软件设计与开发实践Ⅰ课程报告

高级数据结构及其应用

项目题目： **后缀自动机**

学号1：1151200203 姓名1: 万丁

学号2：1152710103 姓名2: 杨婉

答辩时间:2017年6月2日

1. **引言**

自动机是[有限状态机](https://zh.wikipedia.org/wiki/%E6%9C%89%E9%99%90%E7%8A%B6%E6%80%81%E8%87%AA%E5%8A%A8%E6%9C%BA" \o "有限状态自动机)（FSM）的数学模型。FSM是给定符号输入，依据（可表达为一个表格的）[转移函数](https://zh.wikipedia.org/wiki/%E8%BD%AC%E7%A7%BB%E5%87%BD%E6%95%B0" \o "转移函数)“跳转”过一系列状态的一种机器。在常见的FSM的“[米利型有限状态机](https://zh.wikipedia.org/wiki/%E7%B1%B3%E5%88%A9%E5%9E%8B%E6%9C%89%E9%99%90%E7%8A%B6%E6%80%81%E6%9C%BA)”（Mealy）变体中，这个转移函数告诉自动机给定当前状态和当前字符的时候下一个状态是什么。

逐个读取输入中的符号，直到被完全耗尽(把它当作有一个字写在其上的磁带，通过自动机的读磁头来读取它；磁头在磁带上前行移动，一次读一个符号)。一旦输入被耗尽，自动机被称为“停止”了。

依赖自动机停止时的状态，称呼这个自动机要么是“接受”要么“拒绝”这个输入。如果停止于“接受状态”，则自动机“接受”了这个字。在另一方面，如果它停止于“拒绝状态”，则这个字被“拒绝”。自动机接受的所有字的集合被称为“这个自动机接受的[语言](https://zh.wikipedia.org/wiki/%E5%BD%A2%E5%BC%8F%E8%AF%AD%E8%A8%80" \o "形式语言)”。

后缀自动机（单词的有向无环图）是用于表示一组字符串的完整索引的有效数据结构。 它们是表示一组字符串的所有后缀或子串的集合的最小有限自动机，能够在线性时间内解决许多字符串问题，例如在某一字符串中搜索另一字符串的所有出现位置，或者计算不同子串的个数。它以压缩的形式包含了长度为n的字符串子串的所有简明信息，仅需0（n）的空间。并且能在0（n）的时间内被构造。

1. **逻辑结构及其性质**
   1. **自动机**

后缀自动机里的自动机其实就是确定有限状态自动机。定义为：

确定有限状态自动机A是由

* 一个非空有限的状态集合 Q
* 一个输入字母表 Σ（非空有限的字符集合）
* 一个转移函数 （例如：）
* 一个开始状态sQ
* 一个接受状态的集合

所组成的5-元组。可以表示成。

后缀自动机即可以识别后缀的自动机。字符串s的后缀自动机A是识别s的后缀的最小有限自动机。 这意味着A是与初始节点i，一组终端节点的有向非循环图，并且节点之间的箭头用s的字母标记。 为了测试w是否是s的后缀，从i的初始节点开始，并且遵循对应于w的字母的箭头就足够了。 如果我们能够为w的每个字母做这个，并且终结在一个终端节点，那么w是s的后缀。

* 1. **状态**

在介绍状态的含义之前先介绍几个定义。

给定字符串T与s，定义endpos（T,s）={end|T[begin:end]=s}.即endpos（T,s）表示所有在T中出现的s的结束位置的下标。有两个字符串s1 与 s2, 如果有 endpos(T,s1)=endpos(T,s2), 则 s1 与 s2 是endpos等价的。

所以，根据endpos的定义，将T中所有子串分成若干个等价类，每个等价类中的子串都是endpos等价的，且每个等价类分别对应一个状态。因此我们可以用endpos(u)来表示下标集合，因为u所表示的子串是endpos等价的。

一个状态的入度可以大于等于1，这意味着状态u可以识别T的某个子串集合，将这个集合表示为substrings(u)。可知，substrings(u)中的任意两个子串必然是endpos等价的。

