

Algoritmica grafurilor

Curs 2

BFS - Breadth First Search - Parcurgere în lățime

algoritmul BFS

dându-se un graf $G = (V, E)$ și un vârf **sursă** s , algoritmul de parcurgere în lățime explorează sistematic muchiile lui G pentru a descoperi fiecare vârf **accesibil din s**

→ prima dată orizontal apoi vertical

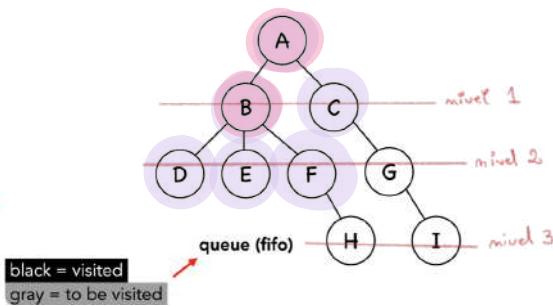
↪ pt grafuri orientate & neorientate

⇒ BFS construiește un arbore cu radă în s , arbore ce conține toate vârfurile accesibile

→ pt fiecare vârf v accesibil din s , lantul simplu din arbore reprez. lantul minim dintr-un $s \rightarrow v$

căutare în lățime pt că descoperă **TOATE** vîrfurile accesibile la distanță k da și sănătatea la distanță $k+1$

```
BFS( $G, s$ )
for fiecare vârf  $u \in G, V - \{s\}$  do // initializare
    u.color = alb
    u.d = ∞
    u.π = NIL
end for
s.color = gri // mod sursă
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s) → coadă (FIFO)
while Q ≠ ∅ do → ești timp coada nu e goală
    u = Dequeue(Q) → pop coadă = u
    for fiecare  $v \in G.Adj[u]$  do → pt fiecare nod incident cu  $u$ 
        if v.color == alb then → dacă e alb (nevizitat)
            v.color = gri → frontieră
            v.d = u.d + 1 → distanță de la  $u$  (făcându-i parcurgerea vecinii) + 1
            v.π = u → predecesorul e  $u$ 
            Enqueue(Q, v)
        end if
    end for
    u.color = negru →  $u$  vizitat
end while
```



Coadă

$s \rightarrow A$

$[A] \rightarrow [B, C] \rightarrow [C, D, E, F] \rightarrow [D, E, F, G] \rightarrow [E, F, G] \rightarrow$

$[F, G] \rightarrow [G, H] \rightarrow [H, I] \rightarrow [I] \rightarrow \emptyset$

Complexitate: $\Theta(V + E)$

\uparrow \uparrow
vîrf muchie

DFS - Depth First Search - Parcursuri în adâncime

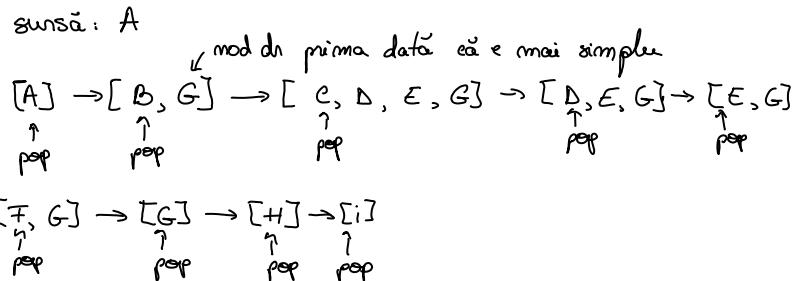
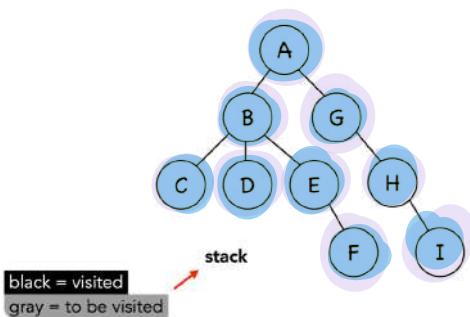
- după ce au fost parcursse TOATE muchiile dintr-un vj. v, alg. se întoarce la vj. muchie care a dus în v și continuă explorarea
- se repetă pînă ce au fost explorate toate vîrfurile **accesibile** din sensă, dacă nămărește vj. NEXPLORE
- DFS alege unul dintre ele ca să sensă și continuă

```
DFS(G)
for fiecare vîrf u ∈ G.V do → initializare
    u.color = alb
    u.π = NIL
end for
time = 0
for fiecare u ∈ G.V do → pt fiecare nod
    if u.color == alb then → dacă e neparcurs
        DFS_VISIT(G,u)
    end if
end for
```

```
DFS_VISIT(G, u)
time = time + 1 → timp overall de parcursere
u.d = time → timpul în care a fost descoperit vj.
u.color = gri
for fiecare v ∈ G.Adj[u] do
    if v.color == alb then
        v.π = u
        DFS_VISIT(G,v)
    end if
end for
u.color = negru
time = time + 1
u.f = time
```

- algoritmul colorează vîfurile pe parcursul căutării similar cu BFS, prin culoare se indică starea nodului
- pe lângă stare DFS marchează și timpul când a fost descoperit vîrful și timpul când a fost explorat complet arborele din vîrful descoperit
 - pentru a măsura performanța algoritmului
 - pentru a descoperi structura grafului
- $u.d$ marchează timpul când a fost descoperit vîrful u
- $u.f$ marchează timpul când a fost explorat vîrful u
- starea unui vîrf: alb - $u.d$ - gri - $u.f$ - negru

Stivă → Last in First Out



Complexitate: $\Theta(V+E)$

Graf tare conex. Graf slab conex

- **graf tare conex** = un graf orientat este tare conex dacă între oricare două vîrfuri există un drum
- **graf slab conex** = între oricare două drumeuri u și v ale grafului există un drum de la u la v sau de la v la u, nu există ambele drumeuri

→ pt dt componentele tare conex dintr-un graf orientat: **Algoritm Kosaraju - Sharir**

→ cel mai scurt lanț / drum, graf neorientat, (ne)orientat: **Algoritmul lui Moore**

Algoritmul lui Dijkstra

- algoritmul lui Dijkstra rezolvă problema drumului minim pentru un graf orientat ponderat pozitiv! $G = (V, E)$ în care $w(u, v) \geq 0, \{u, v\} \in E$
- algoritmul menține un set S de vârfuri pentru care drumul minim de la sursa s a fost determinat
- în implementarea prezentată se folosește o coadă cu priorități pentru vârfuri, cheia fiind $v.d$

DIJKSTRA(G, u)

1. $S := \{u\}, T := V \setminus S, l(u) := 0$
2. **for** fiecare $v \in V, v \neq u$ **do**
3. $l(v) := \infty$
4. $x := u$
5. **while** $T \neq \emptyset$ **do**
6. **for** fiecare $v \in N(x) \cap T$ **do**
7. **if** $l(v) > l(x) + w(x, v)$ **then**
8. $l(v) := l(x) + w(x, v)$
9. $p(v) := x$?
10. fie $x \in T: l(x) = \min_{y \in T} l(y)$
11. $S := S \cup \{x\}, T := T \setminus \{x\}$
12. **return** l, p

; S = sursă, T = toate nod. fără s , $l(u) \rightarrow$ costul drum
 → inițializăm costul fiecărui nod cu ∞
 → căt timp există noduri nevizitate
 → pt fiecare nod \in vecini(x) \wedge totalitate nod nevizitate
 → dacă $\overset{\text{mod vecin}}{l(v)} > l(x) + \text{pondere dinspre } (x, v)$
 $\Rightarrow l(v) = l(x) + \text{pondere}(x, v)$

Complexitate: $O(|E| + |V| \log |V|)$

1. **Pornim de la un punct:** Alege un nod de început. Să spunem că începi din orașul tău.
2. **Costuri inițiale:** Atribuie fiecărui nod din rețea un cost inițial. Nodului de început îi atribuie costul 0 (deoarece ești deja acolo) și tuturor celorlalte noduri le atribuie un cost infinit (pentru că nu știi încă cât de mult te va costa să ajungi acolo).
3. **Explorează vecinii:** Uită-te la toți vecinii nodului curent (adică nodurile care sunt direct conectate la nodul de început printr-o cale). Actualizează costurile pentru a ajunge la acești vecini dacă găsești o cale mai ieftină decât cea cunoscută până acum.
4. **Marchează nodul vizitat:** După ce ai actualizat costurile vecinilor nodului curent, marchează nodul curent ca fiind vizitat. Un nod vizitat nu va mai fi verificat din nou.
5. **Alege următorul nod:** Alege nodul nevizitat cu cel mai mic cost total și repetă pașii de mai sus.
6. **Repetă:** Continuă acest proces până când ai vizitat toate nodurile sau ai găsit cel mai scurt drum către destinația dorită.

Algoritmul lui Bellman - Ford

Algoritmul Bellman-Ford rezolvă problema drumului minim de la un nod sursă s pentru cazul general când avem și ponderi negative.

Bellman_Ford(G, w, s)

```

1: INITIALIZARE_S( $G, s$ )
2: for  $i = 1$  la  $|V| - 1$  do
3:   for fiecare arc  $\{u, v\} \in E$  do
4:     RELAX( $u, v, w$ )
5: for fiecare arc  $\{u, v\} \in E$  do
6:   if  $v.d > u.d + w(u, v)$  then
7:     return FALSE
8: return TRUE

```

- algoritmul rulează în $O(VE)$ timp
- pasul de inițializare (linia 1) durează $\Theta(V)$
- parcurgerea din liniile 2-4 durează $O(VE)$
- bucla for din liniile 5-7 durează $O(E)$

! dacă există eiluri negative \Rightarrow NU EXISTĂ DRUM CEL MAI SCURT

→ iterăm de maximum $|n|$ de moduri $| - 1$ $\xrightarrow{\text{dacă nu există eiluri negative}}$ vom avea drum
 → la fiecare iteratie verificăm toate modurile

Sortare topologică

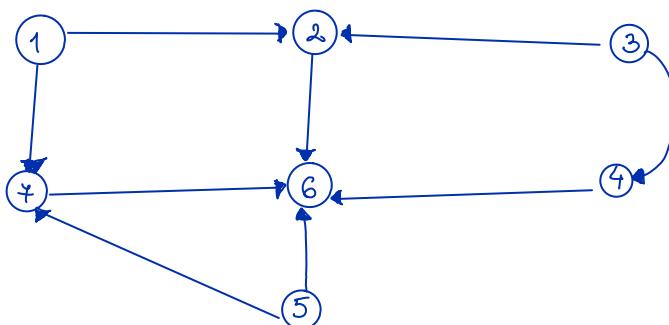
- folosind algoritmul DFS se poate sorta topologic un graf orientat fără circuite (DAG - directed acyclic graph)
- realizează o aranjare liniară a vârfurilor unui graf în funcție de arcele grafului
- Sortare topologică fie un graf orientat aciclic $G = (V, E)$, sortarea topologică reprezintă ordonarea vârfurilor astfel încât dacă G conține arcul (u, v) atunci u apare înaintea lui v în înșiruire.

sortare_topologică(G)

```

1: apel DFS( $G$ ) pentru a determina timpii  $v.f, v \in V$ 
2: sortare descrescătoare în funcție de timpul de finalizare (când fiecare vârf e terminat e inserat
   într-o listă înlănțuită)
3: return lista înlănțuită de vârfuri

```



Suport curs algoritmica grafurilor

III. Parcurgeri, drumuri de cost minim în grafuri orientate și ponderate

3 DFS

```
DFS( $G$ )
  for fiecare vîrf  $u \in G.V$  do
     $u.\text{color} = \text{alb}$ 
     $u.\pi = \text{NIL}$ 
     $\text{time} = 0$ 
    for fiecare  $u \in G.V$  do
      if  $u.\text{color} == \text{alb}$  then
        DFS_VISIT( $G, u$ )
```

```
DFS_VISIT( $G, u$ )
   $time = time + 1$ 
   $u.d = time$ 
   $u.\text{color} = \text{gri}$ 
  for fiecare  $v \in G.\text{Adj}[u]$  do
    if  $v.\text{color} == \text{alb}$  then
       $v.\pi = u$ 
      DFS_VISIT( $G, v$ )
     $v.\text{color} = \text{negră}$ 
     $time = time + 1$ 
     $u.f = time$ 
```

Teorema 3.1 (Teorema parantezelor). În orice căutare în adâncime a unui graf $G = (V, E)$ (orientat sau neorientat), pentru orice două vîrfuri u și v , exact una din următoarele trei afirmații este adevarată:

- intervalele $[u.d, u.f]$ și $[v.d, v.f]$ sunt total disjuncte
- intervalul $[u.d, u.f]$ este conținut în întregime în intervalul $[v.d, v.f]$ iar u este descendenter al lui v în arborele de adâncime
- intervalul $[v.d, v.f]$ este conținut în întregime în intervalul $[u.d, u.f]$ iar v este descendenter al lui u în arborele de adâncime

Teorema 3.2 (Teorema drumului alb). Într-o căutare de adâncime a unui graf $G = (V, E)$ (orientat sau neorientat), vîrful v este descendenter al vîrfului u dacă și numai dacă la momentul $u.d$, când căutarea descoperă vîrful u , vîrful v este accesibil din u printr-un drum format în întregime din vîrfuri albe.

Clasificarea muchiilor

- pentru un graf $G = (V, E)$, fie $(u, v) \in E$, în funcție de timp tipul arcelor pentru DFS:
 - $u.d$ marchează timpul când a fost descoperit vîrful u
 - $u.f$ marchează timpul când a fost explorat vîrful u

III. Drumuri în grafuri

3 DFS

tip arc	d	f
t (tree)	$u.d < v.d$	$u.f > v.f$
b (back)	$u.d > v.d$	$u.f < v.f$
f (forward)	$u.d < v.d$	$u.f > v.f$
c (cross)	$u.d > v.d$	$u.f > v.f$

3.1 Sortare topologică

- folosind algoritmul DFS se poate sorta topologic un graf orientat fără circuite (DAG - directed acyclic graph)
- realizează o aranjare liniară a vârfurilor unui graf în funcție de arcele grafului
- Sortare topologică fie un graf orientat aciclic $G = (V, E)$, sortarea topologică reprezintă ordonarea vârfurilor astfel încât dacă G conține arcul (u, v) atunci u apare înaintea lui v în înșiruire.
- multe aplicații folosesc grafuri orientate fără circuite pentru a indica precedența între evenimente
- un set de acțiuni ce trebuie îndeplinite într-o anumită ordine
- unele sarcini trebuie executate înainte ca alte acțiuni să înceapă
- în ce ordine trebuie executate sarcinile?**
- problema poate fi rezolvată reprezentând sarcinile ca vârfuri într-un graf
- un arc (u, v) indică precedență între activități, activitatea u înaintea activității v
- sortând topologic graful se arată ordinea efectuării acțiunilor

sortare_topologică(G)

- apel DFS(G) pentru a determina timpii $v.f, v \in V$
- sortare descreșătoare în funcție de timpul de finalizare (când fiecare vârf e terminat e inserat într-o listă înlănțuită)
- return** lista înlănțuită de vârfuri

- un graf se poate sorta topologic în timpul $\Theta(V + E)$
 - DFS durează $\Theta(V + E)$
 - pentru a insera un vârf $v \in V$ în listă e nevoie de $O(1)$ timp

Lema 3.3. *un graf orientat G este aciclic dacă și numai dacă DFS aplicat pe el nu găsește arce pentru care $u.d > v.d$ și $u.f < v.f$.*

Teorema 3.4. *procedura sortare_topologică(G) produce o sortare topologică a unui graf orientat aciclic primit ca și parametru.*

3.2 Componente tare conexe

componente_tare_conexe(G)

- 1: apel DFS(G) pentru a determina timpii $v.f, v \in V$
- 2: determină G^T
- 3: apel DFS(G^T) dar în bucla principală a DFS nodurile sunt sortate descrescător după $v.f$
- 4: fiecare arbore din pădurea găsită de DFS în pasul 3 este o componentă tare conexă

- pentru $G = (V, E), G^T = (V, E^T)$ unde $E^T = \{(u, v) : (v, u) \in E\}$
- pentru reprezentarea sub formă de listă de adiacență, pentru a determină G^T e nevoie de $O(V + E)$ timp
- fie G reprezentat ca listă de adiacență, pentru procedura componente_tare_conexe(G) complexitatea în timp este
 - $\Theta(V + E)$

Lema 3.5. fie C și C' două componente tare conexe din graful $G = (V, E)$. $u, v \in C, u', v' \in C'$ și G conține un drum $u \rightsquigarrow u'$. Atunci G nu poate avea un drum $v' \rightsquigarrow v$.

- graful format din componente tare conexe este un graf orientat aciclic
- fie $U \subseteq V$, putem defini $d(U) = \min_{u \in U} \{u.d\}$ și $f(U) = \max_{u \in U} \{u.f\}$
 - $d(U)$ reprezintă timpul pentru primul vârf descoperit de DFS din U
 - $f(U)$ reprezintă timpul pentru ultimul vârf prelucrat de DFS din U

Lema 3.6. fie C și C' două componente tare conexe distincte din graful orientat $G = (V, E)$. Dacă există un arc $(u, v) \in E$, unde $u \in C$ și $v \in C'$ atunci $f(C) > f(C')$.

Corolar 3.6.1. fie C și C' două componente tare conexe distincte din graful orientat $G = (V, E)$. Dacă există un arc $(u, v) \in E^T$, unde $u \in C$ și $v \in C'$ atunci $f(C) < f(C')$.

Teorema 3.7. procedura componente_tare_conexe(G) găsește corect componentele tare conexe din graful orientat G .

3.3 Drum de lungime minimă - sursă unică

Problema drumului de cost minim într-un graf pentru un singur vârf sursă se poate defini în felul următor: fie un graf $G = (V, E)$, se vrea să se găsească un drum de la un vârf **sursă** $s \in V$ la fiecare vârf din graf $v \in V$. Algoritmii care rezolvă problema drumului de cost minim de la un vârf sursă pot fi folosiți pentru a rezolva și alte tipuri de probleme, printre care și variante ale problemei drumului de cost minim:

- **Drum de cost minim pentru o singură destinație:** se caută drumul de cost minim de la toate vârfurile din graf $v \in V$ către un singur vârf **destinație** t .
- **Drum minim între o pereche de vârfuri:** se caută drumul de cost minim între vârfurile u și v (pentru u și v date). Dacă se rezolvă problema drumului de cost minim de la un vârf sursă către toate vârfurile din graf se rezolvă și această variantă a problemei.
- **Drum de cost minim între toate perechile de vârfuri:** se caută drumul de cost minim între toate perechile de vârfuri u și v din graf.

III. Drumuri în grafuri

3 DFS

Structura optimă a unui drum de cost minim

În general algoritmii care rezolvă problema drumului de cost minim se bazează pe proprietatea drumului minim: un drum de cost minim între două vârfuri conține drumuri de cost minim.

Lema 3.8 ((sub)drumuri de cost minim într-un drum de cost minim).

Pentru graf orientat și ponderat $G = (V, E)$ cu funcția de pondere $w : E \rightarrow \mathbb{R}$, fie $p = \langle v_0, v_1, \dots, v_k \rangle$ un drum de cost minim de la vârful v_0 la vârful v_k și pentru oricare i și j , $0 \leq i \leq j \leq k$, fie $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ un drum din p (subdrum) între v_i și v_j . Atunci p_{ij} este un drum de cost minim între v_i și v_j .

Demonstrație.

Drumul p poate fi descompus în $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, în acest caz costul drumului este $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$ (suma ponderilor arcelor ce formează drumul).

Dacă se presupune că există drumul p'_{ij} de la v_i la v_j cu costul $w(p'_{ij}) < w(p_{ij})$, atunci $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ este un drum de la v_0 la v_k de cost $w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$ ceea ce contrazice presupunerea că p este un drum minim de la v_0 la v_k . \square

Arce cu cost negativ

Unele instanțe ale problemei de cost minim permit ca ponderea unui arc să poată lua valori negative. Dacă un graf $G = (V, E)$ nu conține circuite de pondere negativă accesibile din vârful sursă s , pentru toate $v \in V$, ponderea drumului de cost minim $\delta(s, v)$ are o valoare finită (chiar dacă este negativă). În schimb dacă $G = (V, E)$ conține un circuit de pondere negativă accesibil din vârful s ponderea drumului de cost minim nu este bine definită. Nu există un drum de la s către un vârf din $G = (V, E)$ prin circuit care să fie drum minim (se poate găsi un drum mai bun dacă se urmărește drumul și se traversează circuitul negativ). Dacă există un circuit negativ pe un drum de la s la v definim $\delta(s, v) = -\infty$.

Circuite

Un drum de cost minim nu poate conține un circuit.

Dacă $G = (V, E)$ conține pe drumul de la s la v un circuit de pondere negativă tot timpul s-ar găsi un drum mai bun.

În cazul unui circuit de pondere pozitivă dacă se elimină circuitul din drum se obține un drum între vâfurile s și v cu un cost mai mic. Altfel, fie $p = \langle v_0, v_1, \dots, v_k \rangle$ un drum și $c = \langle v_i, v_{i+1}, \dots, v_j \rangle$ un circuit de pondere pozitivă pe drumul p ($v_i = v_j$ și $w(c) > 0$), atunci drumul $p' = \langle v_0, v_1, \dots, v_i, v_{j+1}, v_{j+2}, \dots, v_k \rangle$ are costul $w(p') = w(p) - w(c) < w(p)$ și p' nu poate fi drumul minim de la v_0 la v_k .

Mai rămân circuite de pondere 0. Se poate elimina un circuit de pondere 0 dintr-un drum oarecare pentru a obține un drum cu același cost. Se poate elimina orice circuit de pondere 0 de pe un drum de la vârful sursă s la vârful destinație v . Dacă există un drum minim de la s la v care conține un circuit de pondere 0, există un drum de cost minim de la s la v fără acest circuit.

Un drum de cost minim cu conține circuite. Orice drum aciclic din graful $G = (V, E)$ conține cel mult $|V|$ vârfuri distincte și cel mult $|V| - 1$ arce.

Reprezentarea unui drum de cost minim

Pentru un graf dat $G = (V, E)$, pentru fiecare vârf $v \in V$ se menține un **predecesor** $v.\pi$ care este un vârf sau *NIL*. Algoritmii de drum minim vor modifica atributul π astfel încât lanțul predecesorilor de la vârful v merge înapoi pe drumul minim de la s la v . Astfel, fie un vârf v pentru care

$v.\pi \neq NIL$, procedura AFISARE_DRUM(G, s, v) va afișa drumul de cost minim de la s la v .

```
AFISARE_DRUM( $G, s, v$ )
1: if  $v == s$  then
2:   print  $s$ 
3: else if  $v.\pi == NIL$  then
4:   print "nu este drum de la"  $s$  "la"  $v$ 
5: else
6:   AFISARE_DRUM( $G, s, v.\pi$ )
7:   print  $v$ 
```

Se poate defini graful predecesor $G_\pi = (V_\pi, E_\pi)$ indus de valorile atributului π . Setul V_π este setul vârfurilor lui G pentru care atributul π este diferit de NIL plus vârful sursă s :

$$V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\}.$$

Setul E_π este setul arcelor induse de valorile lui π pentru vârfurile din V_π :

$$E_\pi = \{(v.\pi, v) \in E \mid v \in V_\pi - \{s\}\}.$$

Pentru valorile lui π găsite de algoritmi, la terminare, G_π este un arbore minim (în sensul sumei ponderii arcelor ce compun drumul de la s la v , unde vârful s este rădăcina arborelui). Fie $G = (V, E)$ un graf ponderat și orientat cu funcția de pondere $w : E \rightarrow \mathbb{R}$ și G nu conține circuite de pondere negativă accesibile din vârful sursă $s \in V$, astfel încât drumurile de cost minim sunt bine definite. Un arbore minim de drumuri cu rădăcina în s este un graf orientat $G' = (V', E')$ unde $V' \subseteq V$ și $E' \subseteq E$ astfel încât:

1. V' este setul vârfurilor accesibile din s ;
2. G' este un arbore cu rădăcina în s ;
3. pentru oricare vârf $v \in V'$ drumul de la s la v în G' este un drum de cost minim de la s la v în G .

