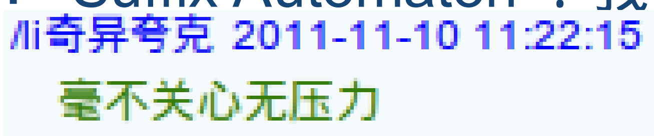
后缀自动机
Suffix Automaton

杭州外国语学校 陈立杰
WJMZBMR

吐槽 & 回答

- Q：你是哪里的弱菜？我听都没听说过！
- A：我是来自杭州外国语学校的陈立杰，确实是弱菜。
- Q： Suffix Automaton ？我根本就没有听说过这种数据结构



- A：这个还是有点用处的，等下我会讲的，你就当长知识了吧。
- Q：呼噜噜 ~~~~~~
- A：睡好。。。

先让我们看 SPOJ 上的一道题目

- 1812. Longest Common Substring II
- 题目大意：给出 N(N <= 10) 个长度不超过 100000 的字符串，求他们的最长公共连续子串。
- 时限： SPOJ 上的 2s

一个简单的做法

- 二分答案之后使用哈希就可以在 O(LlogL) 的时间内解决这个问题。这个做法非常经典就不详细讲了。

看起来很简单。。但是。。。

Longest Common Substring II statistics & best solutions

Users accepted	Submissions	Accepted	Wrong Answer	Compile Error	Runtime Error	Time Limit Exceeded
16	750	57	129	44	89	431

All ADA ASM AWK BASH BF C C# C++ 4.3.2 C99 strict CLPS CLOJ LISP sbcl LISP clisp D ERL F# FORT GO HASK ICON ICK JAR JAVA JS LUA NEM NICE CAML PAS fpc PAS gpc PERL PERL 6 PHP PIKE PRLG PYTH 2.5 PYTH 3.1.2 RUBY SCALA

我们可以看到大部分人都 TLE 了。。为什么呢？

- SPOJ 太慢了
- SPOJ 太慢了
- SPOJ 太慢了
- SPOJ 太慢了
- SPOJ 太慢了
- SPOJ 太慢了
- SPOJ 太慢了

新的算法

2011-07-17 06:32:18 **Tony Beta Lambda**
Are we expected to implement Suffix Array or Suffix Tree?

Jin Bin: Suffix Automaton was expected.

OI 中使用的字符串处理工具

- Suffix Array 后缀数组
- Suffix Tree 后缀树
- Aho-Corasick Automaton AC 自动机
- Hash 哈希

Suffix Automaton又是什么呢？

什么是自动机

- 有限状态自动机的功能是识别字符串，令一个自动机 A ，若它能识别字符串 S ，就记为 A(S)=True ，否则 A(S)=False 。
- 自动机由五个部分组成， alpha ：字符集， state ：状态集合， init ：初始状态， end ：结束状态集合， trans ：状态转移函数。
- 不妨令 trans(s,ch) 表示当前状态是 s ，在读入字符 ch 之后，所到达的状态。
- 如果 trans(s,ch) 这个转移不存在，为了方便，不妨设其为 null ，同时 null 只能转移到 null 。
- null 表示不存在的状态。
- 同时令 trans(s,str) 表示当前状态是 s ，在读入字符串 str 之后，所到达的状态。

trans(s,str)

```
Cur = s;
For i = 0 to Length(str)-1
    Cur = trans(Cur,str[i]);
trans(s,str) 就是 Cur
```

- 那么自动机A能识别的字符串就是所有使得 trans(init,x) ⊂ end的字符串x。
  令其为Reg(A)。

- 从状态s开始能识别的字符串，就是所有使得 trans(s,x) ⊂ end的字符串x。

- 令其为Reg(s)。

后缀自动机的定义

- 给定字符串 S
- S 的后缀自动机 suffix automaton(以后简记为 SAM) 是一个能够识别 S 的所有后缀的自动机。
- 即 SAM(x) = True，当且仅当 x 是 S 的后缀
- 同时后面可以看出，后缀自动机也能用来识别 S 所有的子串。

