Tópicos Avançados em Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

21 de março de 2019

Algoritmos em Árvores

LCP - Lowest Common Parent - Parente Comum mais baixo

- Este é um algoritmo de caminhos mínimos em árvore.
- O menor caminho de um nó a outro em uma árvore passa pelo primeiro ascendente comum aos nós.
- Se pensarmos em uma árvore enraizada e níveis de altura, seria o primeiro ascendente de menor nível (mais baixo ou *lowest*).
- O algoritmo é simples, para um nó, nós listamos todos os ascendentes.
- Para o segundo nó começamos a procurar o primeiro ascendente que estiver na lista daquele nó.

Algoritmo LCP

Algoritmo $\mathrm{LCP}(n_1, n_2)$
    lista.add(n_1)
    enquanto $n_1 \neq root$ **faça**
        lista.add($n_1 \rightarrow pai$)
        $n_1 = n_1 \rightarrow pai$
    enquanto $\neg lista.possui(n_2)$ **faça**
        $n_2 = n_2 \rightarrow pai$
    retorne n_2

Problema URI 1135 - Colônia de Formigas

Um grupo de formigas está muito orgulhoso pois construíram uma grande e magnífica colônia. No entanto, seu enorme tamanho tem se tornado um problema, pois muitas formigas não sabem o caminho entre algumas partes da colônia. Elas precisam de sua ajuda desesperadamente!

A colônia de formigas foi criada como uma série de **N** formigueiros conectados por túneis. As formigas, obssessivas como são, numeraram os formigueiros sequencialmente à medida que os construiam. O primeiro formigueiro, numerado 0, não necessitava nenhum túnel, mas para cada um dos formigueiros subsequentes, 1 até **N**-1, as formigas também construíram um único túnel que conectava o novo formigueiro a um dos formigueiros existentes. Certamente, esse túnel era suficiente para permitir que qualquer formiga visitasse qualquer formigueiro já construído, possivelmente passando através de outros formigueiros pelo percurso, portanto elas não se preocupavam em fazer novos túneis e continuavam construindo mais formigueiros.

O seu trabalho é: dada a estrutura de uma colônia e um conjunto de consultas, calcular, para cada uma das consultas, o menor caminho entre pares de formigueiros. O comprimento do caminho é a soma dos comprimentos de todos os túneis que necessitam ser visitados.

Entrada:

Cada caso de teste se estende por várias linhas. A primeira linha contém um inteiro N representando a quantidade de formigueiros na colônia ($2 \leqslant \mathbf{N} \leqslant 10^5$). Cada uma das próximas $\mathbf{N}-1$ linhas contém dois inteiros que descrevem um túnel. A linha $\mathbf{i}$, para $1 \leqslant \mathbf{i} \leqslant \mathbf{N}-1$, contém $\mathbf{A}_i$ e $\mathbf{L}_i$, indicando que o formigueiro $\mathbf{i}$ foi conectado diretamente ao formigueiro $\mathbf{A}_i$ por um túnel de comprimento $\mathbf{L}_i$ ($0 \leqslant \mathbf{A}_i \leqslant \mathbf{i}-1$ e $1 \leqslant \mathbf{L}_i \leqslant 10^9$). A próxima linha contém um inteiro $\mathbf{Q}$ representando o número de consultas que seguem ($1 \leqslant \mathbf{Q} \leqslant 10^5$). Cada uma das $\mathbf{Q}$ linhas seguintes descreve uma consulta e contém dois inteiros distintos $\mathbf{S}$ e $\mathbf{T}$ ($0 \leqslant \mathbf{S}, \mathbf{T} \leqslant \mathbf{N}-1$), representando, respectivamente, os formigueiros de origem e destino.
O último caso de teste é seguido por uma linha contendo apenas um zero.

Saída:

Para cada caso de teste, imprima uma única linha com $\mathbf{Q}$ inteiros, os comprimentos do menor caminho entre os dois formigueiros de cada consulta. Escreva os resultados para cada consulta na mesma ordem em que aparecem na entrada.