Procedura de relaxare

Algoritmii de drum minim folosesc procedura de *relaxare*. Pentru fiecare vârf $v \in V$ se menține un atribut $v.d$ care reprezintă limita superioară a ponderii (costului) drumului de la s la v . $v.d$ este mează costul minim al drumului. Folosind o procedură în $\Theta(V)$ se initializează estimările drumului de cost minim.

```
INITIALIZARE_S( $G, s$ )
1: for  $v \in G.V$  do
2:    $v.d = \inf$ 
3:    $v.\pi = NIL$ 
4:    $s.d = 0$ 
```

Procesul de *relaxare* a arcului (u, v) presupune testarea dacă drumul de cost minim până la v găsit până în acest moment se poate îmbunătăți dacă se trece prin u , dacă da se actualizează $v.d$ și $v.\pi$. Codul de mai jos face acest pas.

```
RELAXARE( $u, v, w$ )
1: if  $v.d > u.d + w(u, v)$  then
```

III. Drumuri în grafuri

3 DFS

- 2: $v.d = u.d + w(u, v)$
- 3: $v.\pi = u.\pi$

Procedura de relaxare modifică estimatorii drumului de cost minim ($v.d$) și predecesorii vârfurilor.

Diferența între algoritmii de drum de cost minim este numărul de apeluri ale procedurii de relaxare, algoritmul lui *Bellman-Ford* apelează procedura de relaxare de $|V| - 1$ ori pentru fiecare arc iar algoritmul lui *Dijkstra* și algoritmul de drum de cost minim pentru grafuri aciclice apelează procedura de relaxare o singură dată pentru fiecare arc.

3.4 Algoritmul Bellman-Ford

Algoritmul Bellman-Ford rezolvă problema drumului minim de la un nod sursă s pentru cazul general când avem și ponderi negative.

Bellman_Ford(G, w, s)

- 1: INITIALIZARE_S(G, s)
- 2: **for** $i = 1$ la $|V| - 1$ **do**
- 3: **for** fiecare arc $\{u, v\} \in E$ **do**
- 4: RELAX(u, v, w)
- 5: **for** fiecare arc $\{u, v\} \in E$ **do**
- 6: **if** $v.d > u.d + w(u, v)$ **then**
- 7: **return** FALSE
- 8: **return** TRUE

- algoritmul rulează în $O(VE)$ timp
- pasul de inițializare (linia 1) durează $\Theta(V)$
- parcurgerea din liniile 2-4 durează $O(VE)$
- bucla for din liniile 5-7 durează $O(E)$

Lema 3.9. fie $G = (V, E)$ un graf ponderat orientat cu nodul sursă s și funcția de pondere $w : E \rightarrow \mathbb{R}$, presupunem că G nu conține circuite de pondere negativă accesibile din vârful s . După $|V| - 1$ iterații ale buclei for din liniile 2-4 a procedurii $\text{Bellman_Ford}(G)$ avem $v.d = \delta(s, v)$ pentru toate vârfurile v accesibile din s .

Corolar 3.9.1. fie $G = (V, E)$ un graf ponderat orientat cu nodul sursă s și funcția de pondere $w : E \rightarrow \mathbb{R}$. Pentru fiecare vârf $v \in V$ există un drum de la s la v dacă și numai dacă procedura $\text{Bellman_Ford}(G)$ se termină cu $v.d < \infty$.

Teorema 3.10. Corectitudine Bellman-Ford. Fie procedura $\text{Bellman_Ford}(G)$ care este rulată pe un graf orientat și ponderat $G = (V, E)$ din nodul sursă s și funcția de pondere $w : E \rightarrow \mathbb{R}$. Dacă G nu conține circuite de pondere negativă accesibile din s algoritmul va întoarce TRUE, $v.d = \delta(s, v) \forall v \in V$ iar graful predecesorilor G_π este un arbore minim cu rădăcina în s . Dacă G conține un circuit de pondere negativă accesibil din s , algoritmul întoarce FALSE.

3.5 Drum de cost minim în grafuri orientate aciclice

drum_minim_dag(G)

- 1: sortate_topologica(G)
- 2: INITIALIZARE_S(G, s)
- 3: **for** fiecare vârf v sortat topologic **do**
- 4: **for** $v \in G.\text{Adj}[u]$ **do**

3 DFS

III. Drumuri în grafuri

5: RELAX(u,v,w)

- timp de rulare
 - sortate topologică: $\Theta(V + E)$
 - INITIALIZARE_S: $\Theta(V)$
 - bucla for (liniile 4-5) relaxează fiecare arc o singură dată
 - timpul total de rulare $\Theta(V + E)$

Teorema 3.11. Corectitudine $\text{drum_minim_dag}(G)$. Dacă un graf orientat, ponderat și aciclic $G = (V, E)$ are ca și sursă vârful s , la terminarea procedurii $\text{drum_minim_dag}(G)$ v.d. $= \delta(s, v) \forall v \in V$ iar graful predecesorilor G_π este un arbore minim.

3.6 Algoritmul lui Dijkstra

- algoritmul lui Dijkstra rezolvă problema drumului minim pentru un graf orientat ponderat $G = (V, E)$ în care $w(u, v) \geq 0, \{u, v\} \in E$
- algoritmul menține un set S de vârfuri pentru care drumul minim de la sursa s a fost determinat
- în implementarea prezentată se folosește o coadă cu priorități pentru vârfuri, cheia fiind $v.d$

Teorema 3.12. Corectitudine Dijkstra. Fie procedura $\text{Dijkstra_queue}(G)$ rulată pe un graf orientat, ponderat $G = (V, E)$ ce nu conține ponderi negative și s vârful sursă. La terminare $u.d = \delta(s, u) \forall u \in V$.

Cât de rapid este algoritmul Dijkstra_queue?

- menține o coadă cu priorități Q prin apelul operațiilor: INSERT (implicit în linia 3), EXTRACT_MIN (linia 5) și DECREASE_KEY (implicit în RELAX, linia 8)
- algoritmul apelează INSERT și EXTRACT_MIN pentru fiecare vârf
- fiecare vârf este adăugat în setul S o singură dată, fiecare arc din $\text{Adj}[u]$ este examinat o singură dată pe liniile 7-8
- numărul total de arce din lista de adiacență este $|E|$, bucla for iterează de $|E|$ ori (liniile 7-8), algoritmul apelează DECREASE_KEY de cel mult $|E|$ ori
- timpul total de rulare depinde de implementarea cozii cu priorități
- dacă vârfurile sunt numerotate de la 1 la $|V|$
 - $v.d$ e stocat pe poziția v
 - fiecare operație INSERT și DECREASE_KEY necesită $O(1)$ timp iar operația EXTRACT_MIN necesită $O(V)$ timp
 - pentru un timp total $O(V^2 + E) = O(V^2)$
- dacă graful este suficient de rar, coada se poate implementa ca și *binary min-heap*
 - fiecare operație EXTRACT_MIN durează $O(\lg V)$, există $|V|$ astfel de operații
 - timpul necesar construirii *binary min-heap* este $O(V)$
 - fiecare operație DECREASE_KEY necesită $O(\lg V)$ timp, există $|E|$ astfel de operații

- timpul total de rulare este $O((V + E) \lg V)$, dacă toate vârfurile sunt accesibile din sursă
timpul este $O(E \lg V)$
- se poate obține un timp de rulare $O(V \lg V + E)$ dacă coada cu priorități este implementată cu un *heap Fibonacci*

3.7 Referințe

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press.
2. Geir Agnarsson and Raymond Greenlaw. 2006. Graph Theory: Modeling, Applications, and Algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
3. Mark Newman. 2010. Networks: An Introduction. Oxford University Press, Inc., New York, NY, USA.
4. Cristian A. Giumale. 2004. Introducere în analiza algoritmilor, teorie și aplicație. Polirom.
5. cursuri Teoria grafurilor: Zoltán Kása, Mircea Marin, Toadere Teodor.

Suport curs algoritmica grafurilor

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

4.1 Programare Liniara

Problema generală: fie o matrice A de dimensiune $m \times n$, un vector b de dimensiune m și un vector c de dimensiune n . Trebuie să se găsească un vector x de n elemente care maximizează funcția obiectiv

$$\sum_{i=1}^n c_i x_i$$

și satisfacă m constrângeri date de

$$Ax \leq b.$$

In unele cazuri nu prezintă interes funcția obiectiv, se dorește să se găsească unei soluții fezabile (orice vector x ce satisfacă $Ax \leq b$) sau să se arate că nu există astfel de soluții.

Intr-un sistem de constrângeri fiecare rând din matricea A conține o valoare -1 , o valoare 1 și restul valorilor sunt 0 . Astfel constrângările date de $Ax \leq b$ sunt un set de m constrângeri cu n necunoscute unde fiecare constrângere este o inecuație de forma

$$x_j - x_i \leq b_k,$$

unde $1 \leq i, j \leq n, i \neq j$ și $1 \leq k \leq m$.

Exemplu

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

problema cere să se găsească x_1, x_2, x_3, x_4, x_5 pentru cele 8 constrângeri

$$x_1 - x_2 \leq 0,$$

$$x_1 - x_5 \leq -1,$$

...

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

Soluția nu este unică, două posibile soluții:

$$x = (-5, -3, 0, -1, -4)$$

$$x' = (0, 2, 5, 4, 1)$$

4.1.1 Constrângeri sub forma unui graf

Cum se poate modela problema sub forma unui graf?

Grafuri de constrângeri sistemul de constrângeri poate fi interpretat sub forma unui graf, pentru un sistem $Ax \leq b$ de constrângeri, matricea A de dimensiune $m \times n$ poate fi văzută ca transpusa unei matrici de incidentă a unui graf cu n vârfuri și m arce. Fiecare vârf $v_i \in V, i = 1, 2, \dots, n$ corespunde unei variabile x_i . Fiecare arc $(i, j) \in E$ corespunde unei inegalități

Fie un sistem $Ax \leq b$ de constrângeri, graful corespunzător acestui sistem este un graf ponderat și orientat $G = (V, E)$ unde $V = \{v_0, v_1, \dots, v_n\}$ și

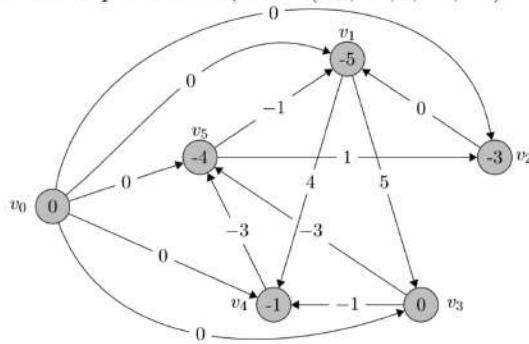
$$\begin{aligned} E = & \{(v_i, v_j) \mid x_j - x_i \leq b_k \text{ este o constrângere}\} \\ & \cup \{(v_0, v_1), (v_0, v_2), \dots, (v_0, v_n)\} \end{aligned}$$

Fie un sistem $Ax \leq b$ de constrângeri și $G = (V, E)$ graful constrângerilor. Dacă G nu conține circuite de pondere negativă, atunci

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \dots, \delta(v_0, v_n))$$

este o soluție fezabilă pentru sistem. Dacă graful G conține un circuit negativ, sistemul nu are soluție.

Exemplu: pentru sistemul definit mai sus se poate desena graful de mai jos, în graf valoarea $\delta(v_0, v_i)$ apare în fiecare nod. O posibilă soluție $x = (-5, -3, 0, -1, -4)$.



Drumuri minime

Problema găsirii drumului minim între toate perechile de vârfuri ale unui graf:

- **se dă:** un graf orientat $G = (V, E)$ cu funcția de pondere $\omega : E \rightarrow \mathbb{R}$,
- **se vrea:** să se găsească pentru fiecare pereche de vârfuri $u, v \in E$ un drum minim de la u la v (suma ponderilor arcelor din drum să fie minimă),

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

- afişarea rezultatului se face sub forma unei matrici $A_{n,n}$, $n = |V|$, unde elementul $a_{i,j} = \delta(i, j)$ arată lungimea drumului de la vârful i la vârful j .

Idee: se poate rezolva problema dacă se rulează un algoritm de drum minim între un vârf sursă s și toate vâfurile din graf $V \setminus \{s\}$ de $|V|$ ori (pentru fiecare vârf din graf ca și sursă). De exemplu:

- dacă există ponderi negative se rulează BELLMAN_FORD pentru fiecare vârf din graf, complexitatea $O(V^2E)$ iar dacă graful este dens $O(V^4)$;
- dacă toate ponderile nu sunt negative se poate rula Dijkstra pentru fiecare vârf din graf, complexitatea $O(VE \lg V)$ dacă se folosește un binary heap pentru coada de priorități ($O(V^3 \lg V)$ pentru un graf dens) și $O(V^2 \lg V + VE)$ dacă se folosește un Fibonacci heap pentru coada de priorități ($O(V^3)$ dacă graful este dens).

Vom vedea cum putem determina drumul de cost minim între oricare două vârfuri din graf în $O(V^3)$ în toate cazurile, fără a folosi structuri de date speciale.

4.2 Drumuri minime și înmulțirea unor matrici

Majoritatea algoritmilor își reprezintă un graf folosind ca și reprezentare matricea de adiacență. Fie un graf $G = (V, E)$, vâfurile sunt numerotate de la 1 la n , reprezentat de o matrice de adiacență de ponderi $A = (a_{i,j})_{i,j=1,n}$ unde:

$$a_{i,j} = \begin{cases} 0 & \text{dacă } i = j, \\ \text{ponderea arcului } (i, j) & \text{dacă } i \neq j, (i, j) \in E, \\ \infty & \text{dacă } i \neq j, (i, j) \notin E. \end{cases}$$

Rezultatul va fi matricea $D = (d_{i,j})$, unde $d_{i,j} = \delta(i, j)$.

Se prezintă o soluție bazată pe programare dinamică pentru a rezolva problema drumului minim între oricare două vârfuri din graf.

Pentru aceasta trebuie:

1. caracterizată structura unei soluții optimale
2. definită recursiv valoarea soluției optimale
3. calculată valoarea soluției optimale

4.2.1 Structura unui drum minim

Pe baza lemei drumului minim (vezi cursurile anterioare): oricare subdrum dintr-un drum minim este drum minim. Graful $G = (V, E)$ este reprezentat de matricea de adiacență $A = (a_{i,j})$.

- fie p drumul minim de la vârful i la vârful j format din m arcuri;
- dacă nu există un circuit negativ m este finit;
- dacă $i = j$ ponderea lui p este 0;
- dacă $i \neq j$ atunci $i \xrightarrow{p'} k \rightarrow j$, unde drumul p' conține $m - 1$ arce;
- din lema drumului minim: p' este drum minim de la i la k și $\delta(i, j) = \delta(i, k) + a_{k,j}$.

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

4.2.2 O soluție recursivă

Fie $l_{ij}^{(m)}$ ponderea minimă a unui drum $i \rightsquigarrow j$ de m arce

$$l_{ij}^{(0)} = \begin{cases} 0, & \text{dacă } i = j, \\ \infty, & \text{dacă } i \neq j. \end{cases}$$

De la $m \geq 1$ se calculează $l_{ij}^{(m)}$ ca minimul lui $l_{ij}^{(m-1)}$ (ponderea drumului minim de la i la j format din cel mult $m-1$ arce) și ponderea minimă a oricărui drum de la i la j format din m arce, obținută dacă ne uităm la toți predecesorii k ai lui j .

Se pot defini recursiv:

$$l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{l_{ij}^{(m-1)} + a_{ik}\}) = \min_{1 \leq k \leq n} \{l_{ij}^{(m-1)} + a_{jk}\} \quad (1)$$

deoarece $a_{jj} = 0 \forall j$.

Care este drumul minim $\delta(i, j)$?

Dacă G nu are circuite negative, pentru orice pereche i și j pentru care $\delta(i, j) < \infty$ există un drum minim de la i la j simplu ce conține cel mult $n-1$ arce. Deci:

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)}.$$

4.2.3 Calculul drumului minim

Ca și date de intrare se primește $A = (a_{ij})$ și se calculează o serie de matrici $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$ unde pentru $m = 1, 2, \dots, n-1$ avem $L^{(m)} = (l_{ij}^{(m)})$. $L^{(n-1)}$ conține drumul minim (ponderile drumului minim)

$$l_{ij}^{(1)} = a_{ij} \forall i, j \in \mathbb{N} \Rightarrow L^{(1)} = A.$$

Următoarea procedură primește ca și parametrii $L^{(n-1)}$, A și întoarce $L^{(n)}$.

EXTINDE_DRUM_MINIM(L,A)

- 1: $n = L.rânduri$
- 2: fie $L' = (l'_{ij})$ o matrice $n \times n$
- 3: **for** $1 \leq i \leq n$ **do**
- 4: **for** $1 \leq j \leq n$ **do**
- 5: $l'_{ij} = \infty$
- 6: **for** $1 \leq k \leq n$ **do**
- 7: $l'_{ij} = \min(l'_{ij}, l_{ik} + a_{kj})$
- 8: **return** L'

Procedura determină matricea L' folosind relația (1), timpul de execuție este $\Theta(V^3)$. Se poate face o paralelă cu procedura de înmulțirea a două matrici.

Practic drumul minim se determină prin extinderea drumului minim arc cu arc. Trebuie să se determine:

$$L^{(1)} = L^{(0)} \cdot A = A$$

$$L^{(2)} = L^{(1)} \cdot A = A^2$$

$$L^{(3)} = L^{(2)} \cdot A = A^3$$

$$\dots$$

$$L^{(n-1)} = L^{(n-2)} \cdot A = A^{n-1}$$

Matricea $L^{(n-1)} = A^{n-1}$ conține drumul minim. Următoarea procedură determină secvența în $\Theta(V^4)$.

 IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

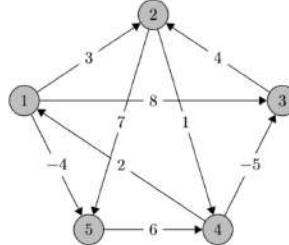


Figura 1: Exemplu determinare drum minim

DETERMINA_TOATE_DRUMURILE_MIN(A)

```

1:  $n = A.\text{rânduri}$ 
2:  $L^{(1)} = A$ 
3: for  $2 \leq m \leq n - 1$  do
4:   fie  $L^{(m)}$  o matrice  $n \times n$ 
5:    $L^m = EXTINDE\_DRUM\_MINIM(L^{m-1}, A)$ 
6: return  $L^{(n-1)}$ 

```

De exemplu, fie graful din figura 1 pentru care se determină matricea $L^{(n-1)}$:

$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

...

$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

4.2.4 Îmbunătățirea timpului de rulare

Nu trebuie determinate m matrici, ne interesează doar $(n-1)$ matrici. $L^{(n-1)}$ se poate calcula din $\lceil \lg(n-1) \rceil$ pași deoarece $2^{\lceil \lg(n-1) \rceil} \geq n-1$ produsul final $L^{(2^{\lceil \lg(n-1) \rceil})}$ este $L^{(n-1)}$.

$$\begin{aligned} L^{(1)} &= A \\ L^{(2)} &= A^2 = A \cdot A \\ L^{(4)} &= A^4 = A^2 \cdot A^2 \\ L^{(8)} &= A^8 = A^4 \cdot A^4 \\ &\dots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= A^{2^{\lceil \lg(n-1) \rceil}} = A^{2^{\lceil \lg(n-1) \rceil}-1} \cdot A^{2^{\lceil \lg(n-1) \rceil}-1} \end{aligned}$$

Procedura este:

MAI_RAPID_TOATE_DRUMURILE_MIN(A)

```
1:  $n = A.\text{rânduri}$ 
```

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

```

2:  $L^{(1)} = A$ 
3:  $m = 1$ 
4: while  $m \leq n - 1$  do
5:   fie  $L^{(2m)}$  o matrice  $n \times n$ 
6:    $L^{2m} = EXTINDE\_DRUM\_MINIM(L^m, L^m)$ 
7:    $m = 2m$ 
8: return  $L^{(m)}$ 
```

Timpul de execuție este $\Theta(V^3 \lg V)$, fiecare produs de matrice ia $\Theta(V^3)$ datorită $\lceil \lg(n-1) \rceil$.

4.3 Floyd-Warshall

Algoritmul a fost discutat în cursurile anterioare.

Recurziv, fie $d_{ij}^{(k)}$ ponderea drumului minim de la i la j , vârfurile intermediare drumului sunt în setul $\{1, 2, \dots, k\}$.

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

4.4 Închiderea tranzitivă a unui graf orientat

Fie $G = (V, E)$ un graf orientat cu setul $V = \{1, 2, \dots, n\}$, trebuie să se determine dacă există un drum în graful G de la vârful i la vârful j , unde $i, j \in V$.

Închiderea tranzitivă a lui G este definită ca și graful $G^* = (V, E^*)$, unde

$$E^* = \{(i, j) | \text{ dacă există } i \rightsquigarrow j \text{ în } G\}.$$

O modalitate de a determina închiderea tranzitivă în $\Theta(V^3)$ e de a rula algoritmul *Floyd-Warshall* pe G cu ponderea 1 pe fiecare arc. Dacă există un drum $i \rightsquigarrow j$ atunci $d_{ij} < n$ altfel $d_{ij} = \infty$.

Există o **altă metodă** pentru a determina închiderea tranzitivă în $\Theta(V^3)$ care poate salva timp și spațiu în practică: se substituie operațiile aritmetice *min* și *+* din *Floyd-Warshall* cu operațiile logice *∨* (*SAU* logic) și *∧* (*SI* logic).

Pentru $i, j, k = 1, \dots, n$ se definește $t_{ij}^{(k)} = 1$ dacă există un drum $i \rightsquigarrow j$ cu vârfurile intermediare în setul $\{1, \dots, k\}$ și $t_{ij}^{(k)} = 0$ în rest.

Se construiește închiderea tranzitivă $G^* = (V, E^*)$ prin adăugarea arcului (i, j) în E^* dacă și numai dacă $t_{ij}^{(k)} = 1$.

