acm.mipt.ru

олимпиады по программированию на Физтехе

Алгоритм Укконена построения суффиксного дерева (бэта :), C++)

Код этот очень приблизительный. В том смысле, что работает-то он правильно (судя по El Judge), но вот вид у него не олимпиадный. Надо бы как-то ужимать, упрощать, от мусора избавляться.

```
/**
 * ukk.cpp
 * My implementation of the Ukkonen algo. Based on
 * Ukkonen's paper on this topic.
 * Daniil Shved, MIPT, 2009.
 */
#include <vector>
#include <algorithm>
#include <string>
#include <limits>
#include <stdio.h>
using namespace std;

const int inf = numeric_limits<int>::max();
typedef unsigned char UChar;

// Represents a link in our suffix tree
struct Link {
   int start, end;
   int to;

   // default: invalid link
   Link() {
      to = -1;
   }

   // a link with given parameters
   Link(int _start, int _end, int _to) {
      start = _start;
      end = _end;
      to = _to;
   }
};

// Represents an explicit vertex in our suffix tree
struct Vertex {
   vector<Link> links;      // state links
   int suffix;            // suffix link

   Vertex() {
      links.assign(256, Link());
      suffix = -1;
   }
};

// The whole suffix tree
vector<Vertex> tree;
int root, dummy;

// The sample it is built for
string sample;
```

Поиск
Поиск
Раздел «Алгоритмы»
Главная
Форум
Ссылки
El Judge
Инструменты:
Поиск
Изменения
Index
Статистика
Разделы
Информация
Алгоритмы
Язык Си
Язык Ruby
Язык Ассемблера
El Judge
Парадигмы
Образование
Сети
Objective C
Logon>>

```
// Gets the character with the given index. Understands negative indices
UChar t(int i) {
   return (i<0) ? (-i-1) : sample[i];
}

// Creates a new vertex in the suffix tree
int newVertex()
{
   int i = tree.size();
   tree.push_back(Vertex());
   return i;
}

// Creates a link in the suffix tree
// to, from - two vertices
// [start, end) - the word on the edge
void link(int from, int start, int end, int to)
{
   tree[from].links[t(start)] = Link(start, end, to);
}

// The f function (goes along the suffix link)
int &f(int v)
{
   return tree[v].suffix;
}

// Prints the tree to stdout, for debugging purposes
void print(int v, int start = 0, int end = 0, string prefix = "") {
   // What's written on the edge that leads here
   printf("%s", prefix.c_str());
   for(int i=start; i<end && i<sample.length(); i++)
      printf("%c", t(i));
   if(end == inf) printf("@");

   // This vertex and its suffix link
   printf(" [%2d]", v);
   if(f(v) != -1)
      printf(" f = %d", f(v));
   printf("\n");

   // The children
   for(int i=0; i<256; i++)
      if(tree[v].links[i].to != -1) {
         print(tree[v].links[i].to, tree[v].links[i].start,
               tree[v].links[i].end, prefix+"   ");
      }
}

// Initializes the suffix tree
// creates two vertices: root and dummy (root's parent)
void initTree()
{
   tree.clear();
   dummy = newVertex();
   root = newVertex();

   f(root) = dummy;
   for(int i=0; i<256; i++)
      link(dummy, -i-1, -i, root);
}

// Canonizes the reference pair (v, (start, end)) of a state (probably implicit)
pair<int, int> canonize(int v, int start, int end)
{
   if(end <= start) {
      return make_pair(v, start);
   } else {
      Link cur = tree[v].links[t(start)];
```

```
      while(end - start >= cur.end - cur.start) {
         start += cur.end - cur.start;
         v = cur.to;
         if(end > start)
            cur = tree[v].links[t(start)];
      }
      return make_pair(v, start);
   }
}

// Checks if there is a t-transition from the (probably implicit)
// state (v, (start, end))
pair<bool, int> testAndSplit(int v, int start, int end, UChar c)
{
   if(end <= start) {
      return make_pair(tree[v].links[c].to != -1, v);
   } else {
      Link cur = tree[v].links[t(start)];
      if(c == t(cur.start + end - start))
         return make_pair(true, v);

      int middle = newVertex();
      link(v, cur.start, cur.start + end - start, middle);
      link(middle, cur.start + end - start, cur.end, cur.to);
      return make_pair(false, middle);
   }
}

// Creates new branches
// (v, (start, end)) - the active point (its canonical reference pair)
//
// We want to add a t(end)-transition to this point, and to f(of it), f(f(of
// it)) and so on up to the end point
//
// NOTE: end must be a correct index in the sample string
pair<int, int> update(int v, int start, int end) {
   Link cur = tree[v].links[t(start)];
   pair<bool, int> splitRes;
   int oldR = root;

   splitRes = testAndSplit(v, start, end, t(end));
   while(!splitRes.first) {
      // Add a new branch
      link(splitRes.second, end, inf, newVertex());

      // Create a suffix link from the prev. branching vertex
      if(oldR != root)
         f(oldR) = splitRes.second;
      oldR = splitRes.second;

      // Go to the next vertex (in the final set of STrie(T_end))
      pair<int, int> newPoint = canonize(f(v), start, end);
      v = newPoint.first;
      start = newPoint.second;
      splitRes = testAndSplit(v, start, end, t(end));
   }
   if(oldR != root)
      f(oldR) = splitRes.second;
   return make_pair(v, start);
}

// Builds the whole suffix tree for the string sample
void ukkonen()
{
   // Initialize the tree
   initTree();

   // Add characters one by one
   pair<int, int> activePoint = make_pair(root, 0);
```

```
    for(int i=0; i<sample.length(); i++) {
       activePoint = update(activePoint.first, activePoint.second, i);
       activePoint = canonize(activePoint.first, activePoint.second, i+1);
    }
}

// Test: check if the word is in the tree
bool present(string word)
{
    int v=root, start=0, end=0;
    for(int i=0; i<word.length(); i++) {
       UChar cur = word[i];
       if(end==start) {
          if(tree[v].links[cur].to==-1) return false;
          start = tree[v].links[cur].start;
          end = start+1;
       } else {
          if(cur != t(end)) return false;
          end++;
       }
       if(end==tree[v].links[t(start)].end) {
          v = tree[v].links[t(start)].to;
          start=0;
          end=0;
       }
    }
    return true;
}

// A small test: "indexes" a text and searches for substrings in it
char inBig[1000], inSmall[1000];

int main() {
    // Ask for a text
    printf("Please enter a text: \n");
    fgets(inBig, sizeof inBig, stdin);
    sample = string(inBig);
    if(sample.length() != 0 && *(sample.end()-1)=='\n')
       sample.erase(sample.end()-1);

    // Build and print the tree
    ukkonen();
    print(root);
    printf("\nHere's your text again: %s\n", sample.c_str());

    // Handle requests
    printf("\nYou can search for substrings now. Type \"exit\" to quit.\n");
    while(true) {
       printf("Search: ");
       fgets(inSmall, sizeof inSmall, stdin);
       string what(inSmall);
       if(what.length() != 0 && what[what.length()-1]=='\n')
          what.resize(what.length()-1);
       if(what=="exit") break;
       printf("Result: %s\n", present(what)? "Positive" : "Negative");
    }
    return 0;
}
```

-- DanielShved - 20 Apr 2009