5

laborator 9

>> Grafuri

Conţinut

- Grafuri
- Clasificare
- Reprezentări. Liste şi matrice de adiacenţă
- Traversare în lăţime (BF)
- Traversare în adâncime (DF)
- Componente conexe
- Componente tare conexe
- Sortarea topologică

Referinţe

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest.** *Introducere în algoritmi: cap. 5.4 şi cap 23*, Editura Computer Libris Agora, 2000 (şi ediţiile ulterioare)
- **R. Ceterchi.** *Materiale de curs*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Temele 9, 10*
- **D.-R. Popescu** *Combinatorică şi teoria grafurilor: cap 2*, Ed. Societatea de Ştiinţe Matematice din România, 2005

Terminologie

Un **graf** G este un cuplu (V,E), format din două mulţimi:

- $V \neq \emptyset$: *mulţimea vârfurilor*, şi
- $E \subseteq V \times V$: *mulţime de perechi de vârfuri*.

Grafic reprezentăm vârfurile prin cercuri.

Dacă mulţimea vârfurilor V este finită ($|V| < \infty$), atunci graful G se numeşte **finit**.

Observaţi că putem avea $E = \emptyset$. Putem considera că E este o relaţie binară.

Cu $|\cdot|$ notăm cardinalul unei mulţimi.

Graf orientat

Elementele mulţimii E sunt perechi ordonate de vârfuri $e = (u,v)$ şi se numesc **arce**. Vârfurile u şi v se numesc **capete** sau **extremităţi**. Arcele de tipul $(u,u) \in E$, care pleacă din şi intră în acelaşi nod se numesc **autobucle**. Grafic reprezentăm arcele prin săgeţi.

Graf neorientat (simplu)

Elementele mulţimii E sunt perechi neordonate de vârfuri şi se numesc **muchii**. Grafic reprezentăm muchiile prin linii. Într-un graf neorientat nu avem autobucle (fiecare muchie este compusă din două vârfuri distincte), adică mulţimea muchiilor este

$$E = \{(u,v) | u,v \in V, u \neq v\}.$$

Incidenţă

Într-un graf orientat $G = (V,E)$, spunem că un arc $(u,v) \in E$ este **incident din** vârful u şi **incident în** vârful v, cu alte cuvinte un arc este incident din acel vârf considerat *originea muchiei* şi incident în vârful considerat *vârf terminus*.

Într-un graf neorientat $G = (V,E)$, spunem că o muchie $(u,v) \in E$ este **incidentă** vârfurilor u şi v.

Adiacenţă

Două vârfuri u şi v din G se numesc **adiacente**, dacă sunt conectate printr-o muchie, adică:

$$u,v \text{ adiacente } \Leftrightarrow (u,v) \in E.$$

Într-un graf neorientat, relaţia de adiacenţă este evident simetrică: dacă u este adiacent cu v, atunci şi v este adiacent cu u:

$$(u,v) \in E \Leftrightarrow (v,u) \in E.$$

Afirmaţia nu este valabilă pentru un graf orientat, chiar dacă este posibil să avem $(u,v), (v,u) \in E$ nu există o implicaţie între cele două apartenenţe.

De aceea, considerăm că o muchiile (u,v) şi (v,u) într-un graf neorientat G reprezintă aceeaşi muchie.

Mulţimea vârfurilor adiacente cu un vârf $v \in V$ se numeşte **mulţime de vecini** şi se notează $N_G(v)$:

$$N_G(v) = \{u \in V | (u,v) \in E \text{ sau } (v,u) \in E\}.$$

Gradul unui vârf

- Într-un graf orientat, **gradul unui vârf** $v \in G$, notat $d_G(v)$, este numărul de arce ce intră în v plus numărul de arce ce pleacă din v:

$$d_G(v) = d_G^-(v) + d_G^+(v).$$

Gradul interior al unui vârf $v \in G$, notat d_G^-, este numărul de arce ce intră în v (v este vârf terminus pentru arce respective):

$$d_G^- = |\{(u,v) \in E\}|$$

Gradul exterior al unui vârf $v \in G$, notat d_G^+, este numărul de arce ce pleacă din v (v este vârf origine pentru arcele respective):

$$d_G^+ = |\{(v,u) \in E\}|$$

- Într-un graf neorientat, **gradul unui vârf** $v \in G$, notat $d_G(v)$, este numărul de muchii incidente ale lui v.

Un vârf al cărui grad este 0 se numeşte **vârf izolat**.

Drumuri

În literatura de specialitate noţiunea de drum *se foloseşte pentru grafuri orientate, iar pentru grafuri neorientate cea de* lanţ.

Un **drum** de **lungime** k de la un vârf u la un vârf v într-un graf $G = (V,E)$ este un şir de vârfuri $\langle v_0, v_1, \ldots, v_k \rangle$ astfel încât $u = v_0$, $v = v_k$ şi $(v_{i-1}, v_i) \in E$, $\forall i = \overline{1,k}$. Un drum se numeşte **elementar** dacă toate vârfurile din el sunt distincte.

Lungimea unui drum este numărul de muchii (arce) din acel drum.

Dacă există un drum p de la un vârf u la un vârf v, notăm $u \overset{p}{\rightsquigarrow} v$, şi spunem că v este **accesibil** din u prin p.

Lungimea celui mai scurt drum $\delta(u,v)$ între vârfurile u şi v este numărul minim de muchii ale oricărui drum între u şi v sau ∞ dacă un asemenea drum nu există. Un drum de lungime $\delta(u,v)$ se numeşte **drum de lungime minimă**.