**引理1**：对于 T 的两个任意的子串 s1 与 s2, 集合 endpos(T,s1) 与 endpos(T,s2)要么相交为空，要么一个是另一个的子集。

**证明：**假设length(s1)length(s2),如果,即s1,s2至少会同时以T的某个字符作为结尾，即s1是s2的后缀，表示为s1s2.（包含s1=s2的情况）。

若s1s2，则，因子串越长，对endpos限制越多，集合就越小。同理，若，则s1s2。由此我们得出如下结论：

* iff s1s2
* 其他情况

下面来讨论，给定一个状态u，substrings(u)的特性。

定义：

1. longest(u): substrings(u) 中最长的子串
2. shortest(u): substrings(u) 中最短的子串
3. maxlen(u): longest(u) 的长度
4. minlen(u): shortest(u) 的长度

根据上文，状态u的划分与endpos等价类的划分一致，所以endpos(shortest(u))=endpos(longest(u))，根据以上结论可知，shortest(u)longest(u)。同理，对于任意的ssubstrings(u),必有s。也就是说，substrings(u)中的所有子串必然是longest(u)的后缀。

同时，若s且,必有ssubstrings(u)。因,由于,所以ssubstrings(u)。这意味着，状态会对应一个连续的后缀集合，每一个子串恰好是上一个子串的后缀且与上一个子串长度相差1.

* 1. **Parent树**

除了通过字符的状态变换，状态间还存在另一种关系。上文给出了endpos集合不是相交为空就是一个是另一个的真子集，通过这个关系可以构造一个树形结构，称其为Parent树。令一个状态u，fa=Parent(u)表示Parent树中u的父亲。那么，且endpos(fa)的大小是其中最小的。考虑长度，u的范围是[minlen(u),maxlen(u)]，长度minlen(u)-1不符合要求是因为该子串出现的地方超出了endpos(u)，因此minlen(u)-1属于fa的范围。即：

maxlen(fa)=minlen(u)-1

qe 
qoqe 
Kldtua 

图 2.1

该图以T=“abcb”为例，其中黑色线表示后缀自动机的状态转移，蓝色线表示Parent树。在每个状态u上标记了longest(u)和shortest(u).

设Ti表示T中以T[i]结尾的前缀。直观来讲，后缀自动机会把Ti的后缀集合切成几个区间，每个区间为一个状态，状态之间的连接形成Parent树。对于某个满足longest(u)= Ti的状态u，从u到树根的路上经过的所用状态对应的substrings集合做并，即可得到Ti所有后缀的集合。

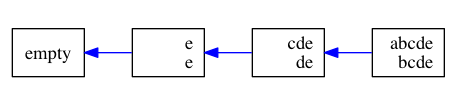


图 2.2

以longest(u)=”abcde”为例，可以看到所有后缀的划分及其状态相连。

* 1. **状态转移**

给定一个状态u与字符c,定义trans(u,c)=v,表示从状态u到状态v存在 状态转移路径,该路径通过字符c标识。

如果有trans(u,c)=v，那么必有,也就是说，往状态u的对应的substrings后面拼接字符c, 得到的新 substrings 集合必然包含于 substrings(v).

考虑到substrings(v)里的substring都是以某个字符c结尾的, 我们可以知道, 对于SAM里所有合法的状态转移trans(ui,ci)=v, 必有 ci=c, 也就是说, 所有以v为终点的状态转移,必然带有相同的字符。

下面要解决的问题是，哪些状态u存在到v的状态转移使其满足后缀自动机的性质。已知对于一个状态v,substrings(v)中的substrings是"连续的".由于到状态v的状态转移都带有相同的字符，所以状态u的集合所包含的子串必然也是连续的。而刚才我们得到的Parent树从叶到根的路径上经过的状态就包含了全部连续子串，所以这些状态的集合即为u的集合。

* 1. **后缀自动机的动态构建**

考虑每次添加一个新字符，更新当前的SAM使它成为包含这个新字符的SAM。如果加入T[i+1],意味着更新后的SAM里必定有一个状态u,满足, 而更新前的SAM必然不存在这样的状态。所以,我们需要创建一个新的状态cur 使：

* Ti+1 substrings(cur)

根据上文，只有Parent树上末节点到根节点上经过的状态u将进行状态转移。这时设状态p为当前正在处理的状态，希望可以建立 trans(p,T[i+1])=cur。但仍存在trans(p,T[i+1])=q，其中 q 是 SAM 里某个已经存在的状态，下面将集中于如何解决这个问题。

