后缀树

在pongba的讨论组上看到一道Amazon的面试题：找出给定字符串里的最长回文。例子：输入XMADAMYX。则输出MADAM。这道题的流行解法是用后缀树（Suffix Tree)。这坨数据结构最酷的地方是用它能高效解决一大票复杂的字符串编程问题：

在文本T里查询T是否包含子串P（复杂度同流行的KMP相当）。

文本T里找出最长重复子串。比如abcdabcefda里abc同da都重复出现，而最长重复子串是abc。

找出字符串S1同S2的最长公共子串。注意不是常用作动态规划例子的LCS哈。比如字符串acdfg同akdfc的最长公共子串为df，而他们的LCS是adf。

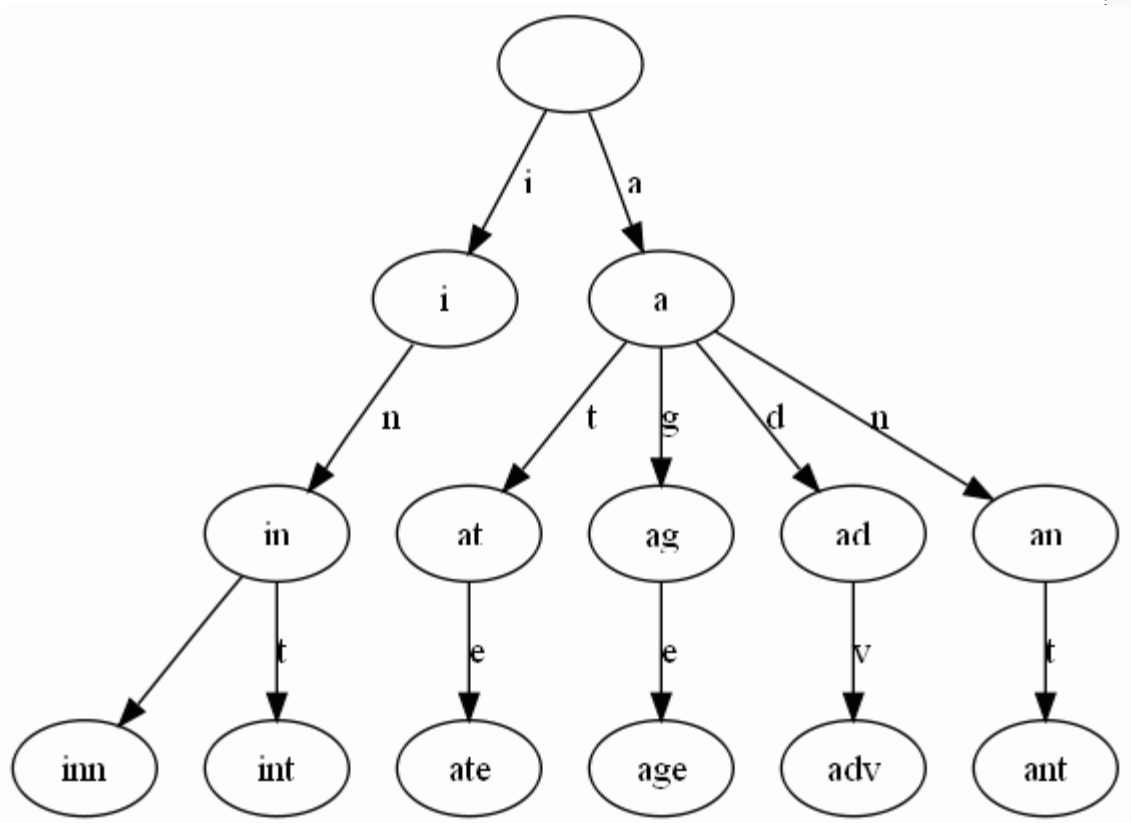
Ziv-Lampel无损压缩算法。

还有就是这道面试题问的最长回文了。

另外后缀树在生物信息学里应该应用广泛。碱基匹配和选取的计算本质上就是操作超长的{C, T, A, G, U}*字符串嘛。

虽说后缀树的概念独立于Trie的概念，但我觉得从Trie推出后缀树自然简洁，所以先简单解释一下Trie。“Trie”这个单词来自于"retrieve"，可见它的用途主要是字符串查询。不过词汇变迁多半比较诡异，Trie不发tree的音，而发try的音。

Trie是坨简单但实用的数据结构，通常用于实现字典查询。我们做即时响应用户输入的AJAX搜索框时，就是Trie开始。谁说学点数据结构没用来 着？本质上，Trie是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树 不同的地方是，相同的字符串前缀共享同一条分支。还是例子最清楚。给出一组单词，inn, int, at, age, adv, ant, 我们可以得到下面的Trie：



可以看出:

每条边对应一个字母。

每个节点对应一项前缀。叶节点对应最长前缀，即单词本身。

单词inn与单词int有共同的前缀“in”, 因此他们共享左边的一条分支，root->i->in。同理，ate, age, adv, 和ant共享前缀"a"，所以他们共享从根节点到节点"a"的边。

查询非常简单。比如要查找int，顺着路径i -> in -> int就找到了。

搭建Trie的基本算法也很简单，无非是逐一把每则单词的每个字母插入Trie。插入前先看前缀是否存在。如果存在，就共享，否则创建对应的节点和边。比如要插入单词add，就有下面几步:

