# 多模式匹配算法——Aho-Corasick自动机

软件17-2-11 李荣杰

### 摘要

我们知道单模式匹配的经典算法kmp可以解决两个串之间的匹配，即模式串与文本串匹配。然后面对有多个模式串的时候，这样匹配就十分消耗时间，复杂度为O(n\*(n+m))。于是AC自动机就是解决这类问题。在数据结构中，有个经典的数据结构为Trie，又称字典树。我们通过对多个模式串建立Trie，对于Trie的每个节点我们都有一个类似kmp中next数组的fail指针。Fail指针指向当前模式串且以当前点结尾的所有后缀与Trie上的所有前缀的匹配的最长位置。有了这个我们就可以在Trie上进行类似kmp的匹配，从而完成多模式匹配。复杂度为O(n)。

【关键字】Trie,kmp,多模式匹配

### 概述

模式匹配算法是信息领域中的重要内容，广泛应用于文本检索，网络入侵检测系统，病毒检测，信息检索，计算机生物学等领域。

模式串匹配可以分为单模式匹配和多模式匹配。其中单模式匹配有个经典的算法是Knuth-Morris-Pratt算法，简称kmp算法。在这个算法中，我们有一个文本串和一个模式串。我们需要的是求模式串与文本串的匹配。当然，直接匹配的时间复杂度有O(n\*m)，所以我们需要对模式串处理。在kmp算法中，有个重要思想，next数组。其表示在当前位置后缀与前缀的最大匹配。所以我们对模式串预处理好next数组。那么在模式串与文本串失配的时候，我们利用next数组将模式串匹配指针跳转到相应位置。因为前面的部分字符我们匹配过了，所以无需重复匹配。这样可以在O(n+m)的时间复杂度解决匹配问题。当然，一旦有两个或两个以上的模式串与文本串匹配，我们可以对每个模式串与文本串进行一次kmp，这样的复杂度为O(n\*(n+m))。但是这个时间复杂度太高，我们发现模式串与模式串之间有一定联系，在文本串与当前模式串匹配时，可能也可以与另一个模式串匹配，那么如果可以通时进行匹配，那么就可以节省很多时间。这时，有个经典的数据结构—Trie。我们可以对所有模式串建立个Trie。Trie每个节点到根节点都是众多模式串中的一个前缀。接下来我们还要在Trie上实现一个类似next数组的fail指针，这样我们就可以在失配时，通过fail指针去转移，从而完成匹配，复杂度为O(n)。

### 相关概念

#### (1).Trie

Trie又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是字符串。对于Trie的每个节点，其到根节点路径上的所有结点组成的串是这个Trie的前缀。Trie可以由一个或者多个字符串构建而成，对于两个具有相同前缀的串，它们在Trie里具有部分重叠节点。

Trie一般是从一个空树开始构建，然后对于每个串依次插入Trie。如果插入串的前缀与Trie的某一前缀有最大匹配，那么从匹配的最后一个节点下将剩下未匹配的插入串部分接上。如下图串”aacb”，在插入这个串前，我们可以知道Trie中存在前缀”aa”，所以我们只需要将”cb”接在最后一个节点下面即可。

例如”aa”,”abab”,”aacb”,”abac”四个串构建的字典树如下:

**图2.1 Trie构建**

#### (2).fail指针

在AC自动机中，存在类似kmp中next数组的工具—fail指针。Fail指针指向当前节点所代表的串的后缀与Trie的所有前缀的最大匹配的位置。

下图是以上文构建的Trie为基础构建的,图中带箭头的曲线为fail指针。

**图2.2 fail指针构建**

其实上图是理想状态的fail指针。正常在构建fail指针时，是使当前节点的fail指针指向父节点的fail指针所指向的节点下面的子节点。这样存在一种情况是父节点的fail指针所指向的节点下面的相应子节点不存在。于是，在构建fail指针的过程中，对于Trie中没有的节点，我们可以建立相应的虚点。所谓虚点为原Trie中不存在的点，没有自己的编号，其编号是其父节点的fail指针所指向的节点下的相应子节点的编号。虚点目的是为了辅助fail指针的建立。当父节点的fail指针所指向的节点下面相应子节点不存在，那么可以指向相应的虚点。其实我们发现这个虚点具有继承性，它始终会指向一个实点（原Trie上的点）。所以fail指针可以通过父节点的fail指针继承下来。

#### (3)多模式匹配

具有多个模式串和一个文本串，现在需要知道这些模式串在文本串中的匹配。这种匹配方式为多模式匹配。

### 详细实现

下面以文本串中出现的模式串的个数为例。（文本串中出现过则记一次）

#### (1)相关结构体

struct tree//字典树

{

int fail;//失配指针

int vis[26];//子节点

int end;//统计以该节点结尾的字符串数量

}AC[N];

#### (2)建立Trie

inline void insert(string s)//向字典树上插入字符串

{

int length = s.length();//获取字符串长度

int now = 0;//初始化插入位置

for(int i = 0; i < length; i ++)

{

if(!AC[now].vis[s[i] - 'a'])//当前位置的子节点无这个字符

AC[now].vis[s[i] - 'a'] = ++ cnt;//分配编号

now = AC[now].vis[s[i] - 'a'];//移动到这个子节点位置

}

AC[now].end ++;//统计以now位置结束的模式串数量

}

因为Trie中，不同串的公共前缀的节点具有相同编号。而一位一位插入这个字符的过程相当于找最长前缀匹配的过程。对于无法匹配的，即无这个节点，所以需要分配编号，建立这个节点。

#### (3)获取fail指针

void get\_fail()//构建fail指针

{

queue <int> Q;//用队列去广度遍历Trie

for(int i = 0; i < 26; i ++)