在识别到这个trans(p,T[i+1])=q之后,我们可以知道,原先的SAM已经可以识别了Ti+1 的后缀集合的子集, 由此, 可以在Parent树上将状态cur 指向某个 状态pre,这样更新后的SAM即可识别Ti+1 的后缀集合。

设,有maxlen(p)=minlen()-1 ①

由于在处理状态p时已经有trans(,T[i+1])=cur，即minlen(cur)=minlen()+1 ②

由①，②式可得minlen(cur)=minlen()+1=maxlen(p)+2 ③

pre=Parent(cur)，有maxlen(pre)+1=minlen(cur) ④

由③，④式可得maxlen(pre)=maxlen(p)+1

检测状态q=trans(p,T[i+1])是否满足作为pre的条件

* 如果maxlen(p)+1=maxlen(q),那么状态q就是我们要找的pre,直接设 Parent(cur)=q.
* 如果maxlen(p)+1<maxlen(q)，这意味着substring(pre)⊂substring(q). 这个时候, 需要将满足 pre 条件的部分从 q 里面剥离出去

对于maxlen(p)+1<maxlen(q)，我们将以状态q为中点的转移分为两部分，以p为界限。

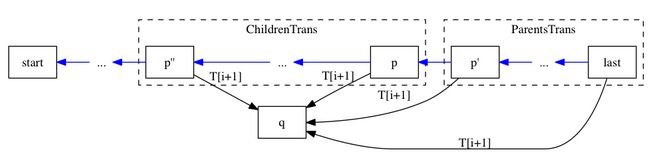


图 2.3

其中状态满足或为初始状态，或trans(Parent(),T[i+1]) 不存在，或trans(Parent(),T[i+1]) 存在，但trans(Parent(),T[i+1])q.

为了将属于pre的部分从状态q中分离出来,可以新建一个状态sq,将 ChildrenTrans中的transitions全部转向sq.如下图所示:

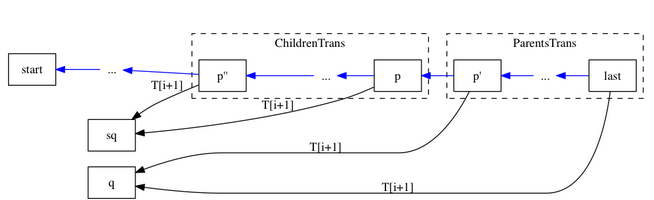


图 2.4

处理完以状态sq为终点的转移之后,还需要考虑以sq为起点的 转移。显然的, 对于所有的 trans(q,c)=q′, 也必须存在有 trans(sq,c)=q′. 所以, 直接把 state q 维护的转移拷贝给 state sq 就可以了.

调整完转移，还需调整Parent树。为保证Parent树的路径状态子串连续性，令Parent(sq)=Parent(q),Parent(q)=sq。最后令Parent(cur)=sq。SAM更新完毕。

1. **存储结构与算法实现**

完成SAM的构建，每个状态u仅需维护maxlen(u),Parent(u),trans(u,\*)即可。

struct state

{

state \*par // Parent(u)

state \*go[26]; // trans(u,\*)

int val; // maxlen(u)

};

把一个字符串对应的后缀自动机定义为一个类。

class SAM

{

public:

bool issuffix(string);//判断一个子串是否是后缀

bool issubstr(string);//判断一个子串是否是其子串

void extends(char);//扩展一个字符

void extendstr(string);//扩展一个字符串

private:

string str;//保存存储的字符串

state \*root = new state(0), \*last = root;//定义起始，终止状态，并且初始化

state\* next(state\*, char);// 对于状态u和字符c返回trans(u,c)

void extend(int);//扩展一个字符所需要的辅助函数

};

算法实现：

bool SAM::issuffix(string st)

{

int b = str.size() - st.size();

int len = str.size();

if (b < 0)

return false;

for(int i=b;i<len;++i)

{

if(str[i]!=st[i-b])

{

return false;

}

}

return true;

}

bool SAM::issubstr(string st)

{

state\* p = root;

for (auto& s : st)//如果tran(init,st)!=null说明是子串，否则中途

//状态变成null说明不是

{

p = next(p, s);

if (p == NULL)

{

return false;

}

}

return true;

}

void SAM::extends(char c)

