Arbori parțiali de cost minim

Probleme – maxim două

1. Implementaţi eficient algoritmul lui Prim pentru determinarea unui arbore parţial de cost minim (economic) al unui graf simplu, conex, ponderat cu n vârfuri şi m muchii (**O(m log n)**). Se vor da la intrare numărul de vârfuri ale grafului, numărul de muchii şi fiecare muchie a grafului cu costul său. (**4p**)
   http://www.infoarena.ro/problema/apm

2. Implementaţi eficient algoritmul lui Kruskal pentru determinarea unui arbore parţial de cost minim (economic) al unui graf simplu, conex, ponderat cu n vârfuri şi m muchii (**O(m log n)**). Se vor da la intrare numărul de vârfuri ale grafului, numărul de muchii şi fiecare muchie a grafului cu costul său. (**4p**)
   http://www.infoarena.ro/problema/apm

3. **Second best minimum spanning tree** – Implementaţi un algoritm eficient pentru determinarea **primilor doi** arbori parţial cu cele mai mici costuri, pentru un graf simplu, conex, ponderat cu n vârfuri şi m muchii, în ipoteza că **muchiile au costuri distincte** (**O(n^2)**). Se vor da la intrare numărul de vârfuri ale grafului, numărul de muchii şi fiecare muchie a grafului cu costul său. Pentru determinarea arborelui parţial de cost minim (primul) se va folosi un algoritm de complexitate **O(m log n)** (**9p**)

4. http://www.infoarena.ro/problema/retea2 (**5p)**

5. http://campion.edu.ro/arhiva/index.php?page=problem&action=view&id=960 (**5p**)

6. http://www.infoarena.ro/problema/online (**5p –** implementarea optimă a algoritmului pentru arbori parţiali de cost minim ales)

7. Orice altă problemă care se reduce la determinarea de arbori parţiali de cost minim

Bibliografie

T.H. Cormen, C.E. Leiserson, R.L. Rivest – Introduction to algorithms, MIT Press, Cambridge, 1990/2001

Algoritmul lui Kruskal

Pe parcursul algoritmului lui Kruskal muchiile selectate formează o pădure.

Inițial mulțimea muchiilor selectate este vidă, fiecare vârf al grafului constituind un arbore.

La fiecare pas este selectată o muchie de cost minim cu extremitățile în arbori diferiți din pădurea deja construită (altfel spus, o muchie de cost minim care nu formează cicluri cu muchiile deja existente).

Pentru a simplifica alegerea unei muchii de cost minim muchiile sunt sortate inițial în ordinea crescătoare a costurilor și vor fi prelucrate de algoritm în această ordine.

Mai trebuie găsită o metodă eficientă de a testa dacă o muchie unește doi arbori din pădurea deja construită sau are extremitățile în același arbore (deci formează ciclu cu muchiile deja selectate). Este suficient să găsim o modalitate de a memora vârfurile arborilor. Pentru aceasta se folosesc **structuri de date pentru mulțimi disjuncte** (mulțimile vârfurilor arborilor sunt mulțimi disjuncte). Fiecărei mulțimi i se asociază un reprezentatnt. Operațiile principale cu mulțimi disjuncte sunt:

`MakeSet`(u) – formează o mulțime cu un singur element u

`FindSet`(u) – returnează reprezentantul mulțimii care conține pe u

`Unoin`(u,v) – reunește mulțimea care conține elementul u cu cea care conține elementul v.

