Arbori și arborescențe

Proprietăți elementare ale arborilor

Definiție: Un *arbore* este un graf conex și fără circuite.

Teorema 1. *Fie G=(V,E) un graf. Următoarele afirmații sînt echivalente:*
(i) *G este arbore.*
(ii) *G este conex și este minimal cu această proprietate.*
(iii) *G este fără circuite și este maximal cu această proprietate.*

Observație: Maximalitatea și minimalitatea din condițiile (i) și (ii) se referă la mulțimea muchiilor grafului G și se consideră în raport cu relația de ordine dată de incluziune. Mai precis, cele două afirmații se pot formula echivalent astfel:

(ii') *G este conex și* $\forall e \in E(G)$, *G-e este neconex.*
(iii') *G este fără circuite și* $\forall e \in E(\overline{G})$, *G+e are un circuit.*

Demonstrația teoremei 1: Demonstrăm *i*) $\Leftrightarrow$ *ii*) și *i*) $\Leftrightarrow$ *iii*).

i) $\rightarrow$ *ii)* Dacă G este arbore rezultă că G este conex. Fie $e = vw \in E(G)$ oarecare. Dacă $G - e$ ar fi conex, atunci în $G - e$ există un vw-drum P care impreună cu vw induc în G un circuit contrazicând faptul că G este arbore. Deci $\forall e \in E(G)$, $G - e$ este neconex, adică ii) are loc.

ii) $\rightarrow$ *i)* Trebuie să demonstrăm că G nu are circuite. Presupunând, prin reducere la absurd, că G are un circuit C și $e = uv \in E(C)$ atunci graful $G - e$ este conex. În adevăr, oricare ar fi P un drum în G, dacă $e \in E(P)$ atunci P este drum în $G - e$, iar dacă $e \in E(P)$, atunci substituind e în P cu drumul $C - e$ se obține un mers în $G - e$ cu aceleași extremități ca și P, din care se poate extrage un drum cu aceleași extremitați ca și P. Cum G este conex, rezultă din această observație că $G - e$ este conex contrazicând ii).

i) $\rightarrow$ *iii)* Trebuie dovedit că $\forall e \in E(\overline{G})$, $G + e$ are un circuit. Evident, putem presupune că $G \neq K_1, K_2$ (situație în care iii are loc întrucât $E(\overline{G})=0$). Fie $e = uv \in E(\overline{G})$. Din conexiunea lui G rezultă că există un uv-drum P în G. Atunci P împreună cu e generează un circuit în $G + e$.

iii) $\rightarrow$ *i)* Trebuie dovedit că G este conex. Dacă, prin reducere la absurd, n-ar fi așa, atunci consderând u, v în componente conexe diferite ale lui G, rezultă că $uv \in E(\overline{G})$ și deci conform condiției iii), $G + uv$ conține un circuit C. Dar atunci $C - uv$ este un uv - drum în G contrazicând alegerea celor două vârfuri.

Definiție: Fie $G=(V, E)$ un (multi) graf. Se numește arbore parțial al lui G, un graf parțial $T = (V, E')(E' \subseteq E)$ care este arbore. Vom nota cu T_G mulțimea arborilor parțiali ai lui G.

Observație: $T_G \neq 0$ dacă și numai dacă G este conex.

În adevăr, dacă $T_G \neq 0$, atunci există un arbore parțial $T=(V, E')$ al lui G. T este conex, deci între orice două vârfuri ale lui G există un drum cu muchii din $E' \subseteq E$. Prin urmare G este conex.

Reciproc, dacă G este conex, atunci considerăm următorul algoritm:

1. $T := G$
2. **while** ($\exists\, e \in E(T)$ astfel încât $T \setminus \{e\}$ este conex) **do**
   $T := T \setminus \{e\}$

Graful T obținut este graf parțial al lui G, este conex (din ipoteză, după atribuirea din 1, așa este și din condiția lui **while**, T este conex după fiecare iterație) și în plus la oprirea algoritmului, T satisface condiția ii) din teorema 1, deci este arbore.

O altă demonstrație a reciprocei anterioare se bazează pe observația că $G = (V, E)$ este conex dacă și numai dacă oricare ar fi o partiție (V_1, V_2) a lui V există $e = v_1v_2 \in E$ cu $v_i \in V_i$ $i=\overline{1,2}$.

Dacă $|V| = n > 0$ atunci următorul algoritm construiește un arbore parțial al lui G.

begin

1. $T_1 := (\{v\}, 0)(v \in V$, oarecare$)$; $k := 1$;
2. **while** $k<n$ **do begin**
   Determină $v_1v_2 \in E$ cu $v_1 \in V(T_k)$, $v_2 \in V \setminus V(T_k)$;
   { există o astfel de muchie din conexiunea lui G }
   $V(T_{k+1}) := V(T_k) \cup \{v_2\}$;
   $E(T_{k+1}) := E(T_k) \cup \{v_1v_2\}$;
   $k := k+1$;

end

end.

Se observă că T_k este arbore $\forall k = \overline{1,n}$ (inductiv, dacă T_k este arbore, atunci din construcție T_{k+1} este conex și nu are circuite), și în plus se verifică imediat că $|V(T_k)| = k$ iar $|E(T_k)| = k-1$ $\forall$ $k=1, 2, \ldots, n$.