{

str.push\_back(c);//在保存的字符串里加入字符

extend((int)(c - 97));//调用辅助函数改变相关结构

}

void SAM::extendstr(string st)

{

for(auto& s:st)

{

extends(s);

}

}

state\* SAM::next(state\* m, char c)

{

return m->go[(int)(c - 97)];

}

void SAM::extend(int w)

{

state \*p = last;//令p=ST(T)

state \*np = new state(p->val + 1);//新建np=ST(Tx)

while (p && p->go[w] == 0)//对p没有标号为x的边的祖先v,trans(v,x)=np

{

p->go[w] = np, p = p->par;

}

if (p == 0)//如果第一个祖先vp没有标号为x的边，那么par(np)=root，结束

{

np->par = root;

}

else //否则令q=trans(vp,x)

{

state \*q = p->go[w];

if (p->val + 1 == q->val)//若max(q)==1+max(vp) par(np)=q 结束

{

np->par = q;

}

else //否则新建立结点nq,trans(nq,\*)=trans(q,\*)

// par(nq)=par(q)

{ //par(q)=nq par(np)=nq

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)// 对所有*trans(v,x) == q*的*p*的祖先

//*v*，*trans(v,x)* 改成*nq*

{

p->go[w] = nq, p = p->par;

}

}

}

last = np;

}

在解决实际问题时，可能需要维护更多信息。

1. **实验与结果分析**
   1. **理论分析**
2. **空间复杂度**

由于SAM本质上是图结构，对空间复杂度的考量需要考虑点数和边数。

点数即为状态数，长度为n的字符串s建立的SAM的状态个数不超过2n-1（对于n>=3）。

最初自动机包含一个初始节点，第一步和第二步都会添加一个状态，余下的n-2步每步至多由于需要分割，增加两个状态即：1+2+2(n-2)=2n-1.

边数即为转移数，长度为n的字符串s建立的后缀自动机中，转移的数量不超过3n-3（对于n>=3）。

首先计算连续转移的个数，即转移trans(u,c)=v满足maxlen(u)+1=maxlen(v). 考虑初始节点的最长路径树。这棵树将包含所有连续的转移，树的边数比结点个数小1，这意味着连续的转移个数不超过2n-2.

考虑每个不连续转移；转移trans(u,c)=v。对自动机运行一个合适的字符串u+c+w，其中字符串u表示从初始状态到u经过的最长路径，w表示从v到任意终止节点经过的最长路径。一方面，对所有不连续转移，字符串u+c+w都是不同的（因为字符串u和w仅包含连续转移）。另一方面，每个这样的字符串u+c+w，由于在终止状态结束，它必然是完整串s的一个后缀。由于s的非空后缀仅有n个，并且完整串s不能是某个u+c+w（因为完整串s匹配一条包含n个连续转移的路径），那么不连续转移的总共个数不超过n-1.

2n-2+n-1=3n-3.由此我们得到了转移个数的上限。

综上，构造一个SAM仅需要0(n)的空间。

1. **时间复杂度**

上文说到可以在线性时间内构造SAM，这需要保证字母表的大小是常数。这样，可证明操作个数是线性的。

每个按字符查询转移的操作、添加转移、寻找下一个转移，所有这些操作认为是O(1)的。

观察算法的所有部分，会发现其中三处的线性时间复杂度并不显然：

* 沿Parent树向上走添加字符c的转移
* 将指向q的转移分离出来给sq
* 将q的转移状态复制给sq

后缀自动机的大小（状态和转移的数目）是线性的。那么显然第一处和第三处是渐进线性的，因为每次操作都会增加新的状态和转移。对于第二处，不妨关注shortest(Parent(last))。注意到，在沿着后缀链接上溯的过程中，当前节点的shortest的长度总是严格变小。

显然，在向s中添加新字符之前，shortest(Parent(last))的长度不小于shortest(p)的长度，因为Parent(last)至多是p。而后假设我们由q拷贝得到了节点sq，并试图从p沿后缀链接上溯，将所有通往q的转移重定向为通往sq。设v是shortest(当前节点)，在sq刚刚建立完成后，v=short(p)。然后，在每次沿后缀链接上溯时，v的值都会变小，而如果当前节点存在经过字符c通往q的转移，就意味着q对应的字符串集合中包含v+c，也意味着sq包含的字符串集合中包含v+c。换言之，我们为sq包含的字符串集合找到了一个更短的元素，即减少了short(sq)的长度。