Lemă: Pentru un graf $G = (V,E)$ şi un vârf $s \in V$, dacă $(u,v) \in E$ este o muchie din graf, atunci

$$\delta(s,v) = \delta(s,u) + 1.$$

Într-un graf orientat, un drum $\langle v_0, v_1, \ldots, v_{k-1}, v_0 \rangle$ se numeşte **ciclu** (primul şi ultimul vârf din ciclu coincid). Un ciclu este **elementar** dacă toate vârfurile din el sunt distincte. O autobuclă este un ciclu de lungime 1.

Un graf fără cicluri se numeşte **aciclic**.

Un graf neorientat este **conex** dacă fiecare pereche de vârfuri este conectată printr-un drum (echivalent cu „graful are exact o componentă conexă", echivalent cu „fiecare vârf este accesibil din fiecare vârf").

O **componentă conexă** a unui graf este o clasă de echivalenţă a vârfurilor sub relaţia *este accesibil din*.

Tare conexitatea se defineşte doar pentru grafuri orientate.

O **componentă tare conexă** a unui graf orientat $G = (V,E)$ este o mulţime maximală de vârfuri $U \subseteq V$, astfel încât pentru fiecare pereche u şi v de vârfuri din U, v este accesibil din v şi v este accesibil din u:

$$(\forall u,\ v \in U) u \rightsquigarrow v \text{ şi } v \rightsquigarrow u.$$

Deci o componentă tare conexă a unui graf este o clasă de echivalenţă a vârfurilor sub relaţia *este reciproc accesibil*.

Un graf orientat este **tare conex** dacă este format dintr-o singură componentă tare conexă.

Alte tipuri de grafuri

Grafurile cu costuri sunt numite şi ***grafuri ponderate***.

Un graf cu costuri este un graf în care fiecărei muchii (respectiv arc) i se asociază un cost, de obicei printr-o funcţie $w : E \to \mathbb{R}$. Deci costul unei muchii (respectiv unui arc) $(u,v) \in E$ va fi valoarea reală $w(u,v)$.

Complementarul $\overline{G} = (V, \overline{E})$ al unui graf *simplu* $G = (V, E)$, este practic graful G care conţine muchii între toate vârfurile ce nu erau conectate prin muchii şi nu conţine nicio muchie din E. Formal,

$$\overline{E} = \{(u, v) \in V \times V | (u, v) \notin E\} = V \times V \setminus E.$$

Transpusul $G^T = (V, E^T)$ al unui graf *orientat* $G = (V, E)$, este practic graful G cu sensurile arcelor inversate. Formal,

$$E^T = \{(v, u) \in V \times V | (u, v) \in E\}.$$

Pătratul $G^2 = (V, E^2)$ al unui graf *orientat* $G = (V, E)$ este un graf în care există muchia (u, w) dacă în G există un drum cu exact două muchii între u şi w. Formal,

$$E^2 = \{(u, w) \in V \times V | (u, v), (v, w) \in E\}.$$

Reprezentarea grafurilor

Vârfurile unui graf sunt reţinute în general într-un vector V sau vor fi „implicite" (de exemplu, dacă le considerăm numerotate de la 1 la $|V|$ atunci nu mai este necesar să stocăm denumirea lor în V). în cele două reprezentări de mai jos în mod uzual nu se foloseşte o structură auxiliară pentru vârfuri.

Uneori ne vom referi la mulţimea de vârfuri a unui graf G prin $V[G]$, la fel cum $E[G]$ indică mulţimea de muchii a grafului G. De cele mai multe ori se face un abuz de notaţie (atunci când nu există posibilitatea de a crea confuzii şi se scrie direct $v \in G$ sau $(u, v) \in G$.

Dacă vârfurile prin natura problemei sau a algoritmilor au asociate diferite atribute se poate folosi şi o implementare ca tip structurat a unui vârf, în care fiecare atribut este implementat ca şi câmp al tipului structurat. A doua variantă este să reţinem valorile tuturor vârfurilor pentru atributul respectiv într-un vector de dimensiune $n = |V|$ ce poartă în general numele atributului. Atunci, pentru un vârf $v \in V$ în primul caz vom nota $v.atribut$, iar în al doilea caz $atribut[v]$.

Inserarea (unui vârf sau a unei muchii) în graf, respectiv ştergerea (unui vârf sau a unei muchii) din graf, vor actualiza corespunzător listele de adiacenţă sau matricea de adiacenţă.

Reprezentarea prin liste de adiacenţă

Reprezentarea prin liste de adiacenţă este preferată îndeosebi pentru **grafuri rare**, acele grafuri unde numărul de muchii existente este mult mai mic decât numărul de muchii posibile între vârfurile din graf, adică:

$$|E| << |V|^2.$$

*Similar, matricele **rare** sunt acelea în care doar o mică parte din elemente sunt nenule, iar polinoamele **rare** sunt acelea ce conţin un număr redus de coeficienţi nenuli.*

Această reprezentare presupune că reţinem o **listă de liste** L. Lista L conţine $|V| = n$ liste de adiacenţă, notate L_v, una pentru fiecare vârf $v \in G$ din graf. Lista L nu trebuie neapărat să aibă aceeaşi structură ca elementele sale L_v, $v \in G$. De exemplu L poate fi o structură liniară în alocare statică (un vector) care conţine primul nod al fiecărei liste simplu înlănţuite L_v.

Lista de adiacenţă a unui vârf v va conţine toate vârfurile sale adiacente u, deci:

$$L_v = \{u \in G | (u, v) \in E\}.$$

De aici provine şi denumirea de adiacenţă.