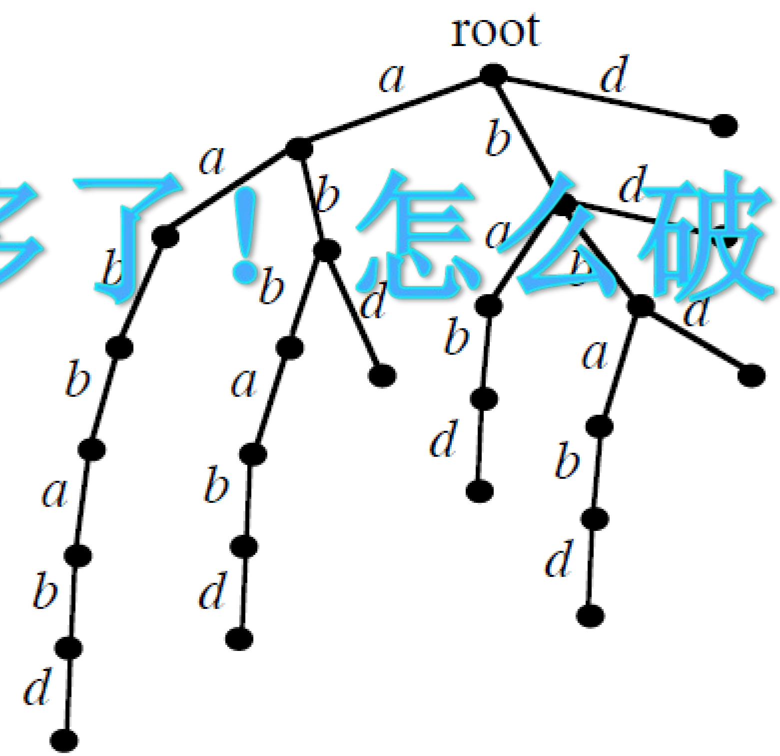
最简单的实现

考虑字符串” aabbabd”

我们可以讲该字符串的所有后缀插入一个Trie中，就像右图那样。那么初始状态就是根，状态转移函数就是这颗树的边，结束状态集合就是所有的叶子。

状态数量太多了！怎么破？

注意到这个结构对于长度为N的串，会有O(N^2)的节点。



最简状态后缀自动机

- 顾名思义，就是状态数最少的后缀自动机，在后面可以证明它的大小是线性的，我们先来看一些性质。
- 假如我们得到了这个最简状态后缀自动机 SAM 。
- 我们令 ST(str) 表示 trans(init,str) 。就是初始状态开始读入字符串 str 之后，能到达的状态。

分析

- 令母串为*S*，它的后缀的集合为*Suf*，它的连续子串的集合为*Fac*。
  从位置*a*开始的后缀为*Suffix(a)*。
  *S[l,r)*表示S中*[l,r)*这个区间构成的子串。
  下标从0开始。

- 对于一个字符串s，如果它不属于*Fac*，那么*ST(s) = null*。因为s后面加上任何字符串都不可能是*S*的后缀了，没有理由浪费空间。

- 同时如果字符串s属于*Fac*，*那么ST(s) ≠ null*。因为s既然是*S*的子串，就可以在后面加上一些字符使得其变成*S*的后缀。我们既然要识别所有的后缀，就不能放过这种可能性。

分析

- 我们不能对每个$s \in Fac$都新建一个状态，因为Fac的大小是$O(N^2)$的。
- 我们考虑$ST(a)$能识别哪些字符串，即$Reg(ST(a))$
- 字符串x能被自动机识别，当且仅当$x \in Suf$。
- $ST(a)$能够识别字符串x，当且仅当$ax \in Suf$。因为我们已经读入了字符串a了。
- 也就是说ax是S的后缀。那么x也是S的后缀。$Reg(ST(a))$是一些后缀集合。
- 对于一个状态s，我们唯一关心的是$Reg(s)$。

分析

- S=ABBBABBABBBBBABBA
- BBABBABBBBBABBA
- BBABBBBBABBA
- BBABBA
- BBA。
- 如果*a*在*S*中的*[l,r)*位置出现，那么他就能识别S从r开始的后缀。
- 例子。
  那么如果*a*在*S*中的出现位置集合是$\{[l_1,r_1),[l_2,r_2),\ldots,[l_n,r_n)\}$
- 那么*Reg(ST(a))*就是$\{Suffix(r_1),Suffix(r_2),\ldots,Suffix(r_n)\}$。
- 不妨令*Right(a)*=$\{r_1,r_2,\ldots,r_n\}$那么*Reg(ST(a))*就完全由*Right(a)*决定。