Această demonstrație aplicată unui arbore G cu n vârfuri dovedește că G

are *n - 1* muchii. Această proprietate poate fi folosită pentru completarea teoremei 1 cu alte caracterizări ale arborilor:

Teorema 1'. Următoarele afirmații sunt echivalente pentru un graf *G = (V, E)* cu *n* vârfuri.
(i) G este arbore.
(ii) G este conex și are n-1 muchii.
(iii) G este fără circuite și are n-1 muchii.
(iv) G = K_n pentru n = 1, 2 și G ≠ K_n pentru n≥ 3 și adăugarea unei muchii la G produce exact un circuit.

Demonstrația teoremei 1' este la fel de simplă ca cea a teoremei 1 și o omitem (*ii* $\Rightarrow$ *iii* se bazează pe existența unei muchii cu extremitățile în clase diferite ale oricărei partiții ale mulțimii vârfurilor unui graf conex; considerând una din clase mulțimea vârfurilor unui circuit și repetând observația precedentă se obține că *G* are măcar *n* muchii, contradicție).

Numărarea și enumerarea arborilor parțiali

Familia T_G a arborilor parțiali ai unui (multi)graf are proprietăți interesante. Vom prezenta o metodă (tip backtrak) de generare a elementelor lui T_G, problemă de interes practic în multe aplicații (de exemplu, în chimie).

Fie *G = (V, E), V= {1, 2, . . . , n}*, |*E*|=m. Considerăm că *E* este dată printr-un tablou *E = array [1...m, 1..2]* of V unde, dacă *v= E[i,1]* și *w=E[i, 2]* , atunci *vw* este muchia *i* a grafului *G (i=*$\overline{1,m}$*)*. Vom presupune în plus, că primele $d_G(v_0)$ muchii din tabloul *E* satisfac *E[i,1] = v_0* unde $v_0 \in V$ este un vârf oarecare.

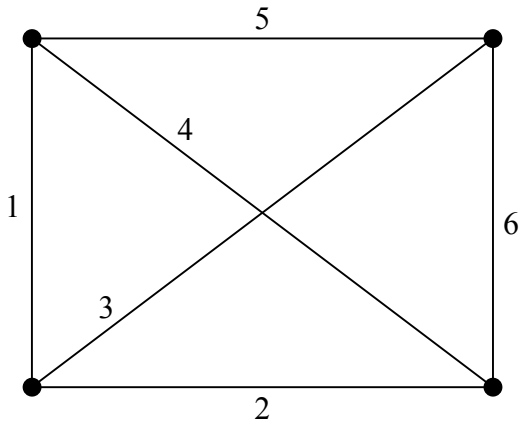
Un arbore parțial $T \in T_G$ va fi identificat cu mulțimea indicilor ce reprezintă muchiile sale în tabloul *E* (submulțime a lui *{1, . . . , m}* de cardinal *n-1*). Pe tot parcursul generării dispunem de un tablou global *T = array[1..n - 1] of 1..m* și de un indicator *i* având semnificația: în arborele curent care se construiește, primele *i - 1* muchii sunt

$$T[1] < T[2] < \ldots < T[i-1] \quad (i \in \{1, \ldots, n\}).$$

procedure *generare-arbori-parțiali(i : integer);*
{ se generează toți arborii parțiali ai lui G
având drept prime *i - 1* muchii elementele *T(1), . . . , T(i-1)*
ale tabloului *E* (ordonate crescător) }
var *j : 1..m; S:* listă de vârfuri; *x : V;*
begin
  if *i = n* **then begin**
    { T(1), . . . , T(n - 1) formează un

```
            arbore parțial}
            prelucrează T ( listează, memorează etc.)
        end
        else  if i = 1 then for j := 1 to d_G(v_0) do begin
            T[i] := j; A:
            generare-arbori-parțiali(i + 1); B:
        end
        else for j := T[i - 1] to m do
            if ⟨{T[1], . . . , T[i - 1]} ∪ {j}⟩_G nu are circuite
            then begin
                T[i] := j; A:
                generare -arbori-parțiali(i + 1); B:
            end
end.
```

Apelul *generare-arbori-parțiali(1)* rezolvă problema enumerării elementelor lui T_G. Dacă $G = K_4$ cu muchiile numerotate ca mai jos:



atunci arborii generați sunt următorii 16: 123, 125, 126, 134, 136, 145, 146, 156, 234, 235, 245, 246, 256, 345, 346, 356.

Pentru implementarea eficientă a testului dacă graful parțial $\langle \{T[1], \ldots, T[i-1]\} \cup \{j\} \rangle_G$ nu are circuite să observăm că din construcție,

$$\langle \{T[1], \ldots, T[i-1]\} \rangle_G$$

nu are circuite, deci componentele sale conexe sunt arbori.

Vom considera o variabilă globală *rad = array*[*1..n*] of *V* cu semnificația *rad*[*v*] = rădăcina arborelui la care aparține vârful *v* (unul din vârfurile acestui arbore).

Înaintea apelului generare-arbori-parțiali(1) se inițializează *rad*[*v*] := *v* ($\forall v \in V$), ceea ce corespunde faptului că $\{T[1], \ldots, T[i-1] = 0$.