Considerăm că nu interesează ordinea în care sunt reţinute vârfurile adiacente u într-o listă L_v.

De cât spaţiu avem nevoie?

- Dacă G este un graf orientat, atunci suma lungimilor (dimensiunilor) tuturor listelor de adiacenţă necesare reprezentării sale este chiar numărul de arce din graf:

$$\sum_{v \in V} |L_v| = |E|,$$

deoarece pentru fiecare arc $(v,u) \in E$, vârful u apare în lista de adiacenţă a lui v, L_v.

- Dacă G este un graf neorientat, atunci suma lungimilor (dimensiunilor) tuturor listelor de adiacenţă necesare reprezentării sale este de două ori numărul de muchii din graf:

$$\sum_{v \in V} |L_v| = 2|E|,$$

deoarece pentru fiecare muchie $(u,v) \in E$, lista L_v va conţine vârful u, iar lista L_u va conţine vârful v.

Listele de adiacenţă pot fi uşor adaptate reprezentării grafurilor cu costuri: pentru fiecare vârf v în loc de a reţine doar fiecare vârf adiacent u, se va reţine (în acelaşi loc) şi costul $w(u,v)$ al muchiei (sau arcului) ce conectează cele două vârfuri.

Reprezentarea prin matrice de adiacenţă

Reprezentarea prin matrice de adiacenţă se foloseşte de obicei în două situaţii.

1. În cazul unui graf **dens**, în care numărul de muchii existente este aproximativ egal cu cel al muchiilor posibile, adică:
$$|E| \approx |V|^2.$$
2. Atunci când timpul de acces mai bun oferit de această reprezentare este necesar. Cu alte cuvinte trebuie să decidem rapid dacă există sau nu muchie între două noduri.

Se presupune că vârfurile din V sunt numerotate $1, 2, \ldots, |V| = n$. Se va reţine o matrice A de dimensiuni $|V| \times |V|$, în care elementul de pe linia $1 \le i \le |V| = n$ şi coloana $1 \le j \le |V| = n$, a_{ij}, indică dacă există muchie între vârful i şi vârful j, adică:

$$a_{ij} = \begin{cases} 1, & \text{dacă } (i,j) \in E \\ 0, & \text{altfel.} \end{cases}$$

Obs.: *Matricea de adiacenţă ce reprezintă un graf neorientat este simetrică faţă de diagonala principală (este propria ei* transpusă*).*

De cât spaţiu avem nevoie?

Indiferent de numărul de muchii din graf, reprezentarea prin matrice de adiacenţă necesită $|V|^2$ spaţiu în memorie. Ca urmare a observaţiei anterioare, spaţiul ocupat poate fi redus aproape la jumătate dacă se folosesc numai elementele de pe diagonala principală şi fie elementele situate deasupra acesteia, fie cele situate sub ea.

Reprezentarea prin matrice de adiacenţă poate fi utilizată şi pentru grafuri cu costuri, stocând în matrice costurile muchiilor existente, astfel:

$$a_{ij} = \begin{cases} c(i,j), & \text{dacă } (i,j) \in E \\ 0, & \text{altfel.} \end{cases}$$

Pentru grafuri orientate se poate defini **matricea de incidenţă** B de dimensiuni $|V| \times |E|$, în care elementul de pe linia $1 \le i \le |V| = n$ şi coloana $1 \le j \le |V| = n$, b_{ij}, indică dacă muchia j, dacă există, pleacă sau intră în nodul i, adică:

$$b_{ij} = \begin{cases} -1, & \text{dacă } j \text{ pleacă din vârful } i \\ 1, & \text{dacă } j \text{ intră în vârful } i \\ 0, & \text{altfel.} \end{cases}$$

Traversarea grafurilor

Traversarea în lăţime (breadth first)

Dat fiind un graf $G = (V, E)$ şi un nod **sursă** s , căutarea în lăţime explorează sistematic muchiile lui G pentru a „descoperi" fiecare nod care este accesibil din s. Algoritmul calculează distanţa (cel mai mic număr de muchii) de la s la toate vârfurile accesibile. Astfel, produce un **arbore de lăţime** cu rădăcina în s, care conţine toate aceste vârfuri accesibile. Pentru fiecare vârf v accesibil din s, calea din arborele de lăţime de la s la v corespunde unui „cel mai scurt drum" de la s la v în G, adică un drum care conţine un drum minim de muchii.

Algoritmul descoperă toate vârfurile aflate la distanţa k faţă de s înainte de a descoperi vreun vârf la distanţa $k+1$.

Algoritmul marchează un nod în trei feluri diferite pentru a reţine starea curentă a sa, după cum urmează:

- un vârf este marcat cu ALB iniţial, apoi când este „descoperit" (întâlnit prima dată) îşi schimbă culoarea;
- toate vârfurile adiacente unui nod NEGRU au fost descoperite, iar
- vârfurile marcate cu GRI reprezintă frontiera dintre cele descoperite şi cele nedescoperite.

Reţinerea acestei valori de marcaj se poate face:

- fie implementând vârfurile ca tip structurat ce conţine şi un câmp `culoare`,
- fie utilizând un vector `culoare` de dimensiune $n = |V|$, care reţine culoarea asociată fiecărui vârf; cum vârfurile sunt numerotate de la 1 la $|V|$ rezultă că pentru un vârf $v \in V$ culoarea sa va fi $culoare[v]$.

În loc de graful G se vor transmite evident datele (lista L) ce reprezintă graful prin liste de adiacenţă.