考察前缀"a"，发现边a已经存在。于是顺着边a走到节点a。

考察剩下的字符串"dd"的前缀"d"，发现从节点a出发，已经有边d存在。于是顺着边d走到节点ad

考察最后一个字符"d"，这下从节点ad出发没有边d了，于是创建节点ad的子节

点add，并把边ad->add标记为d。

继续插播广告。Graph作图软件Graphviz还不错，用的DSL相当简单。上面的图就是用它做的。三步就够了：

实现Trie数据结构。这步不用花哨。10行代码，一坨hash足矣。

把上面的结构翻译成Graphviz的DSL。简单的深度优先足矣。

调用Graphviz的命令。图就生成乐。

多花20分钟，避免了手工作图排版的自虐行为。而且可以自由试验各式例子而不用担心反复画图的琐碎，何乐而不为嘎？

有了Trie，后缀树就容易理解了。先说说后缀的定义。给定一长度为n的字符串S=S1S2..Si..Sn，和整数i，1 <= i <= n，子串SiSi+1...Sn都是字符串S的后缀。以字符串S=XMADAMYX为例，它的长度为8，所以S[1..8], S[2..8], ... , S[8..8]都算S的后缀，我们一般还把空字串也算成后缀。这样，我们一共有如下后缀。对于后缀S[i..n]，我们说这项后缀起始于i。