在“向s中添加新字符”的整个流程结束后，有Parent(last)=Parent(cur)=clone。根据上面的讨论，新的shortest(Parent(last))的长度变小（或保持不变），而且这一长度减小的值和上溯的操作数同阶。

综上，shortest(Parent(last))作为s一个后缀的起始位置在整个过程中不断右移，而且每次沿后缀指针上溯都会导致该位置严格右移。由于在程序结束时这一起始位置不超过n，所以这一过程的时间复杂度是线性的。

* 1. **实验数据**

实验包括四个方面：对建立SAM结点个数的统计，对建立SAM所需要的边个数的统计，构建SAM时间统计以及对识别一个子串需要的时间的统计。

对于点和边数使用了五十组数据，统计的是当SAM对应的字符串长度分别是10000到500000时对应需要的结点数和边数。得到数据如下：

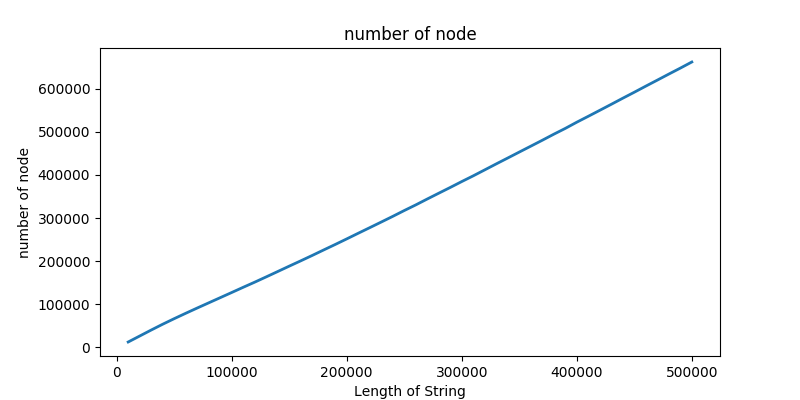


图 4.1

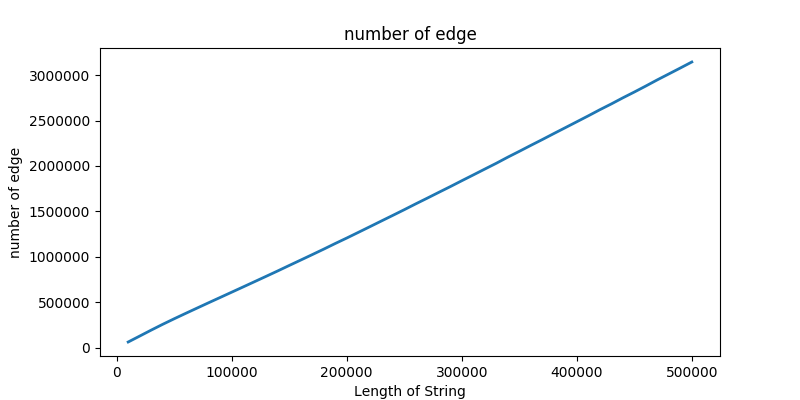


图 4.2

图4.1，4.2表明需要的结点和边数为O(n)。n为SAM对应的串的长度。对上文的理论空间复杂度给出了验证。

对于SAM构建时间采取了和上文同样的方式，得到数据如下：

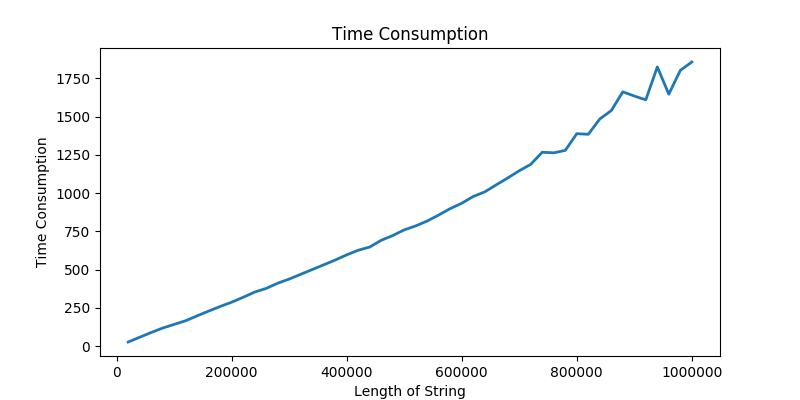


图 4.3

图4.3表明可在线性时间内建立SAM，时间为0(n).