Pentru a calcula recursiv închiderea tranzitivă, putem defini:

$$t_{ij}^{(0)} = \begin{cases} 0 & , \text{ dacă } i \neq j \text{ și } (i, j) \notin E, \\ 1 & , \text{ dacă } i \neq j \text{ și } (i, j) \in E, \end{cases}$$

și pentru $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Se determină matricile $T^k = (t_{ij}^{(k)})$ pentru un $k \nearrow \nearrow$. Procedura pentru determinarea închiderii tranzitive este:

INCHIDERE_TRANZITIVA(G)

```

1:  $n = |V|$ 
2: fie  $T^{(0)} = (t_{ij}^{(0)})$  o matrice  $n \times n$ 
3: for  $1 \leq i \leq n$  do
4:   for  $1 \leq j \leq n$  do
```

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

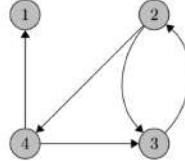


Figura 2: Exemplu determinare închidere tranzitivă

```

5:   if  $i == j$  sau  $(i, j) \in E$  then
6:      $t_{ij}^{(0)} = 1$ 
7:   else
8:      $t_{ij}^{(0)} = 0$ 
9:   for  $1 \leq k \leq n$  do
10:    fie  $T^{(k)} = (t_{ij}^{(k)})$  o matrice  $n \times n$ 
11:    for  $1 \leq i \leq n$  do
12:      for  $1 \leq j \leq n$  do
13:         $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
14: return  $T^{(m)}$ 

```

De exemplu, fie graful din figura 2 pentru care se determină matricile $T^{(k)}$:

$$\begin{aligned}
T^{(0)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} & T^{(1)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} & T^{(2)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & \textcolor{red}{1} \\ 1 & 0 & 1 & 1 \end{pmatrix} \\
T^{(3)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ \textcolor{red}{1} & \textcolor{red}{1} & 1 & 1 \end{pmatrix} & T^{(3)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ \textcolor{red}{1} & 1 & 1 & 1 \\ \textcolor{red}{1} & 1 & 1 & 1 \\ 1 & \textcolor{red}{1} & 1 & 1 \end{pmatrix}
\end{aligned}$$

4.5 Algoritmul lui Johnson pentru grafuri rare

Algoritmul găsește drumul minim în $O(V^2 \lg V + VE)$. Pentru grafuri rare e asymptotic mai rapid decât înmulțirea de matrici și Floyd-Warshall. Algoritmul găsește o soluție pentru grafuri fără circuite negative. Folosește ca și subrutina Dijkstra și Bellman-Ford.

Algoritmul folosește tehnica de reponderare:

- dacă toate ponderile sunt strict pozitive pot rula Dijkstra din fiecare vârf și pot găsi drumul minim în $O(V^2 \lg V + VE)$ (Fibonacci heap)
- dacă G are ponderi negative dar nu are circuit negativ trebuie recalculate ponderile astfel încât să fie pozitive, noul set de ponderi \hat{w} trebuie să satisfacă următoarele:
 1. pentru toate perechile $u, v \in V$, un drum p este minim de la u la v folosind ponderile w dacă p e drum minim de la u la v și pentru ponderile \hat{w} ;
 2. pentru toate arcele (u, v) , $\hat{w}(u, v) \geq 0$;
- \hat{w} se poate determina în $O(VE)$.

Prin reponderare drumul minim trebuie păstrat. Notăm cu:

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

- δ drumul minim din ponderile w ,
- $\hat{\delta}$ drumul minim din ponderile \hat{w} .

Lema 4.1 (Reponderarea nu schimbă drumurile minime) fie un graf orientat și ponderat $G = (V, E)$ cu funcția de pondere $w : E \rightarrow \mathbb{R}$, fie $h : V \rightarrow \mathbb{R}$ o funcție ce mapează vârfurile la numere reale. Pentru fiecare arc $(u, v) \in E$ se definește

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v).$$

Fie $p = \langle v_0, \dots, v_k \rangle$ un drum de la v_0 la v_k , p este un drum minim de la v_0 la v_k cu w dacă e drum minim și pentru \hat{w} , $w(p) = \delta(v_0, v_k) \Leftrightarrow \hat{w}(p) = \hat{\delta}(v_0, v_k)$.

Reponderare pozitivă se vrea ca $\hat{w}(u, v) \geq 0$ pentru $(u, v) \in E$. Fie $G = (V, E)$ cu $w : E \rightarrow \mathbb{R}$, se construiește un nou graf $G' = (V', E')$ unde:

- $V' = V \cup \{s\}$, $s \in V$
- $E' = E \cup \{(s, v) | v \in V\}$, $w(s, v) = 0, \forall v \in V$.

G' nu are circuite negative dacă G nu are circuite negative. Se definește $h(v) = \delta(s, v), \forall v \in V'$

$$h(v) \leq h(u) + w(u, v), \forall (u, v) \in E'$$

deci

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0.$$

Algoritmul lui Johnson este:

JOHNSON(G)

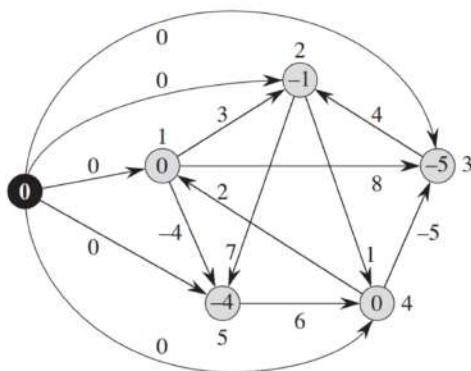
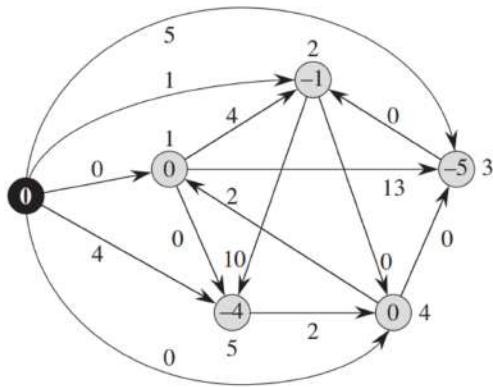
```

1: determină  $G', V' = V \cup \{s\}, E' = E \cup \{(s, v) | v \in V\}$  și  $w(s, v) = 0 \forall v \in V$ 
2: if BELLMAN_FORD( $G', w, s$ ) == FALSE then
3:   exit
4: else
5:   for  $v \in V'$  do
6:     pună  $h(v) = \delta(s, v)$  determinată de BELLMAN_FORD
7:   for  $(u, v) \in E'$  do
8:      $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ 
9:   fie  $D = (d_{uv})$  o matrice  $n \times n$ 
10:  for  $u \in V$  do
11:    rulează DIJKSTRA( $G, \hat{w}, u$ ) pentru a determina  $\hat{\delta}(u, v) \forall v \in V$ 
12:    for  $v \in V$  do
13:       $d_{uv} = \hat{\delta}(u, v) - h(u) + h(v)$ 
14:  return  $D$ 

```

Ca și exemplu fie graful G' cu funcția de pondere w din figura 4. Figura 5 prezintă graful după reponderare. Figurile 6a-6e prezintă rezultatul rulării algoritmului Dijkstra pentru fiecare vârf din G ca și sursă, vârful sursă este negru iar drumul minim este reprezentat de arcele marcate cu gri. Fiecare vârf conține valorile $\hat{\delta}(u, v)/\delta(u, v)$. Valoarea $d_{uv} = \delta(u, v)$ este $\hat{\delta}(u, v) + h(u) - h(v)$.

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

Figura 4: Graful G' cu funcția de pondere w .Figura 5: Graful G' cu funcția de pondere \hat{w} .

IV. Programare liniara, Drumuri minime între toate perechile de vârfuri

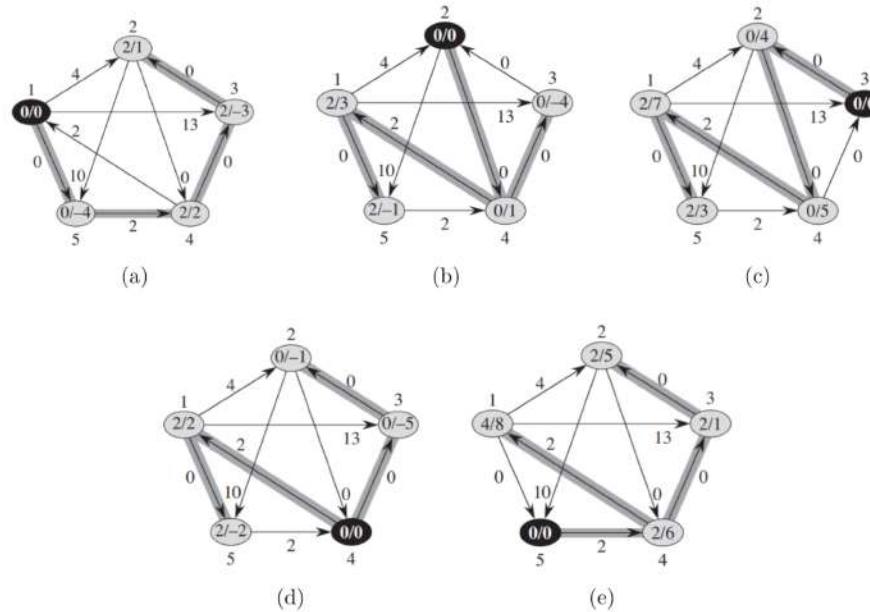
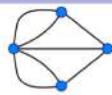
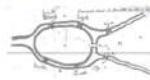


Figura 6: Rezultatul rulării lui *Dijkstra* pentru fiecare vârf din G folosind funcția pondere \bar{w} .

4.6 Referințe

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press.
 2. Geir Agnarsson and Raymond Greenlaw. 2006. Graph Theory: Modeling, Applications, and Algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
 3. Mark Newman. 2010. Networks: An Introduction. Oxford University Press, Inc., New York, NY, USA.
 4. Cristian A. Giumale. 2004. Introducere în analiza algoritmilor, teorie și aplicatie. Polirom.

Curs 4



Algoritmica grafurilor V. Arbori si paduri

Mihai Suciu

Facultatea de Matematică și Informatică (UBB)
Departamentul de Informatică

Martie, 24, 2022

1 / 47



Continut

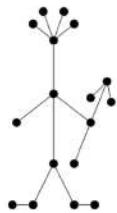
- Arbori si paduri
 - Definii
 - Arbori de acoperire
 - Algoritmul lui Kruskal
 - algoritmul lui Prim
 - Prüfer
 - codare Huffman

2 / 47

Arbori și păduri



- un arbore



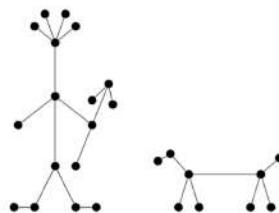
3 / 47

Arbori și păduri



Arbori și păduri (II)

- o pădure

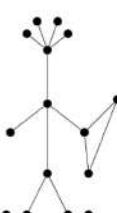


4 / 47

Arbori și păduri



- un graf care nu este arbore sau pădure



5 / 47

Arbori și păduri - Definiții



Arbori și păduri - definiții

Definiții

Un **arbore** este un graf simplu care nu are cicluri.

O **pădure** este un graf $G = (V, E)$ simplu în care fiecare componentă este un arbore.

6 / 47

Arbore și păduri - definiții (II)

Definiție
un vârf u al unui graf simplu $G = (V, E)$ se numește **frunză** dacă $d_G(u) = 1$. Un vârf care nu este frunză se numește **vârf intern**.

Multe proprietăți asociate arborilor pot fi derivate din următoarea teoremă

Teorema 4.2
fiecare arbore cu minim două vârfuri are cel puțin două frunze.

7 / 47

Arbore și păduri - definiții (III)

Demonstrație.

- fie T un arbore cu $n \geq 2$, fie p lanțul de lungime maximă din T și u, v vârfurile lui p
- se arată că u și v sunt frunze, $d(u) = d(v) = 1$, este suficient să se demonstreze pentru un singur vârf
- dacă $d(u) \geq 2 \Rightarrow \exists e \in E, e \notin p$, având vârfurile $u, w \in V$
- avem două cazuri:
 - $w \notin p \Rightarrow$ lanțul compus $p' = (w, e, u)p$ este un lanț din T având lungimea lanțului p plus 1 \rightarrow contradicție (p lanțul de lungime maximă)
 - $w \in p$, dacă p'' este lanțul de la u la v atunci avem un ciclu $c = (w, e, u)p''$ de lungime cel puțin 3 în $T \rightarrow T$ nu este arbore
- $\Rightarrow d(u) = 1$

□

Arbore și păduri - definiții (IV)

Fie $G = (V, E)$ un graf de ordin $n \geq 2$, afirmațiile următoare sunt echivalente și caracterizează un arbore:

- G este un arbore
- G este fără cicluri și are $n - 1$ muchii
- G este conex și are $n - 1$ muchii
- G este conex și suprimând o muchie nu mai este conex
- între oricare două vârfuri ale grafului există un singur lanț
- G este fără cicluri și prin adăugarea unei muchii între două vârfuri neadiacente se formează un singur ciclu

9 / 47

Atenție: - nu are cicluri, este conex, are $n-1$ muchii
 - dacă scoadem o muchie nu mai e conex
 - dacă adăugăm o muchie \Rightarrow ciclu
 - între 2 vârfuri \exists un singur lanț

Arbore și păduri - definiții (V)

Teorema Erdős-Szekeres
dacă $(x_1, x_2, \dots, x_{hk+1})$ este o secvență de numere reale distincte, atunci există o subsecvență crescătoare de $h + 1$ elemente sau o subsecvență descrescătoare de $k + 1$ elemente.

Corolar
fiecare secvență de numere reale distincte de lungime n conține o subsecvență de lungime $\lceil \sqrt{n} \rceil$ strict crescătoare sau strict descrescătoare.

10 / 47

Arbore și păduri - definiții (VI)

Centrul unui arbore
fie $G = (V, E)$ un graf și $u \in V$

- excentricitatea $e_G(u)$ a lui u în G este distanța de la u la vârful cel mai îndepărtat de u din G ,

$$e_G(u) = \max(\delta_G(u, v) | v \in V)$$

- centrul lui G este vârful pentru care

$$\min_{u \in V}(e_G(u))$$

11 / 47

Arbore și păduri - definiții (VII)

Rădăcina unui arbore
fie T un arbore și $r \in V(T)$. Un arbore cu rădăcină este perechea ordonată (T, r) , vârful r se numește **rădăcina** arboreului.

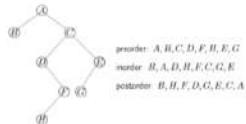


12 / 47

Arbore și păduri - definiții (VIII)

Arbore binar

un **arbore binar** este un arbore ce are o rădăcină, este ordonat și în care fiecare vârf are cel mult doi succesi. Succesorii fiecărui vârf sunt ordonați, fiul stâng și fiul drept.



13 / 47

Arbore și păduri

Arbore de acoperire

Arbore de acoperire (spanning trees)

Ex. realizarea unui circuit electronic

- terminalele mai multor componente electronice trebuie interconectate
- pentru a conecta n terminale e nevoie de $n - 1$ conexiuni, fiecare conectând două terminale
- dintre toate aranjamentele cel mai dezirabil este cel care folosește cât mai puțin cupru pentru a conecta terminalele

14 / 47

Arbore de acoperire (II)

problema poate fi rezolvată cu ajutorul unui graf

Definire problemă

fie un graf $G = (V, E)$ simplu neorientat unde V este setul terminalelor și E este setul conexiunilor posibile între terminalele componentelor. Pentru fiecare muchie $(u, v) \in E$ avem o pondere $w(u, v)$ ce specifică costul legăturii (ex. cantitatea de cupru folosită). Vrem să găsim un subset aciclic $T \subseteq E$ care leagă toate vârfurile având costul total

$$w(t) = \sum_{(u,v) \in T} w(u, v)$$

15 / 47

minim.

Arbore de acoperire (III)

- deoarece T este aciclic și leagă toate vârfurile, T este un arbore numit **arbore de acoperire**
- problema cere determinarea arborelui minim de acoperire



16 / 47

Arbore de acoperire (IV)

Un arbore de acoperire T are următoarele proprietăți

- T este conex
- T este aciclic
- T are n vârfuri
- T are $n - 1$ muchii

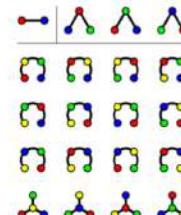
Dacă un subgraf T al unui graf $G = (V, E)$ are oricare trei astfel de proprietăți atunci T este un arbore de acoperire.

17 / 47

Arbore de acoperire - formula lui Cayley

Cayley

fie un graf complet K_n , numărul arborilor etichetați este n^{n-2}



18 / 47

Arbore de acoperire minimă - metoda generică

Fie un graf simplu neorientat $G = (V, E)$ cu funcția de pondere $w : E \rightarrow \mathbb{R}$ și vrem să găsim arborele minim de acoperire a lui G .

- generic, abordarea folosită este surprinsă de procedura
- ```
generic_mst(G)
1: A = ∅
2: while A nu este un arbore minim de acoperire do
3: găsește o muchie (u, v) sigură pentru A
4: A = A ∪ {(u, v)}
5: return A
• arborele minim de acoperire crește muchie cu muchie
```

19 / 47

## Arbore de acoperire minimă - metoda generică (II)

- înainte de fiecare iterație  $A$  este un subset al unui arbore minim de acoperire
- în fiecare pas se găsește o muchie care împreună cu  $A$  formează un subset al unui arbore minim de acoperire (muchie **sigură**)
- **partea dificilă:** găsirea muchiei  $(u, v)$  astfel încât  $A \subseteq T$
- o tăietură  $(S, V - S)$  a unui graf neorientat  $G = (V, E)$  este o partiție a lui  $V$

20 / 47

## Arbore de acoperire minimă - metoda generică (III)

### Teorema

fie  $G = (V, E)$  un graf simplu neorientat ponderat cu funcția de pondere  $w : E \rightarrow \mathbb{R}$ . Fie  $A$  un subset al lui  $E$  inclus într-un arbore minim de acoperire al lui  $G$ , fie  $(S, V - S)$  o tăietură a lui  $G$  ce respectă  $A$  și  $(u, v)$  muchia de pondere minimă ce traversează tăietura  $(S, V - S)$ . În acest caz, muchia  $(u, v)$  este sigură pentru  $A$ .

### Corolar

$G = (V, E)$  un graf simplu neorientat ponderat cu funcția de pondere  $w : E \rightarrow \mathbb{R}$ . Fie  $A$  un subset al lui  $E$  inclus într-un arbore minim de acoperire al lui  $G$ , fie  $C = (V_C, E_C)$  o componentă conexă (arbore) în pădurea  $G_A = (V, A)$ . Dacă  $(u, v)$  este o muchie de pondere minimă ce leagă componenta  $C$  de o altă componentă din  $G_A$ , atunci  $(u, v)$  este sigură pentru  $A$ .

21 / 47

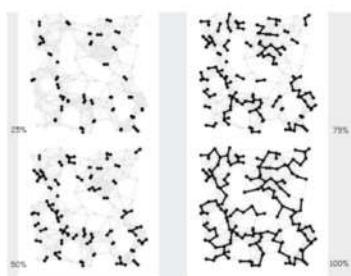
## Algoritmul lui Kruskal

### mst\_kruskal(G,w)

- 1:  $A = \emptyset$
  - 2: for  $v \in V$  do
  - 3: make\_set( $v$ )
  - 4: sortare muchii crescător după ponderea  $w$
  - 5: for  $(u, v) \in E$  luate crescător după  $w$  do
  - 6: if find\_set( $u$ ) ≠ find\_set( $v$ ) then
  - 7:  $A = A \cup (u, v)$
  - 8: union( $u, v$ )
  - 9: return  $A$
- implementarea folosește o structură de date de tipul *disjoint-set* (*union-find, merge-find*)

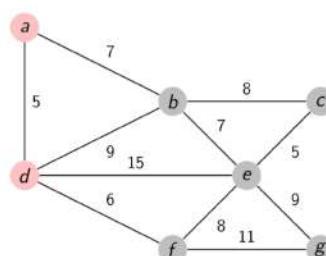
22 / 47

## Algoritmul lui Kruskal - exemplu



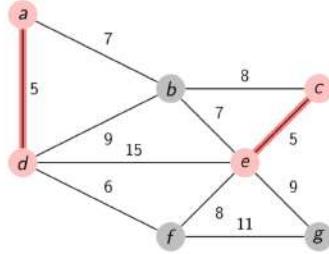
23 / 47

## Algoritmul lui Kruskal - exemplu (II)



Arbore si paduri Algoritmul lui Kruskal

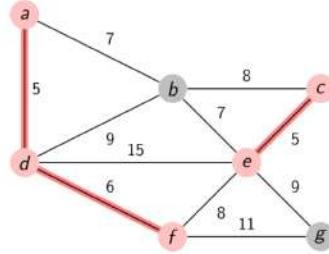
## Algoritmul lui Kruskal - exemplu (II)



25 / 47

Arbore si paduri Algoritmul lui Kruskal

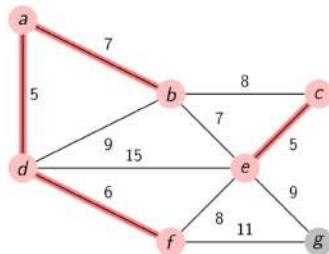
## Algoritmul lui Kruskal - exemplu (II)



26 / 47

Arbore si paduri Algoritmul lui Kruskal

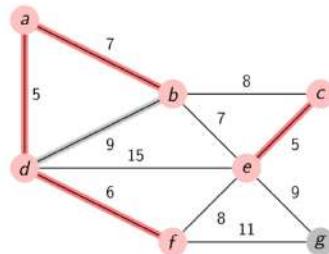
## Algoritmul lui Kruskal - exemplu (II)



27 / 47

Arbore si paduri Algoritmul lui Kruskal

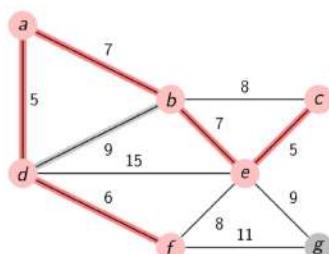
## Algoritmul lui Kruskal - exemplu (II)



28 / 47

Arbore si paduri Algoritmul lui Kruskal

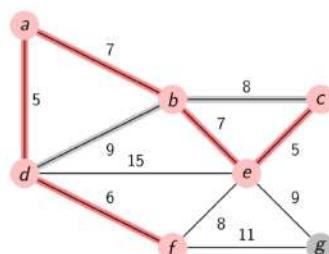
## Algoritmul lui Kruskal - exemplu (II)



29 / 47

Arbore si paduri Algoritmul lui Kruskal

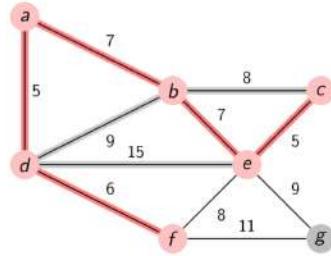
## Algoritmul lui Kruskal - exemplu (II)



30 / 47

Arbore si paduri Algoritmul lui Kruskal

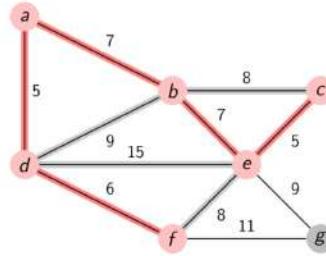
### Algoritmul lui Kruskal - exemplu (II)



31 / 47

Arbore si paduri Algoritmul lui Kruskal

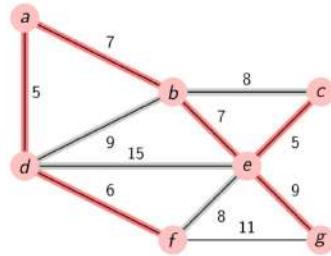
### Algoritmul lui Kruskal - exemplu (II)



32 / 47

Arbore si paduri Algoritmul lui Kruskal

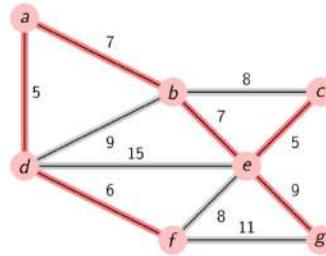
### Algoritmul lui Kruskal - exemplu (II)