S[1..8], XMADAMYX， 也就是字符串本身，起始位置为1

S[2..8], MADAMYX，起始位置为2

S[3..8], ADAMYX，起始位置为3

S[4..8], DAMYX，起始位置为4

S[5..8], AMYX，起始位置为5

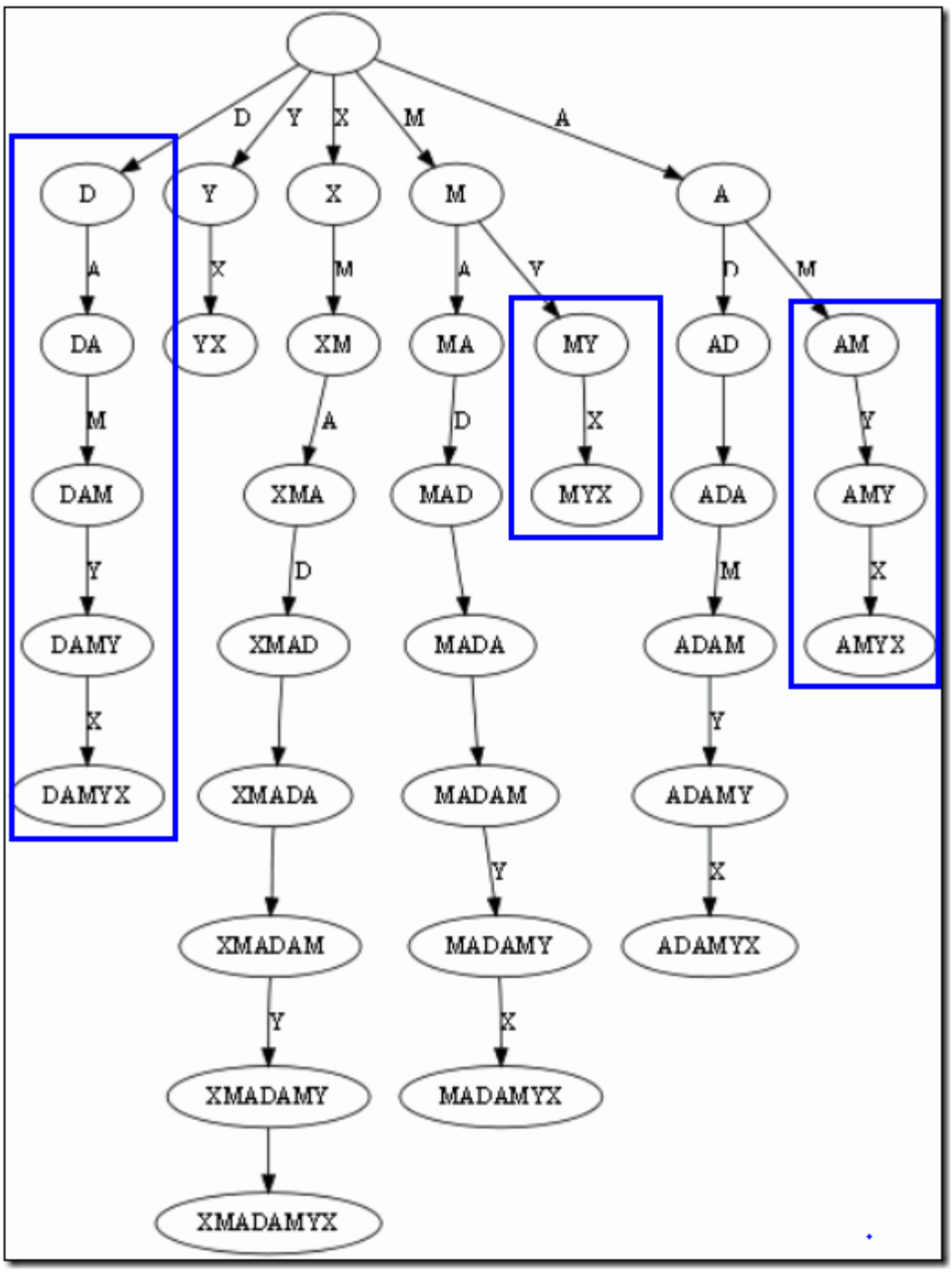
S[6..8], MYX，起始位置为6

S[7..8], YX，起始位置为7

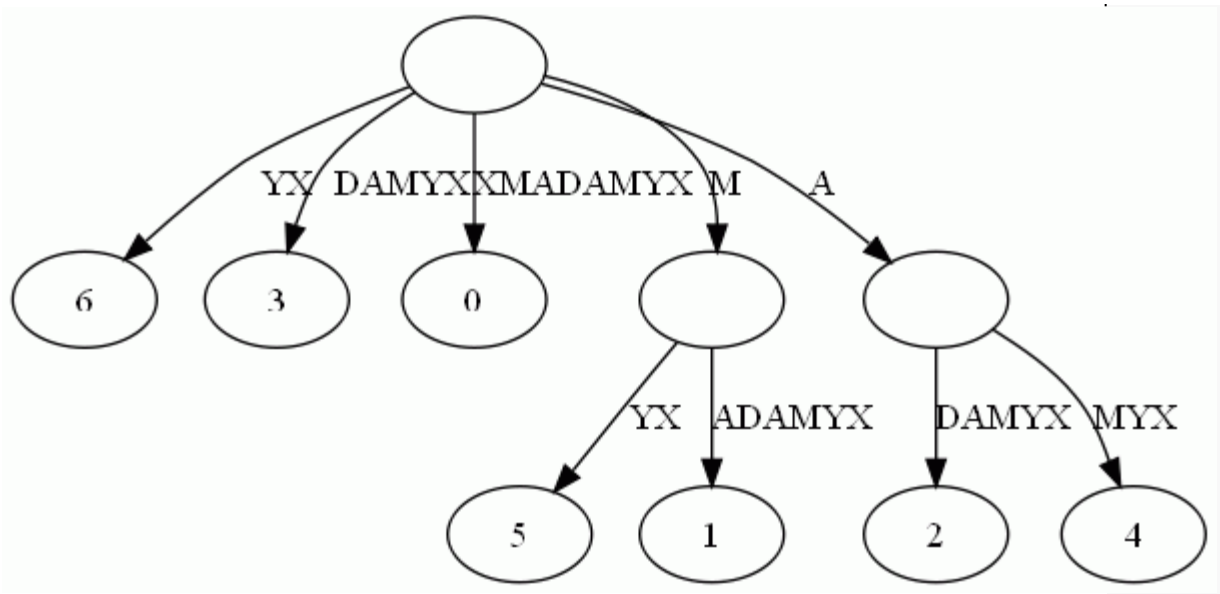
S[8..8], X，起始位置为8

空字串。记为$。

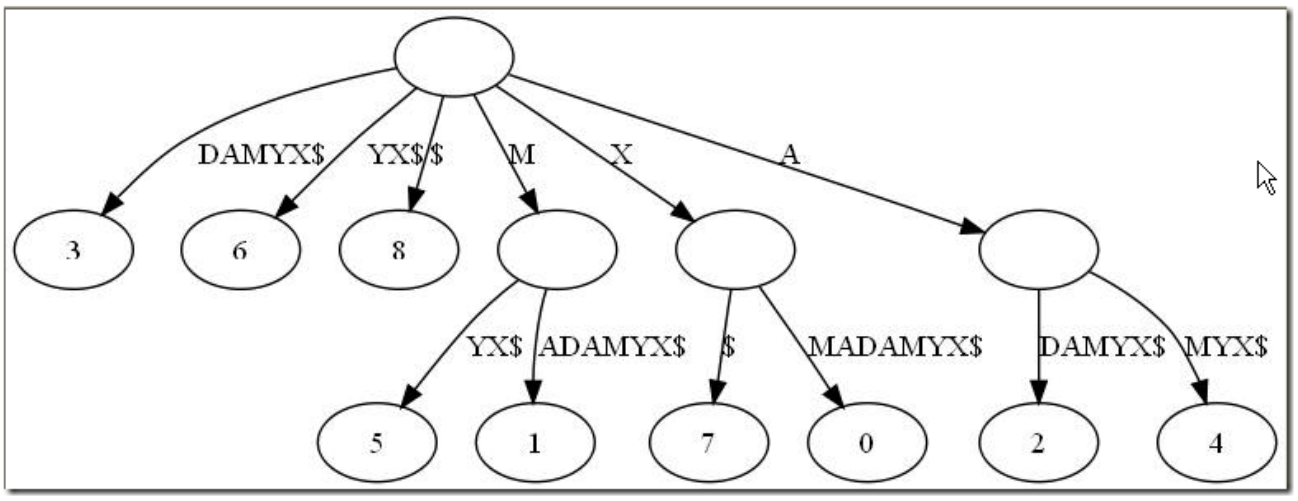
而后缀树，就是包含一则字符串所有后缀的压缩Trie。把上面的后缀加入Trie后，我们得到下面的结构：



仔细观察上图，我们可以看到不少值得压缩的地方。比如蓝框标注的分支都是独苗，没有必要用单独的节点同边表示。如果我们允许任意一条边里包含多个字 母，就可以把这种没有分叉的路径压缩到一条边。另外每条边已经包含了足够的后缀信息，我们就不用再给节点标注字符串信息了。我们只需要在叶节点上标注上每 项后缀的起始位置。于是我们得到下图：



