Capitolul 6

Arbori oarecare

Reamintim definiţia unui *arbore oarecare*:

Definiţia 6.1 *Fiind dată o mulţime M de elemente denumite noduri, vom numi* ***arbore*** *o submulţime finită de noduri astfel încât:*

1. *există un nod cu destinaţie specială, numit rădăcina arborelui;*
2. *celelalte noduri sunt repartizate în n mulţimi disjuncte două câte două, $A_1, A_2, \cdots, A_n$, fiecare mulţime A_i constituind la rândul ei un arbore.*

6.1 Moduri de reprezentare

Pentru reprezentarea arborilor oarecare pot fi utilizate următoarele metode:

- *legături fiu-frate*: fiu_k - este primul descendent al vârfului k, $frate_k$ - următorul descendent al tatălui nodului k, ce urmează după k. Între descendenţii unui vârf se presupune că definim o relaţie de ordine.

  Rădăcina arborelui din figura 6.1 este $Rad = 1$.

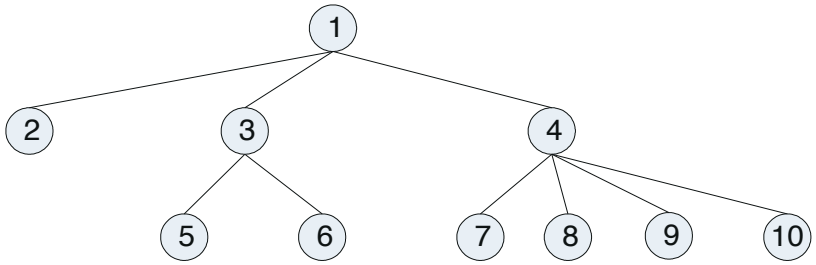

Fig. 6.1: Exemplu de arbore oarecare

Table 6.1: Reprezentarea cu legături fiu–frate a arborelui din figura 6.1.

Nod	1	2	3	4	5	6	7	8	9	10
Fiu	2	0	5	7	0	0	0	0	0	0
Frate	0	3	4	0	6	0	8	9	10	0

Dacă identificăm *Fiu* cu *Stang* şi *Frate* cu *Drept*, unui *arbore oarecare* i se poate asocia un *arbore binar*.

Pentru reprezentarea unui arbore oarecare, modelul de reprezentare *fiu-frate* în varianta de alocare statică poate fi uşor extins la o variantă de alocare dinamică, folosind pointeri pentru legăturile către *primul descendent* respectiv pentru următorul *frate*. Astfel un nod al arborelui poate avea următoarea configuraţie:

```
typedef struct nod {
                      TipOarecare data;
                      struct nod* fiu;
                      struct nod* frate;
                    }Nod;
```

Asemănător cu modelul construit la arbori binari, *rad* ($Nod * rad$;) este o variabilă de tip *pointer la Nod* (variabila păstrează adresa unei zone de memorie). *rad* desemnează adresa rădăcinii arborelui.

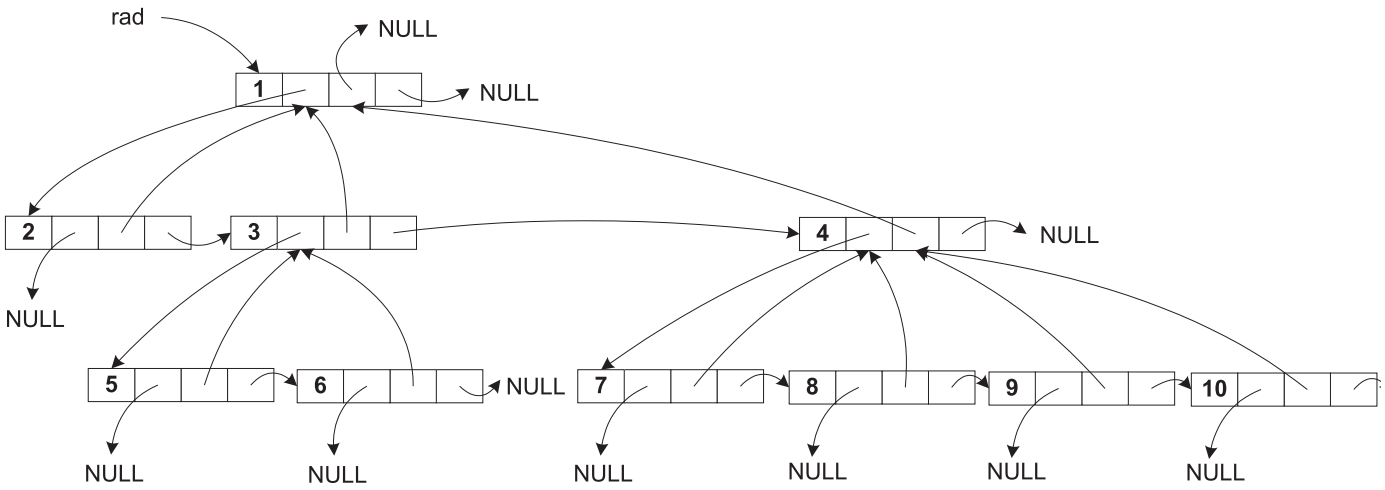

Fig. 6.2: Exemplu de arbore oarecare cu 10 noduri reprezentat prin legături fiu-frate

În figura 6.2 este reprezentat arborele din figura 6.1 prin *legături fiu-frate*.

- *lista descendenţilor*. În cadrul acestui mod de reprezentare fiecare vârf este descris prin lista descendenţilor săi. Pentru memorare se va utiliza un vector cu n componente:

$$C_k = \begin{cases} 0 & , \text{ vârful respectiv nu are descendenţi} \\ j & , \ j \text{ indică adresa (coloana) unde începe lista descendenţilor vârfului } k. \end{cases}$$

Listele de descendenţi se păstrează prin intermediul unei matrice L cu 2 linii şi $N-1$ coloane:

$L_{1,k}$ - un descendent al vârfului a cărui listă conţine coloana k a matricei date.

$$L_{2,k} = \begin{cases} 0 & , \text{dacă descendentul respectiv este ultimul} \\ j & , j \text{ indică coloana unde se află următorul descendent} \end{cases}$$

Table 6.2: Reprezentarea arborelui din figura 6.1 folosind liste cu descendenţi.

Nod	1	2	3	4	5	6	7	8	9	10
C	1	0	4	6	0	0	0	0	0	0

$$L = \begin{pmatrix} 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 3 & 0 & 5 & 0 & 7 & 8 & 9 & 0 \end{pmatrix}$$

Exploatând mai departe această idee, într-un nod al arborelui oarecare se poate păstra o listă cu adresele descendenţilor săi, lista fiind reprezentată sub forma unui tablou alocat static sau dinamic. Oricum modelul se recomandă atunci când numărul descendenţilor unui nod este limitat superior, sau când acesta este cunoscut/stabilit în momentul în care se construieşte arborele. Modificările efectuate asupra arborelui, cum ar fi inserări de descendenţi noi, atunci când intrările alocate pentru aceşti descendenţi sunt deja alocate nu poate conduce decât la realocarea spaţiului de memorie cu un cost reflectat în complexitatea algoritmului. Astfel dacă numărul de descendenţi este limitat superior putem defini

```
#define NMAX 100
typedef struct nod {
                      TipOarecare data;
                      struct nod* children[NMAX];
                    }Nod;
```

sau

```
typedef struct nod {
                      TipOarecare data;
                      int no_of_children;
                      struct nod** children;
                    }Nod;
```

atunci când numărul maxim de descendenţi nu poate fi cunoscut decât în momentul construirii arborelui. Astfel, în acel moment, se alocă spaţiu pentru un tablou de pointeri către structura de tip **Nod** definită (**children = malloc(sizeof(Nod*) * no_of_children)**).

În figura 6.3 este reprezentat arborele din figura 6.1 folosind *liste cu descendenţi*.

- *Tata* - fiecărui nod k se indică nodul părinte.

Table 6.3: Reprezentarea arborelui din figura 6.1 folosind vectorul *tata*.

Nod	1	2	3	4	5	6	7	8	9	10
Tata	0	1	1	1	3	3	4	4	4	4

Acest mod de reprezentare are dezavantajul că nu păstrează în mod explicit ordinea descendenţilor unui nod. Această ordine poate fi dedusă printr–o numerotare adecvată a nodurilor. De obicei reprezentarea cu vectorul *tata* nu este folosită singură ci împreună cu alte moduri de reprezentare, ca o completare.

