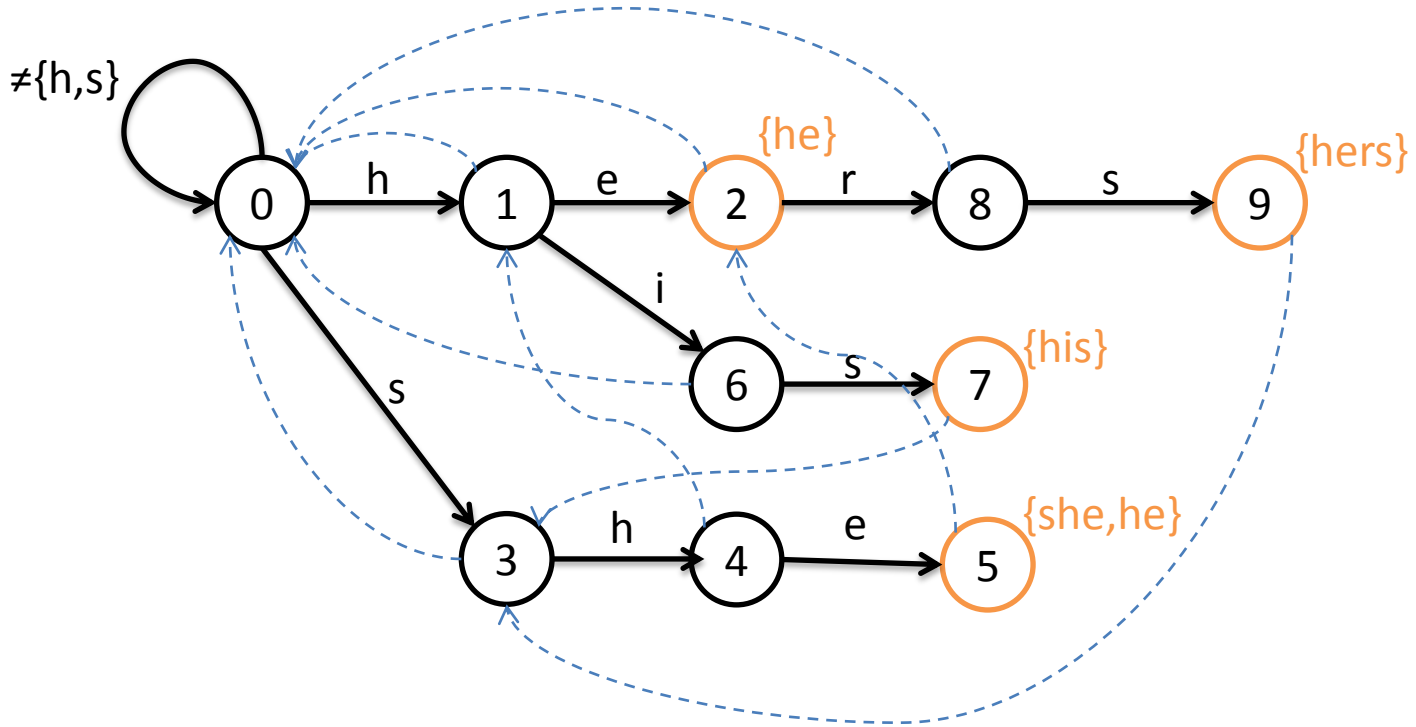
AC自动机
（Aho-Corasick Automaton）

keyword tree的每个结点就是一个**状态**，根结点是**初始状态（状态0）**

自动机的行为被定义为一下3个函数：

1. **goto函数g(q,a)**：返回从当前状态q走值为a的边后所到达的状态
   - 如果从根结点出发没有值为a的边，那么g(0,a)=0，（就是说：当读入不匹配的字符时，自动机保持在初始状态）
   - 如果从结点q出发有一条边值为a到达结点v，那么g(q,a)=v
   - 否则g(q,a)=Φ
2. **fail函数f(q)**：返回从状态q(q!=0)匹配错误时要转移到的状态
   - 如果L(v)是L(q)的一个后缀，且是最长的后缀，则f(q)=v，（就是说：一个错误的匹配后进行的状态转移会最大地利用之前已经匹配的串），因为虽然L(q)不是任何一个模式串的前缀，但是L(v)还是有可能是某个模式串的前缀的
   - f(q)总是能返回一个状态，因为L(0)=Φ是任何模式串的前缀
3. **output函数out(q)**：输出在状态q时，所有匹配的模式串

