11. Grafuri ponderate ("Weighted Graphs")

- Adeseori, modelarea problemelor practice presupune utilizarea unor grafuri în care arcelor li se asociază **ponderi** care pot fi greutăţi, costuri, valori, etc.
- Astfel de grafuri se mai numesc şi **grafuri ponderate** ("**weighted graphs**").
    - Spre exemplu, pe **harta traseelor aeriene** ale unei zone, arcele reprezintă rute de zbor iar ponderile distanţe, timpi sau preţuri.
    - Într-un **circuit electric** unde arcele reprezintă legături, lungimea sau caracteristicile fizice ale acestora sunt de regulă utilizate ca ponderi.
    - Într-o activitate de **planificare în execuţie a taskurilor**, ponderea poate reprezenta fie timpul, fie costul execuţiei unui task, fie timpul de aşteptare până la lansarea în execuţie a taskului.
- Este evident faptul că într-un asemenea context apar în mod natural probleme legate de **minimizarea costurilor**.
- În cadrul acestui paragraf vor fi prezentate mai în detaliu două astfel de probleme referitoare la grafurile ponderate:
    - (1) Găsirea **drumului cu costul cel mai redus** care conectează toate punctele grafului.
    - (2) Găsirea **drumului cu costul cel mai redus** care leagă două puncte date.
- Prima problemă care este în mod evident utilă pentru grafuri reprezentând circuite electrice sau ceva analog, se numeşte **problema arborelui de acoperire minim** ("**minimum spanning tree problem**").
- Cea de-a doua problemă este utilă în grafurile reprezentând hărţi de trasee (aeriene, feroviare, turistice) şi se **numeşte problema drumului minim** ("**shortest-path problem**").
    - Aceste probleme sunt tipice pentru o largă categorie de aspecte ce apar în prelucrarea **grafurilor ponderate**.
- Se impune o precizare.
- De regulă algoritmii utilizaţi presupun parcurgerea grafului, motiv pentru care în mod intuitiv **ponderile** sunt asociate cu **distanţe**.
    - Se spune de obicei "cel mai apropiat nod" cu sensul de poziţionare geografică a nodului.
    - De fapt acest mod de a concepe lucrurile este valabil în contextul **problemei drumului minim.**
- În general, este însă foarte important a se avea în vedere faptul, că **nu** este absolut necesar ca ponderile să fie proporţionale cu distanţele şi că ele pot reprezenta orice altceva, ca spre exemplu **timpi, costuri** sau **valori**.

- În situaţiile în care ponderile reprezintă într-adevăr distanţe, alţi algoritmi cu caracter specific, pot fi mai potriviţi decât cei care vor fi prezentaţi în continuare.

- În figura 11.a (a) apare o reprezentare grafică a unui **graf neorientat ponderat**.

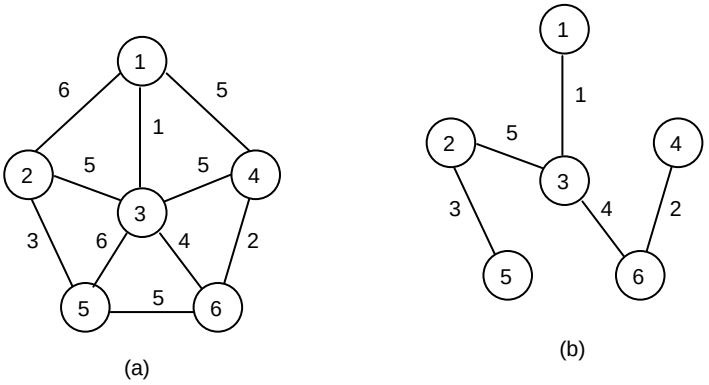


Fig.11.a Reprezentarea unui graf ponderat şi a unui arbore de acoperire minim asociat

- În ceea ce priveşte **reprezentarea structurii de date abstracte graf ponderat**, principial ea se reprezintă ca şi grafurile normale cu următoarele **deosebiri**:
  - (1) În cazul reprezentării prin **matrice de adiacenţe**, matricea va conţine **ponderi** în locul valorilor booleene.
  - (2) În cazul reprezentării prin **structuri de adiacenţe**, fiecărui element al listei i se adăugă un **câmp suplimentar** pentru memorarea **ponderii**.
- Se presupune faptul că **ponderile sunt toate pozitive**.
  - Există însă algoritmi mult mai complicaţi care pot trata şi ponderi negative.
  - Astfel de algoritmi sunt utilizaţi mai rar în activitatea practică.

11.1. Arbori de acoperire minimi ("Minimum-Cost Spanning Trees")

- Se presupune că `G=(N,A)` este un graf conex în care oricare arc `(u,v)` aparţinând lui `A` are ataşat un cost specific `cost(u,v)`.
- Un **arbore de acoperire** al lui `G` este un **arbore liber** care cuprinde toate nodurile din `N` (fig.11.a.(b)).
  - **Costul** unui arbore de acoperire este **suma costurilor** tuturor arcelor cuprinse în arbore.
- **Definiţie**: Un **arbore de acoperire minim** al unui graf ponderat este o selecţie minimă de arce care conectează toate nodurile grafului, astfel încât costul său este cel puţin la fel de mic ca şi costul oricărui alt arbore de acoperire al grafului.
- O altă **definiţie** este următoarea.
  - Dându-se **orice partiţionare** a mulţimii nodurilor unui graf în două submulţimi, **arborele de acoperire minim** conţine **arcele cu ponderea cea**

mai mică care conectează un nod aparţinând unei submulţimi cu un nod aparţinând celeilalte [Se 88].

