Arbori parțiali de cost minim

Arbori parțiali de cost minim

Kruskal

- **Inițial T= (V; $\varnothing$)**
- **pentru i = 1, n-1**
  - alege o muchie uv cu **cost minim** a.î. u,v sunt în **componente conexe diferite (T+uv aciclic)**
  - $E(T) = E(T) \cup uv$

Prim

- **s- vârful de start**
- **Inițial T= ({s}; $\varnothing$)**
- **pentru i = 1, n-1**
  - alege o muchie uv cu **cost minim** a.î. $u \in V(T)$ și $v \notin V(T)$
  - $V(T) = V(T) \cup \{v\}$
  - $E(T) = E(T) \cup uv$

Kruskal

```
sorteaza(E)
for(v=1;v<=n;v++)
    Initializare(v);
nrmsel=0
for(uv ∈ E)
 if(Reprez(u)!=Reprez(v))
  {
     E(T) = E(T) ∪ {uv};
     Reuneste(u,v);
     nrmsel=nrmsel+1;
     if(nrmsel==n-1)
         STOP;
   }
```

Kruskal

Complexitate – dacă folosim arbori

- **Sortare** -> O(m log m) = O(m log n)
- **n * Initializare** -> O(n)
- **2m * Reprez** -> O(m log n)
- **(n-1) * Reuneste** -> O(n log n)

O(m log n)

Prim(G, w, s)

```
pentru fiecare u∈V executa
      d[u] = ∞; tata[u]=0
 d[s] = 0
 inițializează Q cu V
 cat timp Q ≠ ∅ executa
       u=extrage vârf cu eticheta d minimă din Q
       pentru fiecare v adiacent cu u executa
             daca v∈Q si w(u,v)<d[v] atunci
                  d[v] = w(u,v)
                  tata[v] = u
                  //actualizeaza Q - pentru Q heap
 scrie (u, tata[u]), pentru u≠ s
```

Prim

Complexitate

Varianta 1 – cu vector de vizitat

- **Inițializări** -> $O(n)$
- **n * extragere vârf minim** -> $O(n^2)$
- **actualizare etichete vecini** -> $O(m)$

$O(n^2)$

Prim

Varianta 2 – memorarea vârfurilor din într-un min-heap Q (min-ansamblu)

- **Inițializare Q** -> O(n)
- **n * extragere vârf minim** -> O(n log n)
- **actualizare etichete vecini** -> O(m log n)

O(m log n)

Implementare Prim

Problemă – reparare heap după modificare chei

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

1. Heap propriu (implementat), pentru fiecare vârf se memorează și poziția în heap pentru a ști unde trebuie "reparat" => cel mult n elemente în heap

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

2. Priority queue PQ cu reinserarea vârfului cu noua etichetă

=> un vârf poate fi in PQ de mai multe ori

(dimensiunea PQ cât poate fi maxim?)

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

2. Priority queue PQ cu reinserarea vârfului cu noua etichetă

=> un vârf poate fi in PQ de mai multe ori

(dimensiunea PQ cât poate fi maxim?)

=> relaxăm arcele care ies din v doar prima dată când este extras din PQ

=> dimensiunea PQ O(m)

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

3. O structură care să permită ștergere și inserare in O(log(n)) (atunci modificare eticheta v= ștergere+inserare) => set din stl (arbore binar de căutare)

Implementare Prim

Detalii – implementare cu PQ

Implementare Prim

```
//citire graf
vector< pair<int,double> > *la;
f>>n;
f>>m;
la = new vector< pair<int,double> >[n+1];
//graf- memorat cu liste de adiacenta

for(i=1;i<=m;i++){
        f>>x>>y>>c;
        la[x].push_back(make_pair(y,c));//merge si {y,c}
        la[y].push_back(make_pair(x,c));
}
. . .
for(u=1;u<=n;u++){
        viz[u]=tata[u]=0;
        d[u]=infinit;
}
```