►► CAUTĂ–ÎN–LĂŢIME(G, s)

```
1.  for v ∈ V[G] \ {s} do
2.    color[v] ← ALB
3.    d[v] ← ∞
4.    π[v] ← NIL
5.  endfor
6.  color[s] ← GRI
7.  d[s] ← 0
8.  π[s] ← NIL
9.  PUSH(Q, s) // inserăm s în coada Q
10. while Q ≠ ∅ do
11.   POP(Q, u) // extragem capul cozii Q în u
12.   for v ∈ L_u do
13.   // pentru fiecare vârf din lista de adiacenţă a lui u
14.     if color[v] ← ALB then
15.       color[v] ← GRI
16.       d[v] ← d[u] + 1
17.       π[v] ← u
18.       PUSH(Q, v) // inserăm v în coada Q
19.     endif
20.   endfor
21.   color[u] ← NEGRU
22. endwhile
```

Algoritmul de căutare în lăţime construieşte un arbore de lăţime ce conţine, iniţial, numai rădăcina sa, care este vârful s. Oricând un vârf alb v este descoperit (linia 14) în cursul parcurgerii

listei de adiacenţă L_u a unui vârf u deja descoperit (parcurgerea începe la linia 12), vârful v şi muchia (u,v) sunt adăugate în arbore (linia 17). Spunem că u este **predecesorul** sau **părintele** lui v în arborele de lăţime (deoarece un vârf este descoperit cel mult o dată, el are un singur predecesor). Relaţia de strămoş-descendent, din arborele de lăţime, sunt definite relativ la rădăcina s: dacă u se află pe un drum din arbore de la rădăcina s la vârful v, atunci u este un strămoş al lui v, iar v este un descendent al lui u.

Structurile de date utilizate de algoritm:

- vectorul `culoare` care reţine culoarea cu care a fost marcat fiecare vârf;
- vectorul π care reţine predecesorul (unic !) fiecărui vârf; dacă un vârf nu are predecesor (cazul vârfului s şi a vârfurilor ce nu au fost încă descoperite) atunci predecesorul său este `NIL`;
- vectorul `d` care reţine distanţa (calculată de algoritm) la care se află fiecare vârf faţă de s.
- coada `Q` în care se reţin vârfurile gri.

Liniile 1-9 reprezintă partea de iniţializare a algoritmului:

- toate vârfurile mai puţin s sunt marcate cu alb, nu au distanţa calculată şi nici predecesorul stabilit (liniile 1-5);
- vârful s este marcat gri (deoarece tocmai a fost descoperit), se află la distanţă 0 faţă de sine şi nu are predecesor (liniile 6-8);
- în coada Q, a vârfurilor gri, se introduce s, singurul vârf pe care îl conţine iniţial.

Liniile 10-22 conţin algoritmul propriu-zis: cât timp mai există vârfuri gri în coada Q (ciclul `while` de la linia 10),

- se scoate un vârf u din Q (linia 11) şi
- pentru fiecare vârf v din lista sa de adiacenţă L_u (linia 12), dacă vârful v este marcat cu alb (nu a fost încă descoperit) (linia 14), atunci el este descoperit acum (liniile 15-18):
  - este marcat cu gri,
  - este actualizată distanţa sa faţă de s (cu 1 mai mare decât a vârfului u din a cărui listă de adiacenţă a fost descoperit),
  - se reţine u ca predecesor al său şi
  - se introduce în coadă, deoarece a devenit gri.
- La linia 21 se ajunge când s-au examinat toate vârfurile adiacente cu u, prin urmare el este marcat cu negru.

Vârfurile gri sunt vârfuri descoperite a căror listă de adiacenţă nu a fost examinată în întregime.

Algoritmul de căutare în lăţime calculează lungimile drumurilor minime între s şi toate vârfurile $v \in V \setminus \{s\}$. La terminare, valorile din vectorul d (care reţine distanţele) au proprietatea că: $d[v] = \delta(s,v)$.

După ce rulăm programul Caută–În–Lăţime(G,s) putem folosi următorul algoritm pentru afişarea vârfurilor de pe un drum de lungime minimă de la s la u.

►► Afişează–Drum(G, s, u)

```
1.  if v = s then
2.    Afişează(s)
3.  else if π[v] = NIL then
4.    // Nu există drum de la s la v
5.  else
6.    Afişează–Drum(G, s, π[v])
7.    Afişează(v)
8.  endif
```

Traversarea în adâncime (depth first)

Strategia folosită de căutarea în adâncime este de a căuta „mai adânc" în graf oricând acest lucru este posibil. Muchiile sunt explorate pornind din vârful v cel mai recent descoperit care mai are încă muchii neexplorate care pleacă din el. Când toate muchiile care pleacă din v au fost explorate, căutarea „revine" pe propriile urme, pentru a explora muchiile care pleacă din vârful din care v a fost descoperit. Căutarea continuă până când s-au descoperit toate vârfurile accesibile din vârful sursă inițial. Procedeul se reia până când toate vârfurile sunt descoperite.

Căutarea în adâncime creează pentru fiecare vârf și **marcaje de timp**. Fiecare vârf $v \in V$ are două marcaje:

- $d[v]$ reține momentul când v este descoperit întâia oară și colorat gri, iar
- $f[v]$ memorează momentul când căutarea termină examinarea listei sale de adiacență L_v și îl colorează negru.

Aceste marcaje sunt valori întregi între 1 și $2|V|$, deoarece există un eveniment de descoperire și unul de terminare pentru fiecare dintre cele $|V|$ vârfuri. Pentru fiecare vârf v, avem $d[v] < f[v]$. În plus, vârful v este alb înainte de momentul $d[v]$, gri între momentele $d[v]$ și $f[v]$ și negru după momentul $f[v]$.