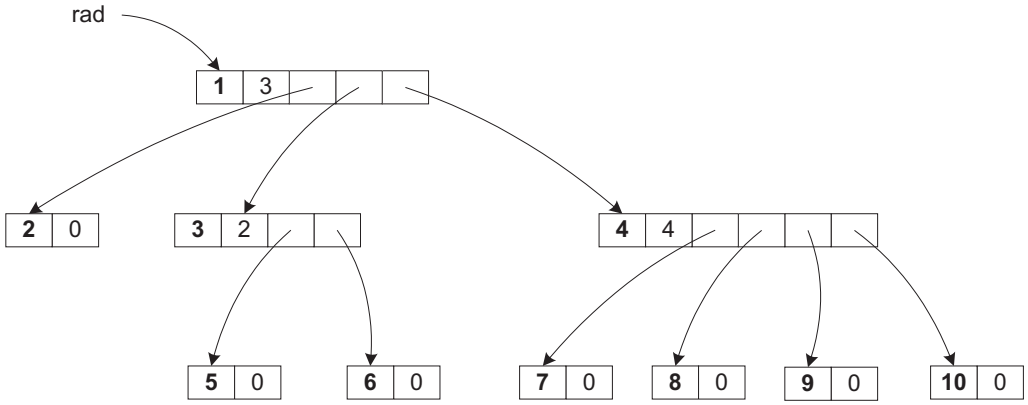

Fig. 6.3: Exemplu de arbore oarecare cu 10 noduri reprezentat prin liste cu descendenţi

De exemplu, reprezentarea unui nod propusă la primul punct, poate fi modifcată astfel încât să incorporeze şi informaţii despre nodul părinte:

```
typedef struct nod {
                    TipOarecare data;
                    struct nod* fiu;
                    struct nod* tata;
                    struct nod* frate;
                  }Nod;
```

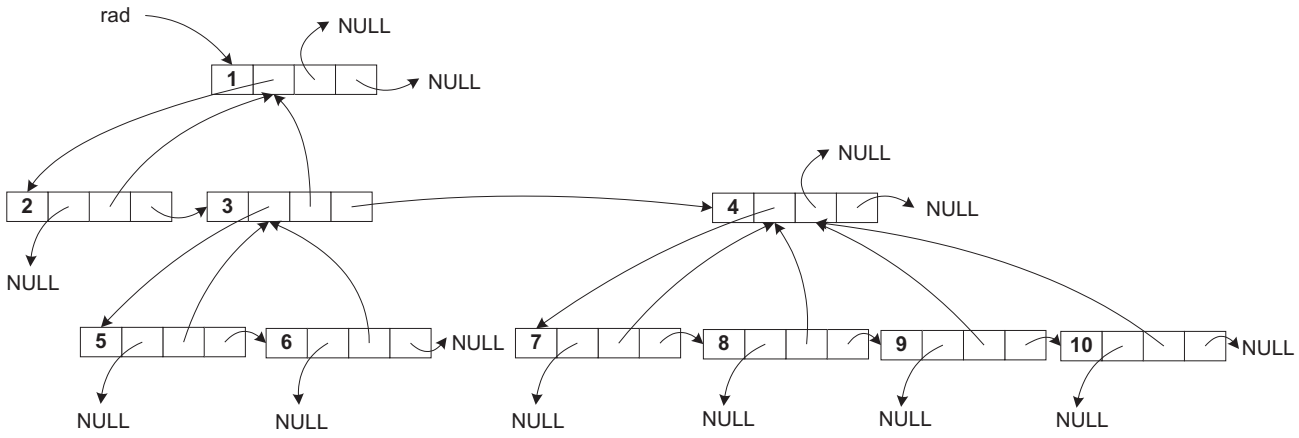

Fig. 6.4: Exemplu de arbore oarecare cu 10 noduri reprezentat prin legături fiu-frate-tata

În figura 6.4, reprezentarea din figura 6.2 este îmbunătăţită prin *legătura tata*.

6.2 Metode de parcurgere

Pentru parcurgerea unui arbore oarecare se pot folosi metodele generale pentru parcurgerea *arborelui binar asociat*. Arborele binar asociat unui arbore oarecare se obţine în urma realizării corespondenţei $Fiu \Rightarrow Stang$ şi $Frate \Rightarrow Drept$.

În plus faţă de acestea, avem şi două metode de parcurgere pe care putem să le numim specifice unui arbore oarecare. Acestea se pot clasifica după momentul în care se realizează vizitarea nodului părinte astfel:

- *A–preordine*: se vizitează mai întâi rădăcina, şi apoi, în ordine, subarborii săi [93], [123]. Metoda este identică cu metoda de parcurgere în preordine a arborelui binar ataşat: $1, 2, 3, 5, 6, 4, 7, 8, 9, 10$ (vezi algoritmul 39). Parcurgerea în *A–preordine* este exemplificată pe arborele oarecare din figura 6.1.

Algoritm 39 Algoritm de parcurgere în A-preordine

```
1: procedure APreordine(Rad, Fiu, Frate)
2:     k ← rad
3:     q ← 0
4:     S ← ∅                                          ▷ iniţializare stivă vidă
5:     while (q = 0) do
6:         if (k ≠ 0) then
7:             call Vizit(k)
8:             S ⇐ k                                  ▷ S.push(k)
9:             k ← fiu_k
10:        else
11:            if (S = ∅) then                        ▷ verificare dacă stiva este vidă
12:                q ← 1
13:            else
14:                S ⇒ k                              ▷ S.pop(k)
15:                k ← frate_k
16:            end if
17:        end if
18:    end while
19:    return
20: end procedure
```

- *A–postordine*: se vizitează mai întâi toţi subarborii ce au drept rădăcini, descendenţii rădăcinii arborelui şi apoi rădăcina arborelui [93], [123]. În urma parcurgerii în *A–postordine* a arborelui din figura 6.1 se obţine $2, 5, 6, 3, 7, 8, 9, 10, 4, 1$.

Exemplul 6.1 *[31] În continuare se prezintă implementarea în limbajul de programare C a variantelor recursive pentru fiecare dintre cele două metode de vizitare ale unui arbore oarecare.*

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

/**
 * Definitii de tipuri necesare cozii
 */
typedef struct nod {
   int id;
   float info;
   struct nod *down, *next;
```

```
}NOD;

typedef NOD* TINFO;

/**
 * Definitii de tipuri pentru reprezentarea arborelui
 */
typedef struct cnod {
   TINFO info; //informatia memorata in nod
   struct cnod *next; //adresa urmatorului element
}CNOD;

CNOD *prim = NULL, *ultim = NULL;

NOD* citListaDescendenti(NOD *up);

/**
 *  Functie ce testeaza daca coada e vida
 * @return  1 daca coada e vida
 *          0 altfel
 */
int isEmpty(void) {
  if (prim == NULL)
    return  1;
  else
    return  0;
}

/**
 *  Functia adauga un element la sfarsitul unei cozi
 * unde n reprezinta elementul ce se adauga.
 */
void add(TINFO n) {
  CNOD *curent;

  curent = (CNOD *) malloc(sizeof(CNOD));
  curent->info = n;
  curent->next = NULL;
  if (!prim)  {
    prim = ultim = curent;
  }  else {
    ultim->next = curent;
    ultim = curent;
  }
}

/**
 * Functia extrage un element din coada. Returneaza elementul scos.
 */
TINFO get(void) {
  CNOD *curent;
  TINFO n;
```

```
  if (prim == ultim) {
    n = prim->info;
    free(ultim);
    prim = ultim = NULL;
  } else {
    curent = prim;
    n = prim->info;
    prim = prim->next;
    free(curent);
  }
  return n;
}

/**
 *  Functia realizeaza crearea unui arbore oarecare. Se introduce initial radacina,
 * apoi descendentii ei, dupa care se introduc descendentii nodurilor de pe
 * nivelul 1, dupa care se introduc descendentii nodurilor de pe nivelul 2 s.a.m.d.
 */
NOD* creare(void) {
  NOD *p, *c;

  p = (NOD *)malloc(sizeof(NOD));
  p->next = NULL;
  printf("Dati id:"); scanf("%d", &p->id);
  add(p);
  while (!isEmpty()) {
    c = get();
    c->down = citListaDescendenti(c);
  }
  return p;
}

