Aho-Corasick算法的Java实现与分析

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。

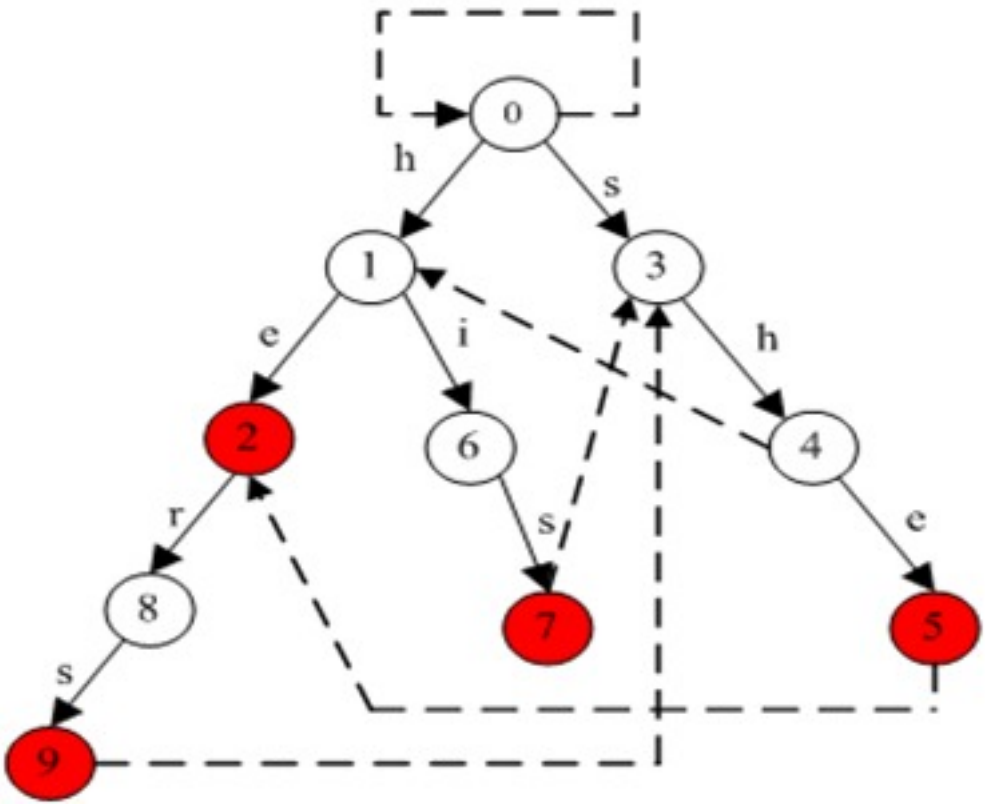
基本构造

AC算法中有三个核心函数，分别是：

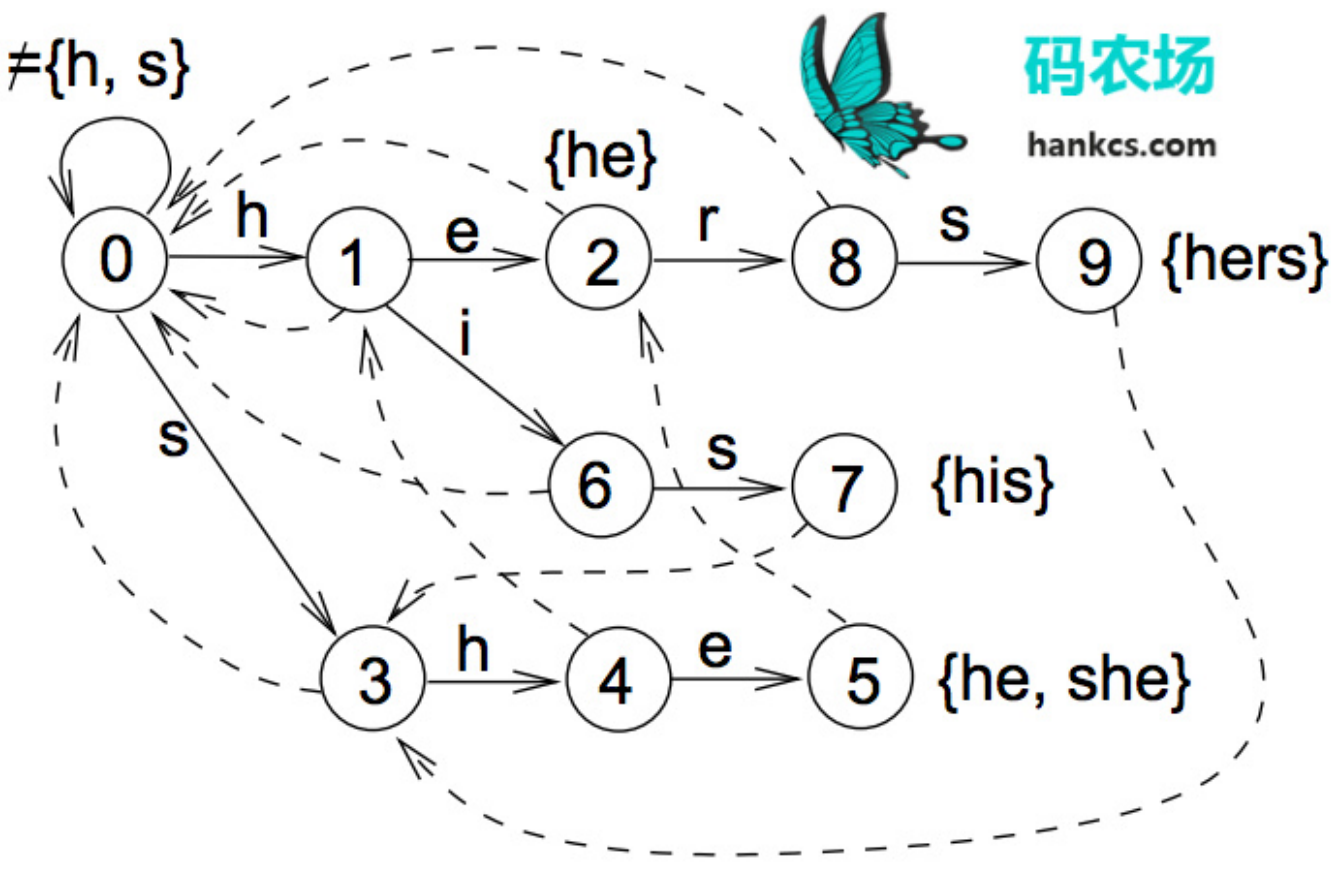
- success; 成功转移到另一个状态（也称goto表或success表）
- failure; 不可顺着字符串跳转的话，则跳转到一个特定的节点（也称failure表），从根节点到这个特定的节点的路径恰好是失败前的文本的一部分。
- emits; 命中一个模式串（也称output表）

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：



匹配过程

自动机从根节点0出发