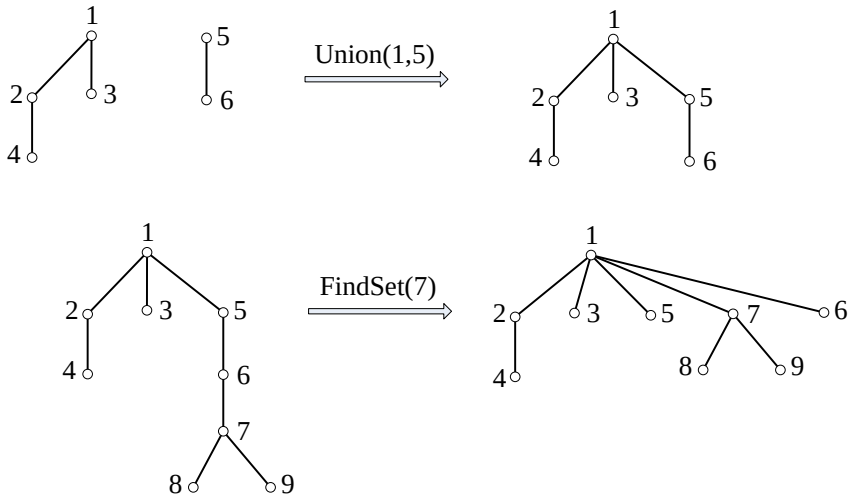
Cu aceste operații **algoritmul lui Kruskal** este următorul:

```
A=∅
for (v ∈ V(G))
     MakeSet(v) //inițial fiecare vârf este un arbore
Sort(E(G),w) //sortează muchiile grafului crescător după cost
for(uv ∈ E(G) în ordinea costurilor)
     if(FindSet(u)!=FindSet(v){
          A=A∪{uv}
          Union(u,v)
          if(|A|==n-1)
               break
     }
return A
```

- Un mod simplu de a reprezenta mulțimile disjuncte este folosirea unui **vector de reprezentanți** r cu n elemente cu semnificația
r[u] = reprezentantul mulțimii căreia aparține u.
Astfel, `MakeSet`(u) se reduce la atribuirea r[u]=u, iar `FindSet`(u) la a returna r[u].
Operația `Unoin`(u,v) presupune însă schimbarea peste tot în vectorul r a valorii r[u] cu valoarea r[v] (reprezentantul mulțimii care îl conține pe u devine reprezentantul mulțimii care îl conține pe v), deci are complexitatea O(n).

- Pentru o implementare mai rapidă a mulțimilor disjuncte, reprezentăm o mulțime printr-un **arbore** având ca rădăcină reprezentantul mulțimii, pentru fiecare element al mulțimii fiind memorat tatăl acestuia în arbore.

Operația `FindSet`(u) presupune urmărirea legăturii de tip tata din vârful u până în rădăcină. Nodurile de pe acest lanț de la vârful u la rădăcină constituie **drumul de căutare**. Operația `Unoin`(u,v) va avea ca efect faptul că rădăcina unui arbore va avea ca tată rădăcina celuilalt arbore. Pentru ca operațiile să fie cât mai eficiente trebuie ca înălțimea arborilor reprezentând mulțimile disjuncte să fie cât mai mică. Pentru aceasta la reuniune rădăcina arborelui cu înălțime mai mică va arăta (va avea ca tată) rădăcina arborelui cu înălțime mai mare – **reuniune ponderată**. În plus, în timpul operației `FindSet`(u) putem modifica legătura de tip tata a vârfurilor de pe drumul de căutare de la u la rădăcină, astfel încât aceasta să arate direct către rădăcină (astfel, la o viitoare căutare, reprezentantul acestor vârfuri va fi chiar tatăl lor, nu va mai fi necesară o urcare în arbore) – **compresie de cale**.



Dacă pentru fiecare vârf u reținem în h[u] o margine superioară a înălțimii lui u (numărul de muchii ale celui mai lung lanț de la u la o frunză descendentă) și în p[u] tatăl vârfului u, pseudocodul pentru cele trei operații este următorul:

```
MakeSet(u)
        p[u]=0
        h[u]=0

FindSet(u)
        if (p[u]==0)
              return u
        p[u]=FindSet(p[u]) //tatal lui u devine radacina
        return p[u]

Union(u,v)
        u=FindSet(u)
        v=FindSet(v)
        if (h[u]>h[v])
              p[v]=u
        else
              p[u]=v
              if(h[u]==h[v])
                    h[v]=h[v]+1
```

În practică se poate considera că un ansamblu de q operații cu mulțimi disjuncte implementate astfel are complexitatea O(q).