Dacă în cursul apelurilor (recursive) se încearcă plasarea muchiei *j* la

mulțimea curentă $T[1], \ldots, T[i-1]$, fie $v = E[j,1]$ și $w = E[j,2]$ Atunci $\langle \{T[1], \ldots, T[i-1]\} \cup \{j\} \rangle_G$ nu are circuite dacă și numai dacă muchia vw nu are extremitățile în aceeași componentă a lui

$$\langle \{T[1], \ldots, T[i-1]\} \rangle_G,$$

adică dacă și numai dacă $rad[v] \neq rad[w]$.
Vectorul *rad* trebuie întreținut pentru a avea semnificația dorită. Acest lucru se obține înlocuind în procedura descrisă, instrucțiunile (vide) etichetate A și B. Astfel: A; se va înlocui cu secvența

$S := \emptyset$; $x := rad[v]$;
for $u \in V$ **do if** $rad[u] = x$ **then begin**
    $S := S \cup \{u\}$;
    $rad[u] := rad[w]$;
end

(deci) arborele cu rădăcina x se "unește" cu arborele cu rădăcina $rad[w]$; se salvează în S vârfurile arborelui cu rădăcina x). După apelul lui trebuie refăcut vectorul *rad* la valoarea de dinainte de apel, deci se va înlocui B; cu

$$\text{for } u \in V \text{ do } rad[u] := x.$$

Numărul elementelor lui T_G, problemă interesantă chiar și numai pentru analiza algoritmului precedent, se poate determina eficient. Prezentăm în continuare una din soluțiile posibile.

Fie $G = (V, E)$ un multigraf cu $V = \{1, 2, \ldots, n\}$. Considerăm $A = (a_{ij})_{n \times n}$ matricea de adiacanță a lui G (a_{ij} = multiplicitatea muchiei ij dacă $ij \in E$, altfel 0. Fie

$$D = diag(d_G(1), d_G(2), \ldots, d_G(n)).$$

Matricea $M[G] = D - A$ se numește *matricea de admitanță a multigrafului G*. Să observăm că în $M[G]$ suma elementelor de pe fiecare linie și fiecare coloană este 0.

Teorema 2. *(Kirchoff-Trent) Dacă G este un multigraf cu mulțimea de vârfuri 1, ... , n și M[G] matricea de admitanță, atunci*

$$| T_G | = det(M[G]_{ii}) \ \forall i \in \{1, \ldots, n\}.$$

Observații: 1⁰ $M[G]_{ij}$ notează minorul lui $M[G]$ obținut prin îndepărtarea liniei i și coloanei j.

2^0 $| T_G | = n^{n-2}$ *(Cayley).* În adevăr,

$$M[K_n] = \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & \ddots & \iddots & \vdots \\ \vdots & \iddots & \ddots & -1 \\ -1 & \cdots & -1 & n-1 \end{pmatrix}$$

$$det(M[K_n]_{11}) = \begin{vmatrix} n-1 & -1 & \cdots & -1 \\ -1 & \ddots & \iddots & \vdots \\ \vdots & \iddots & \ddots & -1 \\ -1 & \cdots & -1 & n-1 \end{vmatrix} = n^{n-2}$$

Observație. Teorema oferă un algoritm polinomial de determinare a lui $|T_G|$.

Arbori parțiali de cost minim.

Considerăm următoarea problemă

(P1)

*Date: G = (V, E) graf și c: **E** -> **R** (c(e) - costul muchiei e).*
Să se determine $T^* \in T_G$ *astfel încât* $c(T^*) = min\{c(T) \mid T \in T_G\}$,
unde $c(T) = \sum_{e \in E(T)} c(e)$.

Algoritmii cunoscuți pentru rezolvarea problemei (P1) au la bază următoarea ideie.

Se consideră inițial, familia $T^0 = (T_1^0, T_2^0, \ldots, T_n^0)$ de arbori disjuncți $T_i^0 = (\{i\},0)$ $i = \overline{1,n}$ (am presupus, ca de obicei, că $V = \{1, 2, \ldots, n\}$).

În pasul general *k (k = 0, 1, . . . , n-2)* al algoritmului se dipune de familia $T^k = (T^k_1, T^k_2, \ldots, T^k_{n-k})$ de *n-k* arbori disjuncți astfel încât $(V(T_i^k))_{i=1,n-1}$ constituie o partiție a lui V și se construiește T^{k+1} astfel:

- se alege T_s^k unul din arborii familiei T^k.
- dintre toate muchiile lui G cu o extremitate în T_s^k și cealaltă în $V - V(T_s^k)$ se alege una de cost minim, $e^* = v_s v_{j^*}$ unde $v_{j^*} \in V(T_{j^*}^k)$ $j^* \neq s$.
- $T^{k+1} = (T^k \setminus \{T_s^k, T_{j^*}^k\}) \cup T$ unde T este arborele obținut din T_s^k și $T_{j^*}^k$ la care adăugăm muchia e^*.

Se verifică imediat că noua familie este formată din arbori disjuncți care partiționează mulțimea de vârfuri ale grafului G. Dacă alegerea umchiei e^* nu este posibilă, atunci rezultă că G nu este conex și deci problema *(P1)* nu are soluție.

Evident, familia T^{n-1} construită în pasul $n-1$ este formată dintr-un singur arbore T_1^{n-1}.

Teorema 3. *Dacă G = (V, E) este un graf conex cu V= {1, 2, . . . , n} atunci arborele construit de algoritmul descris mai sus este arbore parțial de costminim.*

Demonstrație. Vom arăta că $\forall k \in \{1,2, \ldots, n\})$ $\exists T^*$ arbore parțial de cost

minim al lui G, astfel încât $E(\Gamma^k) = \bigcup_{i=1}^{n-k} E(T_i^k) \subseteq E(T^*)$.

În particular, pentru $k = n - 1$, $E(\Gamma^{n-1}) = E(T_i^k) \subseteq E(T^*)$ va implica $T_1^{n-1} = T^*$ (cei doi arbori având același număr de muchii, rezultă că incluziunea are loc cu egalitate) și teorema este demonstrată.