1. 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。

2. 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。

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。

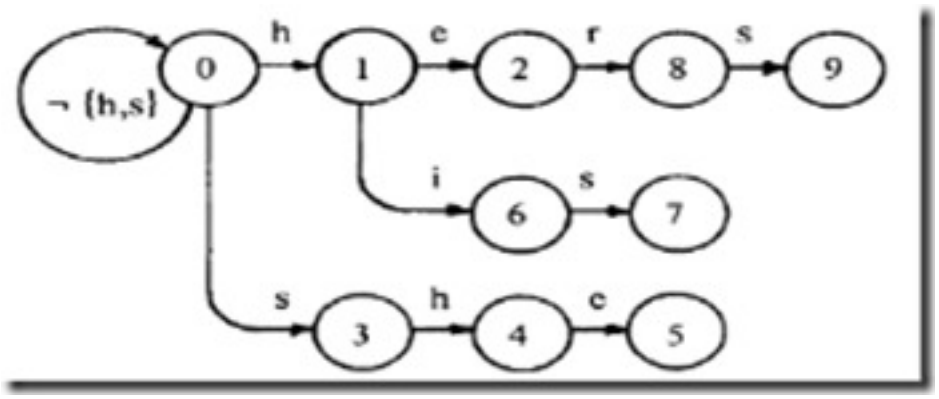
算法高效之处在于，当自动机接受了“ushe”之后，再接受一个r会导致无法按照success表转移，此时自动机会聪明地按照failure表转移到2号状态，并经过几次转移后输出“hers”。来到2号状态的路不止一条，从根节点一路往下，“h→e”也可以到达。而这个“he”恰好是“ushe”的结尾，状态机就仿佛是压根就没失败过（没有接受r），也没有接受过中间的字符“us”，直接就从初始状态按照“he”的路径走过来一样（到达同一节点，状态完全相同）。