/**
 *  Functia citeste informatia asociata unui nod.
 * @return  1 daca citirea s-a facut corect
 *          0 altfel
*/
int citInfo(NOD *pn) {
  printf("Id:");
  pn->next = NULL;
  pn->down = NULL;
  return scanf("%d",&pn->id) == 1;
}

/**
 *  Functia realizeaza citirea listei de descendenti ai nodului up si
 * returneaza adresa primului descendent din lista.
 */
NOD* citListaDescendenti(NOD *up) {
  NOD *prim, *ultim, *p;
  NOD n;
```

```
  printf("\nLista de descendenti pt %d (CTRL+Z) daca nu are\n", up->id);
  prim = ultim = NULL;
  while (citInfo(&n)) {
    p = (NOD *)malloc(sizeof(NOD));
    *p = n;
    if (prim == NULL)
      prim = ultim = p;
    else {
      ultim->next = p;
      ultim = p;
    }
    add(p);
  }
  return prim;
}

/**
 *  Parcurgerea in A-preordine a arborelui cu radacina p.
 */
void aPreordine(NOD *p) {
  NOD *desc;

  printf("%d ", p->id);
  desc = p->down;
  while (desc != NULL) {
    aPreordine(desc);
    desc = desc->next;
  }
}

/**
 *  Parcurgerea in A-postordine a arborelui cu radacina p.
 */
void aPostordine(NOD *p) {
  NOD *desc;

  desc = p->down;
  while (desc != NULL) {
    aPostordine(desc);
    desc = desc->next;
  }
  printf("%d ",p->id);
}

void main(void) {
  NOD *rad;

  rad = creare();
  printf("\n Parcurgerea in A-Preordine este:\n");
  aPreordine(rad);
  printf("\n Parcurgerea in A-Postordine este:\n");
```

```
  aPostordine(rad);
}
```

6.3 Arbori de acoperire de cost minim

Fie $G = (V, E)$ un graf neorientat şi fie $c : E \longrightarrow \mathbb{R}$ o funcţie de cost ce asociează o valoare reală fiecărei muchii. Notăm cu $\mathcal{T}_G$ mulţimea arborilor parţiali ai grafului G (un arbore parţial este un graf parţial conex şi fără cicluri al grafului iniţial).

Cerinţa problemei este aceea de a determina un arbore $T' \in \mathcal{T}_G$ având costul cel mai mic dintre toţi arborii ce aparţin mulţimii $\mathcal{T}_G$:

$$c(T') = \min\{c(T) | T \in \mathcal{T}_G\}$$

unde costul unui arbore T se defineţte ca $c(T) = \sum_{e \in E_T} c(e)$. Arborele T' ce posedă această proprietate se numeşte *arbore de acoperire de cost minim* (eng. *minimum spanning tree - MST*). Se mai întâlneşte şi sub denumirea de *arbore parţial de cost minim* sau *arbore parţial minim*.

Altfel spus, se cere să se determine un *graf parţial conex* $G_1 = (V, E_1)$ $(E_1 \subseteq E)$ cu proprietatea că suma costurilor tuturor muchiilor este minimă, graful parţial de cost minim fiind chiar un arbore.

Determinarea arborelui de acoperire de cost minim are multe aplicaţii practice: de exemplu, dacă se dau n oraşe precum şi costul legăturilor între acestea, se cere să se determine o conectare a tuturor oraşelor astfel încât oricare două oraşe să fie conectate direct sau indirect iar costul conectării să fie minim.

Lema 6.2 *(Proprietatea tăieturii) Fie S o submulţime de noduri a lui V şi e muchia ce are costul minim dintre toate muchiile ce au o singură extremitate în S. Atunci arborele parţial de cost minim va conţine muchia e.*

Demonstraţie: Fie T' un arbore parţial de cost minim. Presupunem prin reducere la absurd că $e \notin E_{T'}$. Dacă adăugăm muchia e la T' se obţine un ciclu C. În acest ciclu există o altă muchie f ce are exact o extremitate în mulţimea S.

Înlocuind muchia f cu muchia e în arborele T', obţinem un alt arbore de acoperire $T'' = T' \cup \{e\} \setminus \{f\}$.

Deoarece $c(e) < c(f)$ vom avea că $c(T'') = c(T') + \underbrace{c(e) - c(f)}_{<0} \Rightarrow c(T'') < c(T')$ adică am obţinut un arbore de acoperire T'' ce are costul mai mic decât T', contradicţie cu faptul că T' este un arbore de acoperire de cost minim. □

Definiţia 6.2 *Se numeşte **tăietură** a grafului G o partiţie de două submulţimi a mulţimii nodurilor V, notată astfel: $< S, T >$ (unde $S \cup T = V$ şi $S \cap T = \emptyset$).*

Lema 6.3 *(Proprietatea ciclului) Fie C un ciclu şi f muchia ce are costul maxim dintre toate muchiile ce aparţin lui C. Atunci arborele parţial de cost minim T' nu conţine muchia f.*

Demonstraţie: Fie T' un arbore parţial de cost minim. Presupunem prin reducere la absurd că $f \in E_{T'}$. Dacă ştergem muchia f din arborele T', se obţine o tăietură $< S, V \setminus S >$. Avem că $f \in C$ şi o extremitate a lui f aparţine lui S. Există atunci o altă muchie e cu proprietatea că $e \in C$ şi e are o extremitate ce aparţine lui S.

Înlocuind muchia f cu muchia e în arborele T', obţinem un alt arbore de acoperire $T'' = T' \cup \{e\} \setminus \{f\}$.

Deoarece $c(e) < c(f) \Rightarrow c(T'') < c(T')$ adică am obţinut un arbore de acoperire T'' ce are costul mai mic decât T', contradicţie. □

Majoritatea algoritmilor ce calculează arborele de acoperire de cost minim prezintă aceeaşi tehnică generală de calcul. La început se porneşte cu o pădure de arbori, $\mathcal{T}^0 = \{T_1^0, T_2^0, \ldots, T_n^0\}$ unde $T_i^0 = \{x_i\}$, $i = \overline{1,n}$ este un arbore format dintr–un singur nod. La pasul k vom avea mulţimea $\mathcal{T}^k$ compusă din $n-k$ arbori: $\mathcal{T}^k = \{T_1^k, T_2^k, \ldots, T_{n-k}^k\}$.

La fiecare pas, se alege un arbore T_i^k şi o muchie (u', v') având costul minim dintre toate muchiile (u, v) cu proprietatea că o extremitate aparţine mulţimii de noduri a arborelui T_i^k ($u' \in T_i^k$) şi cealaltă extremitate aparţine mulţimii $V \setminus V_{T_i^k}$ ($v' \in V \setminus V_{T_i^k}$).

Prin urmare, mulţimea $\mathcal{T}^{k+1}$ se obţine din mulţimea $\mathcal{T}^k$ prin reuniunea arborilor T_i^k şi T_j^k ($i \neq j$), unde $u' \in T_i^k$ şi $v' \in T_j^k$ ($\mathcal{T}^{k+1} = \mathcal{T}^k \setminus \{T_i^k, T_j^k\} \cup \{T_i^k \cup T_j^k\}$).

În final, la pasul $n-1$, mulţimea $\mathcal{T}^{n-1} = \{T_1^{n-1}\}$ va fi compusă dintr–un singur element, acesta fiind *arborele de acoperire de cost minim*.

Algoritmii cei mai cunoscuţi pentru determinarea arborilor de acoperire de cost minim sunt:

1. *Algoritmul lui Boruvka* (vezi algoritmul 40)

Algoritm 40 Algoritmul lui Boruvka (varianta schematica)

