>> # Algoritmi de grafuri

Conţinut

- Arbori de acoperire minimi
- Algoritmul lui Kruskal
- Drumuri minime de sursă unică
- Algoritmul lui Dijkstra
- Drumuri minime între toate perechile de vârfuri
- Algoritmul lui Floyd-Warshall

Referinţe

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest.** *Introducere în algoritmi: cap 24, cap. 26.2, cap. 35*, Editura Computer Libris Agora, 2000 (şi ediţiile ulterioare)
- **R. Ceterchi.** *Materiale de curs*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Tema 10*

Arbori de acoperire minimi (Minimum spanning trees)

Fie dat un graf neorientat, conex $G = (V, E)$ şi pentru fiecare muchie $(u, v) \in E$ costul $w(u, v)$. Dorim să găsim submulţimea aciclică $T \in E$ care conectează toate vârfurile şi al cărei cost total, $w(T) = \sum_{(u,v) \in T} w(u, v)$, este minimizat. Mulţimea T formează un arbore, pe care îl numim *arbore de acoperire*.

Algoritmul lui Kruskal

Algoritmul lui Kruskal consideră iniţial vârfurile din graf ca fiind o pădure. La fiecare pas se adaugă mulţimii T o muchie de cost minim din graf care uneşte doi arbori distincţi din pădure. Pădurea (mulţimea de arbori) va fi reţinută în vectorul P. Acesta poate conţine elemente de un tip ce reprezintă un arbore, dar pentru a determina T este suficient să reţinem doar o mulţime de noduri pentru fiecare arbore.

►► KRUSKAL(G, w)

```
1.  Creează un vector P de dimensiune |V[G]| vid
2.  i ← 1
3.  for v ∈ V[G] do
4.    P[i] ← CREEAZĂ–MULŢIME(v)
5.    i ← i+1
6.  endfor
7.  Creează un vector T de dimensiune |V[G]| vid
8.  i ← 1
9.  Sortează muchiile din E[G] în ordinea crescătoare a costurilor
10. for (u,v) ∈ E[G] do
11.   if GĂSEŞTE–MULŢIME(P, u) ≠ GĂSEŞTE–MULŢIME(P, v) then
12.     T[i] ← {(u,v)}
13.     i ← i+1
14.     REUNIUNE(P, u, v)
15.   endif
16. endfor
17. return T
```

Liniile 1-6 iniţializează pădurea P cu $|V|$ mulţimi (vectori), fiecare formată din câte un vârf din graf. Pentru simplitate, dar fără a face economie de spaţiu, putem aloca acestor $|V|$ vectori $|V|$ poziţii în memorie, deoarece acesta este numărul maxim de vârfuri pe care îl pot conţine. Ştim în plus că o mulţime va avea nevoie de întreg spaţiul. Funcţia CREEAZĂ-MULŢIME(x) construieşte un vector A de dimensiune $|V|$ ce conţine un singur element, pe vârful x.

*Ca urmare a faptului că muchiile sunt sortate anterior, ele vor fi considerate în bucla **FOR** în ordinea crescătoare a costului.*

Liniile 7-8 iniţializează mulţimea T (ce va conţine muchiile arborelui de acoperire) cu mulţimea vidă. Muchiile din E sunt sortate crescător după cost în linia 9. În liniile 10-16 se verifică pentru fiecare muchie (u, v) dacă vârfurile sale terminale, u şi v, aparţin aceleiaşi mulţimi (linia 11). În caz afirmativ adăugarea muchiei (u, v) în mulţime (care ne amintim că reprezintă nodurile unui arbore) ar produce un ciclu. În schimb, dacă nu aparţin aceleiaşi mulţimi, atunci se adaugă muchia (u, v) în mulţimea T şi se combină cele două mulţimi corespunzătoare vârfurilor u, respectiv v (liniile 12-14).