最后对SAM解决的典型问题给出了测试数据，即字符串匹配问题。横坐标表示保证母串和需要识别的子串的长度之比一定的时候（由于我尝试过子串不变改变母串长度，时间并未改变，所以这里认为时间的改变是因为子串），改变子串的长度从1000到50000时，识别子串需要的查询时间。（每个时间代表100次查询长度一样的不同子串）

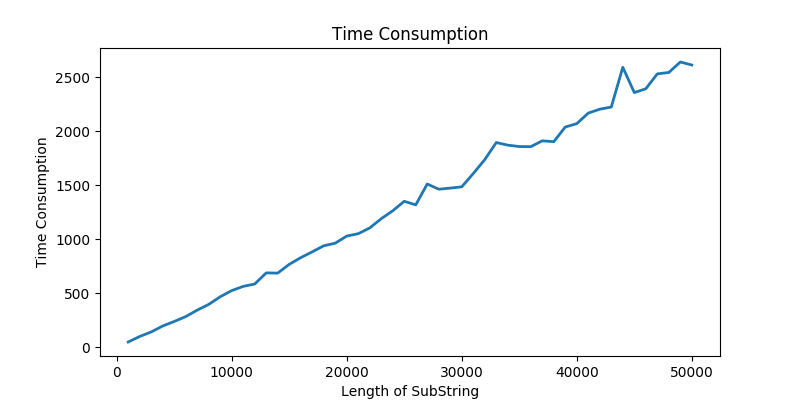


图 4.4

最后得到识别子串的时间为O(m)。m为识别的子串的长度，识别时间与母串长度无关。很好的体现了SAM的性质。

1. **应用**

**5.1 min\_loop\_str（最小循环串）**

**功能：**我们这样定义一个字符串的循环串：把一个字符串S分成左右两部分S1和S2，即S=S1S2，那么我们把S’=S2S1称之为S的一个循环串。对于长度为n的字符串，则有n个循环串。最小循环串定义为字典序最小的循环串。本应用的功能就是给出一个字符串的最小循环串,同时提供向右扩展功能。

**算法思想：**从起始状态开始一直走当前状态对应的所对应的字符最小的边。直到为空，得到了所得最小循环串的左边，然后只需要根据该循环串截断的位置就可以确定这个循环串。

**5.2 nsubstr**

**功能：**我们给一个字符串S定义函数f。f(x)是S的长度为x的所有字符串中重复的次数最多的字符串重复的次数。对于长度为n的字符串。我们求出f(1)f(2)…f(n)。同时提供向右扩展功能。

**算法思想：**我们构造S的SAM，那么对于一个节点s，它的长度范围是[minlen(s),maxlen(s)]，同时他的出现次数是|endpos(s)|。那么我们用

|endpos(s)|去更新F(maxlen(s))的值。

同时最后从大到小依次用F(i)去更新F(i-1)即可。

**5.3 query(查询次数)**

**功能：**给出一个字符串，查询其在模串中出现的次数。同时模串提供向右扩展功能。

**算法思想：**和原来的程序不同，在这个程序中我们加入了对于每个状态对应的endpos集合大小的记录。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endpos集合大小加了1。据此我们修改构造算法，然后在查询出现次数的时候，只要查找到对应状态，查看其保存的endpos集合大小的信息即可。

**5.4 length（求出两个字符串最长连续公共字符串长度）**

**功能：**输入字符串A和B，输出这两个字符串对应的最长连续公共字符串长度。

**算法思想：**对A建立SAM并且依次读入B的字符。若能够输入，则++len；否则意味着当前状态代表的一系列子串后面都没有所需字符，此时需要不断尝试缩短len来扩充子串集合，直到新加入的子串后面有了所需字符为止。注意自动机的每个状态代表的是A的子串，这个串可能很长，长到超出已匹配的长度，超出的这部分和B没有关系，只有重叠的那部分（较小的那个长度）才可以用来更新答案。

主要步骤如下：

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=maxlen(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