Pentru $k = 0$, afirmația este trivial adevărată: $E(\Gamma^0) = \varnothing$ și din conexiunea grafului G, T_G este nevidă deci există T^* soluție a problemei *P1.*

Dacă afirmația este adevărată pentru $0 \le k \le n - 2$, atunci avem $E(\Gamma^k) \subseteq E(T^*)$ (T^* arbore parțial de cost minim) și $E(\Gamma^{k+1}) = E(\Gamma^k) \cup \{e^*\}$. Dacă $e^* \gamma E(T^*)$, atunci, evident, $E(\Gamma^{k+1}) \subseteq E(T^*)$ și deci afirmația are loc pentru $k+1$.

Presupunem, deci, că $e^* \notin E(T^*)$. Atunci $T^* + \{e^*\}$ conține exact un circuit C ce trece prin muchia $e^* = v_s v_{j^*}$. Cum $v_{j^*} \notin V(T_s^k)$ rezultă că C va conține o muchie $e_1 \neq e^*$ cu o extremitate în $V(T_s^k)$ și cealaltă în $V \setminus V(T_s^k)$. Din alegerea muchiei e^*, avem $c(e^*) \le c(e_1)$ și $e_1 \gamma E(T^*) \setminus E(T^k)$.

Fie $T^1 = (T^* + \{e^*\}) - \{e_1\}$. $T^1 \gamma T_G$ (este conex și are $n - 1$ muchii). În plus, $c(T^1) = c(T^*) + c(e^*) - c(e_1) \le c(T^*)$ deci $c(T^1) = c(T^*)$ (T^* este de cost minim) și $E(\Gamma^{k+1}) \subseteq E(T^1)$.

Observații: 1^0 Demonstrația anterioară rămâne valabilă pentru funcții de cost $c : T_G \to \boldsymbol{R}$ astfel încât $\forall T \in T_G$, $\forall e \in E(T)$, $\forall e' \notin E(T)$

$$c(e') \le c(e) \Rightarrow c((T + e') - e) \le c(T)$$

2^0 În algoritmul descris nu s-a precizat modul de alegere al arborelui T^k. Vom considera, în continuare două strategii de alegere a acestui arbore.

Algoritmul lui Prim(1957) (implemetarea este datorată lui *Dijkstra.* 1961). Arborele T_s^k va fi întotdeauna arborele cu cele mai mule vârfuri dintre arborii familiei curente. Rezultă deci, că la fiecare pas $k > 0$, vom avea un arbore cu $k + 1$ vârfuri, ceilalți $n - k - 1$ având câte un singur vârf. Notăm $T_s = (V_s, E_s)$ arborele curent. Considerăm α = *array* [*1.. n*] *of V* și β = *array* [*1..n*] *of real* tablouri cu următoarea semnificație:

(S) $\forall j \in V - V_s$, $\beta[j] = c(\alpha[j]j) = min \{ c(ij) / i \in V_s, ij \in E\}$

Descrierea algoritmului.

begin

1: $V_s := \{s\}$; $\{s \in V$, *oarecare*$\}$.
$E_s := \varnothing$;
for $v \in V \setminus \{s\}$ **do begin** $\alpha[v] := s$; $\beta[v] := c(sv)$ **end;**
{ dacă $ij \notin E$ atunci $c(ij) = \infty$ }

2: **while** $V_s \neq V$ **do begin**
determină $j^* \in V \setminus V_s$; $\beta[j^*] = min\{\beta[j] / i \in V - V_s\}$;
$V_s := V_s \cup \{j^*\}$;

```
            E_s := E_s ∪ {α[j*]j*};
            for j ∈ V − V_s do if β[j] > c(j*j) then begin
                    β[j] := c(j*j);
                    α[j] := j*;
            end
        end
end.
```

Se observă că *(S)* este satisfăcută de inițializările pasului 1, iar în pasul 2 se respectă, pe de o parte, strategia generală de alegere a muchiei de cost minime cu exact o extremitate în V_s (alegerea lui j^*) și pe de altă parte se menține valabilitatea condiției *(S)* pentru iterația următoare (testul asupra valorii curente a lui $\beta[j]$).

Complexitatea algoritmului este $O(n-1 + n-2 + \ldots + 1) = O(n^2)$, dată de operațiile din pasul 2 necesare determinării minimului și actualizărilor tablului β. Se poate introduce testul de conexiune a grafului, după determinarea lui $\beta[j^*]$. Algoritmul este recomandat în cazul grafurilor cu multe muchii, m = $O(n^2)$.

Algoritmul lui Kruskal (1956) În metoda generală prezentată, se va alege la fiecare pas drept arbore T_s^k unul din cei doi arbori cu proprietatea că sunt "uniți" printr-o muchie de cost minim printre toate muchiile cu extremitățile pe arbori diferiți. Această alegere, oarecum complicată, se realizează simplu prin sortarea muchiilor grafului nedescrescător în raport cu costurile și apoi prin examinare în mod "greedy" a listei obținute.

Dacă notăm cu $T = E(\Gamma^k)$, atunci algoritmul poate fi descris, astfel:

```
begin
    1. Sortează E = (e_1, e_2, ..., e_m) astfel încât:
                c(e_1) ≤ c(e_2) ≤ ... ≤ c(e_m).
        1.2     T := ∅; i:= 1;
    2.          while i ≤ m do begin
                        if ⟨T ∪ {e_i}⟩_G nu are circuite then
                                T := T ∪ {e_i};
                        i :=i + 1;
                end
end.
```

Evident, pasul 1 necesită $O(m \log n)$ operații. Pentru realizarea eficientăa a testului din pasul 2 va fi necesar să reprezentăm la fiecare pas k (din metoda generală) $V(T_1^k), V(T_2^k), \ldots, V(T_n^k)$ și să testăm dacă muchia e_i curentă are ambele extremități în aceeași mulțime. Se vor folosi, pentru reprezentarea acestor mulțimi, arbori (care nu sunt în general subarbori ai lui G).