- Se face precizarea că un arbore de acoperire minim **nu** este în mod necesar **unic**.
- O **aplicaţie tipică** a arborilor de acoperire minimi o reprezintă proiectarea **reţelelor de comunicaţii.**
    - Nodurile grafului reprezintă oraşele iar arcele, comunicaţiile posibile dintre ele.
    - Costul asociat unui arc reprezintă de fapt costul selecţiei acelei legături a reţelei.
    - Un **arbore de acoperire minim** reprezintă o reţea care conectează cu un cost minim toate oraşele.

11.1.1. Proprietatea arborilor de acoperire minimi

- Există mai multe moduri de a construi **arbori de acoperire minimi** asociaţi unui graf ponderat.
- Marea lor majoritate se bazează pe următoarea proprietate, denumită şi **proprietatea arborilor de acoperire minimi.**
    - Fie `G=(N,A)` un **graf conex** şi o **funcţie de cost** definită pe arcele sale.
    - Fie `U` o submulţime proprie a **mulţimii de noduri** `N`.
    - Dacă `(u,v)` este **un arc cu costul cel mai scăzut** astfel încât `u∈U` iar `v∈N-U`, atunci există un **arbore de acoperire minim** care include arcul `(u,v)`.
- **Demonstraţia** acestei aserţiuni nu este complicată şi ea se realizează prin **metoda reducerii la absurd.**
    - Se presupune dimpotrivă că **nu** există un arbore de acoperire minim al lui `G` care include **arcul cu costul cel mai scăzut** `(u,v)`.
    - Fie `T` oricare **arbore de acoperire minim** al lui `G`.
    - Adăugarea arcului `(u,v)` arborelui `T` trebuie să conducă la apariţia unui **ciclu**, deoarece `T` este un **arbore liber** şi conform proprietăţii (b) dacă unui arbore liber i se adaugă un arc el devine un graf ciclic, adică va conţine un ciclu (& 10.1.).
        - Acest ciclu include arcul `(u,v)`.
    - În consecinţă, în ciclul nou format, trebuie să existe un alt arc `(u',v')` al lui `T` astfel încât `u'∈U` şi `v'∈N-U`, după cum rezultă din figura 11.1.1.a.
        - Dacă acest lucru **nu** ar fi adevărat, atunci în cadrul ciclului **nu** ar exista o altă posibilitate de a ajunge de la nodul `v` la nodul `u` decât reparcurgând arcul `(u,v)`.

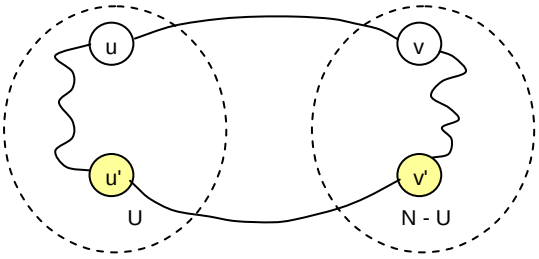


Fig.11.1.1.a. Demonstrarea proprietăţii arborilor de acoperire minimi

- Suprimând arcul `(u',v')`, ciclul dispare şi obţinem arborele de acoperire `T'` al cărui cost **nu** este cu siguranţă mai ridicat decât al lui `T`, deoarece s-a presupus iniţial că `cost(u,v)≤cost(u',v')`, adică `(u,v)` este un **arc cu costul cel mai scăzut**, după cum s-a precizat în condiţiile iniţiale.
- Astfel existenţa lui `T'` **contrazice** presupunerea iniţială şi anume că **nu** există un arbore de acoperire minim care să includă arcul `(u,v)`şi în consecinţă proprietatea arborilor de acoperire minimi este demonstrată.

11.2. Determinarea arborilor de acoperirie minimi

- Există mai multe metode de determinare a unui **arbore minim asociat unui graf ponderat**, metode care în general exploatează proprietatea anterior enunţată pentru aceşti arbori.
- Dintre acestea se remarcă cu deosebire:
  - (1) Algoritmul lui Prim.
  - (2) Metoda căutării "bazate pe prioritate".
  - (3) Algoritmul lui Kruskal.

11.2.1. Algoritmul lui Prim

- Fie `G` **graful ponderat** pentru care se doreşte determinarea unui **arbore de acoperire minim**.
  - Fie `N={1,2,3,...,n}` mulţimea nodurilor grafului `G`.
  - Fie `U` o mulţime vidă de noduri ale grafului.
- **Algoritmul lui Prim**
  - Începe prin a introduce în mulţimea `U` nodul de pornire, să zicem nodul `{1}`.
  - În continuare, într-o manieră ciclică, este construit pas cu pas **arborele de acoperire minim.**
  - Astfel, în fiecare **pas** al algoritmului:
    - (1) Se selectează arcul cu **cost minim** `(u,v)` care conectează mulţimea `U` cu mulţimea `N-U`.