这样的结构丢失了某些后缀。比如后缀X在上图中消失了，因为它正好是字符串XMADAMYX的前缀。为了避免这种情况，我们也规定每项后缀不能是其 它后缀的前缀。要解决这个问题其实挺简单，在待处理的子串后加一坨空字串就行了。例如我们处理XMADAMYX前，先把XMADAMYX变为 XMADAMYX$，于是就得到suffix tree乐。

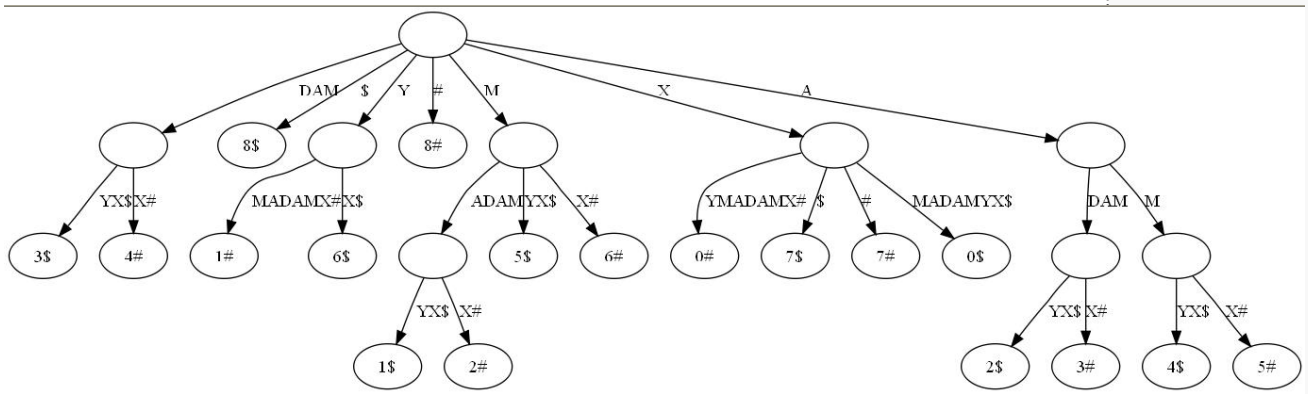


那后缀树同最长回文有什么关系呢？我们得先知道两坨坨简单概念：

最低共有祖先，LCA（Lowest Common Ancestor)，也就是任意两节点（多个也行）最长的共有前缀。比如下图中，节点7同节点10的共同祖先是节点1与借点，但最低共同祖先是5。 查找LCA的算法是O(1)的复杂度，这年头少见。代价是需要对后缀树做复杂度为O(n)的预处理。

广 义后缀树(Generalized Suffix Tree)。传统的后缀树处理一坨单词的所有后缀。广义后缀树存储任意多个单词的所有后缀。例如下图是单

词XMADAMYX与XYMADAMX的广义后缀 树。注意我们需要区分不同单词的后缀，所以叶节点用不同的特殊符号与后缀位置配对。



有了上面的概念，查找最长回文相对简单了。思维的突破点在于考察回文的半径，而不是回文本身。所谓半径，就是回文对折后的字串。比如回文MADAM 的半径为MAD，半径长度为3，半径的中心是字母D。显然，最长回文必有最长半径，且两条半径相等。还是以MADAM为例，以D为中心往左，我们得到半径 DAM；以D为中心向右，我们得到半径DAM。二者肯定相等。因为MADAM已经是单词XMADAMYX里的最长回文，我们可以肯定从D往左数的字串 DAMX与从D往右数的子串DAMYX共享最长前缀DAM。而这，正是解决回文问题的关键。现在我们有后缀树，怎么把从D向左数的字串DAMX变成后缀 呢？到这个地步，答案应该明显：把单词XMADAMYX翻转就行了。于是我们把寻找回文的问题转换成了寻找两坨后缀的LCA的问题。当然，我们还需要知道 到底查询那些后缀间的LCA。这也简单，给定字符串S，如果最长回文的中心在i，那从位置i向右数的后缀刚好是S(i)，而向左数的字符串刚好是翻转S后 得到的字符串S‘的后缀S'(n-i+1)。这里的n是字符串S的长度。有了这套直观解释，算法自然呼之欲出：

预处理后缀树，使得查询LCA的复杂度为O(1)。这步的开销是O(N)，N是单词S的长度

对单词的每一位置i(也就是从0到N-1)，获取LCA(S(i), S(N-i+1)) 以及LCA(S(i+1), S(n-i+1))。查找两次的原因是我们需要考虑奇数回文和偶数回文的情况。这步要考察每坨i，所以复杂度是O(N)

找到最大的LCA，我们也就得到了回文的中心i以及回文的半径长度，自然也就得到了最长回文。总的复杂度O(n)。

用上图做例子，i为3时，LCA(3$, 4#)为DAM，正好是最长半径。当然，这只是直观的叙述。

这篇帖子只大致描述了后缀树的基本思路。要想写出实用代码，至少还得知道下面的知识：

创建后缀树的O(n)算法。至于是Peter Weiner的73年年度最佳算法，还是Edward McCreight1976的改进算法，还是1995年E. Ukkonen大幅简化的算法，还是Juha Kärkkäinen 和 Peter Sanders2003年进一步简化的线性算法，各位老大随喜。