注意到我们求的是对于任意一个endpos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。

1. **结论**

本报告从逻辑结构、存储结构、时空性能、应用几个方面介绍了后缀自动机。可以看出其在解决字符串问题中强有力的作用，能在线性时间空间内构建且能在线性时间内解决一些字符串问题。

后缀自动机虽然实现简单，却有很深的内涵。虽然没有看不出太多明显的效用，但是只要略加修改和补充就可以实现和字符串处理有关的方方面面的许多种不同的应用。想要熟练运用后缀自动机离不开对其更加深入的理解。

构建后缀自动机代码量很少但理解起来相对困难。参考陈立杰讲稿的基础上又参考了俄文机翻的论文和一篇博客后最终总结出这篇报告，受益匪浅。下面贴出这几篇文章供参考。

* 2012年noi冬令营陈立杰讲稿(SAM后缀自动机)

<https://wenku.baidu.com/view/fa02d3fff111f18582d05a81.html>

* 后缀自动机：O(N)的构建及应用

<http://blog.csdn.net/wmdcstdio/article/details/44780707>

* Suffix Automaton Tutorial

<https://huntzhan.org/suffix-automaton-tutorial/>

附代码：

基本实现：

#include <iostream>

#include <cstring>

using namespace std;

struct state

{

state \*par, \*go[26];

int val;

state(int \_val) :

par(0), val(\_val)

{

memset(go, 0, sizeof go);

}

};

class SAM

{

public:

bool issuffix(string);

bool issubstr(string);

void extends(char);

void extendstr(string);

private:

string str;

state \*root = new state(0), \*last = root;

state\* next(state\*, char);

void extend(int);

};

bool SAM::issuffix(string st)

{

int b = str.size() - st.size();

int len = str.size();

if (b < 0)

return false;

for(int i=b;i<len;++i)

{

if(str[i]!=st[i-b])

{

return false;

}

}

return true;

}

bool SAM::issubstr(string st)

{

state\* p = root;

for (auto& s : st)

{

p = next(p, s);

if (p == NULL)

{

return false;

}

}

return true;

}

void SAM::extends(char c)

{

str.push\_back(c);

extend((int)(c - 97));

}

void SAM::extendstr(string st)

{

for(auto& s:st)

{

extends(s);

}

}

state\* SAM::next(state\* m, char c)

{

return m->go[(int)(c - 97)];

}

void SAM::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;

}

最小循环串：（只需要在实现基础上加上）

#include <iostream>

#include <cstring>

string SAM::min\_loop\_str()

{

state\* m = root;

int n = str.length();

string ret;

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

{

int j;

for(j=0;j<26;j++)

{

if (m->go[j])

{

break;

}

}

if(j==26)

{

ret.append(str.begin(), str.begin()+(n-i));

break;

}

else

{

m = m->go[j];

ret.push\_back(char(j + 97));

}

}

return ret;

}

Nsubstr:

#include <iostream>

#include <cstring>

using namespace std;

struct state

{

state \*par, \*go[26];

int val;

int right;

state(int \_val) :

par(0), val(\_val),right(0)

{

memset(go, 0, sizeof go);

}

};

class SAM

{

public:

bool issuffix(string);

bool issubstr(string);

void extends(char);

void extendstr(string);

void nsubstr(vector<int> &result)

{

result.clear();

int n = str.size();

int tot = all\_state.size();

vector<int>dp, cnt;

vector<state\*>b;

state\* node = root;

cnt.resize(2 \* n, 0);

b.resize(3 \* n, 0);

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

cnt[all\_state[i]->val]++;

for (int i = 1; i <= n; i++)

cnt[i] += cnt[i - 1];

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

b[--cnt[all\_state[i]->val]] = all\_state[i];

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

{

node = node->go[str[i] - 'a'];

++(node->right);

}

dp.resize(2 \* n, 0);

for (int i = tot - 1; i > 0; i--)

{

dp[b[i]->val] = (dp[b[i]->val] > b[i]->right) ? dp[b[i]->val] : b[i]->right;

if (b[i]->par)

b[i]->par->right += b[i]->right;

}

for (int i = n - 1; i >= 1; i--)

dp[i] = (dp[i] > dp[i + 1]) ? dp[i] : dp[i + 1];

for (int i = 1; i <= n; i++)

{

result.push\_back(dp[i]);

}

}

private:

string str;

state \*root = new state(0), \*last = root;

vector<state\*>all\_state;

state\* next(state\*, char);

void extend(int);