Fiecare astfel de arbore va avea un vârf, numit *rădăcină*, care va desemna mulțimea de vârfuri ale lui G pe care o reprezintă. Vom folosi o funcție *find(v)*

care determină în ce mulțime este vârful *v*, adică determină rădăcina arborelui care reprezintă mulțimea de vârfuri la care aparține *v*.

Pentru realizarea reuniunilor (disjuncte) de mulțimi de vârfuri care apar în transformarea familiilor T^k (din metoda egenrală) vom folosi o procedură *union(u,v)* cu semnificația: "mulțimile de vârfuri (diferite și disjuncte) la care aparțin *v* și *w* se unesc în una singură".

Cu aceste proceduri, pasul 2 al algoritmului se scrie:

```
2.    while i ≤ m do begin
          fie e_i = vw;
          if find(v) ≠ find(w) then union(u,w);
          i := i + 1;
      end
```

Complexitatea algoritmului va depinde de modul de implementare a funcției *find* și a procedurii *union.*

Soluția I. Considerăm tabloul *rad = array*[*1..n*] *of V* cu semnificația *rad*[*v*] = rădăcina arborelui ce reprezintă mulțimea la care aparține vârful *v*.
Adăugăm pasului 1, inițializarea

```
1.3  for v ∈ V do rad[v] := v;
```

care corespunde familiei T^0. Funcția *find* are în acest caz comlpexitatea *O(1)* și este:

```
function find(v: V): V;
begin
    find := rad[v];
end.
```

Procedura *union* necesită *O(n)* operații:

```
procedure union(v, w: V);
var i: V;
begin
    for i ∈ V do if rad[i] = rad[v] then rad[i] := rad[w];
end.
```

Pasul 2 al algoritmului necesită *O(m)* apeluri ale funcției *find* (exact *m* așa cum l-am descris, sau *O(m)* dacă se introduce un test asupra cardinalului mulțimii *T* curente). În secvența acestor *O(m)* apeluri ale funcției *find* se vor intercala *n – 1* apeluri ale procedurii *union*. Rezultă că în total pasul 2 necesită $O(m + (n-1)O(n)) = O(n^2)$ operații. Deci complexitatea algoritmului este $O(max(m\log n, n^2))$. Dacă graful este plin, $m = O(n^2)$, se observă că acest algoritm este mai puțin eficient

decât cel al lui Prim.

Soluția a II^a. Considerțm *pred = array*[*1..n*] *of integer* un tablou cu interpretarea "*pred*[*v*] = vârful dinaintea lui *v* de pe drumul unic la *v* de la rădăcina arborelui în care este memorată mulțimea la care aparține *v*.

pred[*v*] = *0* ⇔ *v* este rădăcina acestui arbore".

Adăugăm în pasul 1, inițializarea

1.3 **for** $v \in V$ **do** *rad*[*v*] := *0;*

Cu această reprezentare a arborilor, vom modifica pasul 2 al algoritmului pentru a realiza în timp constant fiecare reuniune care apare:

```
2.    while i ≤ m do begin
          fie e_i = vw;
          x := find(v);
          y := find(w);
          if x ≠ y then union(u,w);
          i := i + 1;
      end
```

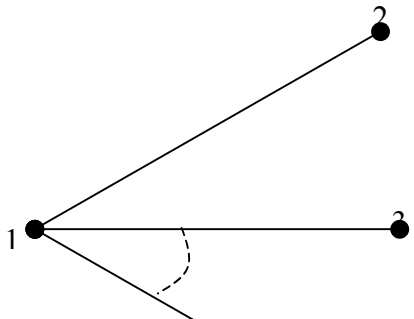
Deci procedura *union* va fi apelată numai pentru argumente reprezentând rădăcini de arbori diferiți

```
procedure union(v, w: V);
begin
    pred[v] = w;
end.
```

Evident, complexitatea procedurii *union* este *O(1)*. Funcția *find* este în acest caz mai complicată:

```
function find(v: V): V;
begin
    i := v;
    while pred[i] > 0 do i := pred[i];
    find := i;
end.
```

Complexitatea lui *find(v)* este *O(h(v))* unde *h(v)* este lungimea drumului din arborelee care-l conține pe *v* de la vârful *v* la rădăcina acestui arbore. Dacă graful *G* este $K_{1,n-1}$ desenat mai jos,



și lista ordonată a muchiilor $E = \{12, 13, ..., 1n\}$, atunci execuția algoritmului provoacă următorul șir de apeluri ale procedurii *union(U)* și functiei *find(F):* *F(1), F(2), U(1,2), F(1), F(3), U(2,3), F(1), F(4), U(3,4),..., F(1), F(n), U(n-1,n).* Apelurile $F(i)$ $(i > 1)$ și $U(i, i+1)$ $i \geq 1$necesită în total $O(n)$ operații. Șirul de $F(1)$ necesită însă $O(1 + 2 + ... + n - 1) = O(n^2)$ operații.

Este deci posibil ca pasul 2 al algoritmului în această implementare să fie de complexitate $\Omega(n^2)$ chiar dacă graful este rar.