►► CREEAZĂ–MULŢIME(x)

```
Creează un vector A de dimensiune |V[G]| vid
A[1] ← x
return A
```

Funcţia GĂSEŞTE-MULŢIME(P,x) returnează un element reprezentativ din mulţimea care îl conţine pe vârful x. Vom presupune că returnează primul element din mulţimea ce îl conţine pe x. Daca am opta pentru implementarea arborilor din pădure prin tipuri structurate această funcţie ar returna rădăcina arborelui. În acest mod, în algoritmul lui Kruskal, se poate efectua verificarea GĂSEŞTE-MULŢIME(P,u) $\neq$ GĂSEŞTE-MULŢIME(P,v) pentru a determina dacă două vârfuri u,v aparţin aceleiaşi mulţimi, deoarece apelurile ar returna acelaşi element.

►► GĂSEŞTE–MULŢIME(P,x)

```
for i ← 1 to |V[G]| do
  if x ∈ P[i] then
    return P[i][1]
  endif
endfor
return NIL
```

Procedura REUNIUNE(P, x, y) combină două mulţimi. Se identifică întâi care sunt cele două mulţimi ce trebuie reunite (liniile 1-10). Apoi, se realizează combinarea lor (elementele din $P[k]$ sunt introduse în mulţimea $P[j]$ (liniile 11-16). În final, a doua mulţime trebuie ştearsă.

►► REUNIUNE(P, x, y)

```
1.  j ← 0
2.  k ← 0
3.  for i ← 1 to |V[G]| do
4.    if u ∈ P[i] then
5.      j ← i
6.    endif
7.    if v ∈ P[i] then
8.      k ← i
9.    endif
10. endfor
11. f ← prima poziţie liberă din P[j]
12. l ← ultima poziţie ocupată din P[k]
13. for i ← 1 to l do
14.   P[j][f] ← P[k][i]
15.   f ← f+1
16. endfor
17. P[k] ← ∅
```

Drumuri minime de sursă unică

Fie dat un graf orientat $G = (V, E)$ şi pentru fiecare arc $(u, v) \in E$ costul $w(u, v)$ nenegativ. Dorim să găsim cel mai scurt drum (de lungime minimă) între două vârfuri din graf. Costul unui drum $p = \langle v_0, v_1, \ldots v_k \rangle$ este suma costurilor corespunzătoare muchiilor componente: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Costul unui drum minim (costul optim) între două vârfuri u şi v se defineşte prin:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{dacă există drum de la } u \text{ la } v \\ \infty, & \text{altfel.} \end{cases}$$

Atunci, un drum minim de la u la v este orice drum p cu proprietatea că $w(p) = \delta(u, v)$.

Problema drumurilor minime de sursă unică cere găsirea pentru un vârf sursă $s \in V[G]$ dat a câte un drum minim de la s la fiecare vârf $v \in V$.

Algoritmul lui Dijkstra

Pentru fiecare vârf $v \in V[G]$ se reţine un predecesor $\pi[v]$ care poate fi fie un alt vârf, fie **NIL**. Vectorul π va fi determinat astfel încât pentru orice vârf v lanţul de predecesori care începe cu v să corespundă unei traversări în ordine inversă a unui drum minim de la s la v. Pentru afişarea unui drum minim de la s la v se foloseşte apelul Afişează-Drum(G,s,v) (procedura a fost dată în laboratorul anterior).

Pentru fiecare vârf $v \in V[G]$ se păstrează un atribut $d[v]$ ce reprezintă o margine superioară a costului unui drum minim de la sursa s la vârful v. Acest atribut este o *estimare a drumului minim*. Procedura următoare realizează iniţializarea vectorilor d şi π.

►► Iniţializează-Sursă-Unică(G, s)

```
for v ∈ V[G] do
   d[v]  ← ∞
   π[v]  ← NIL
endfor
d[s]  ← 0
```

Procedura de relaxare a unei muchii (u, v) verifică dacă se poate îmbunătăţi prin utilizarea vârfului u drumul minim la v (determinat până în momentul respectiv), caz în care se actualizează $d[v]$ şi $\pi[v]$.