Estratégia de solução

- Cada nó é numerado de 0 a n
- O raiz é o nó 0.
- Como só precisamos conhecer o raiz e a relação "pai", cada nó possui como informação o "pai" e o custo para chegar ao pai (`pair<int,long>`)
- Usamos o algoritmo LCP, mas ao invés de retornar o ancestral comum, retornamos o custo para chegar ao ancestral comum.
- Como são 10^5 nós e 10^9 cada custo, teremos 10^{14} o custo máximo, logo `long long int`
- No LCP, na lista dos ancestrais, também guarda-se um par: <pai, custo para chegar ao pai>

O código: main

```
int main() {
 int n, p, q, a, b;
 long l;
 cin >> n;
  while(n) {
  vector<pair<int, long> > v;
  v.push_back(make_pair(0,0));
  for(int i=1; i<n; i++) {
   cin >> p >> l;
   v.push_back(make_pair(p,l));
  }
  cin >> q;
  cin >> a >> b;
  cout << lcp(a,b,v);
  for(int i=1; i<q; i++) {
   cin >> a >> b;
   cout << " " << lcp(a,b,v);
  }
  cout << endl;
  cin >> n;
 }
 return 0;
}
```

Cabeçalhos e um predicado

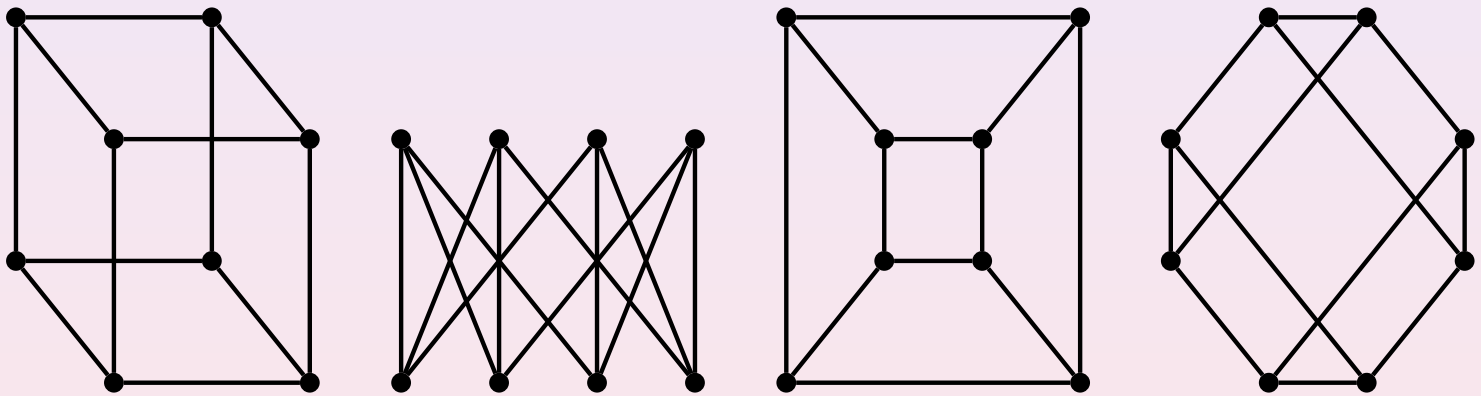
```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class found {
private:
  int i;
public:
  found(int idx):  i(idx) { }
  bool operator()(const pair<int, long long>& v) const {
    return i == v.first;
  }
};
```

LCP

```
long long lca(const int& x, const int& y, const vector<pair<int, long>
>& v) {
 int a = x, b = y;
 vector<pair<int, long long> > pa;
 pair<int, long long> pb;
 pa.push_back(make_pair(a,0));
 int i=1;
 while(a!=0) {
  pa.push_back(make_pair(v[a].first,pa[i-1].second+v[a].second));
  a = v[a].first;
  i++;
 }
 pb = make_pair(b,0);
 auto it=find_if(pa.begin(),pa.end(),found(pb.first));
 while(it==pa.end()) {
  pb = make_pair(v[b].first,pb.second+v[b].second);
  it=find_if(pa.begin(),pa.end(),found(pb.first));
  b = v[b].first;
 }
 return pb.second+it->second;
}
```

Isomorfismo em árvores

- Isomorfismo entre grafos significa que um grafo pode ser desenhado tal qual outro grafo, mantendo as mesmas relações de vizinhos entre os nós envolvidos.
- Um exemplo de grafos isomorfos é dado pelo desenho do 3-cubo:



(a) (b) (c) (d)

(a) tridimensional (b) bipartido (c) planar (d) ciclo hamiltoniano

Isomorfismo em árvores

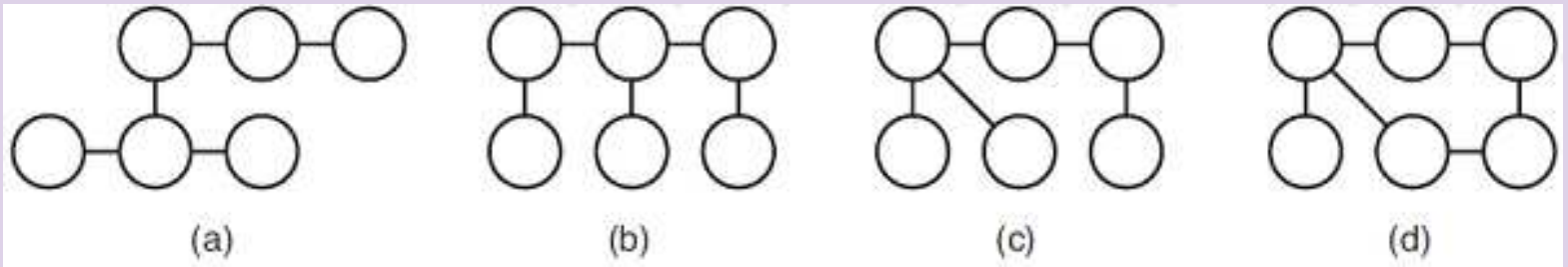
- No caso de isomorfismo em árvores, o tratamento é mais simples, pois a árvore não tem ciclos e ela pode ser processada das folhas para o centro.
- Existe um algoritmo em tempo linear para verificar se duas árvores são isomorfas.
- Para cada árvore se obtém um certificado.
- Os certificados são únicos, e se duas árvores são isomorfas, elas possuem o mesmo certificado.

O Algoritmo para computar certificados

1. Rotule todos os vértices com a string "01"
2. Enquanto existir mais de 2 vértices na árvore:
   1. Para cada nó x (não folha) na árvore, faça:
      1. Seja Y o conjunto de rótulos das folhas adjacentes a x, e o rótulo do próprio x sem o "0" inicial e o "1" final.
      2. Substitua o rótulo de x pelo rótulo formado pela concatenação dos rótulos em Y ordenados alfabeticamente.
      3. Remova todas as folhas adjacentes a x.
3. Se sobrar somente um vértice x na árvore, então o certificado da árvore é o próprio rótulo de x
4. Se sobrarem dois vértices x e y, o certificado é a concatenação de ambos rótulos em ordem alfabética.

Problema 1229 do URI: Combate ao Câncer

Pesquisadores da Fundação Contra o Câncer (FCC) anunciaram uma descoberta revolucionária na Química: eles descobriram como fazer átomos de carbono ligarem-se a qualquer quantidade de outros átomos de carbono, possibilitando a criação de moléculas muito mais complexas do que as formadas pelo carbono tetravalente. Segundo a FCC, isso permitirá o desenvolvimento de novas drogas que poderão ser cruciais no combate ao câncer.
Atualmente, a FCC só consegue sintetizar moléculas com ligações simples entre os átomos de carbono e que não contêm ciclos em suas estruturas: por exemplo, a FCC consegue sintetizar as moléculas (a), (b) e (c) abaixo, mas não a molécula (d).



Devido à agitação térmica, uma mesma molécula pode assumir vários formatos. Duas moléculas são equivalentes se for possível mover os átomos de uma das moléculas, sem romper nenhuma das ligações existentes nem criar novas ligações químicas, de forma que ela fique exatamente igual à outra molécula. Por exemplo, na figura acima, a molécula (a) não é equivalente à molécula (b), mas é equivalente à molécula (c).