实现后缀树用的数据结构。比如常用的子结点加兄弟节点列表，Directed

优化后缀树空间的办法。比如不存储子串，而存储读取子串必需的位置。以及Directed Acyclic Word Graph，常缩写为黑哥哥们挂在嘴边的DAWG。

2,后缀树的用途，总结起来大概有如下几种

(1). 查找字符串o是否在字符串S中。

方案：用S构造后缀树，按在trie中搜索字串的方法搜索o即可。

原理：若o在S中，则o必然是S的某个后缀的前缀。

例如S: leconte，查找o: con是否在S中,则o(con)必然是S(leconte)的后缀之一conte的前缀.有了这个前提，采用trie搜索的方法就不难理解了。

(2). 指定字符串T在字符串S中的重复次数。

方案：用S+'$'构造后缀树，搜索T节点下的叶节点数目即为重复次数

原理：如果T在S中重复了两次，则S应有两个后缀以T为前缀，重复次数就自然统计出来了。

(3). 字符串S中的最长重复子串

方案：原理同2，具体做法就是找到最深的非叶节点。

这个深是指从root所经历过的字符个数，最深非叶节点所经历的字符串起来就是最长重复子串。

为什么要非叶节点呢?因为既然是要重复，当然叶节点个数要>=2。

(4). 两个字符串S1，S2的最长公共部分

方案：将S1#S2$作为字符串压入后缀树，找到最深的非叶节点，且该节点的叶节点既有#也有$(无#)。

一个C++的实现：

```
//
// Suffix tree creation
//
// Mark Nelson, updated December, 2006
//
```

```
// This code has been tested with Borland C++ and
// Microsoft Visual C++.
//
// This program asks you for a line of input, then
// creates the suffix tree corresponding to the given
// text. Additional code is provided to validate the
// resulting tree after creation.
//
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstring>
#include <cassert>
#include <string>

using std::cout;
using std::cin;
using std::cerr;
using std::setw;
using std::flush;
using std::endl;

//
// When a new tree is added to the table, we step
// through all the currently defined suffixes from
// the active point to the end point.  This structure
// defines a Suffix by its final character.
// In the canonical representation, we define that last
// character by starting at a node in the tree, and
// following a string of characters, represented by
// first_char_index and last_char_index.  The two indices
// point into the input string.  Note that if a suffix
// ends at a node, there are no additional characters
// needed to characterize its last character position.
```

```
// When this is the case, we say the node is Explicit,
// and set first_char_index > last_char_index to flag
// that.
//

class Suffix {
   public :
      int origin_node;
      int first_char_index;
      int last_char_index;
      Suffix( int node, int start, int stop )
         : origin_node( node ),
           first_char_index( start ),
           last_char_index( stop ){};
      int Explicit(){ return first_char_index > last_char_index; }
      int Implicit(){ return last_char_index >= first_char_index; }
      void Canonize();
};

//
// The suffix tree is made up of edges connecting nodes.
// Each edge represents a string of characters starting
// at first_char_index and ending at last_char_index.
// Edges can be inserted and removed from a hash table,
// based on the Hash() function defined here.  The hash
// table indicates an unused slot by setting the
// start_node value to -1.
//

class Edge {
   public :
      int first_char_index;
      int last_char_index;
      int end_node;
      int start_node;
```

```
    void Insert();
    void Remove();
    Edge();
    Edge( int init_first_char_index,
        int init_last_char_index,
        int parent_node );
    int SplitEdge( Suffix &s );
    static Edge Find( int node, int c );
    static int Hash( int node, int c );
};

//
//  The only information contained in a node is the
//  suffix link. Each suffix in the tree that ends
//  at a particular node can find the next smaller suffix
//  by following the suffix_node link to a new node.  Nodes
//  are stored in a simple array.
//
class Node {
  public :
    int suffix_node;
    int father;
    int leaf_index;
    Node() { suffix_node = -1;
        father=-1;
        leaf_index=-1;}
    static int Count;
    static int Leaf;
};

//
// The maximum input string length this program
// will handle is defined here.  A suffix tree
// can have as many as 2N edges/nodes.  The edges
```

```
// are stored in a hash table, whose size is also
// defined here.
//
const int MAX_LENGTH = 1000;
const int HASH_TABLE_SIZE = 2179;  //A prime roughly 10% larger

//
// This is the hash table where all the currently
// defined edges are stored.  You can dump out
// all the currently defined edges by iterating
// through the table and finding edges whose start_node
// is not -1.
//

Edge Edges[ HASH_TABLE_SIZE ];

//
// The array of defined nodes.  The count is 1 at the
// start because the initial tree has the root node
// defined, with no children.
//

int Node::Count = 1;
int Node::Leaf = 1;
Node Nodes[ MAX_LENGTH * 2 ];

//
// The input buffer and character count.  Please note that N
// is the length of the input string -1, which means it
// denotes the maximum index in the input buffer.
//

char T[ MAX_LENGTH ];
int N;
```

```
//
// Necessary forward references
//
void validate();
int walk_tree( int start_node, int last_char_so_far );



//
// The default ctor for Edge just sets start_node
// to the invalid value.  This is done to guarantee
// that the hash table is initially filled with unused
// edges.
//

Edge::Edge()
{
   start_node = -1;
}

//
// I create new edges in the program while walking up
// the set of suffixes from the active point to the
// endpoint.  Each time I create a new edge, I also
// add a new node for its end point.  The node entry
// is already present in the Nodes[] array, and its
// suffix node is set to -1 by the default Node() ctor,
// so I don't have to do anything with it at this point.
//

Edge::Edge( int init_first, int init_last, int parent_node )
{
   first_char_index = init_first;
   last_char_index = init_last;
   start_node = parent_node;
```

```
    end_node = Node::Count++;
    Nodes[end_node].father=start_node;
}

//
// Edges are inserted into the hash table using this hashing
// function.
//

int Edge::Hash( int node, int c )
{
    return ( ( node << 8 ) + c ) % HASH_TABLE_SIZE;
}

//
// A given edge gets a copy of itself inserted into the table
// with this function.  It uses a linear probe technique, which
// means in the case of a collision, we just step forward through
// the table until we find the first unused slot.
//

void Edge::Insert()
{
    int i = Hash( start_node, T[ first_char_index ] );
    while ( Edges[ i ].start_node != -1 )
        i = ++i % HASH_TABLE_SIZE;
    Edges[ i ] = *this;
}

//
// Removing an edge from the hash table is a little more tricky.
// You have to worry about creating a gap in the table that will
// make it impossible to find other entries that have been inserted
// using a probe.  Working around this means that after setting
// an edge to be unused, we have to walk ahead in the table,
```

```