```
1: procedure BORUVKA1(G, C, n; L)
2:     iniţializează pădurea de arbori P compusă din n arbori, fiecare arbore fiind compus dintr–un
           singur nod
3:     L ← ∅
4:     while (|P| > 1) do
5:         for T ∈ P do
6:             alege e muchia de cost minim de la T la G \ T
7:             L ⇐ e ▷ adaugă muchia e la lista de muchii alese ce vor forma arborele de acoperire
                   de cost minim
8:         end for
9:         adaugă toate muchiile selectate în cadrul for-ului anterior la P
10:    end while
11: end procedure
```

2. *Algoritmul lui Prim* (vezi algoritmul 41)

3. *Algoritmul lui Kruskal* (vezi algoritmul 42)

Algoritmul lui Prim implementat simplu are o complexitate $O(n^2)$[30] şi atinge o complexitate de $O(m \log n)$ dacă se folosesc heap–uri Fibonacci [54], sau *pairing heaps* [115].

În tabelul 6.4 sunt prezentaţi mai mulţi algoritmi dezvoltaţi de–a lungul timpului pentru determinarea arborelui de acoperire minimal şi complexităţile lor. Karger, Klein şi Tarjan [79] pornind de la algoritmul lui Boruvka au realizat un algoritm randomizat pentru determinarea arborelui de acoperire minimal, având o complexitate liniară, iar Chazelle [26] a dezvoltat un algoritm având complexitatea $O(n\alpha(m,n))$ ($\alpha(m,n)$ este inversa funcţiei lui Ackerman). Pe de altă parte, Pettie şi Ramachandran [105] au propus un algoritm demonstrat ca fiind optimal, având complexitatea cuprinsă între $O(n+m)$ şi $O(n\alpha(m,n))$.

Algoritm 41 Algoritmul lui Prim (varianta schematica)

```
1: procedure PRIM1(n, C, u; L)
2:     S ← {u}, L ← ∅
3:     for i ← 1, n do
4:         d_i ← c_{u,i}
5:     end for
6:     for i ← 1, n − 1 do
7:         k ← min {d_k | k ∈ V \ S}
8:         L ⇐ (tata_k, k)
9:         S ← S ∪ {k}
10:        for each j ∈ V \ S do
11:            d_j ← min {d_j, c_{k,j}}
12:        end for
13:    end for
14: end procedure
```

Algoritm 42 Algoritmul lui Kruskal (varianta schematica)

```
1: procedure KRUSKAL1(G, C, n; L)
2:     ordonează muchiile în ordine crescătoare după cost
3:     L ← ∅
4:     for each u ∈ V do
5:         crează o mulţime compusă din {u}
6:     end for
7:     count ← 0
8:     while count < n − 1 do
9:         alege muchia (u, v)
10:        if (u şi v sunt în mulţimi diferite) then
11:            L ⇐ (u, v)
12:            reuneşte mulţimile ce conţin pe u şi v
13:            count ← count + 1
14:        end if
15:    end while
16: end procedure
```

Fie $G(V, E)$ un graf neorientat unde $V = \{1, 2, ..., n\}$ este mulţimea nodurilor şi E este mulţimea muchiilor ($E \subseteq V \times V$). Pentru reprezentarea grafului se utilizează matricea costurilor C:

$$c_{i,j} = \begin{cases} 0 & \text{, dacă } i = j \\ \infty & \text{, dacă } (i,j) \notin E \\ d > 0 & \text{, dacă } (i,j) \in E \end{cases}$$

6.3.1 Algoritmul lui Boruvka

Algoritmul lui Boruvka [77] a fost descoperit de către matematicianul ceh Otakar Boruvka în 1926 [21], şi redescoperit apoi de către alţi cercetători. Dintre aceştia, Sollin (1961) este cel care a mai dat numele algoritmului, acesta fiind cunoscut în literatura de specialitate şi sub numele de *algoritmul lui Sollin*. Pentru ca acest algoritm să poată fi aplicat, trebuie ca muchiile grafului să aibă costuri distincte (vezi algoritmul 43).

Table 6.4: Algoritmi pentru determinarea arborelui de acoperire minim

Anul	Complexitate	Autori
1975	$E \log\log V$	Yao
1976	$E \log\log V$	Cheriton-Tarjan
1984	$E \log^* V, E + V \log V$	Friedman-Tarjan
1986	$E \log log^* V$	Gabow-Galil-Spencer-Tarjan
1997	$E\alpha(V) \log \alpha(V)$	Chazelle
2000	$E\alpha(V)$	Chazelle [26]
2002	optimal	Pettie-Ramachandran [105]

Algoritm 43 Algoritmul lui Boruvka

```
1: procedure BORUVKA2(G, C, n; L)
2:     for i ← 1, n do
3:         V_i ← {i}
4:     end for
5:     L ← ∅, M ← {V_1, ..., V_n}
6:     while (|T| < n − 1) do
7:         for U ∈ M do
8:             fie (u, v) muchia pentru care se obţine valoarea minimă min{c(u', v')|(u', v') ∈ E, u' ∈
               U, v' ∉ V \ U}
9:             determină componenta U' ce c      e pe v
10:            L ⇐ (u, v)
11:        end for
12:        for U ∈ M do
13:            reuneşte mulţimile ce conţin pe u şi v, U şi U'
14:        end for
15:    end while
16: end procedure
```

Exemplul 6.4 *Să considerăm graful din figura 6.5:*

$$G = (V, E), V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

Aplicând algoritmul lui Boruvka, la pasul întâi vor fi selectate muchiile $(1, 2)$, $(3, 6)$, $(4, 5)$, $(4, 7)$ *şi* $(7, 8)$. *În urma operaţiilor de reuniune a componentelor conexe pe baza muchiilor selectate, vor mai rămâne trei componente conexe în mulţimea* M.

La pasul al doilea sunt alese muchiile $(2, 5)$ *şi* $(1, 3)$ *ce conduc, în urma operaţiilor de reuniune, la o singură componentă conexă.*

6.3.2 Algoritmul lui Prim

Algoritmul a fost descoperit mai întâi de V. Jarnik (1930) [76], şi apoi independent de Prim (1957) [106] şi Djikstra (1959) [38].

Se porneşte cu o mulţime S formată dintr-un singur nod ($S = \{v_0\}$). La fiecare pas se alege muchia de cost minim ce are numai o extremitate în mulţimea S. Procesul se încheie după $n - 1$ paşi, rezultând un graf parţial aciclic. Din teorema 5.1 rezultă faptul că acest graf parţial aciclic cu $n - 1$ muchii este un arbore (de acoperire).

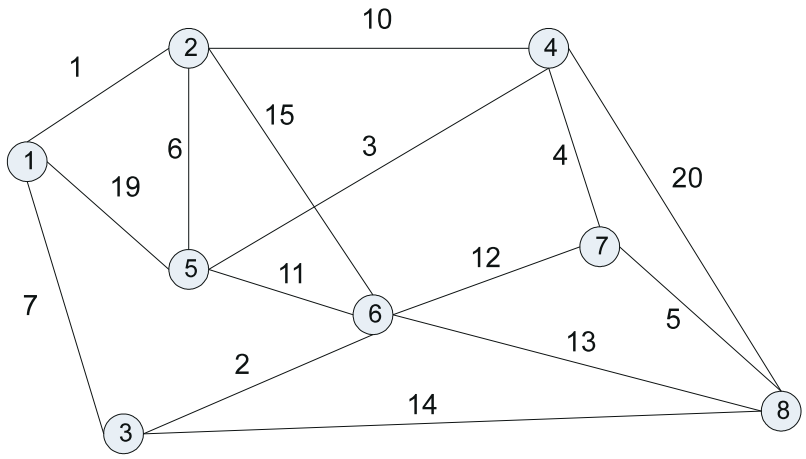

Fig. 6.5: Exemplu de graf ponderat - aplicaţie algoritmul lui Boruvka

Conform *Proprietăţii tăieturii*, toate muchiile alese aparţin arborelui parţial de cost minim de unde rezultă că acest arbore de acoperire este un *arbore parţial minim*.

Vom utiliza trei vectori de dimensiune n, unde n reprezintă numărul de vârfuri al grafului (vezi algoritmul 44):

- *vizitat* - vector caracteristic

$$vizitat_k = \begin{cases} 1 & \text{, dacă nodul } k \in S \\ 0 & \text{, dacă nodul } k \in V \setminus S \end{cases}$$

- d - pentru un nod $k \notin S$, d_k va conţine distanţa minimă de la k la un nod $j \in S$. La început, $d_j = c_{v_0,j}$. Pentru un nod k ales la un moment dat, d_j $(j \in S)$ se modifică numai dacă $c_{k,j} < d_j$ astfel $d_j = c_{k,j}$.

- *tata* - conţine pentru fiecare nod $k \notin S$ nodul $j \in S$ astfel încât $c_{k,j} = \min\{c_{k,i} | i \in S\}$. La început,

$$tata_k = \begin{cases} 0 & \text{, dacă nodul } k = v_0 \\ v_0 & \text{, dacă nodul } k \neq v_0 \end{cases}$$

  În momentul în care se modifică d_j, se va modifica şi valoarea lui $tata_j = k$.

Exemplul 6.5 *Fie graful din figura 6.6:*

$$G = (V, E), V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

Vom lua nodul iniţial $v_0 = 1$. La început, după etapa de iniţializare avem:

	1	2	3	4	5	6	7	8
d		14	6	∞	5	∞	∞	∞
tata	0	1	1	1	1	1	1	1
vizitat	1	0	0	0	0	0	0	0

După primul pas al ciclului, selectăm muchia $(1, 5)$.

	1	2	3	4	5	6	7	8
d		14	6	21	5	16	∞	∞
tata	0	1	1	5	1	5	1	1
vizitat	1	0	0	0	1	0	0	0

Algoritm 44 Algoritmul Prim (varianta detaliata)