33 / 47

Arbore si paduri Algoritmul lui Kruskal

### Algoritmul lui Kruskal - exemplu (II)



34 / 47

Arbore si paduri algoritmul lui Prim

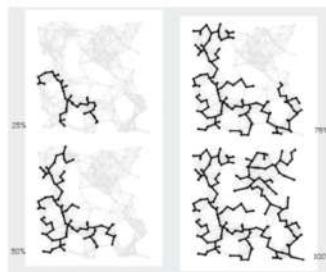
### Algoritmul lui Prim

```
mst_prin(G,w,r)
1: for u ∈ V do
2: u.key = ∞
3: u.π = NIL
4: r.key = 0
5: Q = V
6: while Q ≠ ∅ do
7: u = extract_min(Q)
8: for v ∈ Adj[u] do
9: if v ∈ Q și w(u,v) < v.key then
10: v.π = u
11: v.key = w(u,v)
```

35 / 47

Arbore si paduri algoritmul lui Prim

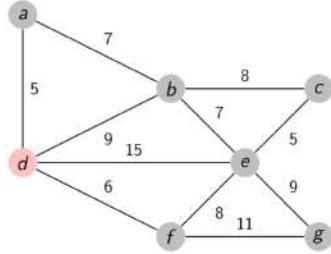
### Algoritmul lui Prim - exemplu



36 / 47

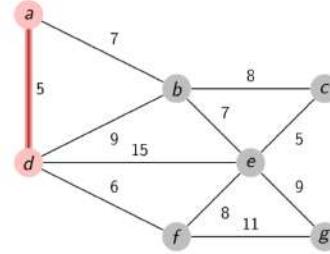
Arbore si paduri algoritmul lui Prim

### Algoritmul lui Prim - exemplu (II)



Arbore si paduri algoritmul lui Prim

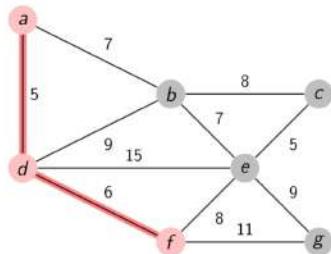
### Algoritmul lui Prim - exemplu (II)



38 / 47

Arbore si paduri algoritmul lui Prim

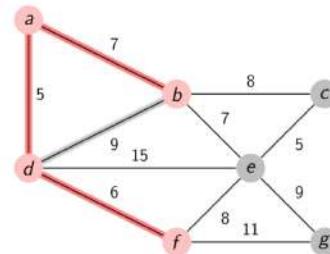
### Algoritmul lui Prim - exemplu (II)



39 / 47

Arbore si paduri algoritmul lui Prim

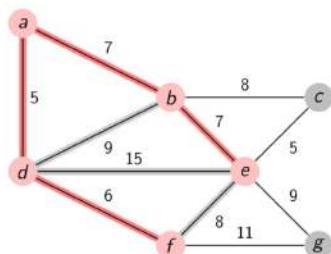
### Algoritmul lui Prim - exemplu (II)



40 / 47

Arbore si paduri algoritmul lui Prim

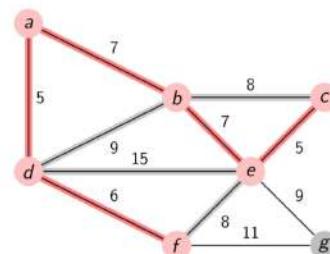
### Algoritmul lui Prim - exemplu (II)



41 / 47

Arbore si paduri algoritmul lui Prim

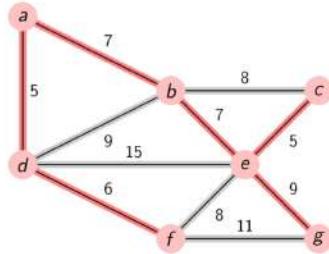
### Algoritmul lui Prim - exemplu (II)



42 / 47

Arbore si paduri algoritmul lui Prim

## Algoritmul lui Prim - exemplu (II)



43 / 47

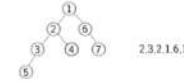
Arbore si paduri Prüfer

## Codare Prüfer

CODARE\_PRÜFER( $F$ )

1.  $K = \emptyset$
2. **while**  $T$  conține și alte vârfuri decât rădăcina **do**
3.     fie  $v$  frunza minimă din  $T$
4.      $K \leftarrow$  predecesor( $v$ )
5.      $T = T \setminus \{v\}$
6. **return**  $K$

exemplu:



2.3.2.1.6.1

44 / 47

Arbore si paduri Prüfer

## Decodare Prüfer

DECODARE\_PRÜFER( $K, n$ )

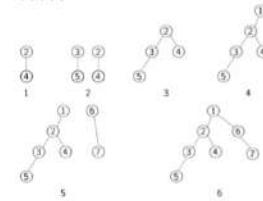
1.  $T = \emptyset$
2. **for**  $i = 1, 2, \dots, n - 1$  **do**
3.      $x$  primul element din  $K$
4.      $y$  cel mai mic număr natural care nu se găsește în  $K$
5.      $(x, y) \in E(T)$ ,  $x$  părintele lui  $y$  în  $T$
6.     șterg  $x$  din  $K$ , adaugă  $y$  în  $K$
7. **return**  $T$

45 / 47

Arbore si paduri Prüfer

## Decodare Prüfer - exemplu

2.3.2.1.6.1  
3.2.1.6.1.4 || 5  
2.1.6.1.4.5 || 3  
1.6.1.4.5.3 || 2  
6.1.4.5.3.2 || 7  
1.4.5.3.2.7 || 6  
4.5.3.2.7.6



46 / 47

Arbore si paduri codare Huffman

## Codare Huffman

HUFFMAN( $C$ )

- 1:  $n = |C|$
- 2:  $Q = C$
- 3: **for**  $1 \leq i \leq n - 1$  **do**
- 4:     alocă un nou vârf  $z$
- 5:      $z.stang = x = \text{EXTRACT\_MIN}(Q)$
- 6:      $z.drept = y = \text{EXTRACT\_MIN}(Q)$
- 7:      $z.fr = x.fr + y.fr$
- 8:      $\text{INSERT}(Q, z)$
- 9: **return**  $\text{EXTRACT\_MIN}(Q)$

47 / 47

## Suport/Transcript curs algoritmica grafurilor

### VI. Grafuri euleriene și hamiltoniene

#### 6.1 Grafuri bipartite

**Definiție 6.1.1.** un graf  $G = (V, E)$  simplu și neorientat este bipartit dacă există  $X, Y \subseteq V$  astfel încât

- $V = X \cup Y$
- $X \cap Y = \emptyset$  (sau  $X \neq \emptyset \neq Y$  și  $Y = V \setminus X$ )
- toate muchiile au un capăt în  $X$  și celălalt capăt în  $Y$  (sau  $G(X)$  și  $G(Y)$  sunt grafuri pentru care  $|E| = 0$ )

Figura 1 prezintă exemple de grafuri bipartite și un contra exemplu.

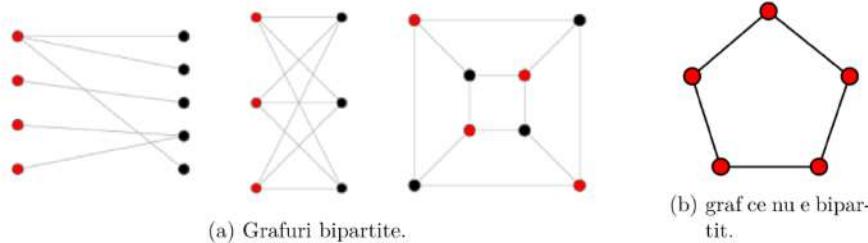


Figura 1: Grafuri bipartite.

Un graf bipartit complet  $K_{n,m}$  este un graf bipartit între  $X$  și  $Y$  cu  $n = |X|$  și  $m = |Y|$  astfel încât există o muchie între oricare pereche de vîrfuri  $(x, y) \in X \times Y$ . Figura 2 prezintă exemple de grafuri bipartite complete.

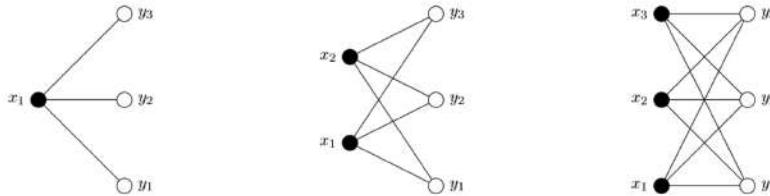


Figura 2: Exemple de grafuri bipartite complete.

## VI. Grafuri euleriene și hamiltoniene

---

### Teorema 6.1 (de caracterizare).

*Un graf cu cel puțin două vârfuri este bipartit dacă și numai dacă nu conține cicluri de lungime impară.*

*Demonstrație.*

" $\Rightarrow$ : " fie  $G = (V, E)$  un graf bipartit între mulțimile  $X$  și  $Y$  și fie  $(v_1, \dots, v_k, v_1)$  un ciclu în  $G$ . Putem presupune că  $v_1 \in X$ . Atunci  $v_i \in X$  și  $v_j \in Y$  dacă  $i$  este par și  $j$  este impar. Deoarece  $(v_k, v_1) \in E$ ,  $k$  trebuie să fie par  $\Rightarrow$  nu putem avea în  $G$  un ciclu de lungime  $k$  impară.

" $\Leftarrow$ : " Putem presupune, fără a reduce din generalitate, că  $G$  este conex (în caz contrar, putem trata separat componentele conexe ale lui  $G$ ). Pentru  $v \in V$  se definește

$X = \{x \in V | \text{cel mai scurt lanț de la } x \text{ la } v \text{ are lungime pară}\}, Y = V \setminus X$ . Se verifică ușor că  $G$  este graf bipartit între  $X$  și  $Y$ .  $\square$

$G$  este bipartit  $\implies$  orice ciclu în  $G$  are lungime pară.

**Observație** dacă  $G$  conține un lanț închis de lungime impară atunci conține un ciclu de lungime impara.

## 6.2 Grafuri Euleriene

Se pot defini următoarele:

- **lanț**: o succesiune de muchii, oricare muchie are o extremitate comună cu muchia precedentă și cealaltă extremitate comună cu muchia următoare;
- **ciclu**: un lanț în care extremitățile coincid;
- **lanț simplu**: un lanț care nu folosește de două ori aceeași muchie;
- **lanț elementar**: un lanț care nu conține (trece) de două ori un (prin) același vârf.

**Definiție 6.2.1.** Pentru un graf simplu  $G = (V, E)$ , putem defini:

- un **lanț Eulerian** în  $G$  ca și un **lanț simplu** ce conține **toate** muchiile din  $G$ ;
- un **ciclu Eulerian** în  $G$  ca și un **lanț simplu** ce conține **toate** muchiile din  $G$  și extremitățile lanțului coincid;
- un **graf Eulerian** ca și un **graf simplu** care conține un **ciclu Eulerian**.

Un graf eulerian se poate caracteriza pe baza:

- gradul vârfurilor,
- existenței unei colecții speciale de cicluri.

### Teorema 6.2 (de caracterizare a grafurilor euleriene).

*pentru un graf conex  $G = (V, E)$ , următoarele afirmații sunt echivalente:*

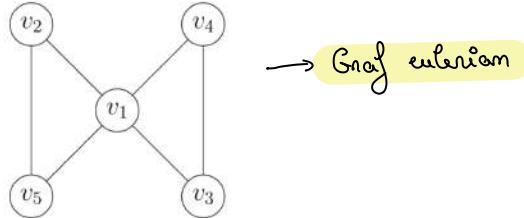
1.  $G$  este eulerian;
2. fiecare vârf al lui  $G$  are grad par;
3. muchiile lui  $G$  pot fi partionate în cicluri care nu au muchii în comun.

## VI. Grafuri euleriene și hamiltoniene

*Demonstrație.*

$1 \rightarrow 2$  se presupune că  $G$  este eulerian  $\Leftrightarrow$  există un ciclu care conține toate muchiile lui  $G$  o singură dată.

Fie graful de mai jos



unde gradul vârfurilor din graf este:  $d(v_1) = 4, d(v_2) = d(v_3) = d(v_4) = d(v_5) = 2$ .

- Ori de câte ori ciclul eulerian intră într-un vârf  $v$  pe o muchie, trebuie să plece din acel vârf pe altă muchie;
- nici o muchie nu apare de două ori în ciclu, numărul muchiilor incidente vârfului  $v$  este par  $\Rightarrow d(v)$  este par;
- exemplu: fie ciclul  $(v_1, v_3, v_4, v_1, v_2, v_5, v_1)$

$2 \rightarrow 3$  Se presupune că fiecare vârf al lui  $G$  are grad par. Ne gândim inductiv după numărul de cicluri disjuncte ale lui  $G$ .  $G$  nu are vârfuri de grad 1  $\implies G$  nu e arbore  $\implies G$  are cel puțin un ciclu  $C_{n_1}$ .

Fie  $G'$  graful produs din  $G$  prin eliminarea muchiilor lui  $C_{n_1} \implies$  toate vârfurile din  $G'$  au grad par  $\implies$  se deduce recursiv ca  $G'$  poate fi partionat în cicluri disjuncte  $C_{n_2}, \dots, C_{n_k}$ .

Rezultă că  $C_{n_1}, C_{n_2}, \dots, C_{n_k}$  este o partitie a lui  $G$  în cicluri (cu muchii) disjuncte. Figura 3 prezintă un exemplu.

$3 \rightarrow 1$  Se presupune că muchiile lui  $G$  pot fi partionate în  $k$  cicluri disjuncte  $C_{n_1}, C_{n_2}, \dots, C_{n_k}$ .  $G$  este conex  $\implies$  fiecare ciclu este un ciclu simplu ce are un vârf comun cu un alt ciclu  $\implies$  ciclurile pot fi înlántuite până se obține un ciclu eulerian.

De exemplu figura 4 prezintă un graf unde se pot forma ciclurile  $C_1 = 1, 6, 8, 1$ ,  $C_2 = 3, 6, 4, 7, 8, 3$ ,  $C_3 = 2, 5, 8, 9, 2$ ,  $C_4 = 1, 3, 2, 7, 6, 5, 4, 1$ . Ciclurile  $C_4$  și  $C_3$  au în comun vârful 2, prin compunere se obține ciclul  $R_1$ . Ciclurile  $R_1$  și  $C_2$  au în comun vârful 3, se obține ciclul  $R_2$ . Ciclurile  $R_2$  și  $C_1$  au în comun vârful 1, se obține ciclul eulerian  $R_3$ .

$$R_1 = 1, 3, 2, 5, 8, 9, 2, 7, 6, 5, 4, 1$$

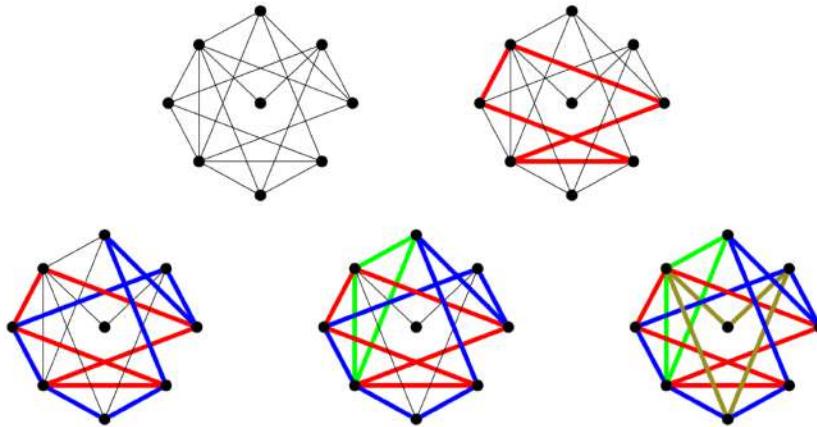
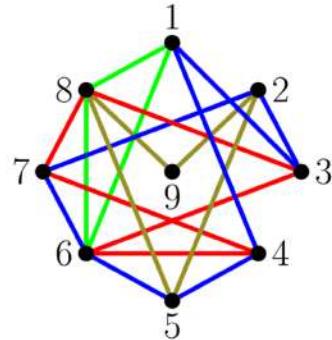
$$R_2 = 1, 3, 6, 4, 7, 8, 3, 2, 5, 8, 9, 2, 7, 6, 5, 4, 1$$

$$R_3 = 1, 6, 8, 1, 3, 6, 4, 7, 8, 3, 2, 5, 8, 9, 2, 7, 6, 5, 4, 1$$

□

Algoritmul de înlántuire a ciclurilor se numește algoritmul lui **Hierholzer**. Algoritmul primește un graf eulerian și caută un ciclu eulerian în graf. Pașii algoritmului sunt:

## VI. Grafuri euleriene și hamiltoniene

Figura 3: Teorema 6.2: echivalență  $2 \rightarrow 3$ .Figura 4: Teorema 6.2: echivalență  $3 \rightarrow 1$ .

1. fie  $i = 1$ , se identifică un ciclu  $R_1$  al grafului și se marchează muchiile lui  $R_1$ ;
2. dacă  $R_i$  conține toate muchiile grafului algoritmul se oprește și  $R_i$  este eulerian;
3. dacă  $R_i$  nu conține toate muchiile grafului, fie  $v_i$  un vârf al ciclului  $R_i$  incident la o muchie nemarcată  $e_i$ ;
4. se construiește un ciclu cu muchii nemarcate  $C_i$ , pornind de la vârful  $v_i$  de-a lungul muchiei  $e_i$ . Muchiile lui  $C_i$  sunt marcate;
5. se creează  $R_{i+1}$  prin înlățuirea lui  $C_i$  în  $R_i$ ;
6.  $i++$  și se revine la pasul 2.

O altă modalitate de a găsi un lanț/ciclu eulerian într-un graf este algoritmul lui **Fleury**. Inițial toate muchiile din graf sunt nemarcate, pașii algoritmului sunt:

1. se alege un vârf curent  $v$ ;
2. dacă toate muchiile lui  $G$  au fost marcate, stop;

---

VI. Grafuri euleriene și hamiltoniene

3. dintre toate muchiile incidente vârfului  $v$  se alege, dacă se poate, o muchie care nu este puncte. Dacă o astfel de muchie nu există, se alege una la întâmplare. Se marchează muchia aleasă iar capătul opus vârfului curent devine noul vârf curent;
4. se revine la pasul 2.

Un graf eulerian conține un lanț eulerian deoarece orice ciclu eulerian este și lanț eulerian. Există grafuri ne-euleriene ce conțin lanțuri euleriene.

**Corolar 6.3.** un graf conex  $G = (V, E)$  conține un lanț eulerian dacă și numai dacă are cel mult două vârfuri de grad impar.

### 6.3 Grafuri hamiltoniene

**Definiție 6.3.1.** Pentru un graf simplu  $G$ , putem defini:

- un *lanț Hamiltonian* în  $G$  ca și un *lanț simplu* ce conține *toate* vârfurile din  $G$ ;
- un *graf traversabil* este un graf simplu ce conține un lanț hamiltonian;
- un *ciclu hamiltonian* în  $G$  ca și un *lanț simplu* ce conține *toate* vârfurile din  $G$  și extremitățile lanțului coincid;
- un *graf hamiltonian* ca și un graf simplu care conține un *ciclu hamiltonian*.

**Observații :**

- toate grafurile hamiltoniene sunt traversabile;
- există grafuri traversabile care nu sunt hamiltonieni.

Problema grafurilor hamiltoniene a apărut ca un joc inventat de matematicianul William R. Hamilton: pe un dodecaedru (figura 5) fiecare vârf reprezintă un oraș, scopul este de a găsi un drum pentru a vizita toate orașele o singură dată și capetele drumului să coincidă.

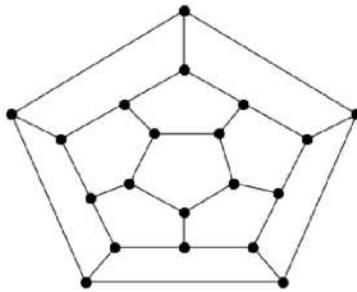


Figura 5: Graf hamiltonian.

Pentru a detecta dacă un graf este hamiltonian se poate utiliza teorema lui Dirac.

**Teorema 6.4** (Dirac).

Fie  $G$  un graf de ordinul  $n \geq 3$ . Dacă  $\delta(G) \geq \frac{n}{2}$  atunci  $G$  este hamiltonian.

sau

fie  $G$  un graf de ordinul  $n \geq 3$ . Dacă  $\forall u \in V, d(u) \geq \frac{n}{2}$  atunci  $G$  este hamiltonian.

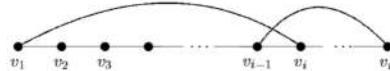
---

VI. Grafuri eulieriene și hamiltoniene

---

*Demonstrație teorema 6.4.*

Presupunem că  $G$  satisface condițiile date, însă  $G$  nu e hamiltonian. Fie  $H = v_1, \dots, v_n$  un lanț simplu în  $G$  de lungime maximă (toți vecinii lui  $v_1$  și  $v_n$  sunt în  $H$ ).  $v_1$  și  $v_n$  au cel puțin  $\frac{n}{2}$  vecini din lanț deoarece  $\delta(G) \geq \frac{n}{2}$ . Arătăm că există  $i \in \{1, \dots, n-1\}$  astfel încât  $v_{i-1} \in N(v_n)$  și  $v_i \in N(v_1)$  ca în figura de mai jos ( $N(v_i)$  reprezintă vecinătatea vârfului  $v_i$ ).



Dacă nu ar fi aşa, pentru fiecare vecin  $v_j$  al lui  $v_n$  din lanț (sunt mai mult de  $\frac{n}{2}$  astfel de  $v_j$ ),  $v_i$  nu ar fi vecin al lui  $v_1$ . Ar rezulta că  $d(v_1) \leq n - 1 - \frac{n}{2} < n - \frac{n}{2} = \frac{n}{2}$  ceea ce contrazice  $\delta(G) \geq \frac{n}{2}$ .

Fie  $L$  ciclul  $v_1, v_2, \dots, v_{i-1}, v_n, v_{n-1}, \dots, v_i, v_1$ , presupunând că  $G$  nu este hamiltonian există un vârf al lui  $G$  care nu e în  $H$ .  $\delta(G) \geq \frac{n}{2}$  și  $n \geq 3 \implies \delta(G) \geq 2$ ,  $G$  este conex  $\implies G$  are un vârf  $w$  care nu este în  $H$  și este adjacent la un vârf  $v_i$  din  $H$ . Dar atunci lanțul care pornește cu  $w, v_i$  și continuă în jurul ciclului  $L$  este mai lung decât  $H \implies$  contradicție.

$\implies G$  trebuie să fie hamiltonian. □

**Teorema 6.5** (Dirac generalizată). *fie  $G$  un graf de ordinul  $n \geq 3$ . Dacă  $d(x) + d(y) \geq n$  pentru toate perechile de vârfuri care nu sunt adiacente  $x, y$ , atunci  $G$  este hamiltonian.*

**Lema 6.6.** *dacă într-un graf cu cel mult  $2k$  vârfuri  $d(x) \geq k, \forall x \in V$  atunci graful este conex.*

*Demonstrație 6.6.*

Presupunem că  $G$  are cel mult  $2k$  vârfuri, fiecare vârf are gradul  $d(x) \geq k$  dar  $G$  nu este conex. În acest caz, graful are cel puțin două componente și există o componentă cu cel mult  $k$  vârfuri. În această componentă gradul maxim este cel mult  $k-1$  ceea ce contrazice presupunerea că fiecare vârf are gradul cel puțin  $k$ . □

Lema 6.6 nu este adevărată pentru multigrafuri.

## Suport curs algoritmica grafurilor

### VII. Rețele de flux

Așa cum se pot modela hărți rutiere folosind grafuri și căuta drumul minim între două puncte, un graf orientat se poate interpreta ca o rețea de flux.