Deficiența acestei implementări este datorată posibilității ca în procedura *union* să declarăm rădăcină nouă pentru cei doi arbori pe cea a celui cu mai puține vârfuri, ceea ce are ca efect posibilitatea ca $h(v)$ să devină mare , $O(n)$, pe parcursul algoritmului. Aecst defect poate fi evitat dacă la execuția lui *union* ținem seama de cardinalul celor două mulțimi. Se poate memora cardinalul unei mulțimi în componenta tabloului *pred* corespunzătoare rădăcinii arborelui care memorează acea mulțime. Mai precis, considerăm inițializarea

1.3 **for** $v \in V$ **do** *rad*[*v*] := -1;

și modificăm procedura *union* astfel încât să asigurăm îndeplinirea condiției

pred[*v*] $< 0 \Leftrightarrow$ *v* este rădăcină a unui arbore și $-$*pred*[*v*] este cardinalul mulțimii memorate în el.

Procedura *union* are, în acest caz tot complexitatea $O(1)$, dar selectează drept rădăcină pe cea care corespunde cu mai multe vârfuri:

```
procedure union(v, w: V);
        { v și w sunt rădăcini }
var t: integer;
begin
        t := pred[v] + pred[w];
        if pred[v] > pred[w]
                then begin pred[v] := w; pred[w] := t; end
                else begin pred[w] := v; pred[v] := t; end
end.
```

Cu această implemetare a funcției *find* și procedurii *union* pe tot parcursul algoritmului are loc:

(*) $\forall v \in \; -pred[find(v)] \geq 2^{h(v)}$

(reamintim că $h(v)$ notează lungimea drumului de la v la rădăcina $find(v)$ a arborelui ce memorează v).
După inițializarea1.3, $\forall v \in V \; h(v) = 0$ și $find(v) = v$ iar $-pred[v] = 1$, deci (*) are loc cu egalitate.

Dacă, înaintea unei iterații din pasul 2, (*) are loc, atunci, dacă în acea interație nu se execută *union*, nu se modifică *pred* și deci (*) rămâne valabilă și după execuție.

Presupunem prin urmare că se apelează $union(x, y)$ și că se execută $pred[y] := x$. Aceasta înseamnă că înaintea acestei iterații avem $-pred[x] \geq -pred[y]$. Să observăm că singurele vârfuri v cărora li se modifică $h(v)$ după execuția iterației curente sunt cele care înaintea iterației satisfăceau $find(v) = y$, pentru care aveam $-pred[y] \geq 2^{h'(v)}$.
După execuția iterației avem $h'(v) = h(v) + 1$ iar $find(v)' = x$, și deci trebuie să verificăm că $-pred'[y] \geq 2^{h'(v)}$. Avem $-pred'[x] = -pred[x] - pred[y] \geq 2*(-pred[y]) \geq 2*2^{h(v)} = 2^{h(v)+1} = 2^{h'(v)}$.

Rezultă că (*) are loc pe tot parcursul algoritmului, deci $\forall v \in V \; h(v) \leq log(-pred'[find[v]] < logn$. Complexitatea pasului 2 va fi deci $O(n - 1 + 2m \log n) = O(m \log n)$ ceea ce-l face superior algoritmului lui Prim pentru grafuri rare.

Soluția a III[a]. Complexitatea pasului 2, cu implementarea precedentă, este datorată apelurilor succesive ale lui *find*. *Tarjan(1976)* a propus ca fiecare apel al lui *find* care necesită parcurgerea unui drum de lungime mai mare decât 1, să "comprime" acest drum, aducându-i vârfurile drept descendenți imediați ai rădăcinii. Mai precis, avem

```
function find(v: V): V;
var i, j, k : integer;
begin
        i := v;
        while pred[i] > 0 do i := pred[i];
        j := v;
        while pred[j] > 0 do begin
                k := pred[j];
                pred[j] := i;
                j : =k;
        end
        find := i;
```

end.

Dacă $A: \mathbf{N}\times\mathbf{N} \to \mathbf{N}$ este funcția lui Ackermann dată de: $A(i, 0) = 0 \ \forall i \geq 0$; $A(i,1) = 2 \ \forall i \geq 1$; $A(0, x) = 2x \ \forall x \geq 0$; $A(i+1, x+1) = A(i, A(i+1, x)) \ \forall i \geq 0 \ \forall x \geq 1$, atunci considerând $\forall m \geq n > 0$

$$\alpha(m, n) = min \{ z \mid A(z, 4\lceil m/n \rceil) \geq \log n, z \geq 1 \}$$

avem: Complexitatea pasului 2, utilizând *union* din soluția a II-a și *find* descris mai sus, este $O(m\alpha(m, n))$. Notăm că $\alpha(m, n)$ crește extrem de încet (pentru valorile practice ale lui n, $\alpha(m, n) \leq 3$ și deci se obtine ca această ultimă implementare este practic liniară (în raport cu m).

Arborescențe

Fie $G = (V, E)$ un digraf. Se numște *rădăcină* a lui G un vârf $v_0 \in V$ astfel încât $\forall v \in V, \mathbf{D}_{v_0 v} \neq \varnothing$ (orice vârf al digrafului care este accesibil din v_0 printr-un drum în G).

Definție. Se numește *arborescență* un graf orientat $H = (V, E)$ (oricare ar fi două vârfuri , există cel mult un arc cu extremitățile aceste două vârfuri $\Leftrightarrow G(H)$ este graf), astfel încât:

(i) H are o rădăcină

(ii) $G(H)$ este arbore.