- (2) Se adaugă acest arc arborelui de acoperire minim.
- (3) Se adaugă nodul v mulţimii U.

- Ciclul se repetă până când U=N.

- Schiţa **algoritmului lui Prim** apare în secvenţa [11.2.1.a] .

```
-------------------------------------------------------------
/*Construcţia unui arbore de acoperire minim AAM al unui
graf G*/

procedure PRIM(Tip_Graf G; Multime_De_Arce AAM);

/*construieşte mulţimea AAM care conţine arcele unui arbore
de acoperirie minim al grafului G*/

Multime_De_Noduri U;
Tip_Nod u,v;

  AAM= {mulţimea vidă};
  U= {nodul de pornire};
  cât timp (U<>N) execută                    /*[11.2.1.a]*/
    *fie (u,v) arcul cu costul cel mai redus care
      satisface condiţia (u apartine lui U) && (v apartine
                          lui N-U);
    AAM= AAM ∪ {(u,v)}; /*adaugă arcul (u,v) mulţimii AAM*/
    U= U ∪ {v};     /*adaugă nodul v mulţimii U*/
  □ /*cât timp*/
/*PRIM*/
-------------------------------------------------------------
```

- În figura 11.2.1.a apare reprezentată pas cu pas secvenţa de construcţie a arborelui de acoperire minim pentru graful (0) din aceeaşi figură.

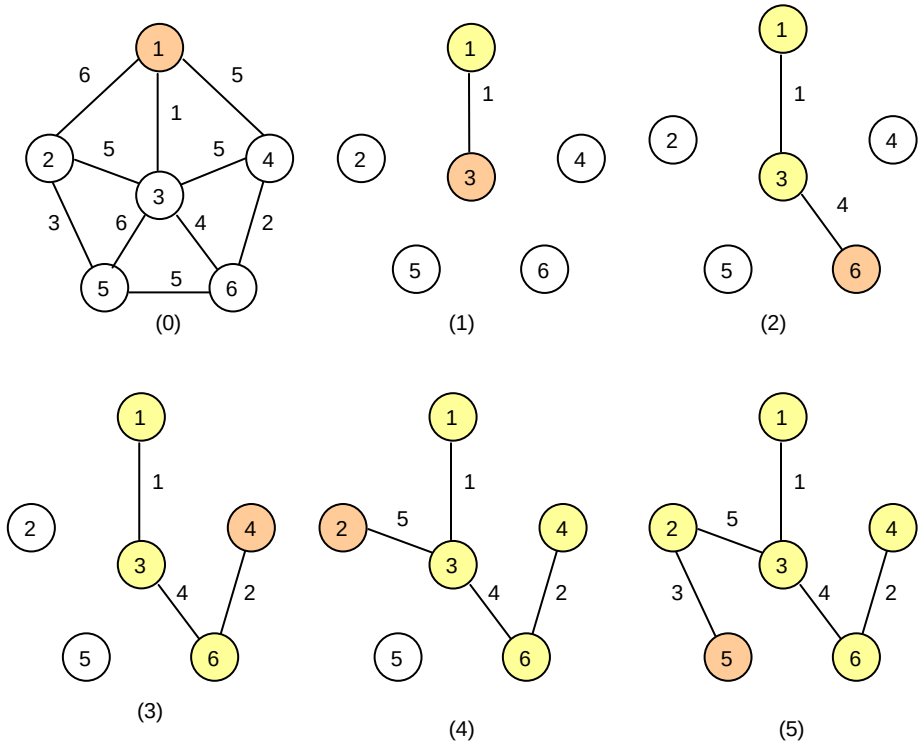


Fig.11.2.1.a. Construcţia unui arbore de acoperire minim al unui graf pe baza algoritmului lui Prim

11.2.1.1. Exemplu de implementare a algoritmului lui Prim

- Se presupune un graf definit prin:
  - (1) Mulţimea nodurilor sale `{1,2,3,...,n}`.
  - (2) Matricea `COST` care memorează costurile arcelor sale.
- O modalitate simplă de a afla arcul cu costul cel mai redus care conectează mulţimile `U` şi `N-U`, este aceea de a utiliza două **tablouri**.
  - Primul dintre tablouri, denumit `APROPIAT` memorează în locaţia `APROPIAT[i]` acel nod care este la momentul respectiv cel mai apropiat de nodul `i` şi care aparţine mulţimii `N-U`. Nodul `i` aparţine mulţimii `U`.
    - Acest tablou materializează de fapt **mulţimea** `N-U`.
  - Cel de-al doilea tablou denumit `COSTMIN,` memorează în locaţia nodului `i` costul arcului `(i,APROPIAT[i])`.
- **Algoritmul lui Prim** de construcţie al **arborelui de acoperire minim** este următorul:
  - (1) Se iniţializează `U` cu mulţimea vidă şi se selectează un nod de pornire care se introduce în mulţimea `U`.
  - (2) Se iniţializează corespunzător tablourile `APROPRIAT` şi `COSTMIN`.
  - (3) În fiecare pas al algoritmului se balează tabloul `COSTMIN` pentru a găsi acel nod, fie acesta `k`, aparţinând mulţimii `N-U,` care este cel mai apropiat de nodurile mulţimii `U`, adică nodul pentru care valoarea `COSTMIN[k]` este minimă.
  - (4) Se adaugă nodul `k` mulţimii `U` şi se tipăreşte arcul `(k,APROPIAT[k])` ca şi aparţinând **arborelui de acoperire minim.**
  - (5) În continuare se actualizează tablourile `COSTMIN` şi `APROPIAT` luând în considerare faptul că nodul `k` a fost adăugat mulţimii `U`. În consecinţă:
    - (1) Pe de o parte apar noi posibilităţi de conectare între mulţimile `U` şi `N-U`.
    - (2) Pe de altă parte costurile unor conexiuni existente se pot reduce prin introducerea noului nod `k` în mulţimea `U`.
  - (6) Algorituml se reia de la pasul (3) până când toate nodurile au fost adăugate mulţimii `U`.
- O versiune C a algoritmului lui Prim apar în secvenţa [11.2.1.1.a].
- Se presupune că `COST` este un tablou de dimensiuni `nxn,` astfel încât `COST[i,j]` reprezintă costul arcului `(i,j)`.
  - Dacă arcul `(i,j)`nu există, `COST[i,j]` are o **valoare mare specifică**. Vom nota această valoare cu simbolul ∞.