Implementare Prim

```
#citire graf
n,m=(int(x) for x in f.readline().split())

la=[[] for i in range(n+1)]



for i in range(m):
    x,y,c = (int(x) for x in f.readline().split())
    la[x].append((y,c))
    la[y].append((x,c))

d = [float("inf")]*(n+1)
tata=[0]*(n+1)
viz=[0]*(n+1)
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
while(!Q.empty()){
    u=Q.top().second;//varful nevizitat cu d minim
    Q.pop();
    viz[u]++;

    //daca este prima extragere din Q a lui u relaxam arcele









}
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
while(!Q.empty()){
    u=Q.top().second;//varful nevizitat cu d minim
    Q.pop();
    viz[u]++;

    //daca este prima extragere din Q a lui u relaxam arcele
    if(viz[u]==1){
        for(j=0;j<la[u].size();j++){
            v=la[u][j].first;
            w_uv=la[u][j].second;
            if(viz[v]==0) {









            }
        }
    }
}
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
while(!Q.empty()){
    u=Q.top().second;//varful nevizitat cu d minim
    Q.pop();
    viz[u]++;

    //daca este prima extragere din Q a lui u relaxam arcele
    if(viz[u]==1){
        for(j=0;j<la[u].size();j++){
            v=la[u][j].first;
            w_uv=la[u][j].second;
            if(viz[v]==0) {
                if(d[v]>w_uv){
                    tata[v]=u;
                    d[v]=w_uv;
                    Q.push(make_pair(-d[v],v));
                }
            }
        }
    }
}
```

```
d[s]=0
Q=[]
heapq.heappush(Q,(d[s],s))
while len(Q)>0:
    u=heapq.heappop(Q)[1] #varful nevizitat cu d minim
    viz[u]+=1

    #daca este prima extragere din Q a lui u relaxam arcele
    if viz[u]==1:
        for (v,w_uv) in la[u]:
            if viz[v]==0:
                if d[v]>w_uv:
                    tata[v]=u
                    d[v]=w_uv
                    heapq.heappush(Q,(d[v],v))
```

Arbori parțiali de cost minim

Problema 1 (Cormen).

- Dacă toate muchiile au cost între 1 și $|V|$ cât de rapid poate deveni algoritmul lui Kruskal? Dar algoritmul lui Prim?
- Dar dacă toate muchiile au costul între 1 și W, unde W = constantă?

Arbori parțiali de cost minim

Problema 2.

Fie $G = (V, E, w)$ un graf conex ponderat și T_{min} un apcm în G.

a) Fie k un număr pozitiv și graful $G_1 = (V, E, w_1)$ unde

$$w_1(e) = w(e) + k$$

Este T_{min} apcm și în G_1?

b) Fie k un număr pozitiv și graful $G_2 = (V, E, w_2)$ unde

$$w_2(e) = w(e) * k$$

Este T_{min} apcm și în G_2?

Arbori parțiali de cost minim

Problema 2. – problema similara pentru drumuri minime

Fie G = (V, E, w) un graf orientat ponderat (fără circuite negative), s și t două vârfuri distincte din G și P un s-t drum minim în G.

a) Fie k un număr pozitiv și graful $G_1 = (V, E, w_1)$ unde

$$w_1(e) = w(e) + k$$

Este P un s-t drum minim și în G_1?

b) Fie k un număr pozitiv și graful $G_2 = (V, E, w_2)$ unde

$$w_2(e) = w(e) * k$$

Este P un s-t drum minim și în G_2

Arbori parțiali de cost minim

Temă. Fie G un graf conex ponderat cu ponderile muchiilor distincte. Arătați că există un unic apcm al lui G.

Arbori parțiali de cost minim

Problema 3.

Fie $G=(V, E, w)$ conex cu ponderi distincte

Fie T_{min} unicul apcm al lui G

Fie T_s un arbore second best

Atunci există $uv \in T_{min}$ și $xy \notin T_{min}$ astfel încât

$$T_s = T_{min} - uv + xy$$

Arbori parțiali de cost minim

Problema 4.

Fie G un graf conex ponderat și T un arbore parțial de cost minim.

Poate fi T obținut prin algoritmul lui Kruskal?

Arbori parțiali de cost minim

Problema 4.

Fie G un graf conex ponderat și T un arbore parțial de cost minim.

Arătați că dacă în lista de muchii ale lui G ordonate după cost în caz de egalitate se dau prioritate muchiilor din T, atunci rezultatul aplicării algoritmului lui Kruskal este chiar T

(**orice arbore parțial de cost minim al lui G poate fi obținut de algoritmul lui Kruskal**)

Arbori parțiali de cost minim

Temă. Fie G un graf conex ponderat și e o muchie de cost minim.

a) Arătați că există un apcm care conține e.

b) Orice apcm din G conține e? Dar o muchie de cost w(e)?

Drumuri minime

Problemă. Fie G un graf orientat ponderat și s un vârf în G. Dacă toate arcele au cost pozitiv cu excepția arcelor care ies din s care pot avea și cost negativ (**dar fără a forma circuite negative**), putem determina distanțele de la s la celelalte vârfuri folosind algoritmul lui Dijkstra? Justificați.