- **Example：** 依次插入模式串"he","she","his","hers"，构建出的各个结点序号的顺序如下，虚线表示fail指针（状态0没有），橙色结点是模式串的结束，旁边有output函数的输出值，注意output(5)={"she","he"}



匹配目标串T[m]

- 遍历目标串T[m]，找出有哪些模式串在T[m]中出现过？

伪代码：

```
q = 0;
for (i = 0; i < m; i++)
{
    while ( g(q, T[i]) == Φ ) //当匹配失败时
        q = f(q);             //沿着fail指针走，直到匹配成功或者到初始状态
    q = g(q, T[i]);
    out(q);
}
```

- 用AC自动机来匹配目标串T[m]的时间复杂度是O(m+z)，其中z是模式串出现的次数。

AC自动机的构建

- AC自动机的构建分为两个阶段：

◆ 阶段一

1．为模式串集P构建keyword tree：依次把pi插入tree，并且对于模式串的终止结点v，设置out(v)={pi}

2．对于根结点出发不存在值为a的边，设置g(0,a)=0

如果所有可能出现的字符的集合∑是固定的，则阶段一的时间复杂度为O(n)，n=|p1|+...+|pk|

◆ 阶段二

用BFS顺序遍历trie树的结点，遍历过程中计算结点的fail和output函数，伪代码：

```
Q = emptyQueue();
for ( a ∈ ∑ )
    if ( (q = g(0, a)) != 0 )
    {
        f(q) = 0; //根结点下的第一层结点的fail指针都指向根结点
        Q.push(q);
    }
while ( !Q.empty() )
{
    r = Q.pop();
    for ( a ∈ ∑ )
        if ( (u = g(r, a)) != Φ )
        {
            Q.push(u);
            v = f(r);
            while ( g(v, a) == Φ )
                v = f(v);
            f(u) = g(v, a);
            out(u) += out( f(u) );
        }
}
```

- 阶段二的复杂度可看成是BFS的，所以也是O(n)

trie图

- 对目标串进行匹配时：当g(q,a)=Φ时，会让q转向它的失败指针所指向的状态（q=f(q)），直到g(q,a)!=Φ为止，然后状态就转移到q=g(q,a)
- 那么我们可以简化这一繁杂的过程，直接让所有为空的g(q,a)不再为空，而是指向一个合适的状态。

举例：目标串T="abch"，走到状态3时，

q=3,由于g(3,h)=Φ;q=f(q)=6;