};

bool SAM::issuffix(string st)

{

int b = str.size() - st.size();

int len = str.size();

if (b < 0)

return false;

for (int i = b; i<len; ++i)

{

if (str[i] != st[i - b])

{

return false;

}

}

return true;

}

bool SAM::issubstr(string st)

{

state\* p = root;

for (auto& s : st)

{

p = next(p, s);

if (p == NULL)

{

return false;

}

}

return true;

}

void SAM::extends(char c)

{

str.push\_back(c);

extend((int)(c - 97));

}

void SAM::extendstr(string st)

{

for (auto& s : st)

{

extends(s);

}

}

state\* SAM::next(state\* m, char c)

{

return m->go[(int)(c - 97)];

}

void SAM::extend(int w)

{

if (all\_state.size() == 0)

all\_state.push\_back(root);

state \*p = last;

state \*np = new state(p->val + 1);

all\_state.push\_back(np);

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);

all\_state.push\_back(nq);

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;

}

Query:

#include <iostream>

#include <cstring>

using namespace std;

struct state

{

state \*par, \*go[26];

int val;

int size;

state(int \_val) :

par(0), val(\_val), size(0)

{

memset(go, 0, sizeof go);

}

};

class SAM

{

public:

bool issuffix(string);

bool issubstr(string);

void extends(char);

void extendstr(string);

void init()

{

root = new state(0), last = root;

str.clear();

}

int query(string substr)

{

int len = substr.length();

state\* x = root;

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

{

if (next(x, substr[i]))

{

x = next(x, substr[i]);

}

else

{

return 0;

}

}

return x->size;

}

private:

string str;

state \*root = new state(0), \*last = root;

state\* next(state\*, char);

void extend(int);

};

bool SAM::issuffix(string st)

{

int b = str.size() - st.size();

int len = str.size();

if (b < 0)

return false;

for (int i = b; i<len; ++i)

{

if (str[i] != st[i - b])

{

return false;

}

}

return true;

}

bool SAM::issubstr(string st)

{

state\* p = root;

for (auto& s : st)

{

p = next(p, s);

if (p == NULL)

{

return false;

}

}

return true;

}

void SAM::extends(char c)

{

str.push\_back(c);

extend((int)(c - 97));

}

void SAM::extendstr(string st)

{

for (auto& s : st)

{

extends(s);

}

}

state\* SAM::next(state\* m, char c)

{

return m->go[(int)(c - 97)];

}

void SAM::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;

nq->size = q->size;

while (p&&p->go[w] == q)

p->go[w] = nq, p = p->par;

}

}

last = np;

for (; np != root; np = np->par)

++np->size;

}

Length:

#include <iostream>

#include <cstring>

#include <cstdio>

#include <algorithm>

using std::max;

using std::min;

using namespace std;

struct state

{

state \*par, \*go[26];

int val;

state(int \_val) :

par(0), val(\_val)

{

memset(go, 0, sizeof go);

}

};

class SAM

{

public:

bool issuffix(string);

bool issubstr(string);

void extends(char);

void extendstr(string);

int leng(string ano)

{

state \*now = root;

int ans = 0, pref = 0;

for (auto p:ano)

{

int x = p - 'a';

for (; now && !now->go[x]; now = now->par)

;

if (!now)

{

now = root;

pref = 0;

continue;

}

pref = min(pref, now->val) + 1;

ans = max(ans, pref);

now = now->go[x];

}

return ans;

}

private:

string str;

state \*root = new state(0), \*last = root;

state\* next(state\*, char);

void extend(int);

};

bool SAM::issuffix(string st)

{

int b = str.size() - st.size();

int len = str.size();

if (b < 0)

return false;

for (int i = b; i<len; ++i)

{

if (str[i] != st[i - b])

{

return false;

}

}

return true;

}

bool SAM::issubstr(string st)

{

state\* p = root;

for (auto& s : st)

{

p = next(p, s);

if (p == NULL)

{

return false;

}

}

return true;

}

void SAM::extends(char c)

{

str.push\_back(c);

extend((int)(c - 97));

}

void SAM::extendstr(string st)

{

for (auto& s : st)

{

extends(s);

}

}

state\* SAM::next(state\* m, char c)

{

return m->go[(int)(c - 97)];

}

void SAM::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;

}