**基于ac算法的敏感字过滤系统**

**简介:**

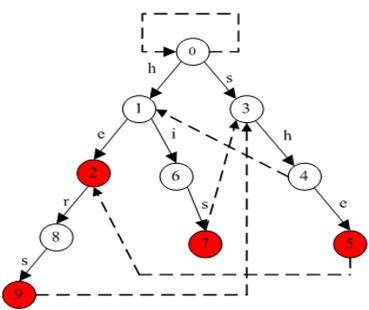
Aho-Corasick算法简称AC算法，通过将模式串预处理为确定有限状态自动机，扫描文本一遍就能结束。其复杂度为O(n)，即与模式串的数量和长度无关。

**思想:**

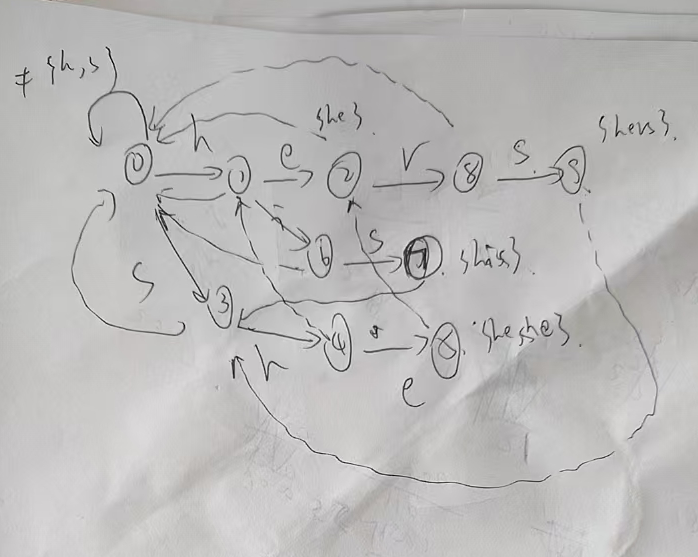
自动机按照文本字符顺序，接受字符，并发生状态转移。这些状态缓存了“按照字符转移成功（但不是模式串的结尾）”、“按照字符转移成功（是模式串的结尾）”、“按照字符转移失败”三种情况下的跳转与输出情况，因而降低了复杂度。

**原理:**

AC算法中有三个核心函数，分别是：success; 成功转移到另一个状态（也称goto表或success表）failure; 不可顺着字符串跳转的话，则跳转到一个特定的节点（也称failure表），从根节点到这个特定的节点的路径恰好是失败前的文本的一部分。emits; 命中一个模式串(也称output表)



其实上图省略了到根节点的fail边，完整的自动机如下图：



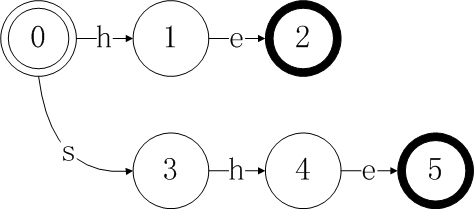
**goto表构造过程****:**

把上图的虚线去掉，实线部分就是一棵trie树了。

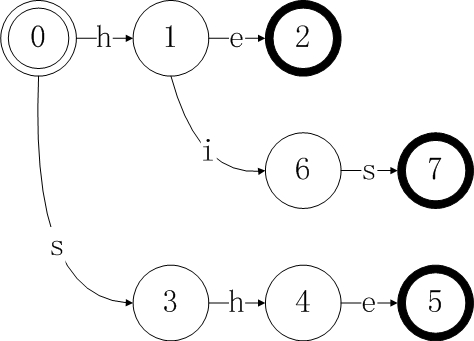
第一步，将模式he加入goto表：



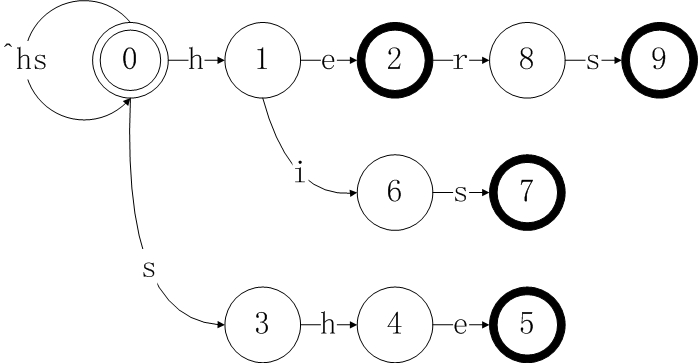
第二步，将模式she加入goto表：



第三步，将模式his加入goto表：



第四步，将模式hers加入goto表：



**failure表构造过程:**

失效函数fail决定了当goto函数得到的下一个状态无效时，应该回退到哪一个状态。在构造fail函数时，我们首先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。以我们上面构造的状态转移图为例，起始状态的深度为0，状态1和3的深度是1，状态2、6、4的深度是2，依次类推。计算失效函数的思路是这样的：首先计算深度为1 的状态的失效函数值，然后是深度为2的，以此类推，直到所有状态的失效函数值都被计算出。同时，我们规定所有深度为1的状态的fail值为0，假设所有深度小于d的状态的fail值都已经计算出，考虑每个深度为d-1的状态r，基于这些已经被计算出的深度为d-1的fail值，我们是可以得到深度为d的fail值的。

