**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。

**基本构造**

AC算法中有三个核心函数，分别是：

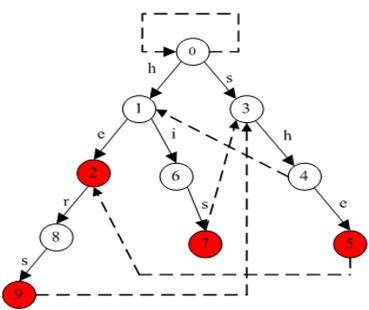
success; 成功转移到另一个状态（也称goto表或success表）

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。

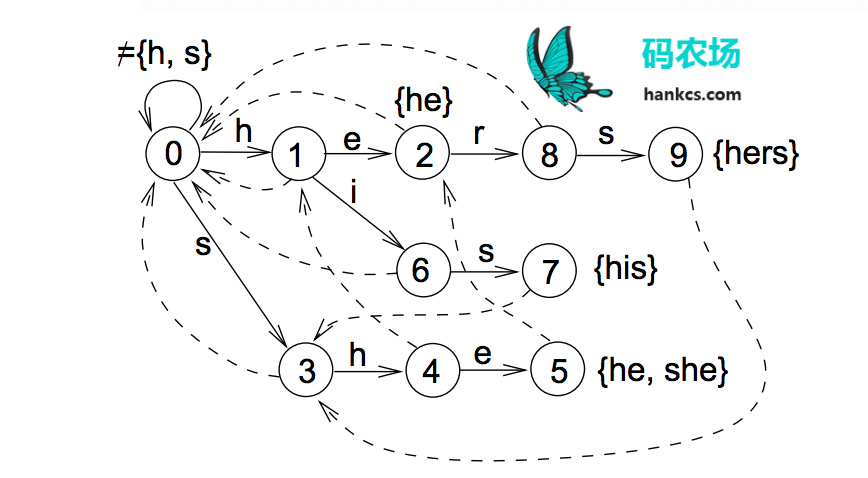
emits; 命中一个模式串（也称output表）

**举例**

以经典的ushers为例，模式串是he/ she/ his /hers，文本为“ushers”。构建的自动机如图：



其实上图省略了到根节点的fail边，完整的自动机如下图：



**构造过程**

看来这三个表很厉害，不过，它们是怎么计算出来的呢？

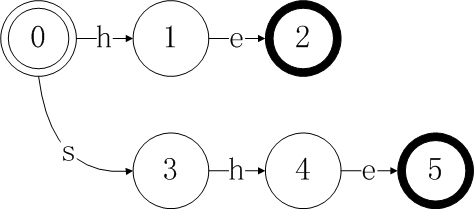
**goto表**

很简单，了解一点trie树知识的话就能一眼看穿，goto表就是一棵trie树。把上图的虚线去掉，实线部分就是一棵trie树了。

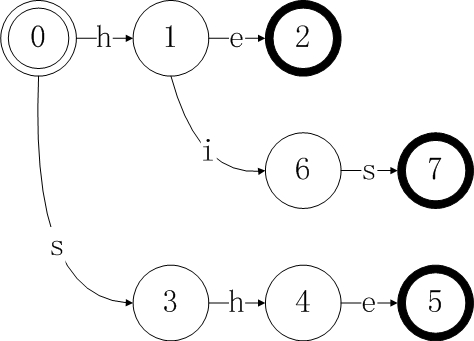
第一步，将模式he加入goto表：



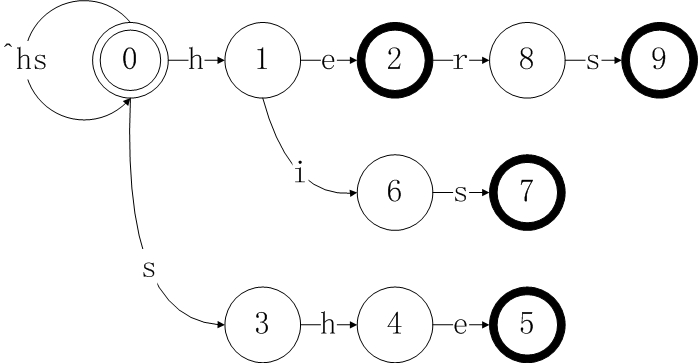
第二步，将模式she加入goto表：



第三步，将模式his加入goto表：



第四步，将模式hers加入goto表：



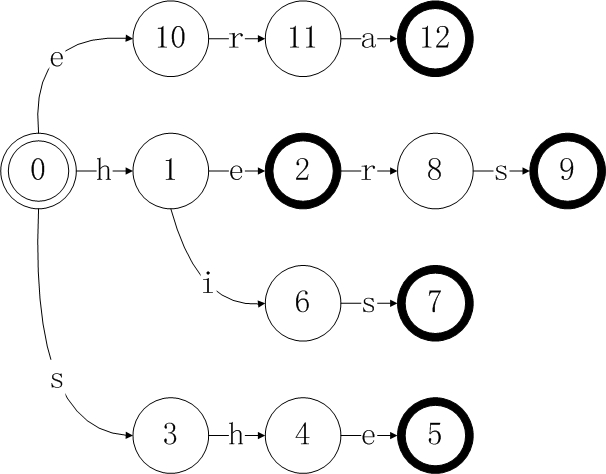
对于第一和第二步而言，两个模式没有重叠的前缀部分，所以每输入一个字符，都对应一个新状态。第三步时，我们发现，D[0][p3[1]]=D[0]['h']=1，所以对于新模式p3的首字母'h'，我们不需要新增加一个状态，而只需将D的当前状态转移到D[1]即可。而对于模式p4其前两个字符he使状态机转移至状态D[2]，所以其第三字符对应的状态D[8]就紧跟在D[2]之后。