Intuitiv putem să ne imaginăm un material curgând de la o sursă (unde este produs) către o destinație (unde e consumat). Sursa (*source*) produce material constant și destinația (*sink*) consumă material la aceeași rată. Fluxul materialului, la un moment dat, este rata cu care se mișcă.

Rețelele de flux pot modela multe probleme: curgerea lichidelor, traseul unor componente pe linia de asamblare, curentul într-o rețea electrică, informația într-o rețea de comunicații.

Ne putem gândi la arcele din graf ca la un canal de transport pentru material, fiecare canal are definit o capacitate pentru rata maximă cu care materialul curge pe canal (ex. 20A într-un cablu electric, 350Mbit/s pentru un cablu CAT5E). Vârfurile grafului reprezintă intersecția canalelor de transport, materialul curge prin vârfuri fără să se acumuleze.

Rata cu care materialul intră într-un vârf este egală cu rata cu care ieșe din vârf, fluxul se conservă. În problema fluxului maxim trebuie determinată rata maximă cu care se poate transporta material de la sursă la destinație fără a încălca diferite constrângeri aplicate capacitaților.

#### 7.1 Rețele de flux

O rețea de flux  $G = (V, E)$  este un graf orientat în care fiecare arc  $(u, v) \in E$  are o capacitate pozitivă  $c(u, v) \geq 0$ .

- Dacă  $E$  conține arcul  $(u, v)$ ,  $E$  nu trebuie să conțină și arcul  $(v, u)$ ;
- dacă  $(u, v) \notin E \implies c(u, v) = 0$ , nu există bucle;
- se disting 2 vârfuri într-o rețea de flux:
  - $s$  = sursă (*source*)
  - $t$  = destinație (*sink*)

Se presupune că fiecare vârf se află pe un drum de la  $s$  la  $t$  ( $\forall v \in V, s \rightsquigarrow v \rightsquigarrow t$ ).

**Definiție 7.1.1** (Rețea de flux).

Fie  $G = (V, E)$  o rețea de flux cu funcția de capacitate  $c$ ,  $G$  e graf orientat. Fie  $s$  sursă din rețea și  $t$  vârful destinație. Un flux este o funcție  $f : V \times V \rightarrow \mathbb{R}$  ce satisfac restricțiile:

- se respectă capacitatea arcului:  $\forall u, v \in V, 0 \leq f(u, v) \leq c(u, v)$ ;

## VII. Rețele de flux

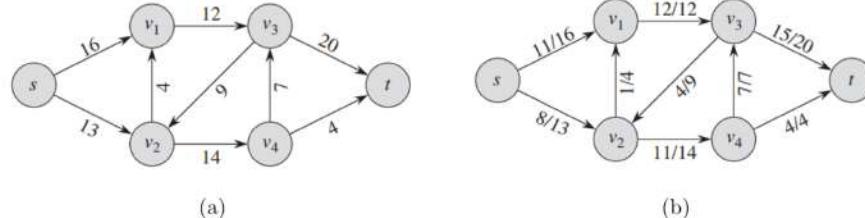


Figura 1: (a) O rețea de flux  $G = (V, E)$  ce modelează problema de transport între fabrică (vârful  $s$ ) și depozit (vârful  $t$ ). Ponderea fiecărui arc reprezintă capacitatea de transport pe zi pe ruta respectivă. (b) Un flux în rețeaua de flux de la (a) de valoare  $|f| = 19$ . Fiecare arc este etichetat sub forma  $f(u, v)/c(u, v)$  (simbolul "/" este folosit doar pentru a separa fluxul de capacitate).

- conservarea fluxului:  $\forall u \in V \setminus \{s, t\}, \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$  (fluxul nu se acumulează sau pierde într-un vârf).

Dacă  $(u, v) \notin E \Rightarrow f(u, v) = 0$ . Valoarea pozitivă  $f(u, v)$  se numește fluxul de la  $u$  la  $v$ , valoarea  $|f|$  a fluxului  $f$  se definește ca

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s).$$

adică diferențele între valoarea totală a fluxului ce iese minus valoarea totală a fluxului ce intră, notația  $|.|$  reprezintă valoarea fluxului și nu modul sau cardinal.

Pentru o rețea de flux

$$\sum_{v \in V} f(v, s) = 0$$

nu există arce ce intră în sursa  $s$ . În problema fluxului maxim se dă o rețea de flux  $G$  cu vârfurile  $s$  și  $t$  și se cere să se găsească un flux maxim.

**Exemplu** Se modelează problema de transport din figura 1a. O companie deține o fabrică (sursa  $s$ ) în Timișoara ce produce mingi de tenis și un depozit (destinația  $t$ ) în Cluj.

Compania închiriază spațiu pe camioane de la o altă firmă pentru a transporta mingile de tenis. Camioanele călătoresc pe rute prestabilite (arce) între orașe (vârfuri)  $\rightarrow$  capacitate limitată de transport. Compania poate expedia maxim  $c(u, v)$  mingi pe zi între două orașe. Trebuie să se determine numărul maxim de lăzi ce pot fi transportate pe zi și să se producă doar atât. Nu contează durata transportului, ce contează este ca  $p$  lăzi să iasă din fabrică pe zi și  $p$  lăzi să ajungă în depozit pe zi. Problema se poate modela cu o rețea de transport (figura 1b), capacitatea arcelor trebuie respectată altfel lăzile să acumuleze în orașele intermediare.

**Arce antiparalele** De exemplu, compania de transport oferă posibilitatea închirierii de spațiu pentru 10 lăzi pe ruta  $v_1 \rightsquigarrow v_2$  (situație prezentată în figura 2).

Rețeaua încalcă constrângerea/presupunerea inițială  $(v_1, v_2) \in E \rightarrow (v_2, v_1) \notin E$  (figura 2a). Arcele  $(v_1, v_2)$  și  $(v_2, v_1)$  se numesc arce antiparalele. Dacă se vrea modelarea problemei ca o rețea de flux trebuie să se transforme rețeaua: se alege un arc antiparalel (de ex.  $(v_1, v_2)$  și se împarte în două  $(v_1, v')$  și  $(v', v_2)$  ambele cu aceeași capacitate  $c(v_1, v_2) = c(v_1, v') = c(v', v_2)$  (figura 2b).

## VII. Rețele de flux

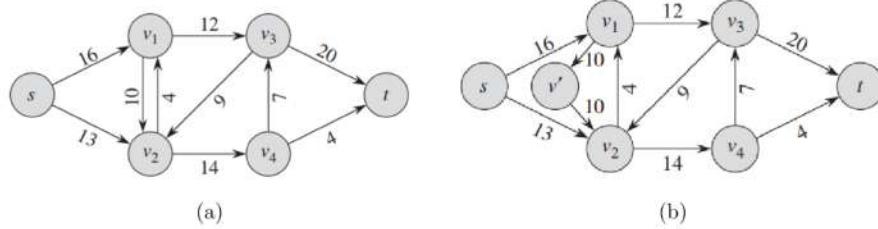


Figura 2: Conversia unei rețele de flux ce conține arce paralele într-o rețea fără astfel de arce. (a) O rețea de flux  $G = (V, E)$  ce conține arcele paralele  $(v_1, v_2)$  și  $(v_2, v_1)$ . (b) Rețeaua de flux fără arce paralele obținută prin adăugarea vârfului  $v'$ .

**Rețele cu noduri sursă și destinație multiple** De exemplu, compania producătoare de mingi de tenis poate avea mai multe fabrici și depozite:

- un set de  $m$  surse  $\{s_1, s_2, \dots, s_m\}$ ;
- un set de  $n$  destinații  $\{t_1, t_2, \dots, t_n\}$ .

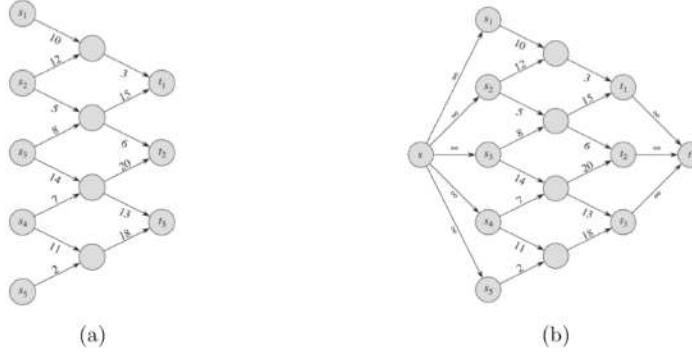


Figura 3: Rețea de flux cu surse și destinații multiple.

Pentru a rezolva problema se adaugă un vârf sursă, **supersursa**  $s$ , și arcele  $(s, s_i)$  cu capacitatele  $c(s, s_i) = \infty, \forall i = 1, \dots, m$ . Se adaugă și un vârf destinație, **superdestinație**  $t$ , și arcele  $(t_i, t)$  cu capacitatele  $c(t_i, t) = \infty, \forall i = 1, \dots, n$ .  $s$  oferă atât flux cât e necesar pentru fiecare  $s_i$  și  $t$  consumă atât flux cât primește de la fiecare  $t_i$ , situația este ilustrată de figura 3.

## 7.2 Metoda Ford-Fulkerson

Aceasta este o metodă care permite rezolvarea problemei de flux maxim, există mai multe implementări și timpi de rulare diferiți. Metoda Ford-Fulkerson depinde de 3 concepte:

- rețea reziduală;
- drum de creștere - cale reziduală;
- tăieturi.

## VII. Rețele de flux

---

Metoda Ford-Fulkerson crește iterativ valoarea fluxului. Pornește cu  $f(u, v) = 0, \forall u, v \in V$  și în fiecare pas se crește fluxul prin găsirea unui drum "de creștere" (cale reziduală sau drum rezidual) într-o rețea reziduală  $G_f$  asociată lui  $G$ . După ce se cunosc arcele unei căi reziduale în  $G_f$ , se pot identifica arce în  $G$  pentru care se poate schimba fluxul astfel încât să crească valoarea fluxului total. În fiecare pas se poate ca fluxul să și descrească. Acești pași se repetă atât timp cât există drumuri reziduale în rețeaua reziduală.

METODA\_FORD\_FULKERSON( $G, s, t$ )

- 1: inițializare flux cu 0
- 2: **while** există o cale reziduală  $p$  în graful rezidual  $G_f$  **do**
- 3:     crește fluxul  $f$  de pe  $p$
- 4: **return**  $f$

### 7.3 Rețea reziduală/Graf rezidual

**Intuitiv** fie o rețea de flux  $G = (V, E)$  și fluxul  $f$ , rețeaua reziduală  $G_f$  e formată din arce a căror capacitate arată cum pot schimba fluxul arcelor din  $G \implies$  un arc din rețeaua de flux admite un flux suplimentar egal cu capacitatea legăturii minus fluxul curent de pe arc.

Dacă această valoare este pozitivă se trasează arcul în  $G_f$  cu "capacitatea reziduală"  $c_f(u, v) = c(u, v) - f(u, v)$ . Singurele arce din  $G$  ce se regăsesc în  $G_f$  sunt cele ce admit mai mult flux.

$G_f$  poate conține și arce ce nu se regăsesc în  $G$ . Algoritmul poate ajunge în situația în care trebuie să scadă fluxul unui arc pentru a maximiza fluxul total din rețea. Pentru a reprezenta o posibilă descreștere a fluxului  $f(u, v)$  pentru un arc din  $G$  se adaugă un arc  $(v, u)$  în  $G_f$  cu capacitate reziduală  $c_f(v, u) = f(u, v)$ .

În rețeaua reziduală se pot trasa arce ce admit flux în direcția opusă lui  $(u, v)$  și cel mult anulând fluxul de pe  $(u, v)$ . Se poate trimite înapoi flux  $\iff$  se reduce fluxul de pe un arc (operație necesară).

**Formal** fie rețeaua de flux  $G = (V, E)$  cu  $s$  și  $t$ . Fie  $f$  un flux în  $G$  și perechea de vârfuri  $u, v \in V$ . Definim capacitatea reziduală  $c_f(u, v)$ :

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v), & \text{dacă } (u, v) \in E, \\ f(v, u), & \text{dacă } (v, u) \in E, \\ 0, & \text{în rest.} \end{cases}$$

Deoarece, dacă  $(u, v) \in E$  se presupune că  $(v, u) \notin E \implies$  doar un caz apare pentru fiecare arc.

De exemplu:  $c(u, v) = 16$  și  $f(u, v) = 11 \implies$  se poate crește  $f(u, v)$  cu până la  $c_f(u, v) = 5$  unități până să se încalce constrângerea capacitații arcului. De asemenea se vrea ca algoritmul să poată întoarce max 11 unități de flux de la  $u$  la  $v \implies c_f(v, u) = 11$ .

**Definiție 7.3.1** (Graf rezidual/Rețea reziduală).

fie o rețea de flux  $G = (V, E)$  și fluxul  $f$  prin  $G$ , numim rețea reziduală a lui  $G$  (sau graf rezidual al lui  $G$ ), indusă de  $f$ , o rețea de flux  $G_f = (V, E_f)$  unde:

$$E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}.$$

Fiecare arc din rețeaua reziduală, arc rezidual, poate admite un flux pozitiv. O cale reziduală este un drum  $s \rightsquigarrow t$  în  $G_f$  și  $c_f(u, v)$  este capacitatea reziduală. Arcele din  $E_f$  sunt cele din  $E$  sau inverse

$$|E_f| \leq 2|E|.$$

## VII. Rețele de flux

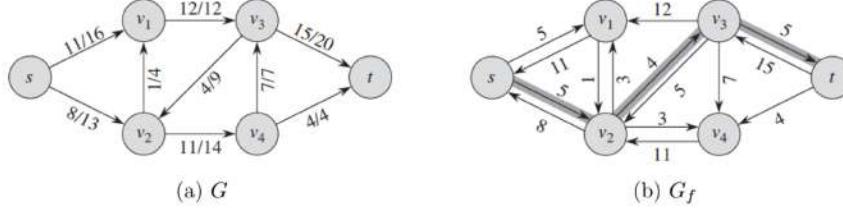


Figura 4: Rețea de flux și graful rezidual.

$G_f$  e similară cu o rețea de flux, cu capacitați date de  $c_f$ ,  $G_f$  nu satisfacă definiția dată pentru o rețea de flux deoarece poate conține arcele  $(u, v)$  și  $(v, u)$  (singura diferență dintre  $G$  și  $G_f$ ).

Un flux în rețeaua reziduală  $G_f$  dă indicații unde se pot adăuga unități de flux în rețeaua de flux originală  $G$ . Figura 4 prezintă o rețea de flux și graful rezidual asociat.

**Definiție 7.3.2** (îmbunătățirea fluxului).

Dacă  $f$  este un flux în  $G$  și  $f'$  este un flux în  $G_f$ , definim  $f \uparrow f'$  îmbunătățirea fluxului  $f$  de  $f'$  ca o funcție de la  $V \times V$  la  $\mathbb{R}$  definită de

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u), & \text{dacă } (u, v) \in E \\ 0, & \text{în rest} \end{cases}$$

Intuitiv se urmărește definiția rețelei reziduale:

- se mărește fluxul pe  $(u, v)$  cu  $f'(u, v)$  dar se scade cu  $f'(v, u)$  deoarece fluxul de pe arcul  $(v, u)$  în  $G_f$  înseamnă descreșterea fluxului în rețeaua originală;
- împingerea fluxului pe arcul invers se mai numește anulare.

De exemplu dacă se trimit 5 lazi de la  $u \rightarrow v$ , 3 lazi de la  $v \rightarrow u$  din perspectiva rezultatului final se trimit doar 2 lazi de la  $u \rightarrow v$  și 0 de la  $v \rightarrow u$ .

**Lema 7.1.**

Fie  $G = (V, E)$  o rețea de flux cu  $s$  și  $t$  și  $f$  un flux în  $G$ . Fie  $G_f$  rețeaua reziduală a lui  $G$  indusă de  $f$  și fie  $f'$  un flux în  $G_f$ . Funcția  $f \uparrow f'$  este un flux în  $G$  cu valoarea  $|f \uparrow f'| = |f| + |f'|$ .

#### 7.4 Creșterea fluxului pe un drum

Fie o rețea de flux  $G = (V, E)$  și un flux  $f$ , o cale reziduală (care îmbunătățește)  $p$  este un drum de la  $s$  la  $t$  în graful rezidual  $G_f$ . Din definiția grafului rezidual, se poate crește fluxul pe un arc  $(u, v)$  de pe un "drum mai bun"  $p$  cu  $\max c_f(u, v)$  fără a încălca constrângerea capacității deoarece capacitatea minimă a drumului de creștere este capacitatea arcului  $c_f(v_2, v_3) = 4$ .

Drumul reprezentat de arcele îngroșate din figura 4b este un drum care îmbunătățește fluxul, se poate crește fluxul de pe fiecare arc din  $p$  până la 4 unități fără a încălca constrângerea capacității deoarece capacitatea minimă a drumului de creștere este capacitatea arcului  $c_f(v_2, v_3) = 4$ .

Capacitate reziduală este valoarea maximă cu care se poate îmbunătăți fluxul pe fiecare arc din  $p$ , este dată de:

$$c_f(p) = \min\{c_f(u, v) | (u, v) \in p\}.$$

**Lema 7.2.**

Fie  $G = (V, E)$  o rețea de flux, fie  $f$  un flux în  $G$  și  $p$  o cale reziduală,  $p \in G_f$ . Definim funcția  $f_p : V \times V \rightarrow \mathbb{R}$

$$f_p(u, v) = \begin{cases} c_f(p), & \text{dacă } (u, v) \in p \\ 0, & \text{în rest} \end{cases}$$

## VII. Rețele de flux

---

### Corolar 7.3.

Fie  $G = (V, E)$  o rețea de flux și  $f$  un flux în  $G$  și  $p$  în  $G_f$ . Fie  $f_p$  definit de Lema (7.2) și presupunem că se îmbunătățește  $f$  cu  $f_p$ . Atunci funcția  $f \uparrow f_p$  este un flux în  $G$  cu valoarea  $|f \uparrow f_p| = |f| + |f_p| > |f|$ .

### 7.5 Tăieturi și fluxuri în rețele

Metoda Ford-Fulkerson îmbunătățește iterativ fluxul până când găsește fluxul maxim. Cum știm că am găsit fluxul maxim? Fluxul e maximal dacă graful rezidual nu conține un drum care poate îmbunătăți fluxul.

O tăietură  $(S, T)$  a rețelei de flux  $G = (V, E)$  este o partiție a lui  $V$  în  $S$  și  $T = V \setminus S$  astfel încât  $s \in S$  și  $t \in T$ . Dacă  $f$  este un flux atunci fluxul net  $f(S, T)$  peste tăietura  $(S, T)$  este definit de

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

iar capacitatea tăieturii  $(S, T)$  este

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).$$

Figura 5 prezintă o tăietură în graf.

### Lema 7.4.

Fie  $f$  un flux într-o rețea de flux  $G$  și  $(S, T)$  o tăietură în  $G$ , fluxul prin tăietura  $(S, T)$  este fluxul prin rețea  $f(S, T) = |f|$ .

### Teorema 7.5 (Teorema flux maxim, tăietura minimă).

Fie  $G = (V, E)$  o rețea de flux. Următoarele afirmații sunt echivalente:

1.  $f$  este o funcție de flux în  $G_f$  astfel încât  $|f|$  este flux maxim total în  $G$ ;
2. rețeaua reziduală  $G_f$  nu are căi reziduale (drumuri  $s \rightsquigarrow t$ );
3. există o tăietură  $(S, T)$  a lui  $G$  astfel încât  $|f| = c(S, T)$ .

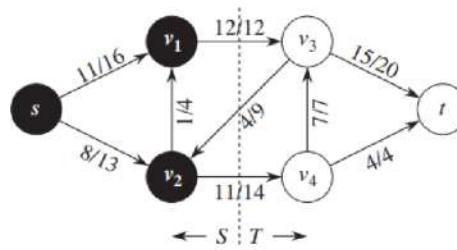


Figura 5: Tăietură într-o rețea de flux.

---

VII. Rețele de flux

### 7.6 Algoritmul Ford-Fulkerson

Timpul de rulare depinde de calculul căii reziduale, dacă se utilizează algoritmul *BFS* pentru a determina calea reziduală  $p \implies$  algoritmul rulează în timp polinomial.

Algoritmul *Ford-Fulkerson* este (atributul  $(u, v).f$  indică fluxul arcului):

```
FORD_FULKERSON(G, s, t)
1: for $(u, v) \in E$ do
2: $(u, v).f = 0$
3: while există o cale reziduală p de la s la t în graful rezidual G_f do
4: $c_f(p) = \min\{c_f(u, v) | (u, v) \in p\}$
5: for fiecare arc $(u, v) \in p$ do
6: if $(u, v) \in E$ then
7: $(u, v).f = (u, v).f + c_f(p)$
8: else
9: $(v, u).f = (v, u).f - c_f(p)$
10: return f
```

Fiecare iterație a buclei *while* durează  $O(V + E') = O(E)$  dacă folosim *DFS* sau *BFS* pentru a determina un drum rezidual. Bucă *while* se execută de cel mult  $|f^*|$  ori,  $f^*$  - fluxul maxim în rețea. Complexitatea în timp pentru Ford-Fulkerson este  $O(E|f^*|)$  (dacă se folosește *DFS* pentru a determina un drum rezidual în graful rezidual).

Figura 6 prezintă iterațiile algoritmului *Ford-Fulkerson* (liniile 3 – 10).

## VII. Rețele de flux

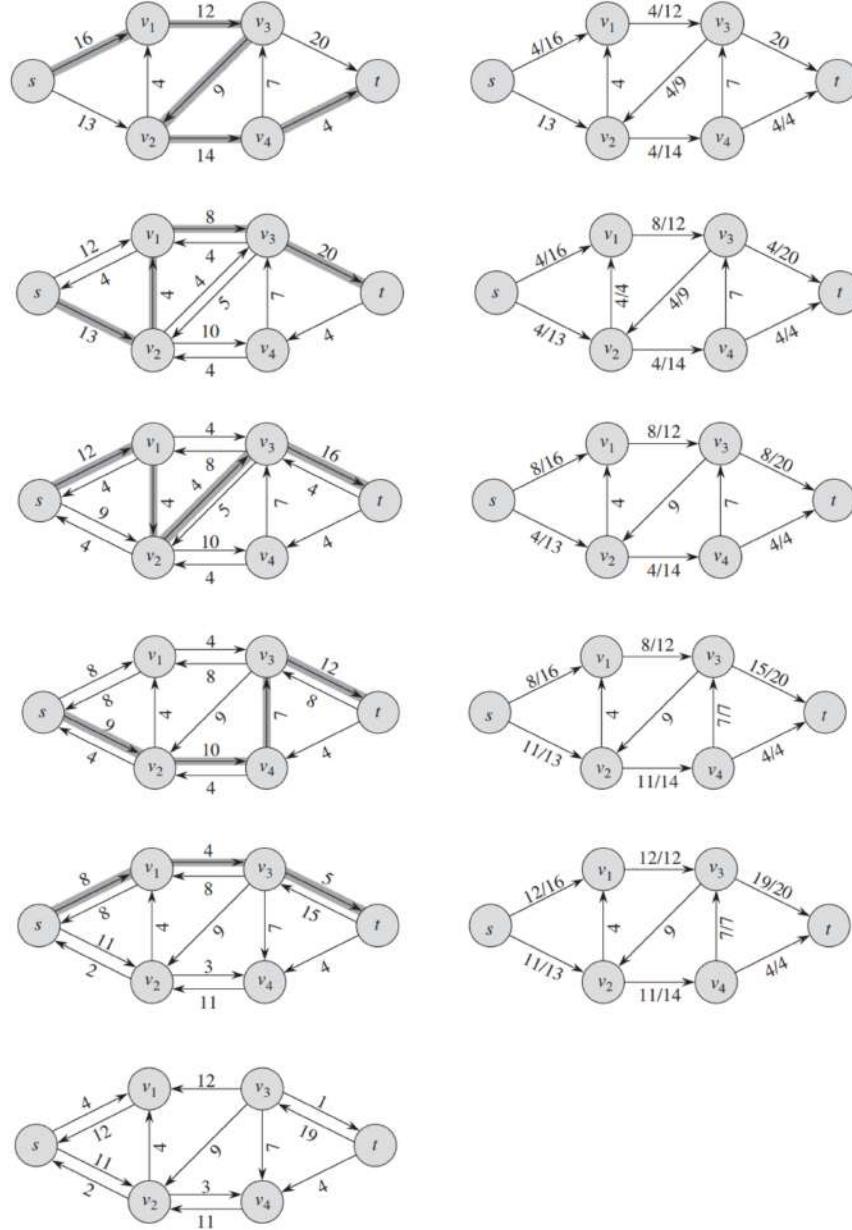


Figura 6: Pașii urmăți de algoritmul Ford-Fulkerson. Grafurile din partea stângă reprezintă rețeaua reziduală iar grafurile din partea dreaptă prezintă fluxul obținut în urma îmbunătățirilor descoperite în graful rezidual.