Entrada:

Você deve escrever um programa que, dadas as estruturas de duas moléculas, determina se elas são equivalentes.
A primeira linha de um caso de teste contém um inteiro **N** indicando o número de átomos nas duas moléculas. Os átomos são identificados por números inteiros de 1 a **N** ($2 \leqslant \mathbf{N} \leqslant 10^4$). Cada uma das 2**N** - 2 linhas seguintes descreve uma ligação química entre dois átomos: as primeiras **N** - 1 linhas descrevem as ligações da primeira molécula; as **N** - 1 últimas descrevem as ligações químicas da segunda molécula. Cada linha contém dois inteiros **A** ($1 \leqslant \mathbf{A}$) e **B** ($\mathbf{B} \leqslant \mathbf{N}$) indicando que existe uma ligação química entre os átomos **A** e **B**.

Saída:

Para cada caso de teste seu programa deve imprimir uma única linha, contendo um único caractere: `S` se as moléculas são equivalentes ou `N` caso contrário.

Resolvendo o problema

- Primeira dificuldade é criar uma estrutura para a árvore. Como não é uma árvore enraizada, não será suficiente apenas a ligação para o pai, logo cada nó precisa possuir uma ligação aos demais nós.
- Informações necessárias:
  - Grau do nó (para saber se é folha).
  - Rótulo do nó.
  - Marcação de folha (para marcar quais folhas serão removidas).
  - ID do nó (número do nó)
  - Ligações do nó (quais são os outros nós ligados a este nó) $\rightarrow$ Um vetor de árvores
- A árvore é criada como um vetor de nós, cada nó é da classe `tree`

Bibliotecas necessárias

```
#include <iostream>
#include <utility>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
```

- Os vizinhos de um nó, na árvore, é um vetor de ponteiros para nós.
- Como dentro da classe precisamos fazer referência a objetos da classe, então só podemos usar ponteiros.
- Ao declarar a classe, o método `removeLink` precisa de uma classe que em sua implementação usa referência à classe `tree`, logo o método é implementado após a definição da outra classe.

Classe tree

```
class tree {
public:
 string cert; // rótulo do nó
 int id; // identificação do nó
 int degree; // grau do nó
 bool isleaf; // este nó é uma folha?
 vector<tree *> neigh; // nós vizinhos deste nó

 tree(int d):  id(d) { // iniciação da árvore
  cert="01"; // inicialmente só tem o rótulo
  degree=0; // árvore vazia
  isleaf=false; // não é folha
 }
 void removeLink(int d); // remove o vizinho de ID = d
 void addLink(tree& t) { // Adiciona um vizinho para este nó
  neigh.push_back(&t); // uma referência ao vizinho
  degree++; // Aumentou o grau deste nó.
  return;
 }
};
```

Método removeLink e o objeto função para o find_if

```
class found { // Predicado para achar um vizinho na lista (find_if)
private:
  int id; // ID do vizinho que vamos procurar
public:
  found(int d):  id(d) { } // Iniciando o ID
  bool operator()(const tree* n) { // Predicado para o find_if
    return id==n->id; // É o vizinho que queremos?
  }
};

void tree::removeLink(int d) { // Remove o vizinho de nó ID = d
  auto it=find_if(neigh.begin(),neigh.end(),found(d)); // busca
  neigh.erase(it); // Achou o vizinho, então remove
  degree--; // O grau deste nó diminuiu
  return;
}
```

Função para ordenar e concatenar rótulos (podia usar multiset)

```
string mergcert(vector<string>& v) { // Criando o certificado
 string cert=v[0]; // v contém uma lista de rótulos
 v[0]=cert.substr(1,cert.size()-2); // O rótulo deste nó sem 0 e 1
 cert="0"; // primeiro 0
 sort(v.begin(),v.end()); // Ordenando os rótulos
 for(int i=0; i< v.size(); i++) {
  cert+=v[i]; // concatenando os rótulos
 }
 cert+="1"; // último 1
 return cert;
}
```

Main: iniciando as árvores

```
int main() {
 int n, no1, no2;

 while(cin >> n) {
  string cert1, cert2;
  vector<tree> t1, t2; // As duas árvores para comparação.
  for(int i = 0; i < n; i++) {
   t1.push_back(tree(i+1)); // Iniciando os n nós
   t2.push_back(tree(i+1)); // de ambas
  }
  for(int i = 0; i < n-1; i++) { // Lendo Árvore 1
   cin >> no1 >> no2; // Lendo as ligações
   t1[no1-1].addLink(t1[no2-1]); // Atribuindo a ligação ao nó 1
   t1[no2-1].addLink(t1[no1-1]); // Atribuindo a ligação ao nó 2
  }
  for(int i = 0; i < n-1; i++) { // Lendo Árvore 2
   cin >> no1 >> no2;
   t2[no1-1].addLink(t2[no2-1]);
   t2[no2-1].addLink(t2[no1-1]);
 }
```