```
1: function DistantaMinima(n, vizitat, d)
2:     min ← ∞
3:     for j ← 1, n do
4:         if (vizitat_j ≠ 1) ∧ (d_j < min) then
5:             min ← d_j
6:             j_0 ← j
7:         end if
8:     end for
9:     if (min = ∞) then
10:        return −1
11:    else
12:        return j_0
13:    end if
14: end function

15: procedure Prim2(n, C, v_0; d, tata)
16:    vizitat_{v_0} ← 1
17:    tata_{v_0} ← 0
18:    for i ← 1, n do
19:        if (i ≠ v_0) then
20:            vizitat_i ← 0
21:            d_i ← c_{v_0,i}
22:            tata_i ← v_0
23:        end if
24:    end for
25:    for i ← 1, n − 1 do
26:        k ← DistantaMinima(n, vizitat, d)
27:        if (k < 0) then
28:            Output 'Graful nu este conex!'
29:            return
30:        end if
31:        vizitat_k ← 1          ▷ (k, tata_k) este o muchie ce aparţine arborelui parţial minim
32:        for j ← 1, n do
33:            if (vizitat_j ≠ 1) ∧ (d_j > c_{k,j}) then
34:                tata_j ← k
35:                d_j ← c_{k,j}
36:            end if
37:        end for
38:    end for
39: end procedure
```

La pasul al doilea se alege nodul 3 *şi muchia* $(1, 3)$*:*

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
d		*14*	*6*	*21*	*5*	*12*	∞	*12*
tata	*0*	*1*	*1*	*5*	*1*	*3*	*1*	*3*
vizitat	*1*	*0*	*1*	*0*	*1*	*0*	*0*	*0*

La pasul al treilea avem două noduri ale căror distanţe la noduri din mulţimea S *sunt egale:* 6 *şi* 8*. Alegem primul nod -* 6 *şi muchia* $(3, 6)$*:*

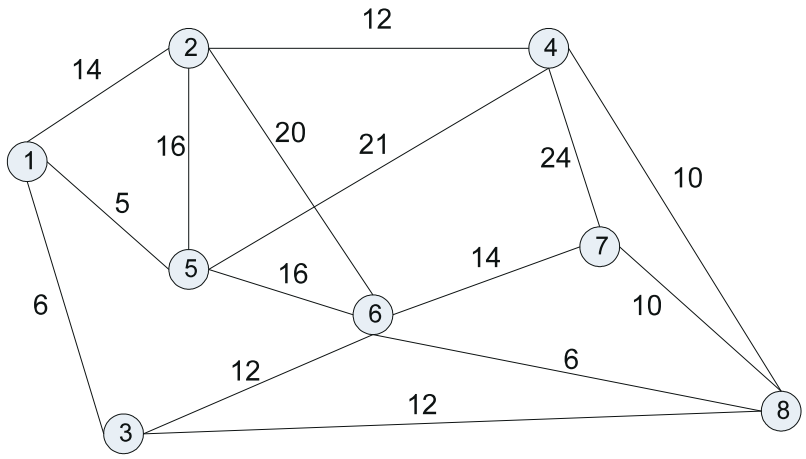

Fig. 6.6: Exemplu de graf ponderat - aplicaţie algoritmul lui Prim

	1	2	3	4	5	6	7	8
d		*14*	*6*	*21*	*5*	*12*	*14*	*6*
tata	*0*	*1*	*1*	*5*	*1*	*3*	*6*	*6*
vizitat	*1*	*0*	*1*	*0*	*1*	*1*	*0*	*0*

Al patrulea nod ales este 8*:*

	1	2	3	4	5	6	7	8
d		*14*	*6*	*10*	*5*	*12*	*10*	*6*
tata	*0*	*1*	*1*	*8*	*1*	*3*	*8*	*6*
vizitat	*1*	*0*	*1*	*0*	*1*	*1*	*0*	*1*

La pasul cinci, nodul aflat la distanţă minimă este 7*:*

	1	2	3	4	5	6	7	8
d		*14*	*6*	*10*	*5*	*12*	*10*	*6*
tata	*0*	*1*	*1*	*8*	*1*	*3*	*8*	*6*
vizitat	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*1*

La pasul şase este ales nodul 4*:*

	1	2	3	4	5	6	7	8
d		*12*	*6*	*10*	*5*	*12*	*10*	*6*
tata	*0*	*4*	*1*	*8*	*1*	*3*	*8*	*6*
vizitat	*1*	*0*	*1*	*1*	*1*	*1*	*1*	*1*

La final, ultimul nod ales este 2 *împreună cu muchia* $(4, 2)$*.*

Trebuie să remarcăm faptul că algoritmul lui Prim (vezi algoritmul 44) este aproape identic cu algoritmul lui Dijkstra (vezi algoritmul 59).

După cum am subliniat, implementarea optimă se realizează folosind nişte structuri de date avansate - *heap–uri Fibonacci* sau *pairing heaps* (vezi algoritmul 45).

6.3.3 Structuri de date pentru mulţimi disjuncte

O partiţie a unei mulţimi A este o secvenţă finită de mulţimi (submulţimi) $A_1, \ldots, A_m$ disjuncte două câte două, cu proprietatea că reuniunea acestora este chiar mulţimea A ($A = \bigcup_{i=1}^{m} A_i$ şi $A_i \cap A_j = \emptyset, \forall i, j = \overline{1, m}, i \neq j$).

Algoritm 45 Algoritmul lui Prim folosind structuri de date avansate

```
1: procedure PRIM3(n, C, u)
2:     for fiecare v ∈ V do
3:         d_v ← ∞
4:     end for
5:     Q ← ∅                          ▷ iniţializează coada cu prioritate Q cu mulţimea vidă
6:     for fiecare v ∈ V do
7:         Q ⇐ v
8:     end for
9:     S ← ∅                          ▷ iniţializează mulţimea S cu mulţimea vidă
10:    while Q ≠ ∅ do
11:        u ← deleteMin(Q)           ▷ extrage nodul de prioritate minimă din Q
12:        S ← S ∪ {u}
13:        for fiecare muchie e = (u, v) ∈ E do
14:            if (v ∉ S) ∧ (c(e) < d_v) then
15:                actualizeaza prioritatea lui v: d_v ← c(e)
16:            end if
17:        end for
18:    end while
19: end procedure
```

Există mai multe probleme ai căror algoritmi de rezolvare depind de următoarele operaţii efectuate asupra elementelor partiţiei: verificarea dacă două elemente fac parte din aceeaşi submulţime precum şi operaţia de reuniune a două submulţimi.

O *structură de date pentru mulţimi disjuncte* memorează o colecţie de mulţimi disjuncte dinamice. Fiecare mulţime este identificată printr–un *reprezentant* [30]. Operaţiile de bază ale acestei structuri de date sunt [2]:

- $init(x, B)$ - procedura crează o mulţime B formată dintr–un singur element x;
- $find(x)$ - întoarce reprezentantul mulţimii căreia îi aparţine x;
- $merge(A, B)$ - reuneşte mulţimile distincte A şi B ($x \in A$, $y \in B$) într–o nouă mulţime ce conţine elementele celor două mulţimi.

O astfel de *structură de date pentru mulţimi disjuncte* se mai numeşte şi *structură de date union–find* (eng. *union–find data structure*).

Reprezentarea folosind vectori

Vom folosi un vector C de dimensiune n, unde n reprezintă numărul de elemente, iar $c_k = u$ indică faptul că elementul k aparţine mulţimii u.

$Init$ constă din iniţializarea lui c_x cu identificatorul mulţimii, u. $Find$ întoarce valoarea lui c_x (valoarea identificatorului mulţimii). În funcţia $Merge$ se caută toate elementele ce fac parte din mulţimea de *identificator* c_y şi se trec în mulţimea al cărui *identificator* este c_x (vezi algoritmul 46).

Exemplul 6.6 *Pentru o putea opera cu modul de reprezentare ales, cel cu vectori, trebuie ca elementele mulţimii să ia valori naturale în intervalul [1,. . . ,n]. Dacă acest lucru nu este posibil atunci folosim un vector auxiliar A ce păstrează valorile elementelor, în cadrul reprezentării utilizându–se indicele acestora. Să presupunem că avem mulţimea de elemente*

Algoritm 46 Algoritmi pentru operaţiile init, find, merge (varianta întâi)