// filling in gaps until all the elements can be found.
//
// Knuth, Sorting and Searching, Algorithm R, p. 527
//

void Edge::Remove()
{
   int i = Hash( start_node, T[ first_char_index ] );
   while ( Edges[ i ].start_node != start_node ||
         Edges[ i ].first_char_index != first_char_index )
      i = ++i % HASH_TABLE_SIZE;
   for ( ; ; ) {
      Edges[ i ].start_node = -1;
      int j = i;
      for ( ; ; ) {
         i = ++i % HASH_TABLE_SIZE;
         if ( Edges[ i ].start_node == -1 )
            return;
         int r = Hash( Edges[ i ].start_node, T[ Edges[ i ].first_char_index ] );
         if ( i >= r && r > j )
            continue;
         if ( r > j && j > i )
            continue;
         if ( j > i && i >= r )
            continue;
         break;
      }
      Edges[ j ] = Edges[ i ];
   }
}

//
// The whole reason for storing edges in a hash table is that it
// makes this function fairly efficient.  When I want to find a
```

```
// particular edge leading out of a particular node, I call this
// function.  It locates the edge in the hash table, and returns
// a copy of it.  If the edge isn't found, the edge that is returned
// to the caller will have start_node set to -1, which is the value
// used in the hash table to flag an unused entry.
//

Edge Edge::Find( int node, int c )
{
   int i = Hash( node, c );
   for ( ; ; ) {
      if ( Edges[ i ].start_node == node )
         if ( c == T[ Edges[ i ].first_char_index ] )
            return Edges[ i ];
      if ( Edges[ i ].start_node == -1 )
         return Edges[ i ];
      i = ++i % HASH_TABLE_SIZE;
   }
}

//
// When a suffix ends on an implicit node, adding a new character
// means I have to split an existing edge.  This function is called
// to split an edge at the point defined by the Suffix argument.
// The existing edge loses its parent, as well as some of its leading
// characters.  The newly created edge descends from the original
// parent, and now has the existing edge as a child.
//
// Since the existing edge is getting a new parent and starting
// character, its hash table entry will no longer be valid.  That's
// why it gets removed at the start of the function.  After the parent
// and start char have been recalculated, it is re-inserted.
//
// The number of characters stolen from the original node and given
// to the new node is equal to the number of characters in the suffix
```

```
// argument, which is last - first + 1;
//

int Edge::SplitEdge( Suffix &s )
{
   Remove();
   Edge *new_edge =
    new Edge( first_char_index,
              first_char_index + s.last_char_index - s.first_char_index,
              s.origin_node );
   new_edge->Insert();
   Nodes[ new_edge->end_node ].suffix_node = s.origin_node;
   first_char_index += s.last_char_index - s.first_char_index + 1;
   start_node = new_edge->end_node;
   Insert();
   Nodes[end_node].father=start_node;
   return new_edge->end_node;
}

//
// This routine prints out the contents of the suffix tree
// at the end of the program by walking through the
// hash table and printing out all used edges.  It
// would be really great if I had some code that will
// print out the tree in a graphical fashion, but I don't!
//

void dump_edges( int current_n )
{
   cout << " Start  End  Suf  First Last  String\n";
   for ( int j = 0 ; j < HASH_TABLE_SIZE ; j++ ) {
      Edge *s = Edges + j;
      if ( s->start_node == -1 )
         continue;
```

```
      cout << setw( 5 ) << s->start_node << " "
           << setw( 5 ) << s->end_node << " "
           << setw( 3 ) << Nodes[ s->end_node ].suffix_node << " "
           << setw( 5 ) << s->first_char_index << " "
           << setw( 6 ) << s->last_char_index << "  ";
      int top;
      if ( current_n > s->last_char_index )
         top = s->last_char_index;
      else
         top = current_n;
      for ( int l = s->first_char_index ;
            l <= top;
            l++ )
         cout << T[ l ];
      cout << "\n";
   }
}

//
// A suffix in the tree is denoted by a Suffix structure
// that denotes its last character.  The canonical
// representation of a suffix for this algorithm requires
// that the origin_node by the closest node to the end
// of the tree.  To force this to be true, we have to
// slide down every edge in our current path until we
// reach the final node.

void Suffix::Canonize()
{
   if ( !Explicit() ) {
      Edge edge = Edge::Find( origin_node, T[ first_char_index ] );
      int edge_span = edge.last_char_index - edge.first_char_index;
      while ( edge_span <= ( last_char_index - first_char_index ) ) {
         first_char_index = first_char_index + edge_span + 1;
         origin_node = edge.end_node;
```

```
        if ( first_char_index <= last_char_index ) {
          edge = Edge::Find( edge.end_node, T[ first_char_index ] );
          edge_span = edge.last_char_index - edge.first_char_index;
        };
      }
    }
}

//
// This routine constitutes the heart of the algorithm.
// It is called repetitively, once for each of the prefixes
// of the input string.  The prefix in question is denoted
// by the index of its last character.
//
// At each prefix, we start at the active point, and add
// a new edge denoting the new last character, until we
// reach a point where the new edge is not needed due to
// the presence of an existing edge starting with the new
// last character.  This point is the end point.
//
// Luckily for use, the end point just happens to be the
// active point for the next pass through the tree.  All
// we have to do is update it's last_char_index to indicate
// that it has grown by a single character, and then this
// routine can do all its work one more time.
//

void AddPrefix( Suffix &active, int last_char_index )
{
  int parent_node;
  int last_parent_node = -1;

  for ( ; ; ) {
    Edge edge;
```

```
        parent_node = active.origin_node;
//
// Step 1 is to try and find a matching edge for the given node.
// If a matching edge exists, we are done adding edges, so we break
// out of this big loop.
//
        if ( active.Explicit() ) {
            edge = Edge::Find( active.origin_node, T[ last_char_index ] );
            if ( edge.start_node != -1 )
                break;
        } else { //implicit node, a little more complicated
            edge = Edge::Find( active.origin_node, T[ active.first_char_index ] );
            int span = active.last_char_index - active.first_char_index;