Definiție. Un digraf $G = (V, E)$ se numește *quasi-tare conex* dacă $\forall v, w \in V, \exists z \in V$ astfel încât $\mathbf{D}_{zv} \neq \varnothing$ și $\mathbf{D}_{zw} \neq \varnothing$.

Să observăm că dacă G este tare conex atunci G este quasi-tare conex (se poate lua drept z în definița anterioară, unul din cele două vârfuri). Pe de altă parte, dacă G este quasi-tare conex atunci G este conex.

Lema 1. *G este qusi-tare conex dacă și numai dacă G are o rădăcină.*

Demonstrație. Dacă G are o rădăcină v_0, atunci vârful z din definiția quasi-tarei conexiuni poate fi luat v_0.

Reciproc, dacă G este tare-conex atunci G are o rădăcină. În adevăr, dacă $|V(G)| = 2$ atunci unul din cele două vârfuri este rădăcină.

Dacă, $|V(G)| \geq 3$, atunci considerăm două vârfuri distincte $v_1, v_2 \in V(G)$ oarecare și $z \in V(G)$ *astfel incit* $\mathbf{D}_{zv_1} \neq \varnothing, \ \mathbf{D}_{zv_2} \neq \varnothing$. Fie următorul algoritm:

begin

1: $T := \{v_1, v_2\} \cup \{z\}; \ v_0 := z;$

2: **while** $T \neq V$ **do begin**

fie $u \in V - T;$

fie $t \in V : \boldsymbol{D}_{tu} \neq \varnothing$ și $\boldsymbol{D}_{tv_0} \neq \varnothing$; {$\exists t$};
$T := T \cup \{u\} \cup \{t\}$;
$v_0 := t$;
end
end.

După inițializarea din 1, are loc

(I) $\forall t \in T, \boldsymbol{D}_{v_0 t} \neq \varnothing$

După fiecare iterație din pasul 2, măcar un vârf se adaugă la T și în plus condiția (I) rămâne satisfăcută. Rezultă că vârful v_0 construit de algoritm este rădăcină a digrafului G.

Are loc următoarea teoremă de caracterizarea arborescețelor.

Teorema 4. *Fie H = (V, E) graf orientat cu n ≥ 2 vârfuri. Următoarele afirmații sunt echivalente:*

(i) *H este quasi-tare conex și G(H) nu are circuite.*
(ii) *H este quasi-tare conex și are n – 1 arce.*
(iii) *H este arborescență.*
(iv) $\exists v_0 \in V$ *astfel încât* $\forall v \in V \; |\boldsymbol{D}_{v_0 v}| = 1$.
(v) *H este quasi-tare conexși minimal cu această proprietate (*$\forall e \in E$, *H – e nu este tare conex).*
(vi) *H conex și* $\exists v_0 \in V$ *astfel încât* $d_H^-(v_0) = 0$ *și* $\forall v \neq v_0 \; d_H^-(v) = 1$
(vii) *G(H) nu are circuite și* $\exists v_0 \in V$ *astfel încât* $d_H^-(v_0) = 0$ *și* $\forall v \neq v_0 \; d_H^-(v) = 1$

Demonstrația acestei teoreme este simplă ($i \Rightarrow ii \Rightarrow ... \Rightarrow vii \Rightarrow i$) și o omitem.

Definiție. Fie $G = (V, E)$ un digraf. Se numește *arborescență parțială* a lui G un digraf parțial $H = (V, E')$ $(E' \subseteq E)$ cu proprietatea că H este arborescență. Notăm

$$A_G = \{ H \mid H \text{ arborescență parțială a lui } G \}.$$

Datorită caracterizărilor *vi* și *vii* ale teoremei precedente, în limba engleză, o arborescență parțială se mai numește și *out-tree*.

Consecință. $A_G \neq \varnothing$ *dacă și numai dacă G este quasi-tare conex.*
(evident, datorită lui (v) din teorema 4).

O aplicație în informatică.

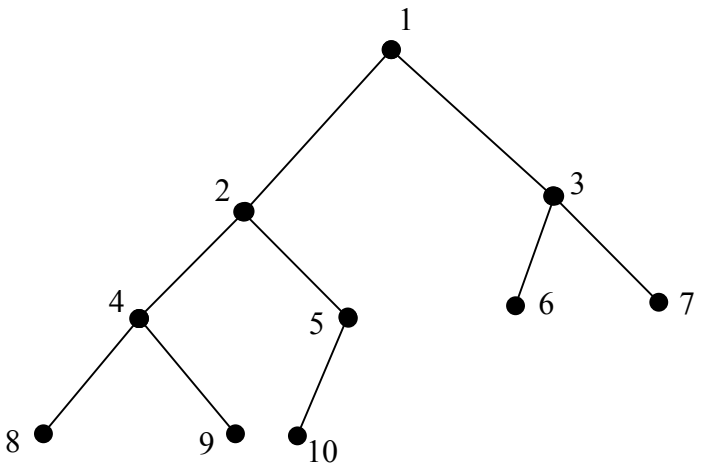
În informatică, arborescențele sunt numite, prin abuz de limbaj, *arbori*,

specificîndu-se rădăcina și considerând implicită orientarea muchiilor corespunzător parcurgerii drumului unic de la rădăcină la fiecare vîrf. Dacă H este o arborescență și $vw \in E(H)$ atunci w se numește descendent direct al lui v, iar $d_H^+(v)$ se numește gradul lui v în H și se notează $d_H(v)$. Dacă H este o arborescență astfel încît $d_H(v) \in \{0, 1, 2\}$ atunci H se numește *arbore binar*. Dacă $d_H(v) = 2$ cei doi descendenți sînt unul *la stînga* și celălalt *la dreapta*. Vârfurile cu $d_H(v) = 0$ se numesc *terminale sau pendante (frunze*), vârfurile cu $d_H(v) > 0$ sunt *interioare*.