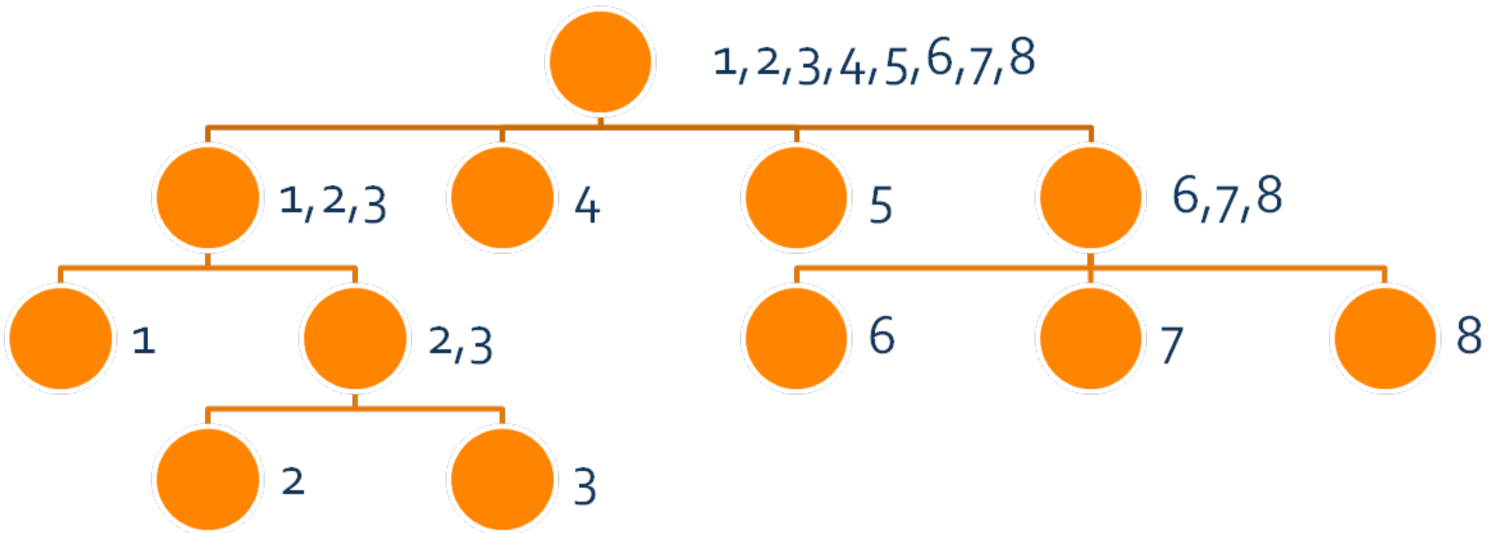
分析

- 那么对于两个子串 $a, b \in Fac$ 如果*Right(a) = Right(b)*,那么*ST(a) = ST(b)*。
- 所以一个状态s,由所有Right集合是Right(s)的字符串组成。
- 不妨令r $\in Right(s)$，那么只要给定子串的长度*len*，该子串就是S[r-len,r)。即给定*Right*集合后，再给出一个长度就可以确定子串了。
- 考虑对于一个Right集合，容易证明如果长度l, r合适，那么长度$l \leq m \leq r$的m也一定合适。所以合适长度必然是一个区间。
- 不妨令*s*的区间是*[Min(s),Max(s)]*

状态数的线性证明

- 我们考虑两个状态a,b。他们的$Right$集合分别为Ra,Rb。
- 假设Ra和Rb有交集，不妨设$r \in Ra \cap Rb$。
- 那么由于a和b表示的子串不会有交集，所以$[Min(a),Max(a)]$和$[Min(b),Max(b)]$也不会有交集。
  不妨令$Max(a)<Min(b)$。那么a中所有长度都比b中短，由于都是由r往前，所以a中所有串都是b中所有串的后缀。因此a中某串出现的位置，b中某串也必然出现了。所以$Ra \subset Rb$。既Ra是Rb的真子集。
- 那么，任意两个串的Right集合，要么不相交，要么一个是另一个的真子集。