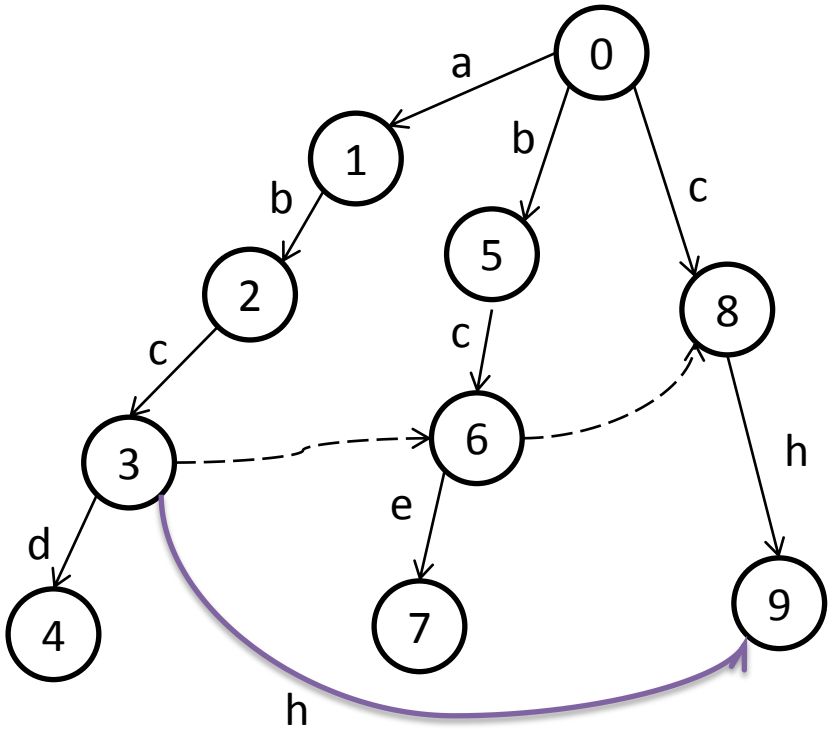
由于g(6,h)=Φ;q=f(q)=8;

由于g(8,h)=9;

所以状态q=9;

现在添加一条蓝色的边使g(3,h)=9

就没有那些多余的步骤了。



◆ 在添加了这些边之后，就转换成了**trie图**，从每个结点出发都有size(Σ)条边，不存在g(q,a)=Φ的情况了。

➢ 我的AC自动机模板代码：

```
class ACAutomaton
{
public:
    static const int MAX_N = 10000 * 50 + 5;
        //最大结点数：模式串个数 x 模式串最大长度
    static const int CLD_NUM = 26;
        //从每个结点出发的最多边数，字符集∑的大小，一般是26个字母

    int n;                              //trie树当前结点总数
    int id['z'+1];                      //字母x对应的结点编号为id[x]
    int fail[MAX_N];                    //fail指针
    int tag[MAX_N];                     //根据题目而不同
    int trie[MAX_N][CLD_NUM];           //trie树，也就是goto函数

    void init();
    void reset();

        //插入模式串s，构造单词树(keyword tree)
    void add(char *s);

        //构造AC自动机：用BFS来计算每个结点的fail指针，并且构造trie图
    void construct();
};
```

```
void ACAutomaton::init()
{
    for (int i = 0; i < CLD_NUM; i++)
        id['a'+i] = i;
}
```

```
void ACAutomaton::reset()
{
    memset(trie[0], -1, sizeof(trie[0]));
    tag[0] = 0;
    n = 1;
}
```

```
void ACAutomaton::add(char *s)
{
    int p = 0;
    while (*s)
    {
        int i = id[*s];
        if ( -1 == trie[p][i] )
        {
            memset(trie[n], -1, sizeof(trie[n]));
            tag[n] = 0;
            trie[p][i] = n++;
        }
        p = trie[p][i];
        s++;
    }
    tag[p]++;     //因题而异
}
```

```
void ACAutomaton::construct()
{
    queue<int> Q;
    fail[0] = 0;
    for (int i = 0; i < CLD_NUM; i++)
    {
        if ( -1 != trie[0][i] )
        {
            fail[trie[0][i]] = 0;    //根结点下的第一层结点的fail指针都指向根结点
            Q.push( trie[0][i] );
        }
        else
            trie[0][i] = 0;     //这个是阶段一中的第2步
    }
    while ( !Q.empty() )
    {
        int u = Q.front();
        Q.pop();
        for (int i = 0; i < CLD_NUM; i++)
        {
            int &v = trie[u][i];
            if ( -1 != v )
            {
                Q.push( v );
                fail[v] = trie[fail[u]][i];
                tag[u] += tag[fail[u]];     //这句因题而异，某些情况下不要这句话
            }
            else           //当trie[u][i]==-1时，设置其为trie[fail[u]][i]，就构造了trie图
                v = trie[fail[u]][i];
        }
    }
}
```