## Suport/Transcript curs algoritmica grafurilor

### VIII. Rețele de flux (II) - Algoritmi de flux maxim

#### 8.1 Algoritmul Edmonds-Karp

Algoritmul Ford-Fulkerson se poate îmbunătăți, se aplică Ford-Fulkerson dar calea reziduală în  $G_f$  este determinată folosind *BFS*. Se alege calea reziduală ca drumul minim  $s \rightsquigarrow t$  în  $G_f$  cu ponderea arcurilor 1. Algoritmul astfel implementat se numește Edmonds-Karp.

Timpul de rulare este  $O(VE^2)$ .

Fie  $\delta_f(u, v)$  drumul de cost minim de la  $u$  la  $v$  în  $G_f$  cu ponderea 1 pentru fiecare arc.

##### Lema 8.1.

*Dacă algoritmul Edmonds-Karp este rulat pe o rețea de flux  $G = (V, E)$  cu vârfurile sursă  $s$  și destinație  $t$ , pentru toate vârfurile  $v \in V \setminus \{s, t\}$ , drumul minim  $\delta_f(s, v)$  în graful rezidual  $G_f$  crește monoton cu fiecare îmbunătățire de flux.*

Următoarea teoremă limitează superior numărul iterărilor algoritmului.

##### Teorema 8.2.

*Dacă algoritmul Edmonds-Karp este rulat pe o rețea de flux  $G = (V, E)$  cu vârfurile sursă  $s$  și destinație  $t$ , numărul total de îmbunătățiri ale fluxului găsite de algoritm este  $O(VE^2)$ .*

#### 8.2 Algoritmul de pompare de preflux (push-relabel sau preflow-push)

Algoritmul face parte dintr-o clasa de algoritmi care încep calculele pe baza unui preflux existent în rețea, urmând apoi distribuirea prefluxului pe arce astfel încât, în final, prefluxul să îndeplinească restricțiile unui flux și să fie maxim.

Funcționarea algoritmului este similară cu curgerea lichidelor printr-un sistem de conducte de diverse capacitați care leagă puncte (vârfuri) aflate la diverse înălțimi. Inițial vârful sursă  $s$  al rețelei este cel mai înalt, celelalte vârfuri fiind la înălțimea 0. Destinația  $t$  rămâne în permanență la înălțimea 0. Prefluxul este inițializat încărcând la capacitate maximă toate conductele ce pornesc din  $s$ . În cursul funcționării, se poate întâmpla ca fluxul (lichidul) să strâns la un vârf  $u$  din conductele ce alimentează vârful să depășească posibilitatea de eliminare prin conductele de scurgere la care este conectat vârful (restricția de conservare de flux nu mai este îndeplinită). Excesul de flux este stocat într-un rezervor  $u.e$  al nodului, cu capacitate nelimitată teoretic. Pentru a echilibra rețeaua și a elimina supraîncărcarea vâfurilor, sunt efectuate două operații de baza:

1. Mărirea înălțimii unui nod. Atunci când un nod supraîncărcat  $u$  are o conductă de scurgere orientată spre un vecin  $v$ , care nu este încărcată la capacitate maximă,  $u$  este înălțat mai sus decât  $v$ , astfel încât fluxul să poată curge din  $u$  în  $v$  prin conducta  $(u, v)$ .

---

VIII. Rețele de flux (II) - Algoritmi de flux maxim

---

2. Pomparea fluxului unui nod. Un nod  $u$  supraîncărcat, aflat la o înălțime mai mare decât un vecin  $v$  și conectat cu  $v$  printr-o conductă subîncărcată, pompează flux din rezervorul  $u.e$  spre  $v$  prin conducta  $(u, v)$ .

Treptat, înălțimile nodurilor cresc monoton și pot depăși înălțimea sursei  $s$ . În acest moment fluxul în exces din rețea se pompează sursei. Numărul operațiilor de mărire a înălțimii vârfurilor și de pompare a fluxului este limitat, iar atunci când nu mai este posibilă nici o operație de acest fel, fluxul din rețea îndeplinește restricțiile impuse și este maxim.

### Operații de bază

Fie  $G = (V, E)$  un graf orientat cu  $n$  vârfuri și  $m$  arce, care desemnează o rețea de flux,  $s$  este vârful sursă și  $t$  este vârful destinație iar  $c : V \times V \rightarrow \mathbb{R}$  este capacitatea arcelor, astfel încât  $c(u, v) \geq 0$  pentru orice arc  $(u, v) \in E$ , iar  $c(u, v) = 0$  dacă  $(u, v) \notin E$ .

Se notează cu  $G_f = (V, E_f)$  graful rezidual al rețelei  $G$ , iar  $c_f(u, v)$  este capacitatea reziduală a arcului  $(u, v)$ .

#### Definiție 8.2.1.

Se numește preflux într-o rețea  $G = (V, E)$  o funcție  $f : V \times V \rightarrow \mathbb{R}$  asociată rețelei  $G$ , astfel încât sunt satisfăcute restricțiile:

1.  $f(u, v) \leq c(u, v) \quad \forall (u, v) \in E \quad$  - respectare capacitate arc;
2.  $f(u, v) = -f(v, u) \quad \forall u \in V \text{ și } v \in V \quad$  - simetrie preflux;
3.  $\sum_{v \in V} f(u, v) \geq 0 \quad \forall u \in V \setminus \{s\} \quad$  - supraîncărcare pozitivă.

#### Definiție 8.2.2.

Se numește supraîncărcare a unui vârf cantitatea

$$u.e = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0.$$

Dacă  $u.e > 0$  spunem că vârful  $u \in V \setminus \{s\}$  este supraîncărcat.

$u.e$  arată fluxul în exces stocat într-un rezervor virtual al vârfului.

În cursul procesului de calcul al fluxului maxim prin  $G$  înălțimea vârfurilor crește treptat, conform unei funcții de înălțime.

#### Definiție 8.2.3.

O funcție  $h : V \rightarrow \mathbb{N}$  este o funcție de înălțime pentru o rețea de flux  $G = (V, E)$  și satisfac următoarele restricții:

- $h(s) = |V|$ ;
- $h(t) = 0$ ;
- $u.h \leq v.h + 1$  pentru orice arc rezidual  $(u, v) \in E_f$ .

#### Lema 8.3.

Fie  $G = (V, E)$  o rețea de flux cu fluxul  $f$ , iar  $h : V \rightarrow \mathbb{N}$  o funcție de înălțime a vârfurilor rețelei. Dacă pentru orice pereche de vârfuri  $u, v \in V$  avem  $u.h > v.h + 1$ , atunci  $(u, v) \notin E_f$  (arcul nu este rezidual = arc în rețeaua reziduală  $G_f$ ).

Cele două operații esențiale ale algoritmului sunt de *pompare* a excedentului de flux al unui vârf și de *înălțare* a unui vârf. Cele două operații sunt aplicabile doar vârfurilor  $V \setminus \{s, t\}$  și doar dacă sunt satisfăcute anumite condiții.

---

VIII. Rețele de flux (II) - Algoritmi de flux maxim

**Pompare a excedentului de flux** are loc doar dacă diferența de înălțime dintre  $u$  și  $v$  este exact 1, ( $u.h = v.h + 1$ ) și dacă există exces de flux  $c_f(u, v) > 0$ . Dacă  $h$  este o funcție de înălțime și  $u.h > v.h + 1$  arcul rezidual  $(u, v)$  nu există și pomparea nu are sens. Procedura de pompare este:

POMPARE( $u, v$ )

- 1: \ conditie de aplicare:  $u \notin \{s, t\} \wedge u.e > 0 \wedge c_f(u, v) > 0 \wedge u.h = v.h + 1$
- 2: \ actiune: pompează cantitatea de flux  $\Delta_f(u, v) = \min(u.e, c_f(u, v))$
- 3:  $\Delta_f(u, v) = \min(u.e, c_f(u, v))$
- 4: if  $(u, v) \in E$  then
- 5:      $(u, v).f = (u, v).f + \Delta_f(u, v)$
- 6: else
- 7:      $(v, u).f = (v, u).f - \Delta_f(u, v)$
- 8:  $u.e = u.e - \Delta_f(u, v)$
- 9:  $v.e = v.e + \Delta_f(u, v)$

Deoarece vârful  $u$  are un exces de flux și capacitatea arcului  $(u, v)$  este pozitivă se poate crește valoarea fluxului de la  $u$  la  $v$  cu valoarea  $\Delta_f(u, v) = \min(u.e, c_f(u, v))$  fără ca  $u.e$  să devină negativă sau capacitatea  $c(u, v)$  să fie depășită. Cantitatea de flux  $\Delta_f$  pompată de arcul rezidual  $(u, v)$  asigură respectarea pozitivității excesului de flux din  $u$  în  $v$  și a capacitatii arcului  $(u, v)$  din  $G$ . Linia 3 determină valoarea lui  $\Delta_f$ , liniile 4-6 actualizează fluxul  $f$  (linia 5 crește fluxul pe arcul  $(u, v)$  și linia 6 descrește fluxul deoarece arcul  $(v, u)$  rezidual este inversul arcului din rețeaua de flux  $G$ ). Liniile 7-8 actualizează excesul de flux pentru vârfurile  $u$  și  $v$ .

Prefluxul  $f$  prin rețea continuă să-si păstreze proprietățile după aplicarea procedurii de pompare. Spunem că pomparea este saturată dacă după pompare  $c_f(u, v) = 0$ , arcul  $(u, v)$  este folosit la întreaga capacitate. Un arc saturat nu mai apare în rețeaua reziduală  $G_f$ . Alternativ pomparea este nesaturată dacă după pompare  $c_f(u, v) > 0$ , această situație are loc atunci când  $u.e < c_f(u, v)$ .

**Înălțarea unui vîrf** se aplică dacă  $u.e > 0$  și  $u.h \leq v.h$  pentru toate arcele  $(u, v) \in E_f$ . Altfel spus, se înălță vârful  $u$  dacă pentru vârful  $v$  pentru care există capacitate reziduală de la  $u$  la  $v$  fluxul nu poate fi pompat de la  $u$  la  $v$  deoarece  $v$  nu este la un nivel inferior lui  $u$ .

ÎNĂLȚARE( $u$ )

- 1: \ conditie de aplicare:  $u \notin \{s, t\} \wedge u.e > 0 \wedge [u.h \leq v.h] \forall v \in V, (u, v) \in E_f$
- 2: \ actiune: mărește înălțimea  $u.h$
- 3:  $u.h = 1 + \min\{v.h | (u, v) \in E_f\}$

Când se aplică procedura de înălțare  $E_f$  trebuie să conțină cel puțin un arc care pleacă din  $u$ . Aceasta reiese din :

$$u.e = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v).$$

Deoarece toate fluxurile sunt pozitive trebuie să existe un vârf  $v$  astfel încât  $(u, v).f > 0$ , dar  $c_f(u, v) > 0 \Rightarrow (u, v) \in E_f$ .

Cele două operații sunt aplicabile doar vârfurilor supraîncărcate din mulțimea  $V \setminus \{s, t\}$ . Lema 8.4 arată că pentru un vârf supraîncărcat  $u \in V \setminus \{s, t\}$  există cel puțin o operație de *pompare* sau *înălțare* care poate fi aplicată vârfului  $u$ .

**Lema 8.4.**

*Fie  $h$  o funcție de înălțime în rețeaua de flux  $G$ . Atunci, un nod supraîncărcat  $u \in V \setminus \{s, t\}$  poate participa fie într-o operație de pompare a prefluxului, fie într-o operație de înălțare a vârfului.*

---

VIII. Rețele de flux (II) - Algoritmi de flux maxim

---

Concluzia lemei 8.4 spune că în momentul în care cele două operații nu mai pot fi aplicate vârfurilor rețelei de flux  $G$ , în  $G$  nu mai există vârfuri supraîncărcate.

### **Algoritmul generic de pompare de flux**

```

INITIALIZARE_PREFLUX(G, s, t)
1: \\\ inițializare $f(u, v)$ și $h(u, v)$, $\forall u, v \in V$
2: for fiecare $v \in V$ do
3: $v.h = 0$
4: $v.e = 0$
5: for fiecare $(u, v) \in E$ do
6: $(u, v).f = 0$
7: $s.h = |V|$
8: for fiecare $v \in s.Adj$ do
9: $(s, v).f = c(s, v)$
10: $v.e = c(s, v)$
11: $s.e = s.e - c(s, v)$

```

```

POMPARE_PREFLUX(G, s, t)
1: INITIALIZARE_PREFLUX(G, s, t)
2: while TRUE do
3: if $\exists u \notin \{s, t\} \wedge u.e > 0 \wedge c_f(u, v) > 0 \wedge u.h = v.h + 1$ then
4: POMPARE(u, v)
5: continue
6: if $\exists u \notin \{s, t\} \wedge u.e > 0 \wedge [u.h \leq v.h] \forall v \in V, (u, v) \in E_f$ then
7: INALTARE(u)
8: continue
9: break

```

Procedura de inițializare creează un preflux inițial  $f$  definit de:

$$(u, v).f = \begin{cases} c(u, v), & \text{dacă } u = s, \\ 0, & \text{în rest.} \end{cases}$$

Totodată algoritmul începe cu o funcție de înălțime  $h$  definită de:

$$u.h = \begin{cases} |V|, & \text{dacă } u = s, \\ 0, & \text{în rest.} \end{cases}$$

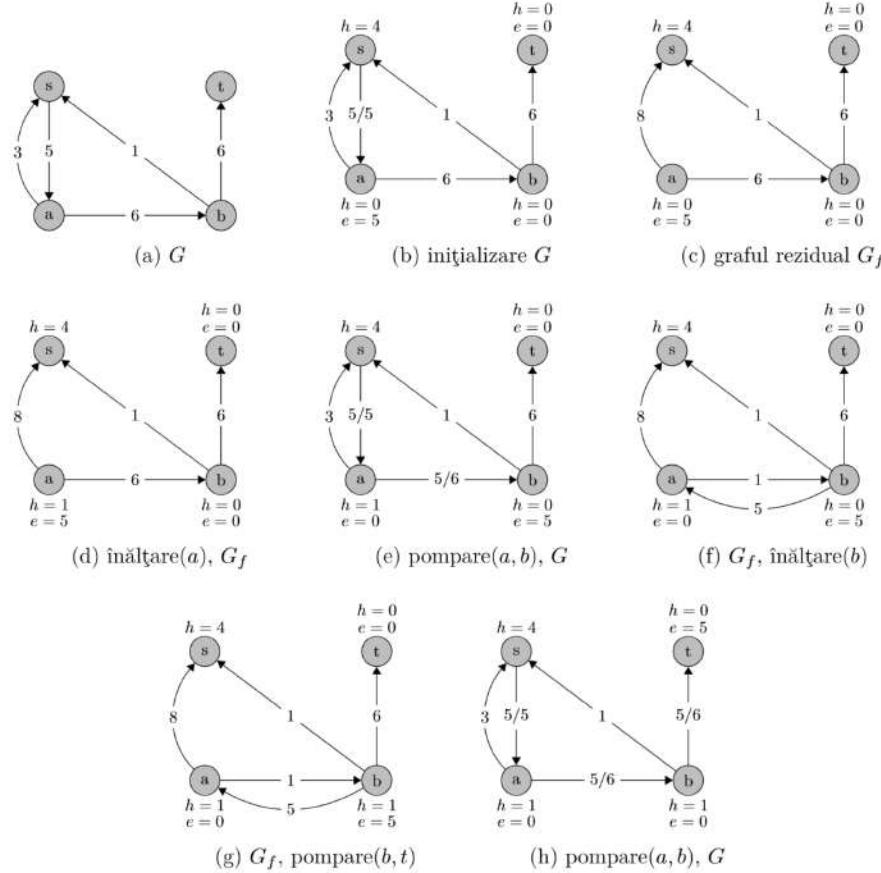
Algoritmul se termină atunci când pentru orice nod din  $u \in V \setminus \{s, t\}$  avem  $u.e = 0$ . La terminare prefluxul din rețea este fluxul maxim.

Complexitatea în timp a algoritmului este  $O(V^2E)$ .

Figura 1 prezintă un exemplu pentru algoritmul de pompare-preflux. Pentru graful original (figura 1a) sunt prezentate atributele înălțime și exces pentru fiecare vârf, pasul de inițializare, grafurile reziduale rezultate și efectul procedurilor de înălțare și pompare (figurile 1b-1h).

O versiune mai rapidă este algoritmul de **pompare-topologică** (relabel-to-front), având complexitatea în timp  $O(V^3)$ .

## VIII. Rețele de flux (II) - Algoritmi de flux maxim

Figura 1: Exemplu: pașii urmați de algoritmul *pompare-preflux*.

## 8.3 Algoritmul de pompare-topologică (relabel-to-front)

Față de algoritmul de pompare\_prelux, pomparea topologică impune o disciplină strictă de secvențiere a operațiilor elementare de pompare a prefluxului prin rețeaua  $G$  și de înălțare a vârfurilor rețelei. Această disciplină este conformă următorilor pași care, cu excepția initializărilor, constituie partea centrală a algoritmului de pompare topologică.

1. Vârful curent prelucrat este selectat dintr-o listă  $L(V)$  care conține toate vârfurile din  $V \setminus \{s, t\}$ . Inițial, ordinea vârfurilor din listă este oarecare.
2. Prelucrarea vârfului curent, cel de la vârful listei  $L(V)$ , fie el  $u = \text{head}(L(V))$ , urmărește eliminare completă a excesului de preflux  $u.e$ , dacă  $u.e > 0$ . Prelucrarea constă în vizitarea vecinilor lui  $u$  cu intenția aplicării operației *POMPARE*( $u, v$ ) și în mărirea treptată a înălțimii lui  $u$ , astfel încât operațiile de pompare să poată fi efectuate. Această prelucrare, numită *DESCĂRCARE*, este principala operație executată de pomparea topologică.
3. Dacă *DESCĂRCARE*( $u$ ) pentru vârful curent din  $L(V)$  a condus la eliminarea efectivă a supraîncărcării vârfului, dacă  $u.e > 0$  înainte de operație și  $u.e = 0$  după operație, atunci  $u$

---

VIII. Rețele de flux (II) - Algoritmi de flux maxim

---

este deplasat la începutul listei  $L(V)$ , iar procesul de selecție de la pasul (1) este reluat cu vârful succesor lui  $u$  din noua listă  $L(V)$ .

4. Repetarea pașilor (1-3) se termină atunci când parcurgând lista  $L(V)$  se ajunge la capătul ei. Această situație apare atunci când  $u.e = 0$  pentru orice vârf  $u$  din  $L(V)$ . În acest moment  $t.e$  desemnează fluxul total prin rețeaua  $G$  și este maxim.

POMPARE\_TOPOLOGICA( $G, s, t$ )

```

1: INITIALIZARE_PREFLUX(G,s,t)
2: $L = V \setminus \{s, t\}$
3: for fiecare $u \in V \setminus \{s, t\}$ do
4: $u.current = u.N.head$
5: $u = L.head$
6: while $u \neq NIL$ do
7: $\text{înălțime_veche} = u.h$
8: DESCARCARE(u)
9: if $u.h > \text{înălțime_veche}$ then
10: mută u în capul listei L
11: $u.next$
```

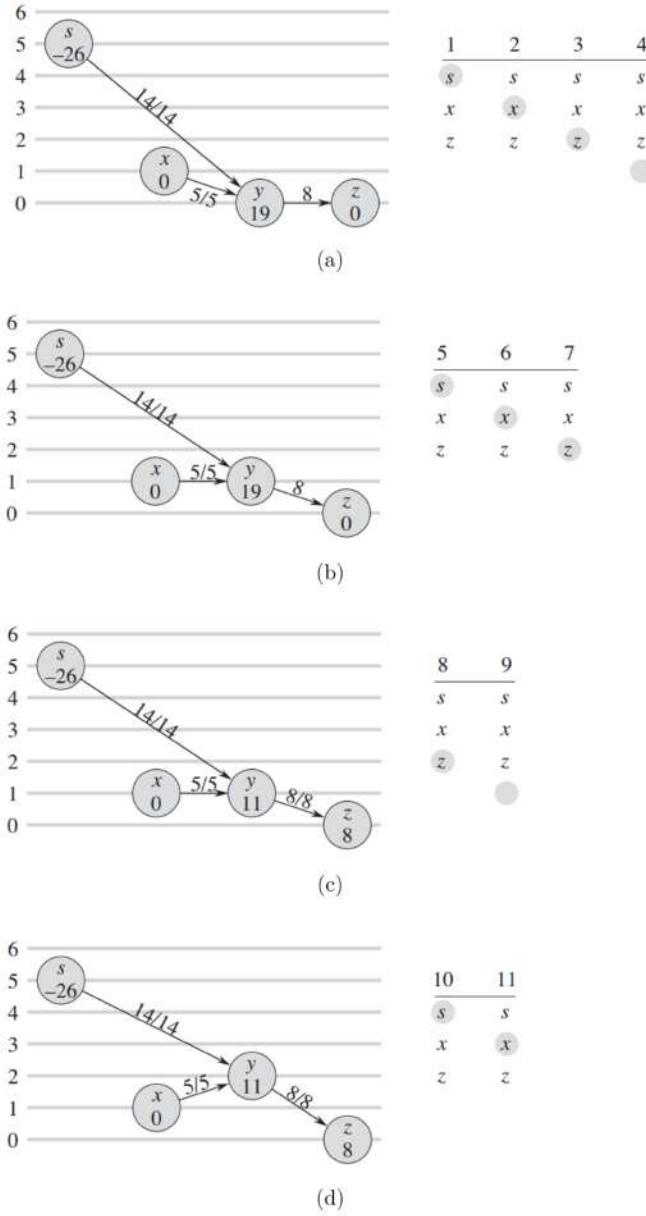
DESCARCARE( $u$ )

```

1: while $u.e > 0$ do
2: $v = u.current$
3: if $v == NIL$ then
4: INALTARE(u)
5: $u.current = u.N.head$
6: else if $c_f(u, v) > 0 \wedge u.h == v.h + 1$ then
7: POMPARE(u,v)
8: else
9: $u.current = u.urmatorul_vecin$
```

Figurile 2 și 3 prezintă un graf pentru care s-a aplicat procedura de  $DESCARCARE(u)$  pentru a scăpa de excesul de flux din vârful  $y$ . Este nevoie de 15 iterații ale buclei *while* din procedura  $DESCARCARE(u)$  pentru a pompa tot excesul din vârful  $y$ . Figurile prezintă doar vârfurile adiacente vârfului  $y$  și arcele ce leagă vârful  $y$ . Pentru figuri, numărul din interiorul vârfului reprezintă excesul de flux iar vârfurile sunt desenate pe nivele ce reprezintă înălțimea vârfului. Lista vecinilor vârfului  $y$  la începutul fiecărei iterații este ilustrată în dreapta fiecărei rețea de flux, numărul iterației este reprezentat de coloanele de pe linia 1. Inițial (figura 2a) vârful  $y$  are în exces 19 unități de flux ce trebuie pompatate ( $y.e = 19$ ) și vârful  $current$  în care se încercă pomparea este  $s$ , ( $y.current = s$ ). Iterațiile 1, 2 și 3 doar modifică atributul  $y.current$  deoarece nu există arce pe care se poate pompa excesul de flux. În iterația 4  $y.current = NIL$  vârful  $y$  este înălțat și  $y.current$  este resetat la începutul listei (figura 2b). Procesul continua și în iterația 7 când  $y.current = z$  se pot pompa 8 unități de flux pe arcul  $(y, z)$  în vârful  $z$ , deoarece în această iterație se pompează exces de flux atributul  $y.current$  nu se modifică. Figura 2c prezintă iterațiile 8 și 9 după care  $y.current = NIL \implies$  vârful  $y$  este înălțat. Procesul continua 2c-3c până când tot excesul de flux este pompat din vârful  $y$ .