Grămezi.

Considerăm $\forall n \in \mathbf{N^*}$ $H_n = (\{1, 2..., n\}, E)$, unde $\forall i \in \{2, ..., n\}$ avem arcul $\left[\frac{i}{2}\right] i \in E$.

Exemplu: Pentru $n=10$



Se observă că vârfurile lui H_n sunt dispuse pe $k+1$ nivele $0, ..., k$, unde $k = \lfloor \log_2 n \rfloor$, un nivel i conținând toate vârfurile aflate la distanța i față de rădăcina 1. Pe nivelul i avem 2^i vârfuri, cu posibila excepție a ultimului nivel, care poate fi incomplet.

Definiție. Tabloul $A[1..n]$ cu elemente dintr-o mulțime total ordonată ($\leq$) formează o *grămadă binară* ("heap" în limba engleză) dacă:

(H) $\qquad \forall i \in \{2,...,n\} \qquad A[i] \geq A[\lfloor i/2 \rfloor]$

(interpretare: dacă se atașează fiecărui vârf i al lui H_n valoarea $A[i]$ atunci condiția *(H)* cere ca valoarea fiecărui vârf să fie $\leq$ decât valoarea descendenților

săi).

Problemă. Dat $A[1..n]$, să se organizeze ca o grămadă.

O primă soluție este aceea de a insera un element într-o grămadă deja existentă. Aplicăm acest algoritm de $n - 1$ ori, prima data într-o grămadă cu 1 element, și continuând până ce toate elementele au fost introduse.

```
procedure insg(var A:tablou; n:integer);
    { inserează valoarea din A[n] în grămada memorată în A[1..n] }
    var j, i: integer; v: tipul elementelor tabloului;
    begin
        j := n; i := ⌊n/2⌋; v := A[n];
        while i > 0 and A[i] > v do begin
            A[j] := A[i]; j := i; i := [j/2];
        end
        A[j] :=v;
    end;
```

Ideea procedurii *insg* este evidentă: pe drumul de la n către 1 în H_n, se caută primul loc în care să-l plasăm pe v, conform cerinței *(H)*. Construirea grămezii este atunci,

```
for i := 2 to n do insg(A,i);
```

Complexitatea construcției: în cazul cel mai nefavorabil, oricare ar fi i, toate cele 2^i noduri de pe nivelul i se plimbă până în rădăcină (pentru a-și găsi locul), deci

$$T_n \le \sum_{i=1}^{\lfloor \log n \rfloor} i2^i < \log n \sum_{i=1}^{\lfloor \log n \rfloor} 2^i = O(n \log n).$$

Există o strategie generală în astfel de probleme, care permite o construcție mai bună : echilibrarea. Dacă în procedura *insg* interpretăm că se combină o grămadă cu $n - 1$ elemente cu o grămadă cu 1 element, atunci următoarea procedură face echilibrarea.

```
procedure combg(i, n: integer);
    { grămada cu rădăcina A[2i] și grămada cu rădăcina A[2i+1] se combină cu
    valoarea A[i] pentru a forma o grămadă cu rădăcina A[i]; tabloul A este
    variabilă globală }
var k, j: integer; v: ...
begin
    k := i; j := 2i; v := A[i];
    while j ≤ n do begin
```

```
        if (j < n) and (A[j] > A[j + 1]) then j:=j+1;
        if v ≥ A[ij] then begin
                A[k] := A[j]; k := j; j:=2j;
                end
        else j := n + 1;
        end;
        A[k] :=v;
end;
```

Se observă că în cel mai rău caz, se va executa o plimbare până la ultimul nivel.

Cu această procedură, construcția grămezii se face astfel

```
procedure    balg(var A:tablou; n:integer);
var i:integer;
begin
        for i := ⌊n/2⌋ downto 1 do combg(i, n);
end;
```

Complexitatea construcției grămezii cu *balg*. Fie $2^{k-1} \le n \le 2^k; (k-1 = \lfloor \log n \rfloor);$ nivelele lui H_n sunt *0, 1,..., k-1,* și pentru fiecare nod de pe nivelul *i* se fac cel mult $k - i$ coborâri. Rezultă :

$$T_n \le \sum_{i=0}^{k-2} 2^i (k-i) = \sum_{j=2}^{k} 2^{k-j} j \le n \sum_{j=2}^{k} j/2^j < 2n = O(n)$$

O coadă cu prioritate este o structură de date, care permite extragerea minimului și introducerea unui nou element, în mod eficient. O grămadă poate fi folosită ca o coadă cu prioritate astfel:

i) Inserția în *O(log n)* operații cu *insg*;

ii) extragerea minimului în *O(log n)* operații, astfel: se extrage elementul din rădăcină; plasăm în rădăcină elementul de pe ultimul și folosim *combg(1, n-1)* pentru a reface grămada .

Notăm că în acest exemplu, utilizarea arborescenței H_n s-a făcut pentru ilustrarea comportării algoritmilor. *Heapsort*, sortarea cu ajutorul grămezilor (*Floyd*) este evidentă:

```
procedure heapsort(A,n);
        { sortează descrescător A[1..n] }
begin
        balg(A,n);
        for i = n downto 2 do begin
                t := A[i]; A[i] := A[1]; A[1] := t;
                combg(A,i-1);
        end
```

end.