状态数的线性证明

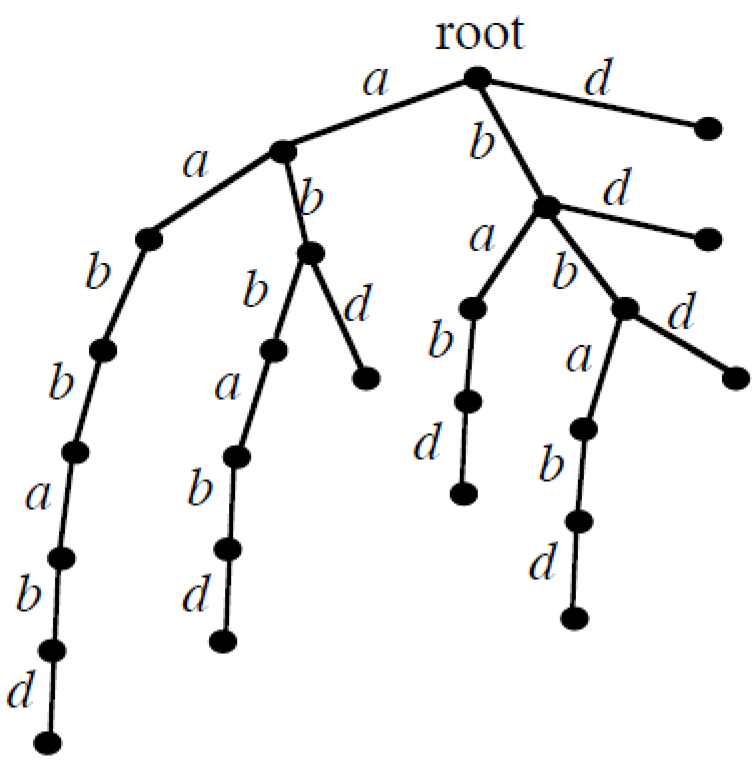


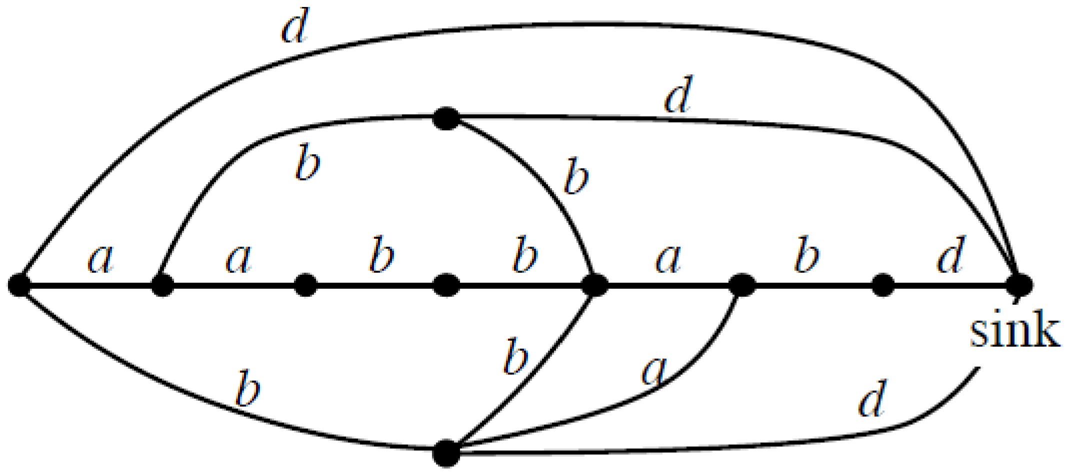
状态数的线性证明

-
- 从上图中我们可以看出*Right*集合实际上构成了一个树形结构。不妨称其为Parent树。
- 在这个树中，叶子节点的个数只有*N*个，同时每内部个节点至少有2个孩子，容易证明树的大小必然是O(N)的。
- 令一个状态*s*，我们令*fa=Parent(s)*表示上面那个图中，它的父亲。那么$Right(fa) \supset Right(s)$，并且Right(fa)的大小是其中最小的。
- 考虑长度，*s*的范围是*[Min(s),Max(s)]*，为什么长度*Min(s)-1*为什么不符合要求？可以发现肯定是因为出现的地方超出了*Right(s)*。同时随着长度的变小，出现的地方越来越多，那么*Min(s)-1*就必然属于*fa*的范围。那么
- *Max(fa) = Min(s)-1*