- Ori de câte ori se găseşte un nod k pentru a fi introdus în mulţimea U, se face COSTMIN[k] egal cu valoarea "**infinit**", pentru ca nodul respectiv să nu mai fie selectat.
    - Valoarea **infinit** reprezintă o valoare mare convenită, astfel încât acest nod să nu mai poată fi selectat în continuare spre a fi inclus în U. Vom nota această valoare cu ∞+.
    - Se face precizarea că valoarea **infinit** trebuie să fie **mai mare** decât costul oricărui arc al grafului, respectiv **mai mare** decât **costul** asociat **lipsei arcului**.

{Implementarea algoritmului lui Prim – varianta C}

```
procedure Prim(float COST[MaxNod,MaxNod], int n)

/*afişează arcele arborelui de acoperire minim pentru un
graf având nodurile {1,2,...,n} şi matricea COST pentru
costurile arcelor*/

float COSTMIN[MaxNod];
int APROPIAT[MaxNod];
int i,j,k,min;

/*i şi j sunt indici. În timpul parcurgerii tabloului
COSTMIN, k este indicele celui mai apropiat nod găsit până
la momentul curent, iar min=COSTMIN[k]*/

COSTMIN[1]= infinit+; /*nodul 1 este nod de start*/

/*iniţializează mulţimea U numai cu nodul 1*/
for (i=2;i<=n;i++)
  {
    COSTMIN[i]=COST[1,i];
    APROPIAT[i]=1;
  } /*for*/                              /*[11.2.1.1.a]*/

/*caută cel mai apropiat nod k din afara mulţimii U,
   faţă de mulţimea U*/
for (i=2;i<=n;i++)
  {
    min=COSTMIN[2];
    k=2;
    for (j=3;j<=n;j++)
      if (COSTMIN[j]<min)
        {
          min=COSTMIN[j];
          k=j; /*k este cel mai apropiat nod de mulţimea U*/
        } /*if*/
    *scrie(k,APROPIAT[k]); /*tipăreşte arcul minim*/
    COSTMIN[k]=infinit+;  /*k se adaugă lui U*/
    for (j=2;j<=n;j++)    /*ajustează costurile minime ale
            nodurilor mulţimii U după includerea nodului k*/
      if(COST[k,j]<COSTMIN[j] && COSTMIN[j]<infinit)
        {
          COSTMIN[j]=COST[k,j];
          APROPIAT[j]=k;
        } /*if*/
```

```
  } /*for*/
 /*Prim*/
-------------------------------------------------------------
```

- În figura 11.2.1.1.b apare urma execuţiei algoritmului lui Prim aplicat grafului din figura 11.2.1.1.a.(0).