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。



output表

output表也很简单，与trie树里面代表这个节点是否是单词结尾的结构很像。不过trie树只有叶节点才有“output”，并且一个叶节点只有一个output。下图却违背了这两点，这是为什么呢？其实下图的output会在建立failure表的时候进行一次拓充。

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

以上两个表通过一个dfs就可以构造出来。关于trie树的更详细内容，请参考：《Ansj分词双数组Trie树实现与arrays.dic词典格式》，《Trie树分词》，《双数组Trie树(DoubleArrayTrie)Java实现》。

failure表

这个表是trie树没有的，加了这个表，AC自动机就看起来不像一棵树，而像一个图了。failure表是状态与状态的一对一关系，别看图中虚线乱糟糟的，不过你仔细看看，就会发现节点只会发出一条虚线，它们严格一对一。

这个表的构造方法是：

1. 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。
2. 然后设当前状态是S_1，求fail(S_1)。我们知道，S_1的前一状态必定是唯一的（刚才说的一对一），设S_1的前一状态是S_2，S_2转换到S_1的条件为接受字符C，测试S_3 = goto(fail(S_2), C)。
3. 如果成功，则fail(S_1) = goto(fail(S_2), C) = S_3。
4. 如果不成功，继续测试S_4 = goto(fail(S_3), C)是否成功，如此重复，直到转换到某个有效的状态S_n，令fail(S_1) = S_n。

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

Java实现

原理谁都可以说几句的，可是优雅健壮的代码却不是那么容易写的。我考察了Git上几个AC算法的实现，发现robert-bor的实现非常好。一趟代码看下来，学到了不少设计上的知识。我fork了下来，针对Ascii做了优化，添加了中文注释。

另外，我实现了基于双数组Trie树的AC自动机：《Aho Corasick自动机结合DoubleArrayTrie极速多模式匹配》。性能更高，内存可控。

开源项目

开源在https://github.com/hankcs/aho-corasick。

调用方法

```
1.          Trie trie = new Trie();
2.          trie.addKeyword("hers");
3.          trie.addKeyword("his");
4.          trie.addKeyword("she");
5.          trie.addKeyword("he");
6.          Collection<Emit> emits = trie.parseText("ushers");
7.          System.out.println(emits);
```

输出：

```
1. [2:3=he, 1:3=she, 2:5=hers]
```

此外，还有一些配置选项：

```
1.      /**
2.       * 大小写敏感
3.       * @return
4.       */
5.      public Trie caseInsensitive()
6.      {
7.          this.trieConfig.setCaseInsensitive(true);
8.          return this;
9.      }
10.
11.     /**
12.      * 不允许模式串在位置上前后重叠
13.      * @return
14.      */
15.     public Trie removeOverlaps()
16.     {
17.         this.trieConfig.setAllowOverlaps(false);
18.         return this;
```

```
19.      }
20.
21.      /**
22.       * 只匹配完整单词
23.       * @return
24.       */
25.      public Trie onlyWholeWords()
26.      {
27.          this.trieConfig.setOnlyWholeWords(true);
28.          return this;
29.      }
```