Structurile de date utilizate de algoritm:

- vectorul `culoare` care reține culoarea cu care a fost marcat fiecare vârf; inițial vârfurile sunt albe, când un vârf e descoperit în timpul căutării el este marcat gri, iar când s-a *terminat* (lista sa de adiacență a fost examinată în întregime) este marcat negru;
- vectorul π care reține predecesorul (unic !) fiecărui vârf; dacă un vârf nu are predecesor (cazul vârfului s și a vârfurilor ce nu au fost încă descoperite) atunci predecesorul său este `NIL`; spre deosebire de căutarea în lățime vectorul de predecesori nu formează un arbore ci poate forma o pădure (mai mulți arbori, unul pentru fiecare vârf inițial s); spunem că obținem o **pădure de adâncime** formată din **arbori de adâncime**; colorarea garantează că arborii aceștia vor fi disjuncți, adică fiecare vârf va face parte din exact un arbore de adâncime;
- vectorul `d` care reține momentele când vârfurile sunt descoperite.
- vectorul `f` care reține momentele când vârfurile sunt terminate.

Variabila `timp` este o variabilă globală pe care o folosim pentru aplicarea marcajelor de timp.

În loc de graful G se vor transmite evident datele ce reprezintă graful.

►► CAUTĂ–ÎN–ADÂNCIME(G)

```
1.  for v ∈ V[G] do
2.    culoare[v] ← ALB
3.    π[v] ← NIL
4.  endfor
5.  timp ← 0
6.  for v ∈ V[G] do
7.    if culoare[v] = ALB then
8.      VIZITEAZĂ(v)
9.    endif
10. endfor
```

Liniile 1-5 reprezintă partea de inițializare a algoritmului:

- toate vârfurile sunt marcate cu alb și niciun predecesor nu este stabilit (liniile 2-3);
- se resetează contorul de timp global (linia 5).

Liniile 6-10 conțin algoritmul propriu-zis: cât timp mai există vârfuri albe (liniile 6-7), se vizitează vârful alb u curent (în procedura VIZITEAZĂ):

- se colorează vârful u cu culoarea gri (linia 1)
- se incrementează variabila globală `timp` şi se memorează timpul de descoperire al vârfului u (liniile 2-3)
- pentru fiecare vârf v din lista sa de adiacenţă L_u (linia 4), dacă vârful v este marcat cu alb (nu a fost încă descoperit) (linia 5), atunci u devine predecesorul său şi el este vizitat (liniile 6-7); pe măsură ce fiecare vârf $v \in L_u$ este examinat, spunem că muchia (u,v) este explorată de către căutarea în adâncime.
- În final, după ce fiecare muchie ce pleacă din u a fost explorată, u este colorat în negru şi timpul său de terminare este memorat în $f[u]$ (liniile 10-12).

Vârfurile gri sunt vârfuri descoperite a căror listă de adiacenţă nu a fost examinată în întregime.

Fiecare vârf este vizitat o singură dată deoarece procedura se apelează doar pentru vârfuri albe şi imediat la intrarea în procedură se colorează vârful cu culoarea gri.

▶▶ VIZITEAZĂ(u)

```
1.  culoare[u]  ← GRI
2.  timp  ←  timp + 1
3.  d[u]  ←  timp
4.  for v∈L_u do
5.    if culoare[v] = ALB then
6.      π[v]  ←  u
7.      VIZITEAZĂ(v)
8.    endif
9.  endfor
10. culoare[u]  ← NEGRU
11. timp  ←  timp + 1
12. f[u]  ←  timp
```

Determinarea componentelor tare conexe

Descompunerea unui grad în componentele sale tare conexe este o aplicaţie a căutării în adâncime şi foloseşte transpusul grafului G.

Această descompunere permite divizarea unei probleme în subprobleme pentru fiecare componentă tare conexă.

▶▶ DETERMINĂ–COMPONENTE–TARE–CONEXE(G)

```
1. Caută-În-Adâncime(G)
2. calculează G^T
3. Caută-În-Adâncime-2(G^T)
4. afişează vârfurile fiecărui arbore în pădurea de adâncime din pasul
   anterior ca o componentă tare conexă separată
```

Apelul procedurii de căutare în adâncime se face pentru a calcula timpii de terminare $f[v]$, $\forall v \in V$. Procedura CAUTĂ–ÎN–ADÂNCIME–2 este la fel ca CAUTĂ–ÎN–ADÂNCIME dar consideră vârfurile din bucla `for` principală (a doua) în ordine descrescătoare a timpilor $f[u]$, care au fost calculaţi prin primul apel (linia 2).

Sortarea topologică

Sortarea topologic ă este o aplicaţie a căutării în adâncime a unui graf.

Grafurile orientate aciclice sunt folosite des pentru a indica precedenţa între evenimente.

Sortarea topologică a unui graf orientat aciclic $G = (V, E)$ este o ordonare liniară a tuturor vârfurilor sale astfel încât, dacă G conţine o muchie (u, v), atunci u apare înaintea lui v în ordonare. Dacă un graf nu este aciclic, atunci nu este posibilă o ordonare liniară. O sortare topologică a unui graf poate fi văzută ca o ordonare a vârfurilor sale de-a lungul unei linii orizontale, astfel încât toate muchiile sale orientate merg de la stânga la dreapta.

►► SORTEAZĂ–TOPOLOGIC(G)

```
1. Caută-În-Adâncime(G)
2. pe măsură ce fiecare vârf este terminat
3.    inserează-l în capul unei liste înlănţuite
4. returnează lista înlănţuită de vârfuri
```

Apelul procedurii de căutare în adâncime se face pentru a calcula timpii de terminare $f[v]$, $\forall v \in V$.