```
1: procedure INIT(x, u)
2:     c_x ← u
3: end procedure
4: function FIND(x)
5:     return c_x
6: end function
7: function MERGE(x, y)
8:     setx ← c_x
9:     sety ← c_y
10:    for k ← 1, n do
11:        if (c_k = sety) then
12:            c_k ← setx
13:        end if
14:    end for
15:    return setx
16: end function
```

$A = \{a, b, e, x, y, z, u, v\}$ şi partiţia $\{a, x, y\}, \{b, z, v\}, \{e, u\}$. Atunci conform modului de reprezentare descris avem:

	1	2	3	4	5	6	7	8
A	'a'	'b'	'e'	'x'	'y'	'z'	'u'	'v'
C	1	2	3	1	1	2	2	3

$a_2 =' b'$ are semnificaţia următoare: elementul de pe poziţia 2 are valoarea $'b'$. $c_2 = 2$ - elementul de pe poziţia 2 face parte din mulţimea de identificator 2. Sau $a_4 =' x'$ - elementul de pe poziţia 4 are valoarea $'x'$, şi $c_4 = 1$ - elementul de pe poziţia 4 face parte din mulţimea de identificator 1.

Reprezentarea folosind liste înlănţuite

În cadrul acestei metode fiecare mulţime este reprezentată printr–o listă simplu înlănţuită, elementul *reprezentant* fiind elementul aflat în capul listei. Un nod al listei va avea un câmp ce păstrează informaţia cu privire la un element al unei mulţimi, şi un câmp ce conţine legătura către următorul nod al listei.

Vom folosi un vector de adrese ($List$), ce conţine adresa primului element din fiecare listă ($List_i$).

Procedura $Init$ alocă un nod şi iniţializează câmpurile acestuia. Adresa nodului alocat este păstrată în $List_k$ (vezi algoritmul 47).

Funcţia $Find$ caută un element printre elementele păstrate de fiecare listă în parte. Se parcurge vectorul $List$ şi se obţine reprezentantul fiecărei submulţimi păstrate. Se parcurge lista (căutare liniară) şi se caută un element cu valoarea x. Dacă elementul este găsit atunci se întoarce capul listei în care a fost găsit. Dacă până la sfârşit nu a fost identificat elementul x, atunci funcţia returnează valoarea $NULL$.

Funcţia $Merge$ reuneşte două liste: practic se adaugă o listă la sfârşitul celeilalte. Pentru a realiza această operaţie pe lângă adresa primului element avem nevoie de adresa ultimului element. Acest lucru se poate obţine indirect prin parcurgerea listei de la primul la ultimul element, fie direct dacă în momentul creării păstrăm pentru fiecare listă încă o variabilă ce conţine adresa ultimului element ($last$).

Algoritm 47 Algoritmi pentru operaţiile init, find, merge (varianta a II-a)

```
1: procedure INIT(x, k)
2:     List_k ← new node                          ▷ alocă spaţiu pentru un nod al listei
3:     List_k.data ← x
4:     List_k.next ← NULL
5: end procedure
6: function FIND(x)
7:     for i ← 1, m do
8:         p ← List_i, reprezentant ← p
9:         while (p ≠ NULL) ∧ (p.data ≠ x) do
10:            p ← p.next
11:        end while
12:        if (p ≠ NULL) then
13:            return reprezentant
14:        end if
15:    end for
16:    return NULL
17: end function
18: function MERGE(x, y)
19:    cap_x ← Find(x)
20:    cap_y ← Find(y)
21:    curent ← cap_x
22:    while (curent.next ≠ NULL) do
23:        curent ← curent.next
24:    end while
25:    curent.next ← cap_y
26:    return cap_x
27: end function
```

Reprezentarea folosind o pădure de arbori

În cadrul acestei modalităţi de reprezentare fiecare mulţime este păstrată sub forma unui arbore. Pentru fiecare nod păstrăm informaţia despre părintele său: $tata_k$ reprezintă indicele nodului părinte al nodului k.

Se poate observa foarte uşor (vezi algoritmul 48) faptul că arborele obţinut în urma unor operaţii repetate *Merge* poate deveni *degenerat* (o listă liniară). Pentru a putea să evităm o astfel de situaţie, reuniunea se poate realiza ţinând cont de una din următoarele caracteristici:

1. *numărul de elemente din fiecare arbore* - rădăcina arborelui ce are mai puţine elemente (notăm arborele cu T_B) va deveni descendentul direct al rădăcinii arborelui ce posedă mai multe elemente (notat cu T_A). Astfel toate nodurile din arborele T_A vor rămâne cu aceeaşi adâncime, pe când nodurile din arborele T_B vor avea adâncimea incrementată cu 1. De asemenea, arborele rezultat în urma reuniunii va avea cel puţin de două ori mai multe noduri decât arborele T_B.

Algoritm 48 Algoritmi pentru operaţiile init, find, merge (varianta a III-a)

```
1: procedure INIT(x)
2:     tata_x ← x
3: end procedure
4: function FIND(x)
5:     while (x ≠ tata_x) do
6:         x ← tata_x
7:     end while
8:     return x
9: end function
10: function MERGE(x, y)
11:     radx ← Find(x)
12:     rady ← Find(y)
13:     tata_rady ← radx
14:     return radx
15: end function
```

```
1: procedure INIT(x)
2:     tata_x ← −1
3: end procedure
4: function FIND(x)
5:     while (tata_x > 0) do
6:         x ← tata_x
7:     end while
8:     return x
9: end function
```

```
1: function MERGE(x, y)
2:     radx ← Find(x)
3:     rady ← Find(y)
4:     size ← |tata_radx| + |tata_rady|
5:     if (|tata_radx| > |tata_rady|) then
6:         tata_rady ← radx
7:         tata_radx ← −size
8:         return radx
9:     else
10:        tata_radx ← rady
11:        tata_rady ← −size
12:        return rady
13:    end if
14: end function
```

Prin această euristică simplă se urmăreşte o echilibrare a arborelui rezultat pentru a se evita cazurile degenerate. Complexitatea timp a procedurii $Find$ este $O(\log n)$.

2. *înălţimea fiecărui arbore* - rădăcina arborelui ce are înălţimea mai mică (notăm arborele cu T_B) va deveni descendentul direct al rădăcinii arborelui cu înălţimea mai mare (notat cu T_A). Dacă înălţimile lui T_A şi T_B sunt egale şi T_A devine subarbore al lui T_B atunci înălţimea lui T_B creşte cu o unitate.

   Înălţimea fiecărui subarbore de rădăcină u se păstrează într–o variabilă nouă h_u. Pentru a economisi cantitatea de memorie utilizată, se poate păstra înălţimea unui arbore în vectorul *tata*: $tata_x = -$valoarea înălţimii arborelui, atunci când x reprezintă nodul rădăcină al unui arbore.