一些性质

- 我们已经证明了状态的个数是*O(N)*的，为了证明这是一个线性大小的结构，我们还需要证明边的大小是*O(N)*的。
- 如果*trans(a,c) == null*，我们就没有必要储存这条边，我们只需要储存有用的边。
- 不妨*trans(a,c)=b*的话，就看成有一条 $a \to b$ 的标号为c的边。
- 我们首先求出一个*SAM*的生成树（注意，跟之前提到的树形结构没有关系），以*init*为根。


root


sink

一些性质

- 那么令状态数为*M*，生成树中的边最多只有*M*-1条，接下来考虑非树边。对于一条非树边*a*→*b*标号为*c*。
- 我们构造：
- 生成树中从根到状态*a*的路径+(*a*->*b*)+*b*到任意一个*end*状态。
- 可以发现这是一条从*init*到*end*状态的路径，由于这是一个识别所有后缀的后缀自动机，因此这必然是一个后缀。
- 那么一个非树边可以对应到多个后缀。我们对每个后缀，沿着自动机走，将其对应上经过的第一条非树边。
- 那么每个后缀最多对应一个非树边，同时一个非树边至少被一个后缀所对应，所以非树边的数量不会超过后缀的数量。
- 所以边的数量也不会超过O(N)

关于子串的性质

- 由于每个子串都必然包含在 *SAM* 的某个状态里。
- 那么一个字符串 *s* 是 *S* 的子串，当且仅当， *ST(s)!=null*

- 那么我们就可以用 *SAM* 来解决子串判定问题。
- 同时也可以求出这个子串的出现个数，就是所在状态 *Right* 集合的大小。

关于子串的性质

- 在一个状态中直接保存 *Right* 集合会消耗过多的空间，我们可以发现状态的 *Right* 就是它 Parent 树中所有孩子 Right 集合的并集，进一步的话，就是 Parent 树中它所有后代中叶子节点的 Right 集合的并集。
- 那么如果按 dfs 序排列，一个状态的 right 集合就是一段连续的区间中的叶子节点的 Right 集合的并集，那么我们也就可以快速求出一个子串的所有出现位置了。
- 树的 dfs 序列：所有子树中节点组成一个区间。

线性构造算法

- 我们的构造算法是 Online 的，也就是从左到右逐个添加字符串中的字符。依次构造 *SAM* 。
- 这个算法实现相比后缀树来说要简单很多，尽管可能不是非常好理解。
- 让我们先回顾一下性质

定义和性质的回顾

- 状态 *s*，转移 *trans*，初始状态 *init*，结束状态集合 *end* 。
- 母串 *S*， *S* 的后缀自动机 *SAM*(*Suffix Automaton* 的缩写）。
- *Right(str)* 表示 *str* 在母串 *S* 中所有出现的结束位置集合。
- 一个状态 *s* 表示的所有子串 *Right* 集合相同 , 为 *Right(s)* 。
- *Parent(s)* 表示使得 *Right(s)* 是 *Right(x)* 的真子集，并且 *Right(x)* 的大小最小的状态 *x* 。
- Parent 函数可以表示一个树形结构。不妨叫它 Parent 树

定义的回顾

- 一个 *Right* 集合和一个长度定义了一个子串。
- 对于状态 *s*，使得 *Right(s)* 合法的子串长度是一个区间，为 |
  [Min(s),Max(s)]
- *Max(Parent(s)) = Min(s)-1* 。
- *SMA* 的状态数量和边的数量，都是 *O(N)* 的。
- 不妨令 *trans(s,ch)==null* 表示从 *s* 出发没有标号为 *ch* 的边，