PROBLEME

1. **(2p)** Fiind dată matricea de adiacenţă a unui graf neorientat cu n vârfuri, scrieţi funcţiile următoare:
   a. `grad(x)` – returnează gradul vârfului `x` al grafului;
   b. `numarMuchii()` – returnează numărul de muchii din graf;
   c. `gradMax()` – afişează vârfurile de grad maxim.
2. **(2p)** Să se determine componentele conexe ale unui graf neorientat folosind reprezentarea grafului prin liste de adiacenţă.
3. **(2p)** Se dă un graf neorientat G conex. Scrieţi algoritmul pentru parcurgerea grafului folosind metoda BF (breadth first - parcurgere în lăţime) pornind dintr-un nod dat.
4. **(2p)** Se dă un graf neorientat G conex. Scrieţi algoritmul pentru parcurgerea grafului folosind metoda DF (depth first - parcurgere în adâncime).
5. **(3p)** Se dă un graf orientat aciclic. Să se aplice algoritmul de sortare topologică.
6. **(10ps)** Se numeşte subsecvenţă a unui vector V cu n elemente întregi un vector cu cel puţin un element şi cel mult n elemente care se găsesc pe poziţii consecutive în vectorul V. Să se scrie un program de complexitate $O(n)$ care citeşte numărul natural n şi vectorul V având n elemente întregi şi afişează subsecvenţa lui V având suma elementelor maximă.

■ TERMEN DE PREDARE: **Săptămâna 13 (7–11 ianuarie 2013) inclusiv.**

■ DETALII: **Studenţii pot obţine un maxim de 23 puncte. Problemele 1-2 sunt obligatorii. Problemele 3–5 sunt suplimentare.Problema 6 este facultativă, iar termenul de predare pentru ele este săptămâna 12 (17–21 decembrie). Un singur student poate rezolva problema facultativă.**

>> Coduri Huffman

Următorul algoritm greedy a fost inventat de Huffman şi construieşte o codificare prefix optimă, numită ***cod Huffman***.

Un cod binar peste un alfabet V dat, împreună cu o probabilitate $w : V \to [0,1]$, cod asociat unui arbore binar strict cu ponderi construit cu algoritmul lui Huffman, se numeşte ***cod Huffman***.

Presupunem că V este o mulţime de n caractere şi că fiecare caracter $a_i \in C$, $i = 1, \ldots, n$ este un obiect cu o pondere w_i care reprezintă frecvenţa sa. Fie cele n ponderi ordonate descrescător:

$$w_1 \geq w_2 \geq \cdots \geq w_{n-1} \geq w_n.$$

Algoritmul construieşte arborele T corespunzător codului optim de jos în sus. Porneşte cu o mulţime de n frunze şi efectuează o serie de $n-1$ operaţii de „fuzionare" pentru a crea arborele final. În termeni de arbori binari stricţi aceasta corespunde următorului raţionament:

- Iniţial, algoritmul formează n arbori binari stricţi, fiecare de tip frunză, cu câte o pondere w_i asociată ei. Obţinem o *pădure* cu n arbori.
- Apoi, se „leagă" doi subarbori cu ponderi minime din pădure cu ajutorul unui nod interior pentru care ei devin cei doi fii, iar acest arbore va fi arbore binar strict cu ponderea asociată egală cu suma ponderilor arborilor pe care i-am legat, pondere pe care o vom asocia nodului intern.

  Astfel, se reduce numărul de arbori din pădure cu 1.

  Se reia acest pas până când se obţine un singur arbore.

  În general, la iteraţia k, algoritmul are $n-k-1$ arbori binari stricţi cu ponderi. Prin legarea descrisă, a doi arbori cu ponderile cele mai mici, se produc $n-k$ arbori binari stricţi cu ponderi.

  La al $n-1$-lea pas iterativ se obţine un singur arbore, ce se numeşte ***arbore Huffman*** asociat ponderilor date, $\{w_1, w_2, \ldots, w_n\}$.

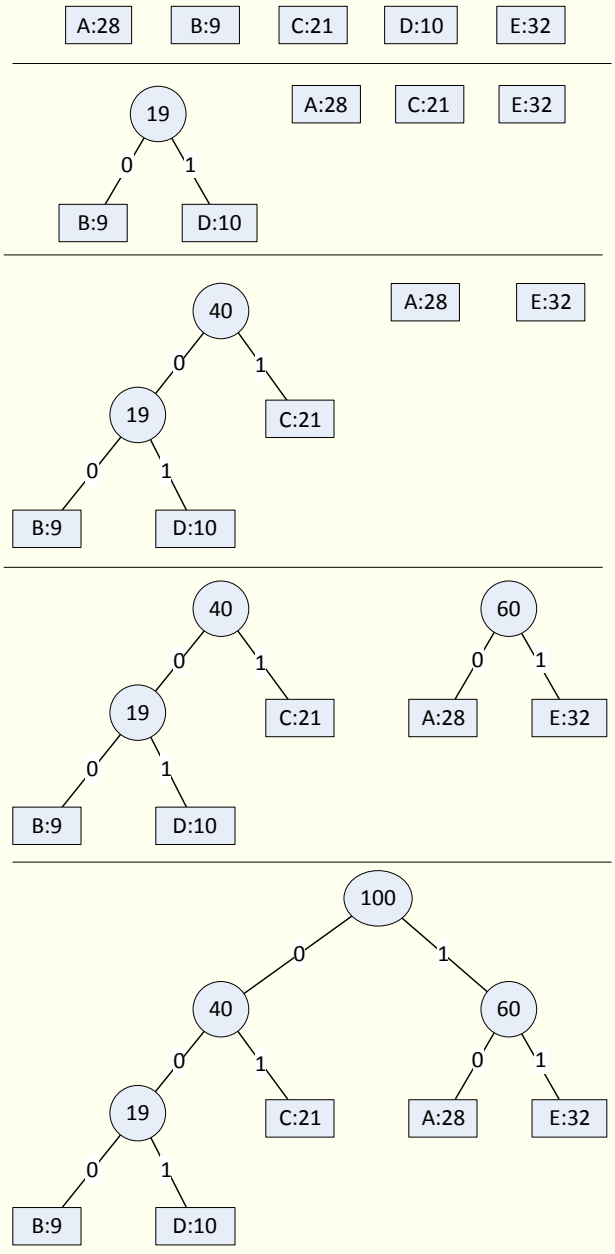
Algoritmul utilizează o coadă de priorităţi Q, ale cărei chei sunt frecvenţele, pentru a identifica cele două cel mai puţin frecvente obiecte ce trebuie combinate. Când două elemente fuzionează, rezultatul este un nou obiect a cărui frecvenţă este suma frecvenţelor celor două obiecte unite.