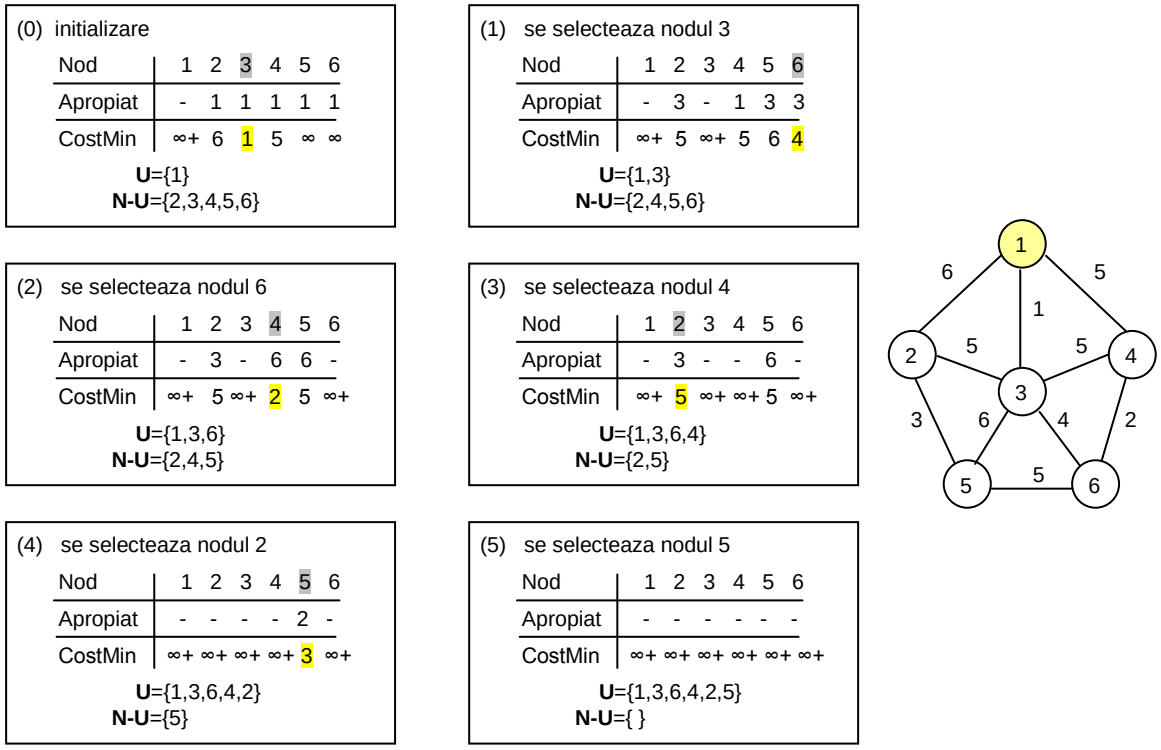


Fig.11.2.1.1.b. Exemplu de aplicare a algoritumului lui Prim

11.2.1.2. Analiza performanţei algoritmului lui Prim

- **Complexitatea** relativă la timpul de execuţie a implementării prezentate a algoritmului lui Prim este $O(n^2)$, deoarece:
  - (1) Se fac `n-1` iteraţii în bucla exterioară **`for`.**
  - (2) Fiecare iteraţie necesită $O(n)$ unităţi de timp datorită celor două bucle **`for`** interioare succesive, prima care determină arcul minim şi cea de-a doua care ajustează costurile lui `U`.
- Pentru valori mari ale lui `n`, performanţa algoritmului poate deveni nesatisfăcătoare.

11.2.2. Metoda căutării "bazate pe prioritate" ("Priority-First Search")

- După cum s-a precizat în &10.4.3, considerând nodurile unui graf divizate în trei clase, "arbore", "vecinătate" şi "neîntâlnite", atunci metodele de traversare a grafului se diferenţiază după maniera în care sunt alese nodurile care trec din clasa "vecinătate" în clasa "arbore".
  - Astfel la traversarea "**în adâncime**" se alege din vecinătate nodul **cel mai recent**

întâlnit (ultimul) ceea ce corespunde utilizării unei **stive** în păstrarea nodurilor clasei "vecinătate".

- La traversarea "**prin cuprindere**" se alege nodul **cel mai devreme întâlnit** (primul) ceea ce corespunde unei **structuri de date coadă**.
- Determinarea **arborelui de acoperire minim** se poate asimila cu acea traversare a grafului în care se alege din clasa "vecinătate" nodul cu **prioritatea maximă**.
  - Aceste poate fi spre exemplu, acel nod la care conduce **arcul** cu **ponderea minimă**.