定义的回顾

- 考虑一个状态*s*,它的*Right*(s)=$\{r_1, r_2, \ldots, r_n\}$，假如有一条$s \rightarrow t$标号为*c*的边，考虑*t*的*Right*集合，由于多了一个字符，s的Right集合中，只有S[r_i]==c的符合要求。那么t的Right集合就是{r_i+1|*S[ri]==c*}
- 那么如果*s*出发有标号为*x*的边，那么*Parent(s)*出发必然也有。
- 同时，对于令*f=Parent(s)*，
- *Right(trans(s,c))* ⊆ *Right(trans(f,c))*。
- 有一个很显然的推论是*Max(t)>Max(s)*

每个阶段

- 我们每次添加一个字符，并且更新当前的*SAM*使得它成为包含这个新字符的*SAM*。

- 令当前字符串为*T*，新字符为*x*,令*T*的长度为*L*
- $SAM(T) \rightarrow SAM(Tx)$
- 那么我们新增加了一些子串，它们都是串*Tx*的后缀。
- *Tx*的后缀，就是*T*的后缀后面添一个*x*

每个阶段

- 那么我们考虑所有表示*T*的后缀(也就是*Right*集合中包含L)的节点$v_1, v_2, v_3, \ldots$。
- 由于必然存在一个*Right(p)={L}*的节点*p(ST(T))*。那么$v_1, v_2, \ldots, v_\mathrm{k}$由于*Right*集合都含有L，那么它们在*Parent*树中必然全是*p*的祖先。可以使用*Parent*函数得到他们。
- 同时我们添加一个字符*x*后，令*np*表示*ST(Tx)*，则*Right(np)={L+1}*
- 不妨让他们从后代到祖先排为$v_1 = p, v_2, \ldots, v_\mathrm{k} = \mathrm{root}$。

每个阶段

- 考虑其中一个v的*Right*集合={$r_1, r_2, \ldots, r_n = L$}。
- 那么在它的后面添加一个新字符x的话，形成新的状态*nv*的话，只有$S[r_i] = x$的r_i那些是符合要求的。
- 同时在之前我们知道，如果从v出发没有标号为x的边(我们先不看r_n)，那v的*Right*集合内就没有满足这个要求的r_i 。
- 那么由于$v_1, v_2, v_3, \ldots$的*Right*集合逐渐扩大，如果v_i出发有标号为x的边，那么v_{i+1}出发也肯定有。
- 对于出发没有标号为x的边的v，它的*Right*集合内只有r_n是满足要求的，所以根据之前提到的转移的规则，让它连一条到*np*标号为x的边。

每个阶段

- 令v_p为$v_1, v_2, \ldots, v_k$中第一有标号为x的边的状态。
- 考虑v_p的*Right*集合=$\{r_1, r_2, \ldots, r_n\}$，令*trans*($v_p$,*x*)=*q*
- 那么*q*的*Right*集合就是{r_i+1}，S[r_i]=x的集合(注意到这是更新之前的情况，所以r_n是不算的)。
- 注意到我们不一定能直接在*q*的*Right*集合中插入*L+1*。

每个阶段

- 最后一个是x，用红色画出v_p的结束位置上，长度为$Max(v_p)$的串。
- 用蓝色画出q的结束位置上，长度为$Max(q)$的串。
- 串：AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAx
- AAAAAAxAAAAAAAAAAAAAAxAAAAAAAAAAAAAAx

  AAAAAAxAAAAAAAAAAAAAAxAAAAAAAABAAAAAx
- 从这里可以看出，如果在$Right(q)$中强行插入L+1，会导致$Max(q)$变小。从而引发一系列的问题。
- 当然如果$Max(q) == Max(v_p)+1$,就不会有这样的问题，直接插入即可，我们只要让$Parent(np)=q$，就可以结束这个阶段了。

每个阶段

- 这个时候，我们注意到q实际上被分成了两份
AAAAAAxAAAAAAAAAAAAAAAxAAAAAAAAAAAAAAx
- AAAAAAxAAAAAAAAAAAAAAxAAAAAAAABAAAAAx
- AAAAAAxAAAAAAAAAAAAAAxAAAAAAAABAAAAAx
- 那么我们新建一个节点*nq*，使*Right(nq)* = *Right(q)* $\cap \{L+1\}$
- 同时可以看出*Max(nq)* = *Max*(v_p)+1。
- 那么由于*Right(q)*是*Right(nq)*的真子集，所以*Parent(q)* = *nq*。
- 同时*Parent(np)* = *nq*。
- 并且容易证明*Parent(nq)* = *Parent(q)* (原来的)