## VIII. Rețele de flux (II) - Algoritmi de flux maxim

Figura 2: Pașii urmăți de procedura de DESCARCARE( $u$ ) (explicațiile se regăsesc în text).

## VIII. Rețele de flux (II) - Algoritmi de flux maxim

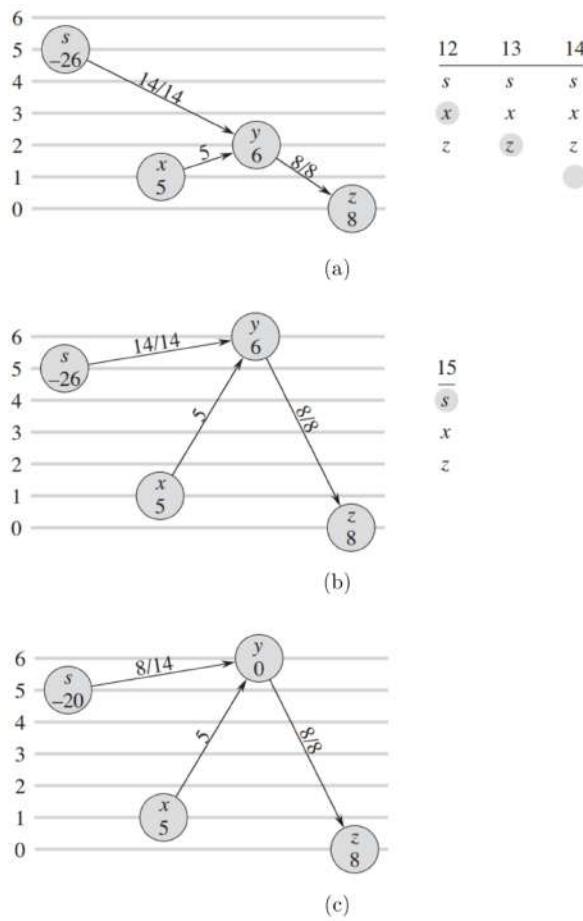


Figura 3: Pașii urmăți de procedura de DESCARCARE( $u$ ). Continuarea exemplului din figura 2 (explicațiile se regăsesc în text).

## Suport/Transcript curs algoritmica grafurilor

### IX-X. Grafuri planare, colorarea grafurilor, cuplaje

#### 9.1 Grafuri planare

**Definiție 9.1.1** (Grafuri planare).

un graf  $G = (V, E)$  este planar dacă poate fi desenat în plan astfel încât muchiile să nu se intersecteze decât în vîrfurile grafului. O astfel de desenare se numește **reprezentare planară** a lui  $G$ .

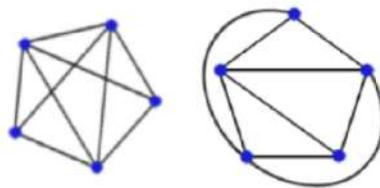


Figura 1: Graf planar.

O **regiune** a unei reprezentări planare a grafului este o porțiune din plan în care orice două vîrfuri pot fi unite cu o curbă care nu intersectează graful  $G$ .

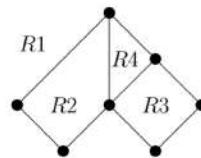


Figura 2: Regiunile unui graf planar.

Graful din figura 2 determină patru regiuni,  $R_1$  este regiunea exterioară.

Oricare regiune este delimitată de muchii, oricare muchie este în contact cu una sau două regiuni. O muchie **mărginește** o regiune  $R$  dacă este în contact cu  $R$  și cu altă regiune.

Pentru graful din figura 3:

- $e_1$  este în contact cu regiunea  $R_1$ ;
- $e_6$  este în contact cu regiunea  $R_2$ ;

## IX-X. Grafuri planare, colorare, cuplaje in grafuri

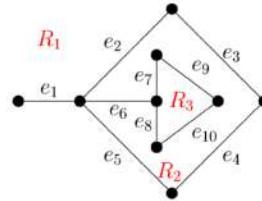


Figura 3: Regiunile unui graf planar.

- regiunea  $R_1$  este mărginită de  $e_2, e_3, e_4, e_5$ ;
- regiunea  $R_2$  este mărginită de  $e_2, e_3, e_4, e_5, e_7, e_8, e_9, e_{10}$ ;
- regiunea  $R_3$  este mărginită de  $e_7, e_8, e_9, e_{10}$ .

Gradul de mărginire  $b(R)$  al unei regiuni  $R$  este numărul de muchii care mărginesc  $R$ . Pentru graful din figura 3:  $b(R_1) = 4$ ,  $b(R_2) = 8$  și  $b(R_3) = 4$ .

**Teorema 9.1** (Formula lui Euler).

Dacă  $G = (V, E)$  este un graf planar conex cu  $n$  vârfuri,  $m$  muchii și  $r$  regiuni atunci:

$$n - m + r = 2.$$

*Demonstrație teorema 9.1.*

Inducție după  $m$ :

1.  $m = 0 \implies G = K_1; n = 1, m = 0, r = 1 \implies n - m + r = 2$ ;
2.  $G$  este un arbore, atunci  $m = n - 1$  și  $r = 1 \implies n - m + r = n - (n - 1) + 1 = 2$ ;
3.  $G$  este un arbore conex cu cel puțin un ciclu. Fie o muchie din ciclul respectiv și  $G' = G - e$ .  $G'$  este conex cu  $n$  vârfuri,  $m - 1$  muchii și  $r - 1$  regiuni  $\implies n - (m - 1) - (r - 1) = 2$ ,  $n - m + r = 2$  are loc și în acest caz.

□

---

 IX-X. Grafuri planare, colorare, cuplaje în grafuri

**Consecințe ale formulei lui Euler**

**Consecință 1**  $K_{3,3}$  nu este un graf planar

*Demonstrație.*

$K_{3,3}$  are  $n = 6$  și  $m = 9$ , dacă ar fi planar ar avea  $r = m - n + 2 = 5$  regiuni  $R_i$  ( $1 \leq i \leq 5$ ), fie  $C = \sum_{i=1}^5 b(R_i)$ .

Orice muchie mărginește două regiuni  $\Rightarrow C \leq 2m = 18$ .  $K_{3,3}$  este bipartit, nu conține  $K_3$  ca subgraf (cel mai scurt ciclu în  $K_{3,3}$  are lungimea 4), deci  $b(R_i) \geq 4$  pentru orice valoare a lui  $i$  și prin urmare  $C \geq 4 \cdot 5 = 20 \Rightarrow$  contradicție, deci  $K_{3,3}$  nu poate fi graf planar.  $\square$

**Consecință 2**

Dacă  $G$  este un graf planar cu  $n \geq 3$  vârfuri și  $m$  muchii atunci  $m \leq 3n - 6$ . Mai mult, dacă  $m = 3n - 6$  atunci  $b(R) = 3$  pentru orice regiune din graf.

*Demonstrație.*

Fie  $R_1, \dots, R_n$  regiunile lui  $G$  și  $C = \sum_{i=1}^n b(R_i)$ . Stîm că  $C \leq 2m$  și că  $C \geq 3r$ ,  $b(R_i) \geq 3$  pentru toate regiunile. Atunci

$$3r \leq 2m \Rightarrow 3(2 + m - n) \leq 2m \Rightarrow m \leq 3n - 6.$$

Dacă egalitatea are loc, atunci

$$3r = 2m \Rightarrow C = \sum_{i=1}^r b(R_i) = 3 \Rightarrow b(R_i) = 3$$

pentru toate regiunile (graf format din triunghiuri).  $\square$

**Consecință 3**  $K_5$  nu este graf planar

*Demonstrație.*

$K_5$  are  $n = 5$  vârfuri și  $m = 10$  muchii deci  $3n - 6 = 9 < 10 = m \Rightarrow K_5$  nu poate fi planar (Consecință 2).  $\square$

**Consecință 4**  $\delta(G) \leq 5$  pentru orice graf planar.

*Demonstrație.*

Presupunem că  $G = (V, E)$  este planar.

Dacă  $n \leq 6$  orice vârf are gradul mai mic sau egal cu 5  $\Rightarrow \delta(G) \leq 5$ .

Dacă  $n > 6$ , putem nota  $D = \sum_{v \in V} d(v)$ . Rezultă următoarele:  $D = 2m \leq 2(3n - 6) = 6n - 12$ .

Dacă  $\delta(G) \geq 6$  atunci  $D = \sum_{v \in V} d(v) \geq \sum_{v \in V} 6 = 6n \Rightarrow$  contradicție.

Deci  $\delta(G) \leq 5$  are loc.  $\square$

**Detectia unui graf planar**

**Teorema 9.2** (Kuratowski).

*Graful  $G = (V, E)$  este un graf planar dacă și numai dacă nu conține subdiviziuni ale lui  $K_{3,3}$  și ale lui  $K_5$ .*

Fie  $G = (V, E)$  un graf neorientat și  $(u, v) \in E$ . Putem defini următoarele:

## IX-X. Grafuri planare, colorare, cuplaje în grafuri

- o subdiviziune a lui  $(u, v)$  în  $G$  este o înlocuire a muchiei  $(u, v)$  din  $G$  cu un lanț de la  $u$  la  $v$  prin vârfuri intermedii noi;
- un graf  $H$  este o subdiviziune a unui graf  $G$  dacă  $H$  se poate obține din  $G$  printr-o secvență finită de subdiviziuni de muchii.



Figura 4: Graf subdiviziune.

Putem spune că un graf  $G$  conține un graf  $H$  dacă graful  $H$  se poate obține prin eliminarea de vârfuri și muchii din  $G$ .

**Observație**

- dacă  $H$  este subgraf al lui  $G$  atunci  $G$  conține  $H$ . Reciproca nu este adevărată ( $G$  conține  $H$  nu implică  $H$  este subgraf al lui  $G$ );
- $H$  este un subgraf al lui  $G$  doar dacă se poate obține din  $G$  prin eliminarea de vârfuri.

De exemplu, fie graful din figura 5 unde dacă se elimină muchiile  $(1, 7)$ ,  $(6, 7)$ ,  $(2, 3)$  și  $(4, 5)$  se obține un graf ce poate fi redus la  $K_{3,3}$ , graful original nu este planar.

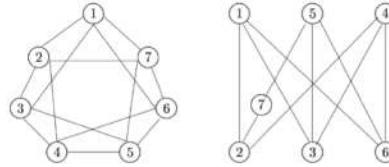


Figura 5: Exemplu Kuratowski.

Orice graf planar poate fi redesenat astfel încât muchiile sale sunt segmente de dreaptă, figura 6.

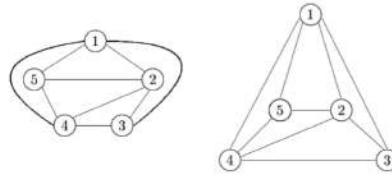


Figura 6: Exemplu graf planar.

## IX-X. Grafuri planare, colorare, cuplaje in grafuri

## 9.2 Colorarea grafurilor

**Definiție 9.2.1** (Colorarea vârfurilor).

O  $k$ -colorare a vârfurilor unui graf  $G = (V, E)$  este o funcție  $K : V \rightarrow \{1, 2, \dots, k\}$  astfel încât  $k(u) \neq k(v)$  dacă  $(u, v) \in E$ .

**Definiție 9.2.2** (Numărul cromatic).

Numărul cromatic  $\chi(G)$  al unui graf  $G$  este valoarea minimă a lui  $k$ ,  $k \in \mathbb{N}$ , pentru care există o  $k$ -colorare a lui  $G$ .

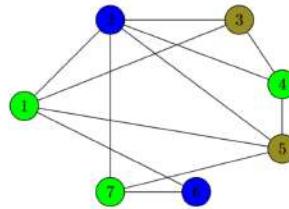


Figura 7: Exemplu de colorare a unui graf.

Pentru graful din figura 7 numărul minim de culori necesar pentru a colora graful este 3.  $k(1) = k(4) = k(7) =$  verde,  $k(2) = k(6) =$  albastru,  $k(3) = k(5) =$  maro.

Determinarea numărului cromatic  $\chi$  pentru un graf  $G$  este o problemă dificilă, o variantă pentru a determina numărul cromatic este de a calcula polinomul cromatic  $c_G(k)$  asociat grafului  $G$ .  $c_G(k)$  este numărul de  $k$ -colorări de vârfuri a grafului  $G$ ,  $\chi(G)$  reprezintă valoarea minimă a lui  $k$  pentru care  $c_G(k) > 0$ .

### Polinoame cromatice pentru grafuri speciale

#### Graful vid $E_n$

Graful vid conține  $n$  vârfuri și  $m = 0$  muchii. Având la dispoziție  $k$  culori, pentru fiecare vârf se pot alege oricare din cele  $k$  culori. Polinomul cromatic pentru graful vid este

$$c_{E_n}(k) = k^n$$

și numărul cromatic este

$$\chi(E_n) = 1.$$

#### Arbore de $n$ vârfuri $T_n$

Rădăcina arborelui poate fi colorată în  $k$  moduri, orice alt vârf poate fi colorat cu orice culoare diferită de cea a vârfului părinte. Polinomul cromatic pentru un arbore de  $n$  vârfuri este

$$c_{T_n}(k) = k(k-1)^{n-1}$$

și numărul cromatic pentru  $n > 1$  este

$$\chi(T_n) = 2.$$

## IX-X. Grafuri planare, colorare, cuplaje în grafuri

**Graf complet  $K_n$** 

Având la dispoziție  $k$  culori, un graf complet poate fi colorat în

$$c_{K_n} = k(k-1)\dots(k-n+1) = \frac{k!}{(k-n)!} = A_k^n$$

moduri și numărul cromatic este

$$\chi(K_n) = n.$$

**Calculul polinomului cromatic**

Pentru a determina polinomul cromatic al unui graf  $G = (V, E)$  neorientat există două metode recursive:

1.  $c_G(K) = c_{G-e}(k) - c_{G/e}(k)$
2.  $c_G(K) = c_{\bar{G}}(k) + c_{\bar{G}/e}(k)$

unde  $\bar{G} = G + e$ . Fie  $G = (V, E)$  un graf neorientat și  $e = (u, v)$  o muchie din  $E$ .  $G - e$  este graful obținut din  $G$  prin eliminarea muchiei  $e$ ,  $G/e$  este graful obținut din  $G$  în care se înlocuiesc vârfurile  $u$  și  $v$  cu un singur vârf care se învecinează cu vecinii lui  $u$  și ai lui  $v$ . De exemplu graful din figura 8a și muchia  $e = (3, 4)$ , figura 8b prezintă graful  $G - (3, 4)$  iar figura 8c prezintă graful  $G/(3, 4)$ .

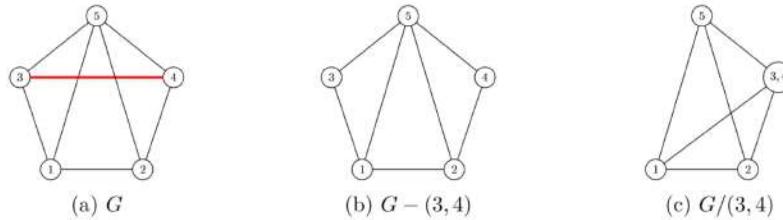


Figura 8: Exemplu determinare polinom cromatic.

Metoda  $c_G(K) = c_{G-e}(k) - c_{G/e}(k)$  presupune determinarea polinomului cromatic recursiv eliminând pe rând câte o muchie  $e \in E$  până când se obțin grafuri speciale  $E_n, T_n$  sau  $K_n$ .

Metoda  $c_G(K) = c_{\bar{G}}(k) + c_{\bar{G}/e}(k)$  presupune determinarea polinomului cromatic recursiv adăugând pe rând muchii  $e$  care lipsesc din graf până când se obțin grafuri speciale  $E_n, T_n$  sau  $K_n$  pentru care se cunoaște polinomul cromatic.

Dacă  $G = (V, E)$  este un graf neorientat cu  $n$  vârfuri și  $m$  muchii polinomul cromatic  $c_G(k)$  trebuie să îndeplinească următoarele proprietăți:

- are gradul  $n$ ;
- coeficientul lui  $k^n$  este 1;
- coeficientul lui  $k^{n-1}$  este  $-m$ .
- coeficienții săi au semne alternante;
- termenul liber este 0.

Algoritmic dacă se dorește colorarea unui graf se poate folosi metoda suboptimală  $COLORARE(G)$ . Algoritmul nu este optimal deoarece nu folosește  $\chi(F)$  culori pentru a colora graful  $G$ . Fie un graf  $G = (V, E)$  și funcția de colorare  $K : V \rightarrow \mathbb{N}$  astfel încât dacă  $e = (u, v) \in E$

---

IX-X. Grafuri planare, colorare, cuplaje in grafuri

---

atunci  $k(u) \neq k(v)$ , atributul  $k$  asociat unui vârf reprezintă culoarea asociată vârfului respectiv, algoritmul este:

```
COLORARE(G)
1: for $v \in V$ do
2: $v.k = 0$
3: $v_1.k = 1$
4: for $j = 2, j \leq n$ do
5: $v_j.k = \min(k \in \mathbb{N} | k > 0 \wedge u.k \neq k \forall u \in A_{v_j})$
```

O altă metodă de a colora un graf presupune o abordare backtracking. Fie  $C$  setul culorilor posibile, algoritmul  $COLORARE\_B(G)$  prezintă această abordare.

```
COLORARE_B(v)
1: for $x \in C$ do
2: if $SIGUR(v, x)$ then
3: $v.k = x$
4: if $v + 1 < |V|$ then
5: COLORARE_B($v+1$)
6: else
7: stop
```

```
SIGUR(v, x)
1: for $0 < i < n$ do
2: if $(a_{vi} = 1 \wedge x = i.k)$ then
3: return FALSE
4: return TRUE
```

O altă variantă de a colora un graf a fost introdusă de Kempe și presupune utilizarea unei stive unde sunt salvate vâfurile din graf ce au gradul mai mic decât numărul culorilor disponibile pentru a colora graful. Astfel vor fi colorate în graf vâfurile în ordine descrescătoare a gradului unui nod.

### 9.3 Cuplaje în grafuri

Fie un graf simplu neorientat  $G = (V, E)$ . Un cuplaj în  $G$  este o mulțime de muchii  $M$  în care nici o pereche de muchii nu are un vârf comun. Vâfurile adiacente la muchiile din  $M$  se numesc vâfururi saturate de  $M$  (sau  $M$ -saturate). Celelalte vâfururi se numesc  $M$ -nesaturate.

Figura 9 prezintă două cuplaje  $M_1$  și  $M_2$  pentru același graf  $G$ . Cuplajul  $M_1$  este format din  $M_1 = \{(1,2), (4,5), (4,6)\}$  având vâfurile saturate 1, 2, 3, 4, 5, 6 iar cuplajul  $M_2$  este format din muchiile  $M_2 = \{(1,2), (3,4)\}$  având vâfurile saturate 1, 2, 3, 4.

Se pot defini următoarele:

- un cuplaj perfect al lui  $G$  este un cuplaj care saturează toate vâfurile lui  $G$ ;
- un cuplaj maxim al lui  $G$  este un cuplaj care are cel mai mare număr posibil de muchii;
- un cuplaj maximal al lui  $G$  este un cuplaj care nu poate fi lărgit prin adăugarea unei muchii.

## IX-X. Grafuri planare, colorare, cuplaje în grafuri

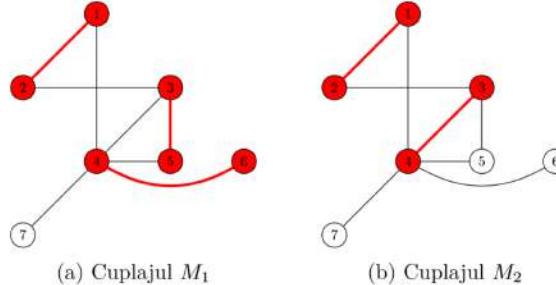


Figura 9: Cuplaje în grafuri - exemple.

**Definiție 9.3.1** (Lanț M-alternant, M-lanț de creștere).

Fie un graf  $G$  și un cuplaj  $M$ , un lanț M-alternant este un lanț în  $G$  în care toate muchiile alternează între muchiile din cuplajul  $M$  și muchiile ce nu aparțin cuplajului. Un M-lanț de creștere este un lanț M-alternant care are ambele capete M-nesaturate.

**Teorema 9.3** (Berge).

Un cuplaj  $M$  al unui graf  $G$  este maxim dacă și numai dacă  $G$  nu conține M-lanțuri de creștere.

*Demonstrație.*

" $\Rightarrow$ ". Se presupune că  $M$  este un cuplaj maxim. Se demonstrează prin contradicție că  $G$  nu are M-lanțuri de creștere. Dacă  $L = (x_1, x_2, \dots, x_k)$  ar fi un M-lanț de creștere atunci, conform definiției,  $k$  ar trebui să fie par astfel încât  $(x_2, x_3), (x_4, x_5), \dots, (x_{k-2}, x_{k-1})$  sunt muchii din cuplaj iar muchiile  $(x_1, x_2), (x_3, x_4), \dots, (x_{k-1}, x_k)$  nu fac parte din cuplaj. Pentru acest lanț se poate defini un cuplaj  $M_1 = \{M \setminus \{(x_2, x_3), (x_4, x_5), \dots, (x_{k-2}, x_{k-1})\}\} \cup \{(x_1, x_2), (x_3, x_4), \dots, (x_{k-1}, x_k)\}$ . Dar  $M_1$  conține cu o muchie mai mult decât  $M$ , ceea ce contrazice ipoteza că  $M$  este maxim.

" $\Leftarrow$ ". Dacă  $M$  nu este maxim, există un cuplaj  $M'$  al lui  $G$  cu  $|M'| > |M|$ . Fie  $H$  subgraful lui  $G$  definit astfel:  $V(H) = V(G)$  și  $E(H)$  este mulțimea muchiilor ce apar exact o dată în  $M$  și  $M'$ .