```
1: procedure INIT(x)
2:     tata_x ← x
3:     h_x ← 0
4: end procedure
5: function FIND(x)
6:     while (tata_x ≠ x) do
7:         x ← tata_x
8:     end while
9:     return x
10: end function
```

```
1: function MERGE(x, y)
2:     radx ← Find(x)
3:     rady ← Find(y)
4:     if (h_radx > h_rady) then
5:         tata_rady ← radx
6:         return radx
7:     else
8:         tata_radx ← rady
9:         if (h_radx = h_rady) then
10:            h_rady ← h_rady + 1
11:        end if
12:        return rady
13:    end if
14: end function
```

Cea de–a doua tehnică utilizată pentru a reduce complexitatea timp a operaţiilor *Find* şi *Merge* este reprezentată de *comprimarea drumului*. Aceasta constă în a *apropia* fiecare nod de rădăcina arborelui căruia îi aparţine. Astfel în timpul operaţiei $Find(x)$ se determină mai întâi rădăcina arborelui căruia îi aparţine (rad) şi apoi se mai parcurge o dată drumul de la nodul x la rădăcină, astfel $tata_y \leftarrow rad$, $\forall y \in$ lanţul de la x la rad.

```
1: function FIND(x)
2:     y ← x
3:     while (tata_y ≠ y) do
4:         y ← tata_y
5:     end while
6:     rad ← y
7:     y ← x
8:     while (tata_y ≠ y) do
9:         y ← tata_y
10:        tata_x ← rad
11:        x ← y
12:    end while
13:    return rad
14: end function
```

6.3.4 Algoritmul lui Kruskal

Algoritmul lui Kruskal [88] este o ilustrare foarte bună a metodei generale *Greedy* (vezi algoritmul 49). La început se pleacă cu o pădure de arbori, fiecare arbore fiind alcătuit dintr–un singur nod. La fiecare pas se alege o muchie: dacă cele două extremităţi (noduri) fac parte din acelaşi arbore atunci nu se va adăuga muchia curentă la arborele parţial respectiv deoarece ar conduce la un ciclu, ceea ce ar strica proprietatea de graf aciclic. Altfel, dacă cele două noduri fac parte din arbori parţiali distincţi, se adaugă muchia curentă la mulţimea de muchii alese anterior, iar cei doi arbori parţiali devin unul singur. La fiecare pas, numărul de arbori parţiali scade cu 1. După $n-1$ alegeri pădurea iniţială compusă din n arbori s–a transformat într–un singur arbore.

Pentru a păstra muchiile grafului vom utiliza o matrice A cu m coloane şi 3 linii: primele două linii conţin extremităţile muchiilor, iar linia a 3-a costul muchiei respective. Pentru a se aplica o strategie Greedy, datele de intrare – muchiile – se vor ordona crescător în funcţie de costul asociat. Deoarece graful parţial respectiv este un arbore cu n noduri el va avea $n-1$ muchii.

O verificare eficientă cu privire la apartenenţa la acelaşi arbore a celor două extremităţi ale unei muchii se poate face dacă vom utiliza o structură de date pentru mulţimi disjuncte.

Ca implementare, am ales reprezentarea cu pădure de arbori. Vom folosi un vector *tata* care, pentru fiecare nod, va păstra *tatăl* acestuia în arborele parţial căruia îi aparţine.

Când sunt preluate ca date de intrare, muchiile vor fi prezentate în ordinea crescătoare a extremităţilor lor, de forma $i\ j\ cost$, cu $i < j$.

Algoritm 49 Algoritmul lui Kruskal