            if ( T[ edge.first_char_index + span + 1 ] == T[ last_char_index ] )
                break;
            parent_node = edge.SplitEdge( active );
        }
//
// We didn't find a matching edge, so we create a new one, add
// it to the tree at the parent node position, and insert it
// into the hash table.  When we create a new node, it also
// means we need to create a suffix link to the new node from
// the last node we visited.
//
        Edge *new_edge = new Edge( last_char_index, N, parent_node );
        new_edge->Insert();
        Nodes[new_edge->end_node].leaf_index=Node::Leaf++;
        if ( last_parent_node > 0 )
            Nodes[ last_parent_node ].suffix_node = parent_node;
        last_parent_node = parent_node;
//
// This final step is where we move to the next smaller suffix
//
        if ( active.origin_node == 0 )
            active.first_char_index++;
```

```
      else
         active.origin_node = Nodes[ active.origin_node ].suffix_node;
      active.Canonize();
   }
   if ( last_parent_node > 0 )
      Nodes[ last_parent_node ].suffix_node = parent_node;
   active.last_char_index++;  //Now the endpoint is the next active point
   active.Canonize();
};

int main()
{
   cout << "Normally, suffix trees require that the last\n"
        << "character in the input string be unique.  If\n"
        << "you don't do this, your tree will contain\n"
        << "suffixes that don't end in leaf nodes.  This is\n"
        << "often a useful requirement. You can build a tree\n"
        << "in this program without meeting this requirement,\n"
        << "but the validation code will flag it as being an\n"
        << "invalid tree\n\n";
   cout << "Enter string: " << flush;
   cin.getline( T, MAX_LENGTH - 1 );
   N = strlen( T ) - 1;
//
// The active point is the first non-leaf suffix in the
// tree.  We start by setting this to be the empty string
// at node 0.  The AddPrefix() function will update this
// value after every new prefix is added.
//
   Suffix active( 0, 0, -1 );  // The initial active prefix
   for ( int i = 0 ; i <= N ; i++ )
      AddPrefix( active, i );
```

```
   for(i=0;i<Node::Count;i++)
      cout<<i<<"   "<<Nodes[i].father<<"   "<<Nodes[i].leaf_index<<endl;
//
// Once all N prefixes have been added, the resulting table
// of edges is printed out, and a validation step is
// optionally performed.
//
 //   dump_edges( N );
 //   cout << "Would you like to validate the tree?"
 //        << flush;
 //   std::string s;
//    std::getline( cin, s );
 //   if ( s.size() > 0 && s[ 0 ] == 'Y' || s[ 0 ] == 'y' )
 //      validate();
   return 1;
};

//
// The validation code consists of two routines.  All it does
// is traverse the entire tree.  walk_tree() calls itself
// recursively, building suffix strings up as it goes.  When
// walk_tree() reaches a leaf node, it checks to see if the
// suffix derived from the tree matches the suffix starting
// at the same point in the input text.  If so, it tags that
// suffix as correct in the GoodSuffixes[] array.  When the tree
// has been traversed, every entry in the GoodSuffixes array should
// have a value of 1.
//
// In addition, the BranchCount[] array is updated while the tree is
// walked as well.  Every count in the array has the
// number of child edges emanating from that node.  If the node
// is a leaf node, the value is set to -1.  When the routine
// finishes, every node should be a branch or a leaf.  The number
// of leaf nodes should match the number of suffixes (the length)
// of the input string.  The total number of branches from all
```

```
// nodes should match the node count.
//

char CurrentString[ MAX_LENGTH ];
char GoodSuffixes[ MAX_LENGTH ];
char BranchCount[ MAX_LENGTH * 2 ] = { 0 };

void validate()
{
   for ( int i = 0 ; i < N ; i++ )
      GoodSuffixes[ i ] = 0;
   walk_tree( 0, 0 );
   int error = 0;
   for ( i = 0 ; i < N ; i++ )
      if ( GoodSuffixes[ i ] != 1 ) {
         cout << "Suffix " << i << " count wrong!\n";
         error++;
      }
   if ( error == 0 )
      cout << "All Suffixes present!\n";
   int leaf_count = 0;
   int branch_count = 0;
   for (i = 0 ; i < Node::Count ; i++ ) {
      if ( BranchCount[ i ] == 0 )
         cout << "Logic error on node "
              << i
              << ", not a leaf or internal node!\n";
      else if ( BranchCount[ i ] == -1 )
         leaf_count++;
      else
         branch_count += BranchCount[ i ];
   }
   cout << "Leaf count : "
       << leaf_count
```

```
        << ( leaf_count == ( N + 1 ) ? " OK" : " Error!" )
        << "\n";
    cout << "Branch count : "
        << branch_count
        << ( branch_count == (Node::Count - 1) ? " OK" : " Error!" )
        << endl;
}

int walk_tree( int start_node, int last_char_so_far )
{
    int edges = 0;
    for ( int i = 0 ; i < 256 ; i++ ) {
        Edge edge = Edge::Find( start_node, i );
        if ( edge.start_node != -1 ) {
            if ( BranchCount[ edge.start_node ] < 0 )
                cerr << "Logic error on node "
                    << edge.start_node
                    << '\n';
            BranchCount[ edge.start_node ]++;
            edges++;
            int l = last_char_so_far;
            for ( int j = edge.first_char_index ; j <= edge.last_char_index ; j++ )
                CurrentString[ l++ ] = T[ j ];
            CurrentString[ l ] = '\0';
            if ( walk_tree( edge.end_node, l ) ) {
                if ( BranchCount[ edge.end_node ] > 0 )
                    cerr << "Logic error on node "
                        << edge.end_node
                        << "\n";
                BranchCount[ edge.end_node ]--;
            }
        }
    }
//
// If this node didn't have any child edges, it means we
```

```
// are at a leaf node, and can check on this suffix.  We
// check to see if it matches the input string, then tick
// off it's entry in the GoodSuffixes list.
//
   if ( edges == 0 ) {
      cout << "Suffix : ";
      for ( int m = 0 ; m < last_char_so_far ; m++ )
         cout << CurrentString[ m ];
      cout << "\n";
      GoodSuffixes[ strlen( CurrentString ) - 1 ]++;
      cout << "comparing: " << ( T + N - strlen( CurrentString ) + 1 )
          << " to " << CurrentString << endl;
      if ( strcmp(T + N - strlen(CurrentString) + 1, CurrentString ) != 0 )
         cout << "Comparison failure!\n";
      return 1;
   } else
      return 0;
}
```

