Árvore geradora

Uma subárvore de um grafo *G* é qualquer árvore *T* que seja <u>subgrafo</u> de *G*. Em outras palavras, um árvore *T* é subárvore de *G* se todo vértice de *T* é vértice de *G* e toda aresta de *T* é aresta de *G*.

Uma subárvore geradora ou árvore geradora (= *spanning tree*) de um grafo *G* é qualquer subárvore de *G* que contenha todos os vértices de *G*.

É claro que somente grafos conexos têm árvores geradoras. Reciprocamente, todo grafo conexo tem uma árvore geradora. (Em geral, um grafo conexo tem muitas árvores geradoras diferentes.)

Primeira Propriedade da Troca: Seja *T* uma árvore geradora de um grafo *G*. Para qualquer aresta *e* de *G* que não esteja em *T*, o grafo $T + e$ tem um único ciclo não-trivial. Para qualquer aresta *t* desse ciclo, o grafo $T + e - t$ é uma árvore geradora de *G*.

Exemplo: Seja *T* a árvore geradora definida pelas cinco arestas "internas" da figura. Seja *e* a aresta `0-1`. O único ciclo não-trivial de *T*+*e* é `0-1-5-4-0`. Para qualquer aresta *t* desse ciclo, *T*+*e*−*t* é uma árvore geradora.

```
0---------------1
|\             /|
| \           / |
|  \         /  |
|   4-------5   |
|  /         \  |
| /           \ |
|/             \|
3---------------2
```

Condição de Otimalidade: Se *T* é uma MST de um grafo então toda aresta *e* fora de *T* tem custo máximo dentre as arestas do único ciclo não-trivial em *T*+*e*.

Segunda Propriedade da Troca: Seja *T* uma árvore geradora de um grafo *G*. Para qualquer aresta *t* de *T* e qualquer aresta *e* de *G* que torne grafo determinado por *T*−*t* +*e conexo*, o grafo $T - t + e$ é uma árvore geradora de *G*.

Seja *T* a árvore geradora definida pelas cinco arestas "internas" da figura. Seja *t* a aresta 4-5. Existem três arestas que reconectam o grafo: 0-1, 4-5 e 3-2. Se *e* é uma qualquer dessas arestas então *T*−*t*+*e* é uma árvore geradora.

```
0---------------1
|\             /|
| \           / |
|  \         /  |
|   4-------5   |
|  /         \  |
| /           \ |
|/             \|
3---------------2
```

Condição de Otimalidade: Se *T* é uma MST de um grafo então cada aresta *t* de *T* é uma aresta mínima dentre as que conectam a árvore T-t.

Algoritmo de Kruskal

1. crie uma floresta *F* (um conjunto de árvores), onde cada vértice no grafo é uma árvore separada
2. crie um conjunto *S* contendo todas as arestas do grafo
3. enquanto *S* for não-vazio, faça:
    a. remova uma aresta com peso mínimo de *S*
    b. se essa aresta conecta duas árvores diferentes, adicione-a à floresta, combinando duas árvores numa única árvore parcial
    c. do contrário, descarte a aresta

Algoritmo de Prim

```
#include <values.h>
const int INF = MAXINT/2;

int fixo[MAXN];
int custo[MAXN];


main:

int total = 0;

for(int i=0; i<n; i++) {
  fixo[i] = 0;
  custo[i] = INF;
}
custo[0] = 0;

for(int faltam = n; faltam>0; faltam--) {

  //Encontra a aresta com custo mínimo que constroem uma árvore
  int no = -1;
  for(int i=0; i<n; i++)
    if(!fixo[i] && (no==-1 || custo[i] < custo[no]))
      no = i;
  fixo[no] = 1;

  //se não encontra termine
  if(custo[no] == INF) {
    total = INF;
    break;
  }
  total += custo[no];

  //atualize os custos
  for(int i=0; i<n; i++)
    if(custo[i] > G[no][i])
      custo[i] = G[no][i];
}
```

Exercício Modifique o algoritmo de Prim para utilizar lista de prioridades (heap)

```
typedef struct edge{
       int x,y,w;
}edge;
int p[101],ordem[101];
int compara(const void *a,const void *b){
    edge ea = *(edge*)a;
    edge eb = *(edge*)b;
    return ea.w - eb.w;
}
void make_set(int x){
     p[x] = x; ordem[x] = 0;
}
int find_set(int x){
    if( x!= p[x] ) p[x] = find_set(p[x]);
    return p[x];
}
void link(int x,int y){
     if(ordem[x] > ordem[y]){ p[y] = x;
     }else{
        p[x] = y;
        if(ordem[x]==ordem[y])ordem[y]++;
     }
}
void une(int x,int y){
     link(find_set(x),find_set(y));
}
int same_componente(int x,int y){
    if(find_set(x)==find_set(y)) return 1;
    else return 0;
```

```
}
```

Main:

```
qsort(edges,m,sizeof(edge),compara);

for(i=0;i<n;i++)make_set(i+1);

i = 0;

S = 0;

while( S < n - 1 ){

    x = edges[i].x;

    y = edges[i].y;

    if( !same_componente(x,y) ){

              une(x,y); S++;

    }

    i++;

 }
```

Heurísticas para melhorar o tempo de execução utilizada

A primeira heurística, união por ordenação, faz a raiz da árvore com menor número de nós apontar para a raiz da árvore com mais nós. Em vez de controlar de modo explícito o tamanho da subárvore com raiz em cada nó, usaremos uma abordagem que facilita a análise. Para cada nó, mantemos uma ordem que é um limitante superior sobre a altura do nó. Na união por ordenação, a raiz com menor ordem é levada a apontar para a raiz com maior ordem durante uma operação UNE.

A segunda heurística, compressão de caminho, também é bastante simples e muito eficiente. Nós usaremos durante a operação de FIND-SET para fazer cada nó no caminho de localização apontar diretamente para a raiz. A compressão de caminho não altera quaisquer ordens.

Considerações sobre o tempo de execução

O tempo de execução da heurística combinada de união por ordenação e compressão de caminho é $O(m\ \alpha(n))$ para m operações de conjuntos disjuntos sobre n elementos. Em qualquer aplicação concebível de uma estrutura de dados de conjuntos disjuntos, $\alpha(n) < 4$; desse modo, podemos considerar o tempo de execução linear em m para todas as situações práticas.