►► Relaxează(u, v)

```
if d[v] > d[u] + w(u,v) then
   d[v]  ←  d[u] + w(u,v)
   π[v]  ← u
endif
```

Algoritmul lui Dijkstra menţine o mulţime S de vârfuri pentru care costurile finale corespunzătoare drumurilor minime de la sursa s au fost deja determinate: $\forall v \in S, d[v] = \delta(u, v)$. La fiecare pas, se selectează un vârf $u \in V \setminus S$ pentru care estimarea drumului minim ($d[v]$) este minimă, inserează u în mulţimea S şi verifică prin procedeul de relaxare dacă drumurile spre vârfurile adiacente cu u (vârfurile v din lista de adiacenţă a lui u, notată L_u) pot fi îmbunătăţite prin utilizarea vârfului u.

►► Dijkstra(G)

```
Iniţializează-Sursă-Unică(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅ do
   u ← Extrage-Min(Q)
   S ← S ∪ {u}
   for v ∈ L_u do
      Relaxează(u, v)
   endfor
endwhile
```

Drumuri minime între toate perechile de vârfuri

Problema drumurilor minime pentru surse şi destinaţii multiple presupune identificarea a câte un drum minim pentru fiecare pereche de vârfuri i şi j într-un graf orientat $G = (V, E)$.

Algoritmul lui Floyd-Warshall

Vom presupune că pot exista arce cu cost negativ, dar că nu pot exista cicluri cu cost negativ în graful G.

Se numeşte vârf *intermediar* al unui drum elementar $p = \langle v_1, v_2, \ldots, v_l \rangle$ oricare vârf diferit de v_1 şi v_l.

Algoritmul Floyd-Warshall se bazează pe următoarea observaţie. Fie $V = \{1, 2, \ldots, n\}$ mulţimea vârfurilor lui G. Considerăm submulţimea $\{1, 2, \ldots, k\}$ pentru un anumit vârf k. Pentru oricare pereche de vârfuri $i, j \in V$, considerăm toate drumurile de la i la j ale căror vârfuri intermediare fac parte din mulţimea $\{1, 2, \ldots, k\}$. Fie p drumul de cost minim dintre aceste drumuri. (Drumul p este elementar deoarece presupunem că G nu conţine cicluri de cost negativ.) Algoritmul Floyd-Warshall exploatează o relaţie între drumul p şi drumul minim de la i la j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$. Relaţia depinde de statutul lui k: acesta poate sau nu să fie vârf intermediar al lui p.

Amintim că un drum este elementar dacă el conţine doar vârfuri distincte.

- Dacă vârful k nu este un vârf intermediar la drumului p, atunci toate vârfurile intermediare ale drumului p fac parte din mulţimea $\{1, 2, \ldots, k-1\}$. Ca urmare, un drum minim de la vârful i la vârful j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$ este, de asemenea, un drum minim de la i la j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$.
- Dacă vârful k este un vârf intermediar al drumului p, atunci vom împărţi p în $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$. Drumul p_1 este drumul minim de la i la k cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$. De fapt, vârful k nu este un vârf intermediar al drumului p_1, deci p_1 este un drum minim de la i la k cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$. Similar, p_2 este un drum minim de la k la j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$.

Definim o estimare recursivă a estimărilor drumului minim. Fie $d_{ij}^{(k)}$ costul drumului minim de la vârful i la vârful j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$. Când $k = 0$, un drum de la vârful i la vârful j, cu niciun vârf intermediar al cărui număr este mai mare decât 0, nu are niciun intermediar. Deci, acest drum conţine cel mult un arc, prin urmare avem $d_{ij}^{(0)} = w_{ij}$. O definiţie recursivă este dată de:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{,dacă } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{, dacă } k \geq 1. \end{cases}$$

Matricea $D^{(n)} = \left(d_{ij}^{(n)}\right)$ conţine răspunsul final, $d_{ij}^{(n)} = \delta(i, j)$ pentru orice $i, j \in V$, deoarece toate vârfurile intermediare se află în mulţimea $\{1, 2, \ldots, n\}$.

Obs. Valorile $d_{ij}^{(k)}$ trebuie determinate în ordine crescătoare a lui k.