org.ahocorasick.trie包

这里封装了Trie树，其中比较重要的类是Trie树的节点State：

org.ahocorasick.trie

public abstract class **State**

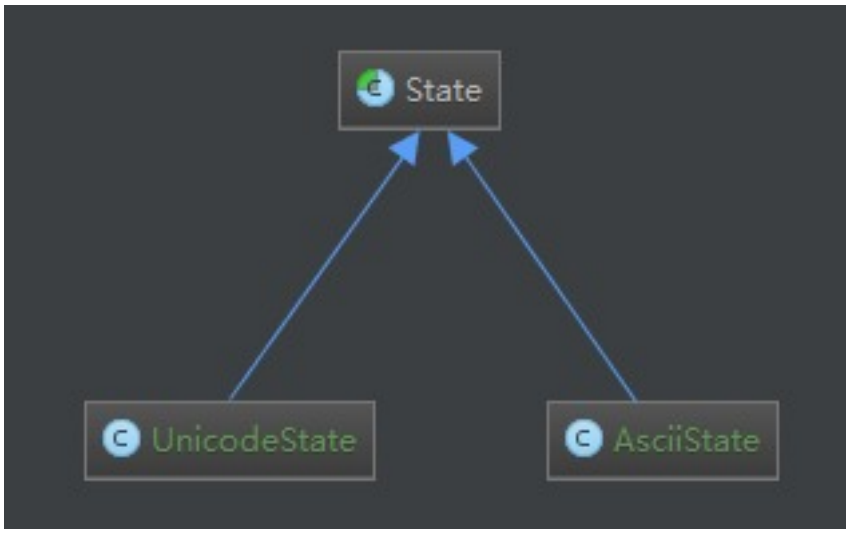
extends Object

一个状态有如下几个功能

- success; 成功转移到另一个状态
- failure; 不可顺着字符串跳转的话，则跳转到一个浅一点的节点
- emits; 命中一个模式串

根节点稍有不同，根节点没有 failure 功能，它的"failure"指的是按照字符串路径转移到下一个状态。其他节点则都有failure状态。

我重构了State，将其异化为UnicodeState和AsciiState类。其中UnicodeState类使用 Map<Character, State> 来储存goto表，而AsciiState类使用数组 State[] success = new State[256]来储存，这样在Ascii表上面，AsciiState的匹配要稍微快一些，相应的在构建时会慢一些，内存占用也会多一些。



从对万字的英语词典的测试结果来看，AsciiState的确有那么一点优势：

```
asciiTrie adding time:1013ms
unicodeTrie adding time:96ms

asciiTrie building time:903ms
unicodeTrie building time:312ms

asciiTrie parsing time:355ms
unicodeTrie parsing time:463ms
```

org.ahocorasick.interval包

这里封装了一棵线段树，关于线段树的介绍请查看：线段树。

线段树用于修饰最后的匹配结果，匹配结果中有一些可能会重叠，比如she和he，这棵线段树对匹配结果（一系列区间）进行索引，能够在log(n)时间内判断一个区间与另一个是否重叠。详细的实现请看代码，都有中文注释，应该很好懂。

基于双数组Trie树的Aho Corasick自动机

AC自动机能高速完成多模式匹配，然而具体实现聪明与否决定最终性能高低。大部分实现都是一个Map<Character, State>了事，无论是TreeMap的对数复杂度，还是HashMap的巨额空间复杂度与哈希函数的性能消耗，都会降低整体性能。

双数组Trie树能高速O(n)完成单串匹配，并且内存消耗可控，然而软肋在于多模式匹配，如果要匹配多个模式串，必须先实现前缀查询，然后频繁截取文本后缀才可多匹配，这样一份文本要回退扫描多遍，性能极低。

如果能用双数组Trie树表达AC自动机，就能集合两者的优点，得到一种近乎完美的数据结构。具体实现请参考《Aho Corasick自动机结合DoubleArrayTrie极速多模式匹配》。

Reference

部分图片和介绍来自：

http://www.cnblogs.com/zzqcn/p/3525636.html

http://blog.csdn.net/sealyao/article/details/4560427

 码农场 » Aho-Corasick算法的Java实现与分析