每个阶段

- 接下来考虑节点 *nq*, 在转移的过程中，结束位置 *L+1* 是不起作用的，所以 *trans(nq)* 就跟原来的 *trans(q)* 是一样的，拷贝即可。

每个阶段

- 接下来，如果新建了节点*nq*我们还得处理。
- 回忆：$v_1, v_2, \ldots, v_\mathrm{k}$是所有*Right*集合包含{L}的节点按后代到祖先排序，其中v_p是第一个有标号为*x*的边的祖先。*x*是这轮新加入的字符。
- 由于$v_\mathrm{p}, \ldots, v_\mathrm{k}$都有标号为*x*的边，并且到达的点的*Right*集合，随着出发点*Right*集合的变大，也会变大，那么只有一段$v_\mathrm{p}, \ldots, v_e$，通过标号为*x*的边，原来是到结点*q*的。
  回忆：$q=Trans(v_\mathrm{p},x)$。
- 那么由于在这里*q*节点已经被替换成了*nq*,我们只要把$v_\mathrm{p}, \ldots, v_e$的*Trans(*,x)*设为*nq*即可。

每个阶段

- 自此我们圆满的解决了转移的问题。
- 嘟嘟噜，搞完啦～～

每个阶段：回顾

- 令当前串为*T*，加入字符为*x*。
- 令*p=ST(T)*，*Right(p)={Length(T)}*的节点。
- 新建*np=ST(Tx)*，*Right(np)={Length(T)+1}*的节点。
- 对*p*的所有没有标号*x*的边的祖先*v*，*trans(v,x)=np*
- 找到*p*的第一个祖先v_p，它有标号*x*的边，如果没有这样的*vp*，那么*Parent(p)=root*，结束该阶段。
- 令*q=trans(vp,x)*，若*Max(q)==Max(vp)+1*，令*Parent(np)=q*，结束该阶段。

每个阶段：回顾

- 否则新建节点 *nq*， *trans(nq,*)=trans(q,*)*
  Parent(nq) = *Parent(q)* (先前的)
  Parent(q) = *nq*
  Parent(np)=nq
- 对所有 *trans(v,x)* == *q* 的 *p* 的祖先 *v*， *trans(v,x)* 改成 *nq*

C++ 的代码实现

```
struct State {
    State*par, *go[26];
    int val;
    State(int _val) :
        par(0), next(0),val(_val) {
        memset(go, 0, sizeof go);
    }
};
State*root,last;
```

C++ 的代码实现

```cpp
void extend(int w) {
    State*p = last;
    State*np = new State(p->val + 1);
    while (p && p->go[w] == 0)
        p->go[w] = np, p = p->par;
    if (p == 0)
        np->par = root;
    else {
        State*q = p->go[w];
        if (p->val + 1 == q->val) {
            np->par = q;
        } else {
            State*nq = new State(p->val + 1);
            memcpy(nq->go, q->go, sizeof q->go);
            nq->par = q->par;
            q->par = nq;
            np->par = nq;
            while (p && p->go[w] == q)
                p->go[w] = nq, p = p->par;
        }
    }
    last = np;
}
```

让我们实战一下吧

实践出真知！

实践是检验真理的唯一标准！

I. 最小循环串

- 给一个字符串 *S*，每次可以将它的第一个字符移到最后面，求这样能得到的字典序最小的字符串。
- 如 *BBAAB*，最小的就是 *AABBB*
- 考虑字符串 *SS*，我们就是要求 *SS* 的长度为 *Length(S)* 且字典序最小的子串，那么我们构造出 *SS* 的 *SAM*，从 *init* 开始每次走标号最小的转移，走 *Length(S)* 步即可以得到结果。
- 为什么这样是对的就留给大家作为小思考了。

II.SPOJ NSUBSTR

- 给一个字符串 *S*，令 *F(x)* 表示 *S* 的所有长度为 *x* 的子串中，出现次数的最大值。求 *F(1)..F(Length(S))*
- *Length(S) <= 250000*
- 我们构造 *S* 的 *SAM*，那么对于一个节点 *s*，它的长度范围是
  [*Min(s),Max(s)*]，同时他的出现次数是 |*Right(s)*| 。那么我们用
- |*Right(s)*| 去更新 *F(Max(s))* 的值。
  同时最后从大到小依次用 *F(i)* 去更新 *F(i-1)* 即可。
- 为什么这样是对的也作为小思考。