{

if(AC[0].vis[i] != 0)//与根节点直接相连的点入队

{

AC[AC[0].vis[i]].fail = 0;

Q.push(AC[0].vis[i]);

}

}

while(!Q.empty())

{

int u = Q.front();//取队首元素

Q.pop();//队首元素出队

for(int i = 0; i < 26; i ++)

{

if(AC[u].vis[i] != 0)//遍历到实点

{

AC[AC[u].vis[i]].fail = AC[AC[u].fail].vis[i];//等于父节点的fail指针所指向的节点的相同字符子节点

Q.push(AC[u].vis[i]);//入队

}

else

AC[u].vis[i] = AC[AC[u].fail].vis[i];//构建虚点

}

}

}

如果父节点的fail指针所指向的节点下无相同字符的实点，那就指向相同字符的虚点。容易发现虚点具有继承性，且终究会指向一个实点。所以对fail指针的构建起到了很大的作用。

#### (4)匹配文本串

int AC\_Query(string s)//多模式匹配

{

int length = s.length(), now = 0, ans = 0;

for(int i = 0; i < length; i ++)

{

now = AC[now].vis[s[i] - 'a'];//沿Trie往下匹配

for(int t = now; t && AC[t].end != -1; t = AC[t].fail)//跳转fail指针

{

ans += AC[t].end;

AC[t].end = -1;//出现过的模式串不能重复计算

}

}

return ans;

}

我们发现沿Trie匹配部分相当于文本串的前缀，而fail指针跳转的是这个前缀的后缀与Trie的所有前缀的最大匹配。所有相当于这个文本串的这个子串与这Trie匹配。若跳转位置有end标记，即为模式串，说明文本串的子串与某一模式串匹配成功。时间复杂度为O(n)。

### 结论分析与扩展

对于上面的例子，我们发现可以在O(n)的时间复杂度内求解。然而如果是求每个模式串在文本串中出现的次数，我们会发现这个查询过程的复杂度超过O(n)。因为每次都是需要跳转fail指针，直到跳转到根节点。

如果我们在Trie中只保留fail指针，那么剩下的这个fail图为一个有向无环图。

**图4.1 fail图**

由于重复跳转某一位置，导致复杂度变大，我们采用拓扑排序的方式遍历这个fail图，从而实现某一位置只遍历一次。这样复杂度又回归O(n)。

相关代码：

struct Tree //Trie结构体

{

int fail;

int vis[26];

int end;//标记模式串结尾

int ans;//记录当前位置被遍历的次数

}AC[N];

int cnt, num[N], Mp[N], inq[N];//inq为记录入度的数组

inline void insert(string s, int id)

{

int length = s.length(), now = 0;

for(int i = 0; i < length; i ++)

{

if(!AC[now].vis[s[i] - 'a'])

AC[now].vis[s[i] - 'a'] = ++ cnt;//分配编号

now = AC[now].vis[s[i] - 'a'];

}

if(!AC[now].end)

AC[now].end = id;

Mp[id] = AC[now].end;//对于重复模式串记录模式串结尾的Trie编号

}

void get\_fail()//获取fail指针

{

queue <int> Q;

for(int i = 0; i < 26; i ++)

{

if(AC[0].vis[i] != 0)//对于与根节点直接相连的节点加入队列

{

inq[0] ++;//入度加一

AC[AC[0].vis[i]].fail = 0;

Q.push(AC[0].vis[i]);

}

}

while(!Q.empty())

{

int u = Q.front();

Q.pop();

for(int i = 0; i < 26; i ++)

{

if(AC[u].vis[i] != 0)

{

AC[AC[u].vis[i]].fail = AC[AC[u].fail].vis[i];

inq[AC[AC[u].fail].vis[i]] ++;//fail所指向的节点入度加一

Q.push(AC[u].vis[i]);

}

else

AC[u].vis[i] = AC[AC[u].fail].vis[i];//构建虚点

}

}

}

void AC\_Query(string s)

{

int length = s.length(), now = 0;

for(int i = 0; i < length; i ++)

{

now = AC[now].vis[s[i] - 'a'];

AC[now].ans ++;//先将文本串在Trie的匹配处理出来

}

}

void topu()//拓扑排序

{

queue <int> Q;

for(int i = 0; i <= cnt; i ++)

if(!inq[i])//入度为零的点入队

Q.push(i);

while(!Q.empty())

{

int u = Q.front();

Q.pop();

num[AC[u].end] = AC[u].ans;//这个位置被遍历的次数

int v = AC[u].fail;

inq[v] --;//入度减一

AC[v].ans += AC[u].ans;//fail指针指向的节点继承父节点的遍历次数，因为按照原来的思路，每次都会遍历fail指针，直到根节点。

if(!inq[v])//入度为零继续入队

Q.push(v);

}

}

通过上面两个例子可知，AC自动机在多模式匹配方面有巨大优势，其时间复杂度稳定在O(n)，在第二个例子中，我们甚至用到了fail指针组成的fail图。通过这个fail图我们可以很快的进行多模式匹配，减少了时间的浪费。当然对于正常的求模式串是否在文本串中出现，我们就可以类似第一个例子，在匹配过程中进行跳转fail指针，边遍历边标记，复杂度也是O(n)。

总之，AC自动机可以在线性时间内解决多模式匹配问题。但由于我们对每个节点下都建立了26个节点（实点和虚点）。所以空间复杂度为S(26\*Max(模式串的长度))。可以发现这样消耗的空间是比较多了，所以可以在空间上进行一些优化。

### 参考文献

[1] 陈新驰，韩建民，贾泂。基于AC自动机的多模式匹配算法FACA[A]。计算机工程，2012，38(11):173-176。