failure表

goto表构建完成之后，我们就要构建fail表，所谓的fail表就是当我们处在状态机的某个状态D[p]时，此时的输入字符c使得D[p][c]=0，那么我们应该转移到状态机的哪个位置来继续进行呢。以输入文本"shers"为例，当输入到字母e时，我们会发现匹配模式(she)rs，对应与状态机的状态D[5]，然后输入字母r，此时我们发现D[6]['r']=0，对于字母r D[6]不存在有意义的跳转。此时我们不能跳转回状态D[0]，这样就会丢掉可能的匹配s(hers)。我们发现s(he)的后缀he是模式(he)rs的一个前缀，所以当匹配模式she时，实际也已经匹配了模式hers的前缀he，此时我们可以将状态D[6]转移到hers中的前缀he在goto表中的对应状态D[2]处，再向后执行跳转匹配。这一跳转，就是AC算法中的fail跳转，要实现正确的fail跳转，还需要满足一系列条件，下面会逐一说明。

对于模式串she，其在字母e之后发生了匹配失败，此时其对应的模式串（回溯到状态D[0]）就是she。对于she来说，它有两个包含后缀（除字符串自身外的所有后缀），he和e，对于后缀he，将其输入自动机D，从状态D[0]可以转移到状态D[2]，对于后缀e，没有可行的状态转移方案。所以对于状态D[5]，如果对于新输入的字符c没有可行的转移方案，我们可以跳转到状态D[2]，考察D[2][c]是否等于0.

AC两人在论文中举出的例子，并不能涵盖在构建fail时遇到的所有情况，这里特别说明一下。前面我们说过，对于she的包含后缀e，没有可行的转移方案，此时如果模式串中还包含一个模式era，那么D[5]可不可以转移到状态D[10]去呢，实际上这是不行的，我们需要找到的是当前所有包含后缀中最长的满足条件者（拗口），如果D[5]对于失败的输入c优先转移到D[10]，那么对于文本串shers，很显然会漏掉可能匹配hers，那么什么时机才应该转移到D[10]呢，当我们处理模式串hers时，处理到D[2]时对于之前的输入he，其最长的包含后缀是e，将e输入自动机，可以转移到D[10]，所以在D[2]处发生匹配失败的时候才应该转移到D[10]。所以当我们在D[5]处匹配失败时，要先跳转到D[2]如果再没有可用的转移，再跳转到D[10]。



这个例子同时说明，对于模式集合P的所有模式pi，我们需要处理的不仅是pi的所有包含后缀，而是pi的所有非前缀子串。以模式hers为例，其在2，8，9三个状态都可能发生匹配失败，所以我们要提取出hers的所有非前缀子串(e，er，r，ers，rs，s)，然后按照这些子串的末尾字符所对应的自动机状态分组（上例就可以分组为{e}对应状态2，{er，r}对应状态8，{ers，rs，s}对应状态9），然后分别将这些组中的子串从D[0]开始执行状态转移，直到没有可行的转移方案，或者整个序列使状态机最终转移到一个合法状态为止。如果一组中的所有子串都不能使状态机转移到一个合法状态，则这组子串所对应的状态的fail值为0，如果存在可行的状态转移方案，则选择其中最长的子串经过转移后的最终状态，令其对应的组的状态的fail值与其相等。

举例说明，当我们要处理模式串hers的fail表，假设已经构建好的goto表如前图所示，首先我们需要考察状态2，此时hers的输入字符是he，其所有包含后缀只有e，我们让e从D[0]开始转移，发现成功转移到D[10]，所以fail[2]=10。然后我们考察状态8，此时hers的输入字符是her，所有包含后缀为er，r，因为我们要找到可以实现转移的最大包含后缀，所以我们先让er从D[0]开始转移，发现成功转移到D[11]，所以fail[8]=11，这是虽然后缀e也可以成功转移到D[10]，但是不是当前包含后缀分组中的子串所能实现的最长跳转，放弃。然后我们考察9，此时hers的输入字符串是hers，所有包含后缀为ers，rs，s，我们依次让其执行状态转换，发现s是可以实现转移的最长子串，转移到D[3]，所以fail[9]=3。

构造方法：

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

最后我们来说一下AC算法中的output表，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。（note1）

**匹配过程**

自动机从根节点0出发

首先尝试按success表转移（图中实线）。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（图中虚线）。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”（图中红色状态）。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