Procedura următoare construieşte matricea finală $D^{(n)}$. Ea primeşte ca parametrii de intrare graful G reprezentat prin matrice de adiacenţă, pe care o notăm a.

▶▶ Floyd–Warshall(G)

```
1. D^(0) ← a
2. for k ← 1 to n do
3.    for i ← 1 to n do
4.       for j ← 1 to n do
5.          D_ij^(k) ← Minim(D_ij^(k-1), D_ik^(k-1) + D^(k-1)_kj)
6.       endfor
7.    endfor
8. endfor
9. return D^(n)
```

La implementare, putem utiliza o singură structuri D în locul celor $n+1$ structuri D^k. Astfel obţinem procedura de mai jos.

▶▶ Floyd–Warshall-2(G)

```
1. D ← a
2. for k ← 1 to n do
3.    for i ← 1 to n do
4.       for j ← 1 to n do
5.          D[i][j] ← Minim(D[i][j], D[i][k]+D[k][j])
6.       endfor
7.    endfor
8. endfor
9. return D
```

Atenţie, linia 1 (în ambele variante), reprezintă copierea matricei de adiacenţă în matricea nou creată D (de aceeaşi dimensiune, $n \times n$). Dacă aţi face atribuirea directă $D = a$, cum a este un pointer, algoritmul ar suprascrie datele din a.

Pentru a putea reconstrui drumurile de cost minim între toate perechile de vârfuri i şi j utilizăm o matrice predecesor $\pi^{(n)}$, ce va fi construită recursiv, conform următoarei recurenţe.

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{, dacă } i = j \text{ sau } a_{ij} = \infty \\ i & \text{, dacă } i \neq j sau a_{ij} < \infty \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{, dacă } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{, dacă } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Modificăm procedura Floyd-Warshall astfel încât să construim şi matricea $\pi = \pi^{(n)}$.

►► Floyd–Warshall-3(G)

```
D ← a
for i ← 1 to n do
    π[i][j] ← NIL
    for j ← 1 to n do
        if i ≠ j or a[i][j] < ∞ then
            π[i][j] ← i
        endif
    endfor
endfor
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if D[i][j] > D[i][k] + D[k][j] then
                D[i][j] ← D[i][k]+D[k][j]
                π[i][j] ← π[k][j]
            endif
        endfor
    endfor
endfor
return D
```

Pentru afişarea drumului de cost minim între două vârfuri i şi j se poate utiliza următorul algoritm.

►► Afişează-Drum-Minim(π, i, j)

```
if i = j then
    Afişează i
else if π[i][j] = NIL then
    // Nu există drum de la i la j
else
    Afişează–Drum-Minim(π, i, π[i][j])
    Afişează j
endif
```

PROBLEME

1. **(5p)** Se dau n oraşe, care pot fi conectate sau nu printr-un drum direct. În caz afirmativ, se dă şi costul utilizării acelui drum. Să se spună care este costul minim pentru a ajunge dintr-un oraş k_1 într-un oraş k_2 (eventual trecând prin alte oraşe intermediare). În acest scop se va utiliza algoritmul lui Dijkstra studiat.
2. **(5p)** Se dau n oraşe şi distanţele dintre oricare două. Se doreşte conectarea acestora cu fire de telefon astfel încât sa fie utilizată o lungime cât mai mică de cablu, dar să poată fi iniţiată o convorbire între oricare două oraşe. Se va folosi algoritmul lui Kruskal pentru determinarea arborelui parţial de cost minim. Datele de intrare se vor citi dintr-un fişier `input.txt`.
3. **(5p)** Se dau n oraşe, care pot fi conectate sau nu printr-un drum direct. În caz afirmativ, se dă şi costul utilizării acelui drum. Să se spună care este costul minim pentru a ajunge din orice oraş în oricare alt oraş, folosind algoritmul Floyd-Warshall.

■ Termen de predare: **Săptămâna 14 (14–18 ianuarie 2013) inclusiv.**

■ Detalii: **Studenţii pot obţine un maxim de 15 puncte. Problemele 1-2 sunt obligatorii. Problema 3 este suplimentară.**