III.BZOJ2555 SubString

- 你要维护一个串，支持在末尾添加一个字符和询问一个串在其中出现了多少次这两个操作。
- 必须在线。
- 构造串的 *SAM*，每次在后面加入一个字符，可以注意到真正影响答案的是 *Right* 集合的大小，我们需要知道一个状态的 *Right* 集合有多大。

III.BZOJ2555 SubString

- 回顾构造算法，对 *Parent* 的更改每个阶段只有常数次，同时最后我们插入了状态 *np* ，就将所有 *np* 的祖先的 *Right* 集合大小 + 了 1 。

- 方法 1 ：使用动态树维护 *Parent* 树。
  方法 2 ：使用平衡树维护 *Parent* 树的 *dfs* 序列。

- 这两种方法跟今天的主题无关，不详细讲了。

IV:SPOJ SUBLEX

- 给一个长度不超过 90000 的串 S ，每次询问它的所有不同子串中，字典序第 K 小的，询问不超过 500 个。

- 我们可以构造出 S 的 SAM ，同时预处理从每个节点出发，还有多少不同的子串可以到达。
- 然后 dfs 一遍 SAM ，就可以回答询问了。
- 具体实现作为小练习留给大家。

V:SPOJ LCS

- 给两个长度小于 100000 的字符串 A 和 B，求出他们的最长公共连续子串。
- 我们构造出 A 的后缀自动机 SAM
- 对于 SAM 的每个状态 s，令 r 为 Right(s) 中任意的一个元素，它代表的是结束位置在 r 的，长度在 [Min(s),Max(s)] 之间的所有子串。
  AAAAAAAAAAAAAAAAAAAAArAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAArAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAArAAAAAAAAAAAAAA
- 我们不妨对状态 s，求出所有 B 的子串中，从任意 r 开始往前能匹配的最大长度 L，那么 min(L,Max(s)) 就可以更新答案了。

V:SPOJ LCS

- 我们考虑用 *SAM* 读入字符串 *B* 。
- 令当前状态为 *s*，同时最大匹配长度为 *len*
- 我们读入字符 *x*。如果 *s* 有标号为 *x* 的边，那么 *s=trans(s,x),len = len+1*
- 否则我们找到 *s* 的第一个祖先 *a*，它有标号为 *x* 的边，令 *s=trans(a,x),len=Max(a)+1* 。
- 如果没有这样的祖先，那么令 *s=root,len=0* 。
- 在过程中更新状态的最大匹配长度

V:SPOJ LCS

- 注意到我们求的是对于任意一个 *Right* 集合中的 *r*，最大的匹配长度。那么对于一个状态 *s*，它的结果自然也可以作为它 *Parent* 的结果，我们可以从底到上更新一遍。
- 然后问题就解决了。

VI:SPOJ LCS2

- 给N个长度小于100000的字符串A和B，求出他们的最长公共连续子串。$N \le 10$。
- 我们构造出其中一个A的后缀自动机SAM，用类似上题的方法求出每个其它串对每个状态的最大匹配长度，
- 考虑一个状态s，如果A之外其它串对它的匹配长度分别是$a_1, a_2, \dots, a_{n-1}$，那么$Min(a_1, a_2, \dots, a_{n-1}, Max(s))$就可以更新答案了。

一些其他的东西

- 其实不仅仅有 Suffix Automaton 还有。。
  Factor Automaton
- Suffix Oracle
- Factor Oracle
- Suffix Cactus
- Oracle 的意思是神谕！听起来很强吧。

Factor Oracle

- 一个串的 Factor Oracle 是一个自动机，可以识别这个串的所有子串的集合，但也可以识别一些别的乱七八糟的东西。
- 其实 Oracle 也有预言的意思，所以这个是不一定准的。
- Factor Oracle 的构造算法非常的简单，不过我也不知道这个在 OI 中有什么用，就不讲了。

Queries are welcomed !

广告

- 我会把课件和代码放在我的博客上，地址是：
- http://hi.baidu.com/wjbzbmr/