Deoarece  $|M'| > |M| \Rightarrow H$  are mai multe muchii în  $M'$  decât în  $M$ . Orice vârf  $x$  al lui  $H$  aparține la cel mult o muchie din  $M$  și la cel mult o muchie din  $M' \Rightarrow d_H(x) \leq 2$  pentru toti  $x \in V(H)$ , deci componentele conexe ale lui  $H$  cu mai multe  $M'$ -muchii decât  $M$ -muchii sunt lanțuri sau cicluri. Dacă este ciclu, trebuie să fie ciclu de lungime pară deoarece muchiile alternează între  $M$ -muchii și  $M'$ -muchii  $\Rightarrow$  singurele componente conexe ale lui  $H$  care pot conține mai multe  $M'$ -muchii decât  $M$ -muchii sunt lanțurile.

$|M'| > |M| \Rightarrow$  există un lanț  $P$  în  $H$  care începe și se termină cu o muchie din  $M' \Rightarrow L$  este un M-lanț de creștere ceea ce contrazice ipoteza.  $\square$

**Teorema 9.4** (Hall).

Fie  $G$  un graf bipartit cu mulțimile partite  $X$  și  $Y$ .  $X$  poate fi cuplat în  $Y$  dacă și numai dacă  $|N(S)| \geq |S|$  pentru toate submulțimile  $S$  ale lui  $X$ .

#### 9.4 Cuplaje în grafuri bipartite

Pentru un graf bipartit  $G = (V, E)$  între submulțimile  $X$  și  $Y$  ale lui  $V$  se poate defini un cuplaj ca și o mulțime de muchii  $C \subseteq E$  cu proprietatea că oricare două muchii din  $C$  nu au un capăt comun.

Pentru a determina un cuplaj într-un graf bipartit se pot folosi algoritmii de flux în felul următor:

- toate muchiile din  $G$  se transformă în arce de la  $X$  la  $Y$  cu capacitatea 1;

---

IX-X. Grafuri planare, colorare, cuplaje în grafuri

---

- graful  $G$  se extinde cu două vârfuri  $s$  (sursă) și  $t$  (destinație),  $s$  se leagă de toate vârfurile din  $X$  folosind arce de capacitate 1, toate vârfurile din  $Y$  se leagă de  $t$  folosind arce de capacitate 1;
- se determină fluxul maxim în rețeaua de flux obținută.

**Teorema 9.5.**

*Fie  $G = (V, E)$  rețeaua de flux construită pentru un graf bipartit și  $f$  un flux maxim în  $G$ . Atunci mulțimea muchiilor  $(u, v)$  ale lui  $f$  cu  $u \in X, v \in Y$  și  $f(u, v) = 1$  este un cuplaj maxim în graful bipartit.*

## Suport/Transcript curs algoritmica grafurilor

### XI. Cuplaje în grafuri.

#### 11.1 Cuplaje în grafuri

**Recapitulare curs 10** Un cuplaj în  $G$  este o mulțime de muchii  $M$  în care nici o pereche de muchii nu are un vârf comun. vârfurile adiacente la muchiile din  $M$  se numesc vârfuri *saturate de  $M$*  (sau  *$M$ -saturate*). Celealte vârfuri se numesc  *$M$ -nesaturate*.

Tipuri de cuplaje:

- un **cuplaj perfect** al lui  $G$  este un cuplaj care saturează toate vârfurile lui  $G$ ,
- un **cuplaj maxim** al lui  $G$  este un cuplaj care are cel mai mare număr posibil de muchii,
- un **cuplaj maximal** al lui  $G$  este un cuplaj care nu poate fi lărgit prin adăugarea unei muchii.

Figura 1a prezintă un graf bipartit și figura 1b un cuplaj aleator în graful bipartit (muchiile care fac parte din cuplaj sunt colorate cu roșu).

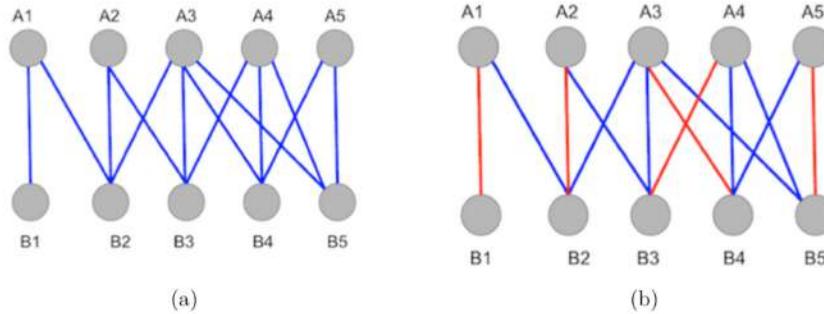


Figura 1: Un graf bipartit și un cuplaj aleator în graful bipartit.

Un *lanț  $M$ -alternant* (*cale  $M$ -alternantă*) este un lanț în  $G$  în care toate muchiile alternează între muchii din  $M$  și muchii ce nu aparțin cuplajului  $M$  (figura 2).

Un  *$M$ -lanț de creștere* ( *$M$ -cale de creștere*) este un lanț  $M$ -alternant care are ambele capete  $M$ -nesaturate.

## XI. Cluplaje.

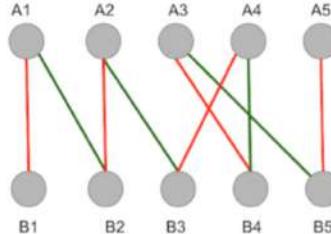


Figura 2: Lanț M-alternant pentru graful bipartit din figura 1a.

**Teorema 9.3** (Berge)

Un cuplaj  $M$  al unui graf  $G = (V, E)$  este maxim dacă și numai dacă  $G$  nu conține M-lanțuri de creștere.

*Demonstrație.*

(vezi cursul 10) □

De exemplu, cuplajul  $M = \{(A1, B2), (A2, B3), (A3, B5)\}$  din figura 3a nu este maxim deoarece conține un M-lanț de creștere  $(B1, A1, B2, A2, B3, A3, B5, A5)$  (figura 3b).

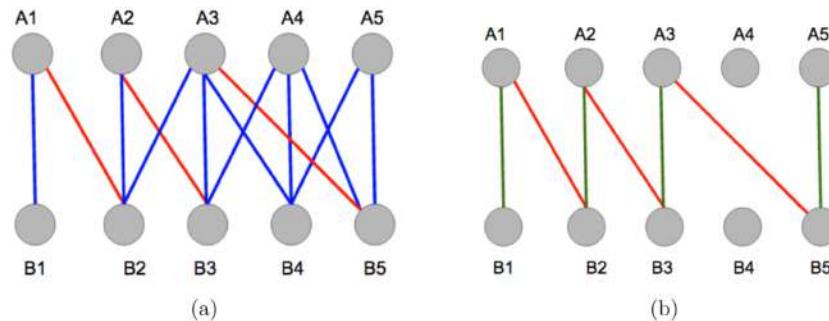


Figura 3: Un graf bipartit și un cuplaj aleator în graful bipartit.

**Teorema 9.4** (Hall)

Fie  $G$  un graf bipartit cu mulțimile partite  $X$  și  $Y$ .  $X$  poate fi cuplat perfect în  $Y$  dacă și numai dacă  $|N(S)| \geq |S|$  pentru toate submulțimile  $S$  ale lui  $X$ .

**11.2 Grafuri bipartite ponderate**

Preferințele pot fi exprimate cu ajutorul ponderilor. Fie un graf bipartit ponderat unde:

- muchia  $(x, y) \in E$  are asociată ponderea  $w(x, y)$
- ponderea cuplajului  $M$  este suma ponderilor muchiilor din cuplajul  $M$

$$w(M) = \sum_{(x,y) \in M} w(x, y)$$

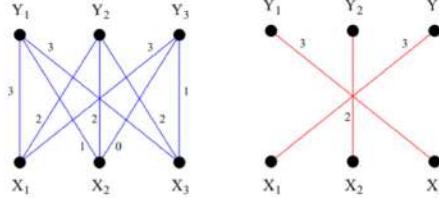
XI. Cluplaje.

Figura 4: Un graf bipartit complet și cuplajul de pondere maximă.

**Problema:** pentru graful bipartit  $G$  găsiți un cuplaj de pondere maximă.

Pentru a forma un graf bipartit complet (figura 4 stânga) se adaugă și muchia  $(X_2, Y_3)$  care are ponderea 0,  $w(X_2, Y_3) = 0$ .

Pentru a găsi cuplajul de pondere maximă se aplică o etichetare a vârfurilor:

- o **etichetare** a vârfurilor este o funcție  $l : V \rightarrow \mathbb{R}$ ,
- o etichetare **fezabilă** respectă:

$$l(x) + l(y) \geq w(x, y), \forall x \in X, y \in Y,$$

- un **graf egal** (ținând cont de  $l$ ) este un graf  $G = (V, E_l)$  unde:

$$E_l = \{(x, y) | l(x) + l(y) = w(x, y)\}.$$

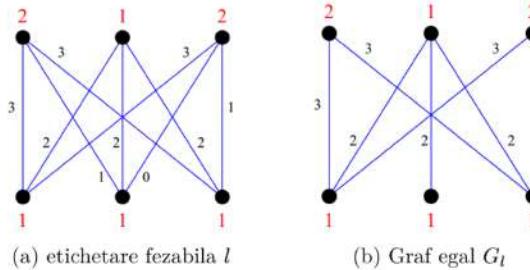


Figura 5: O etichetare fezabilă și graful egal.

**Teorema 11.1** (Teorema Kuhn-Munkres).

Dacă  $l$  este fezabilă și  $M$  este un cuplaj perfect în  $E_l$  atunci  $M$  este un cuplaj de pondere maximă.

*Demonstrație.*

Fie  $e \in E$  și  $e = (e_x, e_y)$ ,  $M'$  un cuplaj perfect în  $G = (V, E)$  (nu neapărat în  $E_l$ ).

Deoarece fiecare vârf  $v \in V$  e acoperit exact o singură dată de  $M$  avem

$$w(M') = \sum_{e \in M'} w(e) \leq \sum_{e \in M'} (l(e_x) + l(e_y)) = \sum_{v \in V} l(v).$$

$\sum_{v \in V} l(v)$  este o limită superioară a oricărui cuplaj.

Fie  $M$  un cuplaj perfect în  $E_l$

$$\rightarrow w(M) = \sum_{e \in M} w(e) = \sum_{v \in V} l(v)$$

---

XI. Cluplaje.

---

și atunci

$$w(M') \leq w(M)$$

deci  $M$  este optimal.  $\square$

Teorema KM transformă problema găsirii unui cuplaj de pondere maximă (problemă de optimizare) într-o problemă combinatorială ce presupune găsirea unui cuplaj perfect. Pentru un cuplaj  $M$  și o etichetare fezabilă  $l$  avem:

$$w(m) \leq \sum_{v \in V} l(v),$$

seamănă cu teorema fluxului maxim și a tăieturii minime.

Într-un cuplaj perfect fiecare vârf este acoperit (saturat) o singură dată și

$$w(M) = \sum_{e \in M} w(e) = \sum_{v \in V} l(v);$$

oricare alt cuplaj  $M'$  din  $G = (V, E)$  satisfacă

$$w(M') = \sum_{e \in M'} w(e) \leq \sum_{v \in V} l(v);$$

astfel  $w(M') \leq w(M)$  și  $M$  trebuie să fie optimă.

### 11.2.1 Metoda maghiară

Un posibil algoritm pentru determinarea cuplajului maxim într-un graf este:

**cuplaj( $G$ )**

- 1: start cu o etichetare fezabilă  $l$  și un cuplaj  $M$  în  $E_l$
- 2: **while** cuplajul  $M$  nu e perfect **do**
- 3:   caută un M-lanț de creștere pentru  $M$  în  $E_l$  (crește dimensiunea lui  $M$ )
- 4:   **if** nu există un M-lanț de creștere **then**
- 5:     îmbunătățește  $l$  la  $l'$  astfel încât  $E_l \subset E_{l'}$

În fiecare pas se crește dimensiunea lui  $M$  sau  $E_l$ . Conform teoremei Kuhn-Munkres,  $M$  va fi un cuplaj de pondere maximă.

Pentru a găsi inițial o etichetare fezabilă se poate folosi:

$$\forall y \in Y, l(y) = 0, \quad \forall x \in X, l(x) = \max_{y \in Y} \{w(x, y)\}.$$

Astfel este evident

$$\forall x \in X, y \in Y, w(x, y) \leq l(x) + l(y).$$

Figura 6 prezintă o astfel de etichetare.

Pentru  $l$  o etichetare fezabilă, se definește un **vecin** al lui  $u \in V$  și un set  $S \subseteq V$  astfel:

$$N_l(u) = \{v | (u, v) \in E_l\}, \quad N_l(S) = \cup_{u \in S} N_l(u).$$

Îmbunătățirea etichetării se poate face conform lemei de mai jos.

**Lema 11.2** (Îmbunătățirea unei etichetări). *Fie  $S \subseteq X$  și  $T = N_l(S) \neq Y$ . Fie*

$$\alpha_l = \min_{x \in S, y \notin T} \{l(x) + l(y) - w(x, y)\} \tag{1}$$

## XI. Cluplaje.

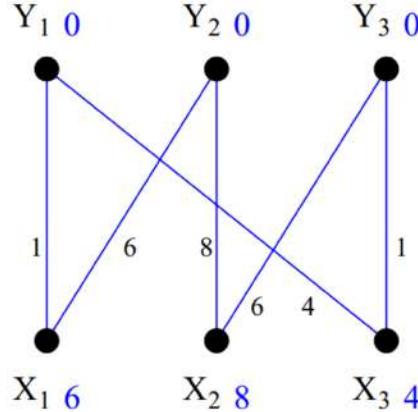


Figura 6: Un graf bipartit complet și cuplajul de pondere maximă.

$$l'(v) = \begin{cases} l(v) - \alpha_l & \text{dacă } v \in S, \\ l(v) + \alpha_l & \text{dacă } v \in T, \\ l(v) & \text{altfel.} \end{cases} \quad (2)$$

Atunci  $l'$  este o etichetare fezabilă și

1. dacă  $(x, y) \in E_l$  pentru  $x \in S, y \in T$  atunci  $(x, y) \in E_{l'}$ ;
2. dacă  $(x, y) \in E_l$  pentru  $x \notin S, y \notin T$  atunci  $(x, y) \in E_{l'}$ ;
3. există o muchie  $(x, y) \in E_{l'}$  pentru  $x \in S, y \notin T$ .

**Metoda maghiară - exemplu matriceal** Să se rezolve următoarea problemă: Vreau să organizez o petrecere, vreau să angajez un muzician, bucătar și serviciu de curătenie. Am la dispoziție 3 companii, fiecare poate furniza un singur serviciu. Ce companie trebuie să furnizeze fiecare serviciu astfel încât costul total să fie minim?

| Companie | Cost muzician | Cost bucătar | Cost curătenie |
|----------|---------------|--------------|----------------|
| A        | 108           | 125          | 150            |
| B        | 150           | 135          | 175            |
| C        | 122           | 148          | 250            |

Fie matricea asociată tabelului:

|     |     |     |
|-----|-----|-----|
| 108 | 125 | 150 |
| 150 | 135 | 175 |
| 122 | 148 | 250 |

## XI. Cluplaje.

**Pas 1.** Se scade valoarea minimă de pe fiecare rând din fiecare element de pe rând:

|    |    |     |
|----|----|-----|
| 0  | 17 | 42  |
| 15 | 0  | 40  |
| 0  | 26 | 128 |

**Pas 2.** Se scade valoarea minimă de pe fiecare coloană din fiecare element de pe coloană:

|    |    |    |
|----|----|----|
| 0  | 17 | 2  |
| 15 | 0  | 0  |
| 0  | 26 | 88 |

**Pas 3.** Se desenează linii pe rândurile și coloanele din matrice ce conțin valoarea 0 astfel încât să se traseze cât mai puține linii:

|    |    |    |
|----|----|----|
| 0  | 17 | 2  |
| 15 | 0  | 0  |
| 0  | 26 | 88 |

Au fost trasate doar două linii ( $2 < n = 3$ ), algoritmul continuă.

Se caută cea mai mică valoare care nu este acoperită de nicio linie. Se scade această valoare de pe fiecare rând pe care nu s-au trasat linii și apoi se adaugă la fiecare coloană pe care am trasat linii. Apoi, se revine la Pasul 3.

|    |    |    |
|----|----|----|
| -2 | 15 | 0  |
| 15 | 0  | 0  |
| -2 | 24 | 86 |

(a) Scade val. minimă

|    |    |    |
|----|----|----|
| 0  | 15 | 0  |
| 17 | 0  | 0  |
| 0  | 24 | 86 |

(b) Adaugă pe col.

Se revine la Pasul 3:

|    |    |    |
|----|----|----|
| 0  | 15 | 0  |
| 17 | 0  | 0  |
| 0  | 24 | 86 |

---

XI. Cluplaje.

Se trasează 3 linii,  $n = 3$ , algoritmul a terminat. Se alege o alocare prin alegerea unui set de valori 0 astfel încât fiecare rând sau coloană să aibă o singură valoare selectată.

|    |    |    |
|----|----|----|
| 0  | 15 | 0  |
| 17 | 0  | 0  |
| 0  | 24 | 86 |

Ex. compania C trebuie să furnizeze muzicianul, compania A trebuie să furnizeze serviciul de curățenie → compania B ne dă bucătarul.

Soluția finală:

|     |     |     |
|-----|-----|-----|
| 108 | 125 | 150 |
| 150 | 135 | 175 |
| 122 | 148 | 250 |

Se poate verifica soluția și exhaustiv:

- $108 + 135 + 250 = 493$
- $108 + 148 + 175 = 431$
- $150 + 125 + 250 = 525$
- $150 + 148 + 150 = 448$
- $122 + 125 + 175 = 422$
- $122 + 135 + 150 = 407$

**Algoritmul** pentru metoda maghiară este prezentat mai jos. Complexitatea algoritmului  $O(V^4)$ , ulterior redusă la  $O(V^3)$ .

**metoda\_maghiara( $G$ )**

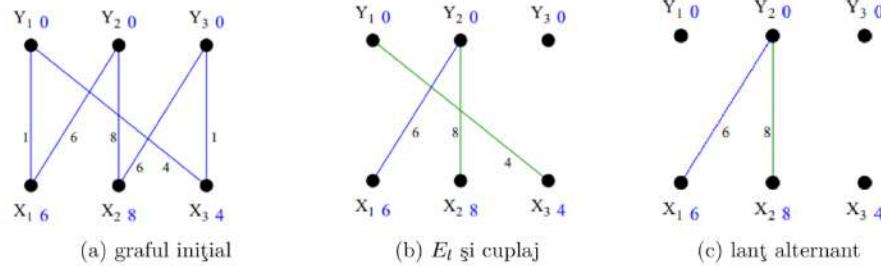
```

1: generează o etichetare inițială l și un cuplaj M în E_l
2: if M nu este un cuplaj perfect then
3: alege un vârf liber $x \in X$
4: $S = \{x\}$, $T = \emptyset$
5: if $N_l(S) = T$ then
6: actualizează etichetele conform (1) și (2) (înănd $N_l(S) \neq T$)
7: if $N_l(S) \neq T$ then
8: alege $y \in N_l(S) - T$
9: if y e liber then
10: $u - y$ este un lanț-M de creștere.
11: îmbunătățește M și sari la linia 2
12: else
13: $T = T \cup \{y\}$ și sari la linia 5

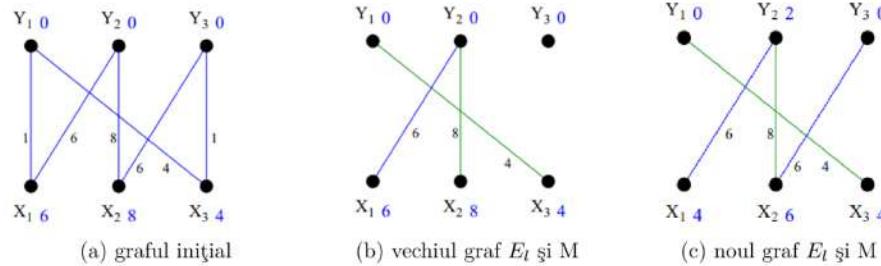
```

## XI. Cluplaje.

**Metoda maghiară - exemplu pe un graf** Figura de mai jos prezintă graful inițial, etichetarea vârfurilor și graful egal asociat.

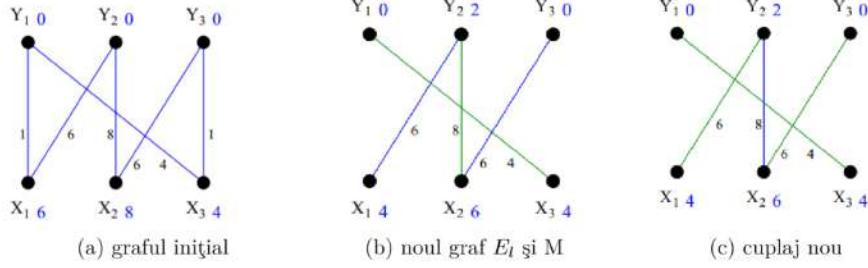


- Cuplajul inițial  $M = \{(x_3, y_1), (x_2, y_2)\}$ ,  $S = \{x_1\}$ ,  $T = \emptyset$ ;
- deoarece  $N_l(S) \neq T$  se merge la linia 7: alege  $y_2 \in N_l(S) - T$ ;
- $y_2$  este în cuplaj, se crește lanțul alternant prin adăugarea lui  $(y_2, x_2)$ ,  $S = \{x_1, x_2\}$ ,  $T = \{y_2\}$ ;
- $N_l(S) = T$ , se sare la linia 5.



- $S = \{x_1, x_2\}$ ,  $T = \{y_2\}$  și  $N_l(S) = T$ ;
  - se determină  $\alpha_l =$
- $$\alpha_l = \min_{x \in S, y \notin T} \begin{cases} 6 + 0 - 1 & (x_1, y_1) \\ 6 + 0 - 0 & (x_1, y_3) \\ 8 + 0 - 0 & (x_2, y_1) \\ 8 + 0 - 6 & (x_2, y_3) \end{cases} = 2$$
- se reduc etichetele lui  $S$  cu 2, se măresc etichetele lui  $T$  cu 2;
  - acum  $N_l(S) = \{y_2, y_3\} \neq \{y_2\} = T$ ,  $S = \{x_1, x_2\}$ .

## XI. Cluplaje.



- Se alege  $y_3 \in N_l(S) - T$  și se adaugă în T;
- $y_3$  nu e în cuplaj, s-a găsit un lanț de creștere  $(x_1, y_2, x_2, y_3)$ ;
- cuplajul  $\{(x_1, y_2), (x_2, y_3), (x_3, y_1)\}$  are costul  $6 + 6 + 4 = 16$  care este egal cu suma etichetelor grafului final.

**11.3 Referințe**

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press.
2. Geir Agnarsson and Raymond Greenlaw. 2006. Graph Theory: Modeling, Applications, and Algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
3. Mark Newman. 2010. Networks: An Introduction. Oxford University Press, Inc., New York, NY, USA.
4. Cristian A. Giumale. 2004. Introducere în analiza algoritmilor, teorie și aplicație. Polirom.
5. cursuri Teoria grafurilor: Zoltán Kása, Mircea Marin, Toadere Teodor.
6. [https://www-m9.ma.tum.de/graph-algorithms/matchings-hungarian-method/index\\_en.html](https://www-m9.ma.tum.de/graph-algorithms/matchings-hungarian-method/index_en.html)