- **Structura de date** care poate fi asociată acestei metode este **coada bazată pe prioritate**.
- În secvenţa [11.2.2.a] se prezintă o metodă de determinare a **arborelui minim** bazată pe considerentele mai sus precizate.
  - Tehnica utilizată se mai numeşte şi **căutare bazată pe prioritate** ("**priority first search**") [Se 88].
- Referitor la această secvenţă se fac următoarele **precizări**:
  - Graful se consideră reprezentat printr-o **structură de adiacenţe**, implementată cu ajutorul **listelor înlănţuite simple** (&10.3.2, Caz 1).
  - **Structura nodului** listelor de adiacenţe se completează cu câmpul "`cost`" utilizat pe post de **prioritate**.
    - În acest câmp se memorează costul arcului care conduce la nodul în cauză.
  - Procedura ***`Initializeaza`*** şi funcţiile ***`Extrage`*** şi ***`Vid`*** implementează operatorii respectivi în contextul **cozilor bazate pe prioritate**.
  - Funcţia *`boolean`* ***`Actualizeaza`***`(Coada_Bazata_Pe_Prioritate q, Tip_Nod k, Tip_Prioritate p)` implementează un operator special referitor la coada bazată pe prioritate.
    - Operatorul verifică dacă nodul `k` precizat ca parametru apare în coada bazată pe prioritate `q`, cu o prioritate cel puţin egală cu prioritatea `p` precizată ca parametru şi acţionază după cum urmează:
    - (1) Dacă nodul `k` nu apare în coadă el este inserat cu prioritatea `p.`
    - (2) Dacă nodul `k` apare în coadă însă are un cost mai mare (adică o prioritate mai mică) decât prioritatea `p` precizată ca parametru, se realizează schimbarea priorităţii sale la valoarea `p`.
    - (3) Dacă nodul `k` apare în coadă însă are un cost mai mic (adică o prioritate mai mare) decât cea precizată ca parametru, operatorul nu face nimic.
    - (4) Dacă se produce vreo modificare (inserţie sau modificare de prioritate) funcţia ***`Actualizeaza`*** returnează valoarea "`true`". Aceasta permite actualizarea corespunzătoare a tablourilor `marc` şi `parinte`.

```
/*Determinarea unui arbore de acoperire minim al unui graf
prin metoda căutării bazate pe prioritate - varianta
pseudocod - se utilizează TDA Coadă Bazată pe Prioritate*/
```

```
/*structuri de date*/

typedef struct Tip_Nod* Ref_Tip_Nod;

typedef struct Tip_Nod /*structura unui nod al listei de
                         adiacenţe*/
{
   int nume;
   int cost;
   Ref_Tip_Nod urm;
} Nod;

Ref_Tip_Nod StrAdj[MaxNod]; /*structura de adiacenţe*/
int id,k;
int marc[MaxNod]; /*tablou pentru evidenţa nodurilor*/
int parinte[MaxNod]; /*arborele de acoperire minim*/
Coada_Bazata_Pe_Prioritate q;

void CautaPrioritar(int k);

/*construieşte arborele de acoperire minim în varianta
indicator spre părinte (tabloul parinte) pornind de la nodul
k*/
Ref_Tip_Nod t;

 [1]  daca Actualizeaza(q,k,nevazut)  parinte[k]=0;
      /*amorsare căutare, valabilă numai pentru rădăcină*/
 [2]  repeta
 [3]    id=id+1;                              /*[11.2.2.a]*/
 [4]    k=Extrage(q); /*k este nodul cel mai prioritar*/
 [5]    marc[k]=-marc[k]; /*k trece în clasa "arbore"*/
 [6]    daca (marc[k]==nevazut) marc[k]=0; /*pt. rădăcină*/
 [7]    t=Stradj[k];  /*lista de adiacenţe a lui k*/
 [8]    cat timp (t<>null)
 [9]        daca (marc[t^.nume]<0)/*nevizitat sau în coadă*/
 [10]         daca Actualizeaza(q,t^.nume,t^.cost)
 [11]           marc[t->nume]=-(t->cost); /*nodul t^.nume
                                  trece în clasa "vecinătate"*/
 [12]           parinte[t->nume]=k; /*marcare părinte*/
              □ /*daca*/
 [13]         t=t->urm;
        □ /*cat timp*/
 [14]   pana cand Vid(q);
      □ /*repeta*/
 /*CautaPrioritar*/

void ArboreMinim
/*programul principal pentru construcţia unui arbore de
acoperire minim*/

id=0;
Initializeaza(q);
pentru (k=1 la N)
  marc[k]=-nevazut;
pentru (k=1 la N)
  daca (marc[k]==-nevazut)
    CautaPrioritar(k);
```

```
/*ArboreMinim*/
-------------------------------------------------------------
```

- Arborele de acoperire minim care în acest caz este un **arbore de căutare bazată pe prioritate** este păstrat în tabloul `parinte` în reprezentarea "indicator spre părinte".
- Fiecare locaţie a tabloului `parinte` memorează părintele nodului în cauză respectiv nodul care a determinat mutarea nodului din clasa "vecinătate" în clasa "arbore".
- Tabloul `mark` memorează starea nodurilor grafului.
  - Pentru fiecare nod `k` al arborelui, `marc[k]` memorează de fapt **prioritatea** nodului în cauză, respectiv **costul arcului** care-l leagă pe `k` de părintele său `părinte[k]`.
- Sunt valabile următoarele **convenţii**:
  - (1) Nodurile din clasa "arbore" sunt marcate cu **valori pozitive** în tabloul `marc`.
  - (2) Nodurile din clasa "vecinătate" sunt marcate cu **valori negative** (linia `[11]`).
  - (3) Nodurile din clasa "neîntâlnite" sunt marcate cu "`-nevăzut`" şi nu cu valoarea zero.
    - Se face precizarea că `nevăzut` reprezintă o valoare mare pozitivă.
- Se observă faptul că atâta vreme cât nodurile se găsesc în **coada bazată pe prioritate** ele sunt marcate în tabloul `marc` cu **valoarea negativă** a costului (priorităţii).
- În momentul în care un nod este trecut în clasa "arbore" i **se schimbă semnul** valorii memorate în tabloul `marc` (linia `[5]`).

11.2.2.1. Analiza performanţei metoda căutării "bazate pe prioritate"

- Analiza algoritmului de **căutare bazată pe prioritate** în determinarea **arborelui de acoperire minim** conduce la performanţa $O((n+a)\ log_2\ n)$.
- **Motivaţia** este următoarea:
  - În procesul construcţiei arborelui sunt parcurse toate nodurile şi toate arcele grafului.
  - Fiecare **nod** conduce la o inserţie şi fiecare **arc** la o eventuală modificare de prioritate în cadrul cozii bazate pe prioritate utilizate.
  - Presupunând că implementarea cozii bazate pe prioritate s-a realizat cu ajutorul **ansamblelor**, atunci atât inserţia cât şi modificarea se realizează în $O(log_2\ n)$ paşi.
  - În consecinţă performanţa totală va fi $O((n+a)\ log_2\ n)$.

11.2.2.2. Consideraţii referitoare la metoda căutării "bazate pe prioritate"

- Metoda **căutării "bazate pe prioritate"** are un pronunţat caracter de **generalitate**.