Main: algoritmo para certificado da árvore

```
int cont=n; // Algoritmo para árvore 1.
while(cont > 2) { // ALGORITMO: Enquanto existir mais de 2 vértices
 for(int i = 0; i < n; i++) {
  if(t1[i].degree==1) t1[i].isleaf=true; // folhas para remover
  else if(t1[i].degree > 1) { // ALGORITMO: para cada nó x não folha
   vector<string> certs; // ALGORITMO: seja Y (certs) o conjunto
   auto viz = t1[i].neigh.begin();
   certs.push_back(t1[i].cert); // do próprio x
   while(viz!=t1[i].neigh.end()) { // e das folhas
    if((*viz)->degree==1) certs.push_back((*viz)->cert);
    viz++;
   } // substitua x pelo rótulo concatenado
   t1[i].cert = mergcert(certs);
  }
 }
```

Main: removendo as folhas

```
  for(int i = 0; i < n; i++) { // ALGORITMO: remova todas as folhas
   if(t1[i].isleaf) {
    t1[i].neigh[0]->removeLink(t1[i].id);
    t1[i].removeLink(t1[i].neigh[0]->id);
    t1[i].isleaf=false;
    t1[i].cert="";
    cont--;
   }
  }
}
```

Main: gerando o certificado da árvore 1

```
auto vi = t1.begin();
vector<string> s; // ALGORITMO:
while(vi!=t1.end()) { // os rótulos dos vértices que sobraram
 while(vi!=t1.end() && vi->cert=="") vi++;
 if(vi!=t1.end()) {
  s.push_back(vi->cert);
  vi++;
 }
}
sort(s.begin(),s.end()); // ordenados alfabeticamente
auto si = s.begin();
while(si!=s.end()) {
 cert1+=*si; // concatenado como certificado final.
 si++;
}
```

Main: algoritmo para certificado da árvore

```
cont=n; // Algoritmo para árvore 2.
while(cont > 2) { // ALGORITMO: Enquanto existir mais de 2 vértices
 for(int i = 0; i < n; i++) {
  if(t2[i].degree==1) t2[i].isleaf=true; // folhas para remover
  else if(t2[i].degree > 1) { // ALGORITMO: para cada nó x não folha
   vector<string> certs; // ALGORITMO: seja Y (certs) o conjunto
   auto viz = t2[i].neigh.begin();
   certs.push_back(t2[i].cert); // do próprio x
   while(viz!=t2[i].neigh.end()) { // e das folhas
    if((*viz)->degree==1) certs.push_back((*viz)->cert);
    viz++;
   } // substitua x pelo rótulo concatenado
   t2[i].cert = mergcert(certs);
  }
 }
```

Main: removendo as folhas

```
  for(int i = 0; i < n; i++) { // ALGORITMO: remova todas as folhas
   if(t2[i].isleaf) {
    t2[i].neigh[0]->removeLink(t2[i].id);
    t2[i].removeLink(t2[i].neigh[0]->id);
    t2[i].isleaf=false;
    t2[i].cert="";
    cont--;
   }
  }
}
```

Main: gerando o certificado da árvore 2

```
vi = t2.begin();
s.clear(); // ALGORITMO:
while(vi!=t2.end()) { // os rótulos dos vértices que sobraram
 while(vi!=t2.end() && vi->cert=="") vi++;
 if(vi!=t2.end()) {
  s.push_back(vi->cert);
  vi++;
 }
}
sort(s.begin(),s.end()); // ordenados alfabeticamente
auto si = s.begin();
while(si!=s.end()) {
 cert2+=*si; // concatenado como certificado final.
 si++;
}
```

Main: terminando o código - comparando certificados

```
  if(cert1==cert2) cout << "S" << endl;
    else cout << "N" << endl;
  }
  return 0;
}
```