Complexitatea algoritmului Kruskal

```
T(n, m) ≤ n·complexitate(MakeSet)+complexitate(sort)+
      +2m·complexitate(FindSet) + (n-1)·complexitate(Union)
```

Pentru reprezentarea mulțimilor disjuncte folosind vectori de reprezentanți obținem complexitatea $O(n^2+m \log n)$. Dacă folosim însă arbori (cu reuniune ponderată şi compresie de cale) obținem **O(m log n).**

Algoritmul lui Prim

Pe parcursul algoritmului lui Prim muchiile selectate formează întotdeauna un singur arbore.

Arborele porneşte de la un vârf arbitrar r si creşte până acoperă toate vârfurile din V. La fiecare pas se adaugă arborelui o muchie de lungime minimă care uneşte un vârf al arborelui A cu un vârf neselectat până la acel pas.

Cheia implementării eficiente a algoritmului lui Prim este să procedăm astfel încât sa fie uşor să selectăm o nouă muchie de cost minim între un vârf selectat şi un vârf neselectat pentru a fi adăugată la arbore. Pentru aceasta, asociem fiecărui vârf neselectat o cheie reprezentând **costul minim al unei muchii care îl unește de un vârf deja selectat**. La fiecare selectare a unui nou vârf pentru a fi adăugat în arbore, se actualizează cheile vârfurilor adiacente cu el rămase neselectate.

În pseudocodul de mai jos graful conex G si rădăcina r sunt date de intrare. Pentru fiecare vârf v, `cheie`[v] este minimul costurilor muchiilor care unesc pe v cu vârfurile selectate (din arborele A). Vârful din arbore pentru care se realizează acest minim se reţine în `p`[v]. Astfel, la fiecare pas, muchia (v, `p`[v]) va fi muchia de cost minim care uneşte pe v de un vârf selectat, iar lungimea acestei muchii este `cheie`[v] şi reprezintă distanţa minimă de la v la arborele A.

Când un vârf u aflat la distanţa minimă de A este adăugat la arbore, `p`[u] va reprezenta părintele lui în arbore.

Prin convenţie `cheie`[v]=∞ dacă nu există nici o muchie care să unească pe v de un vârf selectat.

În timpul execuţiei algoritmului toate vârfurile care nu sunt în arbore se află într-o structură Q (care poate fi de exemplu un ansamblu bazat pe câmpul cheie).

Funcţia `extrageMinim` (Q) extrage din Q vârful cu cheie minimă şi îl returnează.

În timpul algoritmului mulţimea A din algoritmul generic este păstrată implicit ca `A={(v, p[v]), v∈V-{r}-Q}`

Algoritmul se termină când Q este vidă. Arborele parţial de cost minim A al lui G este astfel `A={(v, p[v]), v∈V-{r}}`

```
procedure APCM_PRIM(G, w, r)
      creare(Q, V[G])
      pentru fiecare u∈Q executa
          cheie[u] = ∞
      cheie[r] = 0
      p[r] = null
      cat timp Q ≠ ∅ executa
            u = extrageMinim(Q)
            pentru fiecare v∈Adj[u] executa
                  daca v∈Q si w(u, v)<cheie[v] atunci
                        p[v] = u
                        cheie[v] = w(u, v)
                        /*actualizare(Q)-daca structura pentru Q
      necesita reactualizare, cum este în cazul ansamblului*/
```

În liniile 1-4 se iniţializează Q astfel încât să conţină toate vârfurile şi se iniţializează cheia fiecărui vârf cu ∞, excepţie făcând rădăcina r a cărei cheie este iniţializată cu 0.

p[r] se iniţializează cu `null` deoarece rădăcina r nu are nici un părinte. Pe parcursul algoritmului Q va conţine vârfurile neselectate, iar mulţimea V-Q conţine vârfurile arborelui curent (cele selectate).