- Astfel, după cum se va vedea în continuare, pornind de la această metodă se poate dezvolta un algoritm care rezolvă **problema drumului minim**.
- De asemenea, pornind tot de la aceeaşi metodă va fi dezvoltat un algoritm care rezolvă aceleaşi probleme în cazul **grafurilor dense** cu un efort de calcul proporţional cu O (n^2).
- Istoric lucrările au evoluat însă altfel.
    - În anul 1957 **Prim** publică algoritmul pentru determinarea **arborelui de acoperire minim.**
    - În anul 1959 **Dijkstra** publică algoritmul referitor la **determinarea drumului minim.**
- Pentru clarificare, s-a convenit ca:
    - (1) În cazul **grafurilor dense** cele două soluţii să fie numite:
        - **Algoritmul lui Prim** pentru determinarea **arborelui de acoperire minim** al unui graf.
        - **Algoritmul lui Dijkstra** pentru determinarea **drumului minim** într-un graf .
    - (2) În cazul **grafurilor rare** algoritmul să poarte numele:
        - **Algoritm de căutare bazată pe prioritate**.
- După cum se va vedea, de fapt soluţiile propuse se întrepătrund şi ele nu sunt decât cazuri particulare ale unui **algoritm generalizat de căutare bazat pe prioritate**.
- Este evident faptul că metoda **căutării "bazate pe prioritate**" este aplicabilă cu preponderenţă în cazul **grafurilor ponderate** considerând **ponderile** drept **priorităţi**.
- Cu toate acestea, această metodă poate fi aplicată şi **grafurilor neponderate**.
- Într-un astfel de context, **căutarea bazată pe prioritate** poate **generaliza** tehnicile de traversare bazate pe **căutarea "în adâncime"** respectiv pe **căutarea "prin cuprindere"** prin atribuirea corespunzătoare de **priorităţi** nodurilor grafului.
    - Se reaminteşte faptul că `id` este o variabilă a cărei valoare se incrementează de la `1` la `n` pe măsură ce sunt procesate nodurile grafului în timpul execuţiei procedurii **`CautaPrioritar`** din secvenţa [11.2.2.a].
    - Variabila `id` poate fi utilizată în **atribuirea de priorităţi** nodurilor examinate, în baza convenţiei "**minim = prioritar**".
- Astfel, dacă în procesul de **căutare bazată pe prioritate**:
    - (1) Se consideră **prioritatea** unui nod egală cu `n-id` se obţine **căutarea** "**în adâncime**".
    - (2) Se consideră **prioritatea** unui nod egală chiar cu `id,` se obţine **căutarea** "**prin cuprindere**".
- În primul caz nodurile **nou întâlnite** au cea mai mare prioritate.
- În cel de-al doilea caz nodurile **cele mai vechi** adică cel mai devreme întâlnite, au cea mai mare prioritate.
- De fapt aceste atribuiri de priorităţi determină structura **coadă bazată pe prioritate** să se comporte ca o **stivă** respectiv ca şi o **coadă normală**, structuri specifice celor două tipuri de parcurgeri.

11.2.3. Algoritmul lui Kruskal

- Fie graful conex G=(N,A)cu N={1,2,...,n} şi cu funcţia de cost c definită pe mulţimea arcelor A.
- O altă metodă de construcţie a unui **arbore de acoperire minim**, este **algoritmul lui Kruskal**.
    - **Algoritmul lui Kruskal** porneşte de la un graf T=(N,Φ) care constă doar din cele n noduri ale grafului original G, dar care nu are nici un arc.
        - În această situaţie fiecare nod este de fapt o **componentă conexă** a grafului care constă chiar din nodul respectiv.
    - În continuare pornind de la mulţimea curentă a componentelor conexe, algoritmul selectează pe rând câte un **arc de cost minim** pe care îl adaugă componentelor conexe care cresc în dimensiune dar al căror număr se reduce.
    - În final rezultă **o singură componentă conexă** care este chiar **arborele de acoperire minim.**
- Pentru a construi componente din ce în ce mai mari se examinează arcele din mulţimea A în **ordinea crescătoare a costului** lor.
    - Dacă arcul selectat conectează două noduri aparţinând unor **componente conexe distincte**, arcul respectiv este adăugat grafului T.
    - Dacă arcul selectat conectează două noduri aparţinând unei **aceleeaşi componente conexe**, arcul este neglijat.
        - **Motivul**: introducerea sa ar conduce la apariţia unui ciclu în respectiva componentă şi în final la un ciclu în arborele de acoperire, lucru nepermis prin definiţie.
    - Aplicând în manieră repetitivă acest procedeu, la momentul la care toate nodurile grafului aparţin unei singure componente conexe, algoritmul se termină şi T reprezintă arborele de acoperire minim al grafului G.
- Cu alte cuvinte algoritmul lui Kruskal porneşte de la **o pădure** cu n arbori.
    - În fiecare din cei n-1 paşi, algoritmul combină doi arbori într-unul singur, utilizând ca legătură **arcul cu costul cel mai redus** curent.
    - Procedeul continuă până în momentul în care rămâne un singur arbore.
    - Acesta este **arborele de acoperire minim**.
- Spre exemplu considerând **graful ponderat** din fig.11.2.3.a (0)
    - În succesiunea (1) - (5) din cadrul aceleaşi figuri se prezintă maniera de determinare a unui arbore de acoperire minim al grafului în baza algoritmului lui Kruskal.
    - Ordinea în care se adaugă arcele rezultă din figură.
    - Iniţial se adaugă arcele cu costurile 1,2,3 şi 4, toate acceptate, întrucât nici unul dintre ele nu generează vreun ciclu.
    - Arcele (1,4) şi (3,4) de cost 5 **nu** pot fi acceptate deoarece conectează noduri aparţinând unei aceleeaşi componente (fig.11.2.3.a (d)) şi conduc la

cicluri.