这个计算的过程是这样的：

1.对于所有的字符a，如果goto(r,a) = fail，那么什么也不做（当r为我们上面构造的trie树的叶子节点时，就符合这种情况）

2.如果goto(r,a) == s，我们记state = fail®，执行state = f(state)零次或者若干次，直到使得goto(state,a) != fail，因为goto(0,a) != fail，所以这个状态是一定存在的。

3.记fail(s) = goto(state,a)。

首先规定与状态0距离为1（即深度为1）的所有状态的fail值都为0。

然后设当前状态是S1，求fail(S1)。我们知道，S1的前一状态必定是唯一的（刚才说的一对一），设S1的前一状态是S2，S2转换到S1的条件为接受字符C，测试S3 = goto(fail(S2), C)。

如果成功，则fail(S1) = goto(fail(S2), C) = S3。

如果不成功，继续测试S4 = goto(fail(S3), C)是否成功，如此重复，直到转换到某个有效的状态Sn，令fail(S1) = Sn。

我们还是以上面构造出的状态转移图为例，计算每个节点的fail值，根据规定，fail(1) = fail(3) = 0，因为1和3是深度为1的状态。

考虑深度为2的状态2、6、4：

计算fail(2)，令state = fail(1) = 0，由于goto(0，e) = 0，所以fail(2) = 0

计算fail(4)，令state = fail(3) = 0，由于goto(0，h) = 1，所以fail(4) = 1

计算fail(6)，令state = fail(1) = 0，由于goto(0，i) = 0，所以fail(6) = 0

考虑深度为3的节点8、7、5：

计算fail(8)，令state = fail(2) = 0，因为goto(0，r) = 0，所以fail(8) = 0

计算fail(7)，令state = fail(6) = 0，因为goto(0，s) = 3，所以fail(7) = 3

计算fail(5)，令state = fail(4) = 1，因为goto(1，e) = 2，所以fail(5) = 2

最后考虑深度为4的节点9：

计算fail(9)，令state = fail(8) = 0，因为goto(0，s) = 3，所以fail(9) = 3

这样一来我们构造的fail表如下：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **状态** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| fail值 | None | 0 | 0 | 0 | 1 | 2 | 0 | 3 | 0 | 3 |

**Out表构造过程:**

在构建goto表的过程中，我们知道，状态2，5，7，9是输入的4个模式串的末尾部分，所以如果在执行匹配过程中，达到了如下四个状态，我们就知道对应的模式串被发现了。对于状态机D的某些状态，对应某个完整的模式串已经被发现，我们就用output表来记录这一信息。完成goto表的构建后，D中各状态对应的output表的情况如下：

2 he

5 she

7 his

9 hers

但是这并不是我们最终的output表。下面以构建状态5的fail表为例，说明一下fail表的构建是如何影响output表的。首先根据之前我们的介绍，当我们开始计算D[5]的fail值时，我们要将模式she的所有包含后缀提取出来，包括he，e。这里我们需要注意，在output表中，状态5是一个输出状态。当我们用he在状态机中执行转移时，我们会成功转移到2，这里output[2]也是一个输出状态，这就意味着在发现模式串she的同时，实际上也发现了模式串he，所以如果通过某种转换，我们到达了状态5，则意味着我们发现了she和he两个模式，此时fail[5]=2，所以我们需要将output[2]所包含的输出字符串加入到output[5]中。完成goto和fail表构建后，我们所得到的最终output表为：

2 he

5 she，he

7 his

9 hers

这实际上是一个后缀包含问题，也就是模式p1实际上是模式p2的后缀，所以当发现模式p2时，p1自然也被发现了。

**匹配过程:**

自动机从根节点0出发

首先尝试按success表转移。按照文本的指示转移，也就是接收一个u。此时success表中并没有相应路线，转移失败。

失败了则按照failure表回去。按照文本指示，这次接收一个s，转移到状态3。

成功了继续按success表转移，直到失败跳转步骤2，或者遇到output表中标明的“可输出状态”。此时输出匹配到的模式串，然后将此状态视作普通的状态继续转移。

算法高效之处在于，当自动机接受了“ushe”之后，再接受一个r会导致无法按照success表转移，此时自动机会聪明地按照failure表转移到2号状态，并经过几次转移后输出“hers”。来到2号状态的路不止一条，从根节点一路往下，“h→e”也可以到达。而这个“he”恰好是“ushe”的结尾，状态机就仿佛是压根就没失败过（没有接受r），也没有接受过中间的字符“us”，直接就从初始状态按照“he”的路径走过来一样（到达同一节点，状态完全相同）.

**代码实现:**

<https://github.com/shenwei1234ab/DirtyWordFilter:> 基于ac状态机实现的敏感字过滤系统c++

<https://jihulab.com/software954306/wordfilter> 基于ac状态机实现的敏感字过滤系统golang

**程序说明:**

1.acMach.h 和acMach.cpp c++实现ac自动状态机功能(支持中文)

2.acsmx.h 和acsmx.cpp 源版c实现ac自动状态机功能(仅仅支持英文)

3.dirtywords.txt 记录敏感字库