posted on 2010-10-29 20:55 superKiki 阅读(2301) 评论(0) 编辑 收藏 引用 所属分类: 字符串问题

Remember Me?

登录 使用高级评论 新用户注册 返回页首 恢复上次提交

[使用Ctrl+Enter键可以直接提交]

相关文章:

AC自动机(续)

常用字符串算法综述
简易HTML解析器(HDU1088)
常见字符串算法小结
最长公共子串问题的后缀数组
解法
AC自动机
后缀数组两种算法的分析比较
RMQ问题与LCA问题
最长公共子序列（LCS）

随笔档案

文章分类

相册

我的好友

一个很好用的数字图像处理库
python中国
python中文参考站
shell基础十二篇
whitefirer[PG]
编程爱好者论坛
发芽题库
简明python教程
科学松鼠会
大话很多高深理论
普特英语

一些常去的网站

C/C++编程论坛
CSDN网址导航
CSNA网络分析社区
nocow
OpenCV中文站
Roba大牛的主页
topcoder
ubuntu中文论坛
unix-center
ural
windows API 一日一练
飞燕之家论坛
非常批处理论坛
看雪安全论坛
驱动开发网
数据挖掘研究院
信息爱好者之家
云架构之道

搜索

搜索

最新评论 XML

1. re: A star算法概要[未登录]
@liangzehua
同意
--grace
2. re: 中国剩余定理
评论内容较长,点击标题查看
--zhangyi
3. re: 最长不降子序列[未登录]
评论内容较长,点击标题查看
--zzw
4. re: A star算法概要
如果节点是在closed表里面就不用考虑了，直接跳过。。你上面貌似写错了吧。

--liangzehua

5. re: 计算空间中最近点对[未登录]

吧

--.

阅读排行榜

1. 后缀树(2301)
2. 最长不降子序列(1375)
3. BM算法的改进的算法 Sunday Algorithm(958)
4. 计算空间中最近点对(792)
5. 阶乘计算的高效算法(762)

评论排行榜

1. 数论问题总结(3)
2. 计算空间中最近点对(3)
3. AC自动机(2)
4. A star算法概要(2)
5. 各种字符串hash效率比较(1)