```
1:  procedure KRUSKAL2(n, m, A; L, cost)
2:      for i ← 1, n do
3:          call Init(i)
4:      end for
5:      cost ← 0
6:      j ← 1, L ← ∅
7:      for i ← 1, n − 1 do
8:          q ← 0
9:          while (q = 0) do
10:             r1 ← Find(A1,j)
11:             r2 ← Find(A2,j)
12:             if (r1 ≠ r2) then
13:                 cost ← cost + A3,j
14:                 call Merge(r1, r2)
15:                 L ← L ∪ {(A1,j, A2,j)}
16:                 q ← 1
17:             end if
18:             j ← j + 1
19:         end while
20:     end for
21: end procedure
```

Se ordoneaza muchiile dupa cost crescator
Se adauga muchii de la cost minim catre cost maxim
Se evita adaugarea de muchii ce formeaza ciclu

Exemplul 6.7 *Să considerăm graful din figura 6.6. Muchiile grafului şi costurile lor sunt (am ales să reprezentăm matricea A sub forma unei matrice cu m coloane şi 3 linii din motive de spaţiu):*

A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
linia 1	1	1	1	2	2	2	3	3	4	4	4	5	6	6	7
linia 2	2	3	5	4	5	6	6	8	5	7	8	6	7	8	8
linia 3	14	6	5	12	16	20	12	12	21	24	10	16	14	6	10

După aşezarea în ordine crescătoare a muchiilor după cost avem:

A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
linia 1	1	1	6	4	7	2	3	3	1	6	2	5	2	4	4
linia 2	5	3	8	8	8	4	6	8	2	7	5	6	6	5	7
linia 3	5	6	6	10	10	12	12	12	14	14	16	16	20	21	24

În figura 6.7, sunt ilustraţi paşii algoritmului lui Kruskal aplicat pe graful considerat. La început se iniţializează pădurea de arbori, fiecare arbore fiind alcătuit dintr–un singur nod, acesta fiind şi rădăcina arborelui. Apoi la fiecare pas se alege o muchie şi se verifică dacă extremităţile acesteia fac parte din arbori diferiţi. Dacă răspunsul este afirmativ atunci muchia respectivă este selectată, altfel se trece la muchia următoare.

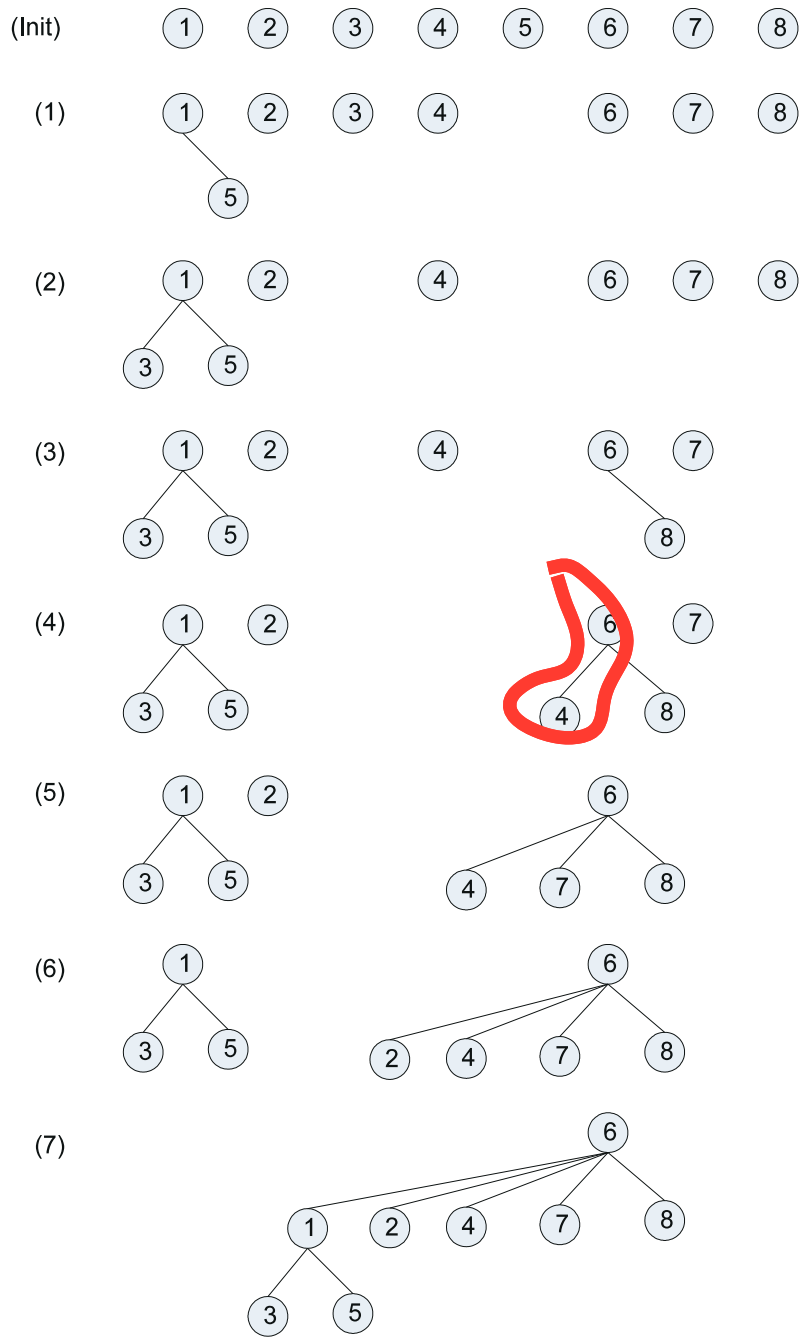

Fig. 6.7: Algoritmului lui Kruskal exemplificat pe graful din figura 6.6

La pasul întâi evaluăm muchia $(1, 5)$*, şi deoarece cele două extremităţi fac parte din arbori distincţi, vom selecta această muchie (*$L = \{(1, 5)\}$*). Reunim arborii din care fac parte cele două extremităţi, (vezi figura 6.7 (1)).*

La pasul al doilea, se ia în considerare muchia $(1, 3)$ *şi mulţimea muchilor selectate devine* $L = \{(1, 5), (1, 3)\}$ *(vezi figura 6.7 (2)).*

La pasul al patrulea, ajungem la muchia $(4, 8)$ *ce are costul* 10*. Nodul* 4 *face parte din arborele de rădăcină* 4 *iar nodul* 8 *face parte din arborele de rădăcină* 6*. Deoarece extremităţile muchiei sunt amplasate în arbori distincţi selectăm muchia curentă pentru arborele parţial de cost minim,* $L = \{(1, 5), (1, 3), (6, 8), (4, 8)\}$*. În urma reuniunii arborilor corespunzători celor două noduri se obţine configuraţia din figura 6.7* (4)*.*

Subliniem faptul că arborii reprezentaţi în figura 6.7 sunt diferiţi de arborii ce conduc la obţinerea soluţiei problemei: arborii din figură constituie suportul necesar pentru reprezentarea structurii de date pentru mulţimi disjuncte, ce este utilizată pentru efectuarea eficientă a operaţiilor Find şi Merge. Acest lucru justifică faptul că deşi la pasul al patrulea selectăm muchia $(4, 8)$ *pentru arborele parţial de cost minim, ea nu se regăseşte în configuraţia* (4) *(nodul* 4 *este unit direct cu nodul* 6*, vezi figura 6.7). Singura legătură dintre un arbore suport pentru reprezentarea unei mulţimi şi un arbore folosit pentru obţinerea arborelui parţial de cost minim, este mulţimea de noduri ce este comună.*

La paşii următori, cinci şi şase, sunt alese muchiile $(7, 8)$ *şi respectiv* $(2, 4)$*:* $L = \{(1, 5), (1, 3), (6, 8), (4, 8), (7, 8), (2, 4)\}$.

La pasul al şaptelea se evaluează muchia $(3, 6)$*. Nodurile* 3 *şi* 6 *fac parte din arbori diferiţi, astfel încât muchia curentă poate fi selectată pentru soluţia problemei,* $L = \{(1, 5), (1, 3), (6, 8), (4, 8), (7, 8), (2, 4), (3, 6)\}$.

6.4 Exerciţii

1. Să se realizeze o subrutină ce întoarce numărul de noduri dintr–un arbore oarecare.

2. Se dă o secvenţă formată din n numere naturale $d_1, d_2, \ldots, d_n$. Să se realizeze un algoritm prin care să se verifice dacă există un arbore cu n noduri ale căror grade sunt $d_1, d_2, \ldots, d_n$.

   Dacă există un astfel de arbore, acesta va fi reprezentat prin liste ale nodurilor. O listă a unui nod conţine numărul nodului urmat de fii săi.

Intrare	Ieşire
1 1 2 3 1 1 3	*DA*
	1 2 3 4
	2 5 6
	3 7
	4
	5
	6
	7

   (Timişoara-pregătire, 1996)

3. Fie un graf G cu n vârfuri şi m muchii de cost pozitiv. Alegând un nod, numit *nod central*, să se determine un subarbore al lui G astfel încât drumurile de la nodul central la toate celelalte noduri să aibă lungimea minimă.

4. Fiind date n puncte în spaţiul $\mathcal{R}^3$ determinate prin coordonatele (x, y, z), să se elaboreze un algoritm ce determină sfera de rază minimă cu centrul într–unul din punctele date şi care conţine în interiorul ei toate cele n puncte.

5. Se consideră procesul de proiectare a unei plăci electronice cu N componente ($1 \leq N \leq 100$). Pentru fiecare componentă electronică C se cunoaşte numărul de interconexiuni. Se consideră graful determinat de mulţimea pinilor şi mulţimea interconectărilor posibile ale tuturor componentelor, precum şi lungimile lor. Dintre toate modalităţile de interconectare posibile se cere cea corespunzătoare arborelui de acoperire minim (interconectarea pentru care suma tuturor circuitelor imprimate are lungimea minimă).

Datele de intrare sunt alcătuite din descrierile mai multor plăci electronice. Descrierea fiecărei plăci conţine numărul N de componente şi numărul M de interconexiuni. O conexiune se caracterizează prin trei valori: două vârfuri, u şi v, precum şi lungimea acesteia l.

Datele de ieşire constau dintr-un răspuns pentru fiecare mulţime de date de test, compus din numărul testului (se începe numerotarea cu 1), precum şi costul interconectării de lungime minimă.

Exemplul 6.8 *Pentru datele de intrare*

```
5 7
1 2 40  2 3 35  1 5 27  1 4 37  4 5 30  2 4 58  3 4 60
0 0
```

vom obţine rezultatul

```
Cazul 1: [1 5] [4 5] [2 3] [1 2]
Interconectarea de cost minim are valoarea 132.
```

În figura 6.8 este reprezentat graful corespunzător datelor de intrare.

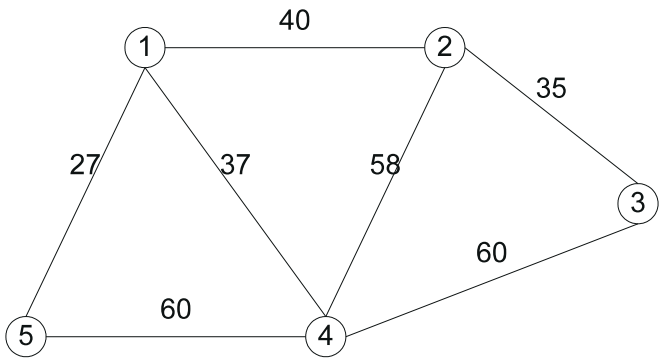

Fig. 6.8: Descrierea conexiunilor posibile dintre componentele unei plăci electronice

6. Realizaţi o subrutină nerecursivă care să determine cheia minimă dintr–un arbore oarecare.

7. Determinaţi înălţimea unui arbore oarecare printr–o funcţie nerecursivă.

8. Pentru asigurarea securităţii activităţilor dintr–un combinat chimic s–a apelat la o companie de pompieri. Desfăşurarea activităţilor companiei presupune stabilirea locurilor de instalare a comandamentului precum şi a posturilor de supraveghere. Pentru aceasta sunt disponibile în cadrul combinatului n puncte de control. Pentru fiecare pereche de puncte de control se cunoaşte dacă există o legătură directă între ele şi, în caz afirmativ, distanţa dintre ele. Se cunoaşte, de asemenea, faptul că între oricare două puncte de control există un drum direct sau indirect.

   Odată stabilite amplasamentele comandamentului şi ale punctelor de supraveghere în câte unul dintre cele n puncte de control este posibil să se ajungă de la comandament

la fiecare punct de supraveghere parcurgând un anumit drum. Evident este de dorit ca lungimea acestui drum să fie minimă şi valoarea maximă dintre lungimile drumurilor minime pentru fiecare punct de supraveghere în parte să fie cât mai mică posibilă.

Se cere să se determine punctul de control în care trebuie instalat comandamentul cât şi drumurile ce vor fi parcurse de la comandament la fiecare punct de supraveghere, astfel încât aceste drumuri să aibă valoarea lungimii minimă şi cel mai lung dintre ele să fie cât mai scurt posibil.