Presupunem că folosim o coadă cu priorităţi Q implementată ca min-ansamblu.

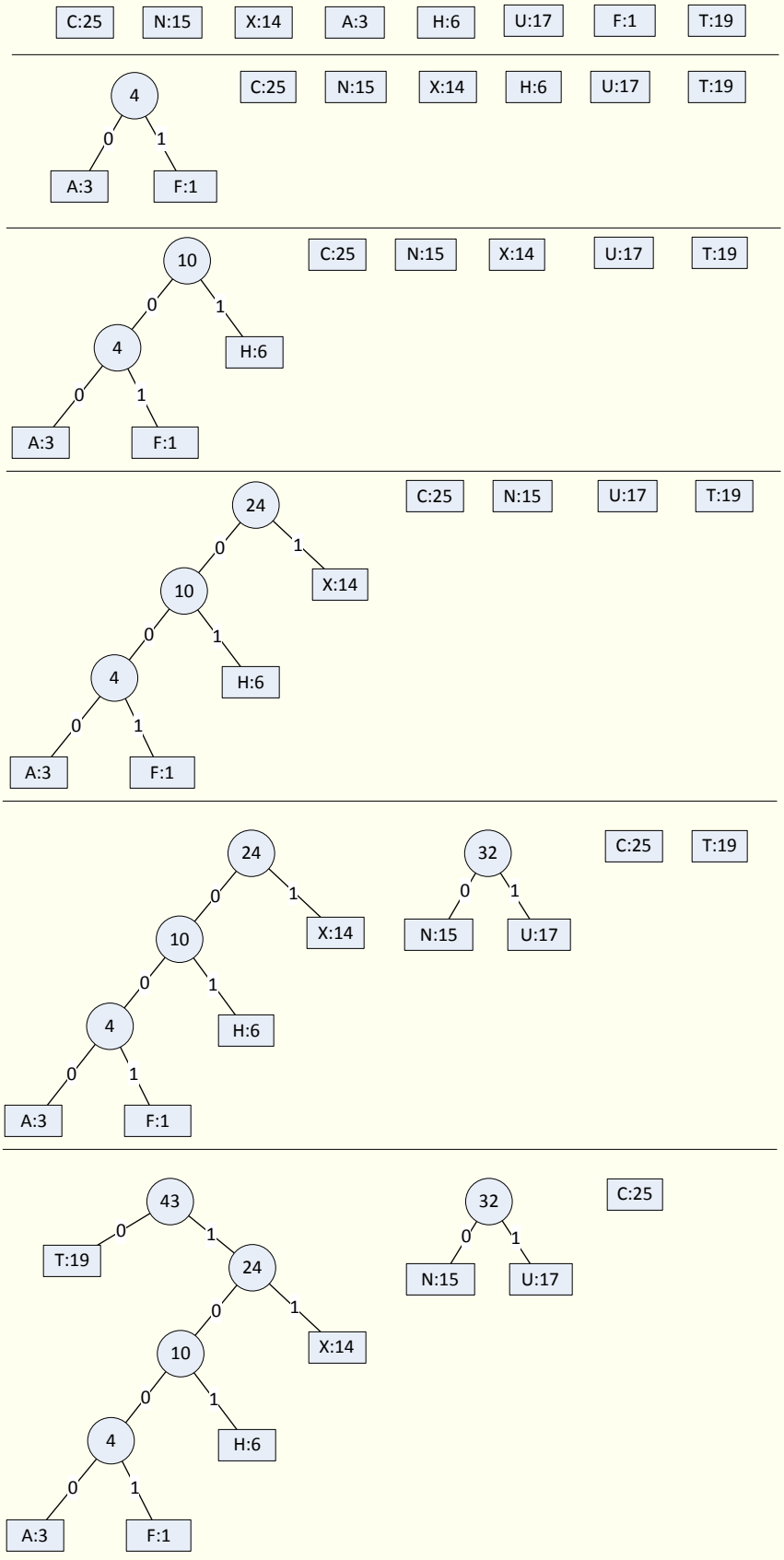
```
HUFFMAN(V)
1   n = |V|
2   Q = V
3   for i = 1 to n − 1
4     alocă un nou nod z
5     z−>left = x = EXTRAGE-MIN(Q)
6     z−>right = y = EXTRAGE-MIN(Q)
7     w(z) = w(x) + q(y)
8     INSEREAZĂ(Q, z)
9.  return EXTRAGE-MIN(Q) // se returnează rădăcina arborelui
```