Vârful care va fi selectat la fiecare pas este, conform algoritmului lui Prim, elementul u cu cheie minimă din Q. Muchia (u, `p`[u]) are costul `cheie`[u], deci aceasta este o muchie de cost minim care uneşte un vârf neselectat cu unul selectat. Elementul u selectat va fi eliminat din mulţimea vârfurilor neselectate Q. Primul vârf extras din Q va fi r.

După selectarea lui u se actualizează câmpurile `cheie` si `p` ale fiecărui vârf v adiacent lui u, dar care nu a fost încă selectat. Vechea cheie a lui v (reprezentând costul minim al unei muchii care îl uneşte pe v cu un vârf anterior selectat) este comparată cu lungimea muchiei care îl uneşte pe v de noul vârf selectat u, pentru a vedea dacă prin adăugarea vârfului u distanţa de la v la arborele construit nu s-a micşorat. În cazul când această distanţă s-a micşorat, se actualizează corespunzător cheia şi părintele lui v.

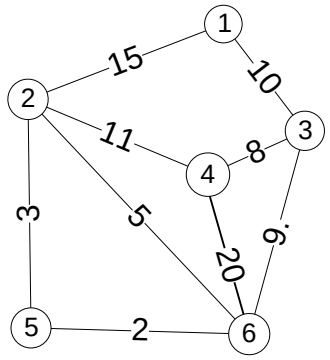
Complexitatea algoritmului Prim

```
T(n, m) ≤ complexitate(creare(Q, V[G]))+
      + n·complexitate(extrageMinim(Q))+
      + m·complexitate(actualizare(Q))
```

Complexitatea diferă în funcţie de structura aleasă pentru Q.
Folosind chiar vectorul de chei pe post de Q obţinem complexitatea $O(n^2)$.
Folosind ansamblu (heap) obţinem O(m log n).

Exemplu

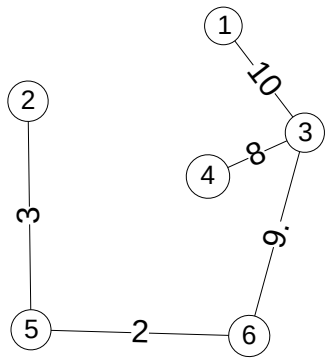
Kruskal :



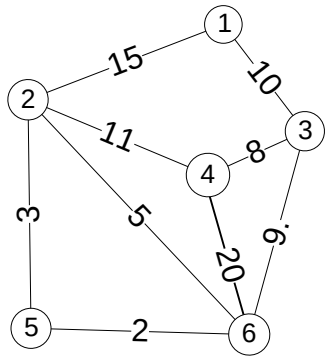
Muchiile vor fi prelucrate în ordinea

(5, 6) – cost 2
(2, 5) – cost 3
(2, 6) – cost 5
(3, 4) – cost 8
(3, 6) – cost 9
(1, 3) – cost 10
(2, 4) – cost 11
(1, 2) – cost 15
(4, 6) – cost 20

Algoritmul va selecta muchia (5, 6), apoi (2, 5), nu va selecta muchia (2, 6) deoarece formează cicluri cu cele deja selectate (are extremitățile în același arbore), va selecta apoi (3, 4), (3, 6), (1, 3) și se va opri deoarece muchiile selectate formează un arbore. Obținem următorul arbore de cost minim



Prim:



Vârf de pornire – 1
Marcăm cu * vârful selectat la pasul curent

vârf	1	2	3	4	5	6
Cheie/p	0/null *	∞/null	∞/null	∞/null	∞/null	∞/null
	-	15/1	10/1 *	∞/null	∞/null	∞/null
	-	15/1	-	8/3 *	∞/null	9/3
	-	11/4	-	-	∞/null	9/3 *
	-	5/6	-	-	2/6 *	-
		3/5 *				

După ce au fost selectate toate vârfurile, muchiile se pot reconstitui ținând cont de părintele p la momentul selectării. Astfel, muchiile vor fi

(5, 2) - deoarece p[2] = 5
(1, 3) - deoarece p[3] = 1
(3, 4) - deoarece p[4] = 3
(6, 5) - deoarece p[5] = 6
(3, 6) - deoarece p[6] = 3

Obținem arborele următor.