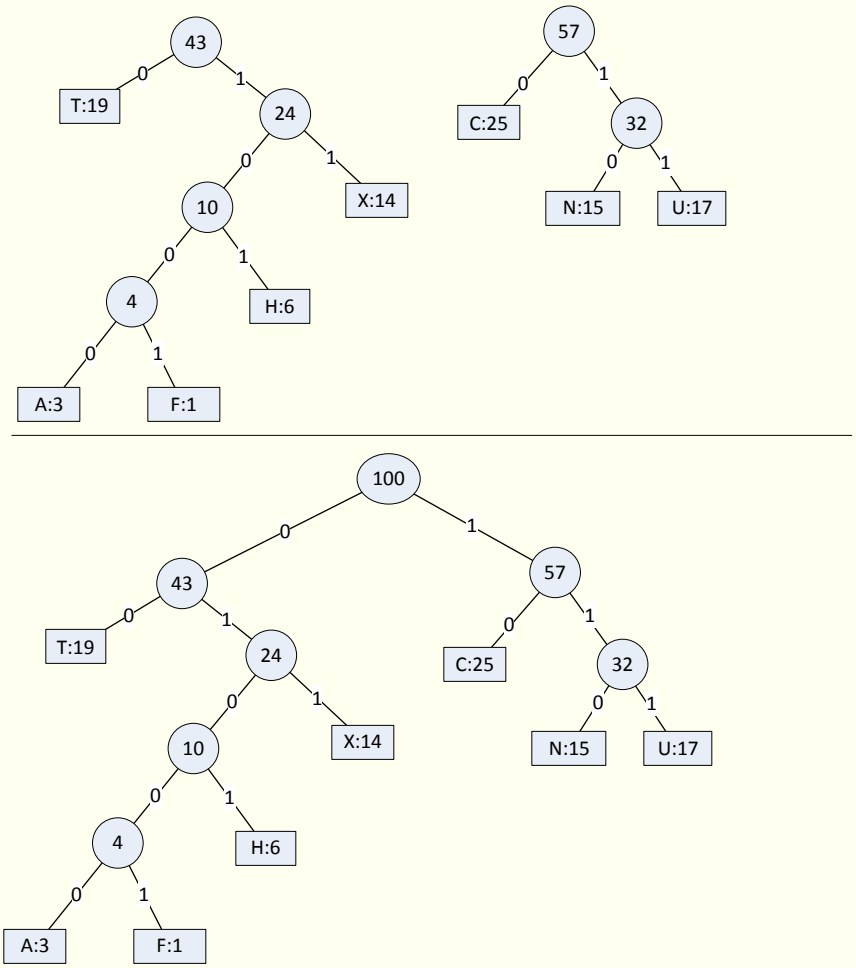
Linia 2 iniţializează coada cu priorităţi Q cu caractere din V. Ciclul **for** extrage la fiecare iteraţie două noduri x şi y corespunzătoare frecvenţelor minime din coadă, înlocuindu-le în Q cu

un nod nou z, ce reprezintă nodul format prin fuziunea între x şi y. Frecvenţa lui z este suma frecvenţelor nodurilor x şi y. Nodul z va avea pe x ca fiu stâng şi pe y ca fiu drept. Ordinea este arbitrară însă, deoarece am obţine un cod diferit cu acelaşi cost (aceeaşi lungime externă ponderată) dacă x devine fiu drept şi y fiu stâng. După $n-1$ fuziuni este returnat nodul unic rămas în coadă, care este rădăcina arborelui de codificare.



Figură 1 Exemplu al comportamentului algoritmului Huffman pentru $V = \{[A,28],\ [B,9],\ [C,21],\ [D,10],\ [E,32]\}$. Caracterul A se codifică prin 10, B prin 000, C prin 01, D prin 001, E prin 11.


C:25
N:15
X:14
A:3
H:6
U:17
F:1
T:19
4
0
1
A:3
F:1
C:25
N:15
X:14
H:6
U:17
T:19
10
0
1
4
H:6
0
1
A:3
F:1
C:25
N:15
X:14
U:17
T:19
24
0
1
X:14
10
0
1
4
H:6
0
1
A:3
F:1
C:25
N:15
U:17
T:19
24
0
1
X:14
10
0
1
4
H:6
0
1
A:3
F:1
32
0
1
N:15
U:17
C:25
T:19
43
0
1
T:19
24
0
1
X:14
10
0
1
4
H:6
0
1
A:3
F:1
32
0
1
N:15
U:17
C:25



Figură 2 Exemplu al comportamentului algoritmului Huffman pentru $V = \{[C,25], [N,15], [X,14], [A,3], [H,6],]U,17], [F,1], [T,19]\}$. Caracterul C se codifică prin 10, N prin 010, X prin 011, A prin 01000, H prin 0101, U prin 111, F prin 01001, T prin 00.

PROBLEME

1. **(5p)** Scrieţi un algoritm care să construiască un arbore Huffman pentru un alfabet cu ponderi dat, arbore reprezentat în aşa fel încât să poată fi folosit atât la codificare, cât şi la decodificare. Scrieţi proceduri pentru codificare şi decodificare.

■ TERMEN DE PREDARE: **Săptămâna 13 (7–11 ianuarie 2013) inclusiv.**

■ DETALII: **Studenţii pot obţine un maxim de 5 puncte. Problema 3 este obligatorie.**