- În consecinţă se acceptă arcul `(2,3)` de cost `5` care nu produce nici un ciclu încheind astfel construcţia arborelui de acoperire minim.
  - Ar putea fi selectat şi arcul `(5-6)` de cost `5`. Arborele de acoperire minim **nu** este unic.

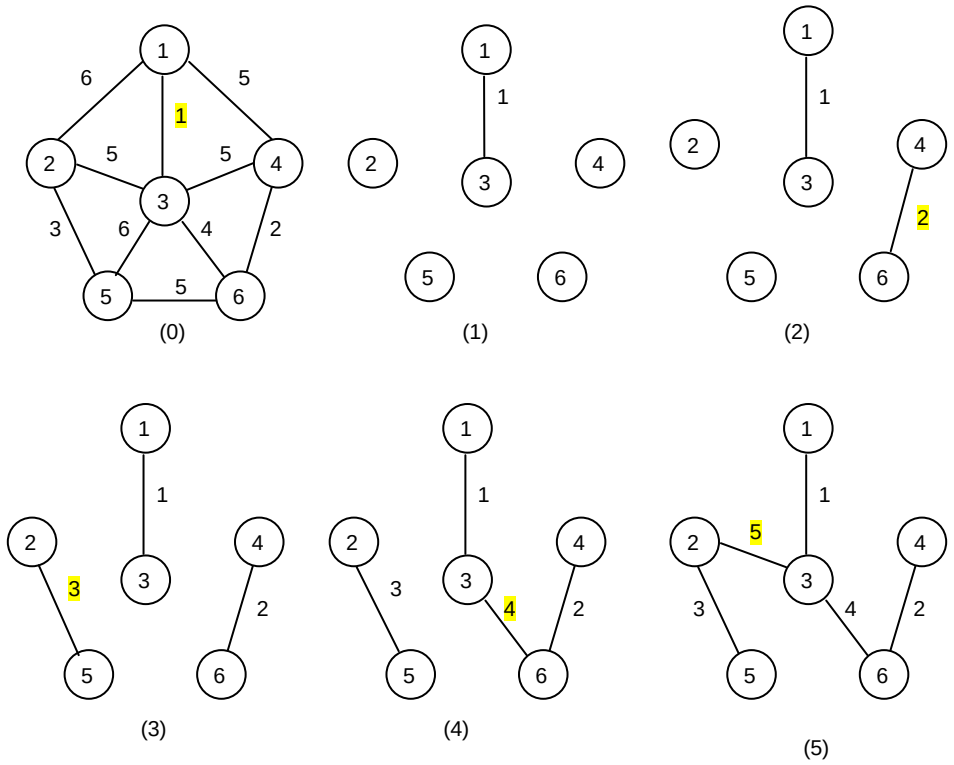


Fig.11.2.3.a. Construcţia unui arbore de acoperire minim pe baza algoritmului lui Kruskal

- Algoritmul lui Kruskal publicat în anul 1956 poate fi implementat în mai multe moduri.
- În continuare se prezintă un exemplu de implementare.

11.2.3.1. Exemplu de implementare a algoritmului lui Kruskal

- Algoritmul lui Kruskal poate fi implementat utilizând structura de date **mulţime de submulţimi** pe care sunt definiţi operatorii ***Uniune*** şi ***Caută***, aşa cum apare ea prezentată în paragraful &9.6.
  - Schiţa de principiu a unei astfel de implementări apare în secvenţa [11.2.3.1.a].

```
-------------------------------------------------------------
/*Implementarea algoritmului lui Kruskal - varianta
pseudocod C-like - se utilizează TDA Mulţime pe care sunt
definiţi operatorii Uniune şi Caută, TDA Coadă bazată pe
prioritate şi TDA Mulţime*/

procedura KRUSKAL(Mulţime_De_Noduri N, Mulţime_De_Arce A,
                  Mulţime_De_Arce T)

 int ncomp; /*numărul curent de componente*/
 Coada_Bazata_Pe_Prioritate CPArce; /* coadă bazată pe
```

```
    prioritate care păstrează arcele în vederea selecţiei
    arcului minim*/
 Mulţime_De_Submulţimi componente; /*mulţime de submulţimi
    pe care sunt definiţi operatorii Uniune şi CautăComp*/
 Tip_Nod x,y;
 Tip_Arc a;
 int compUrm; /*numele noii componente*/
 int xComp,yComp; /*nume de componente*/

[1]Vid(T); /*face mulţimea T (arborele minim) vidă*/
 /*iniţializează coada bazată pe prioritate pe coada vidă*/
[2] Initializeaza(CPArce);
[3] compUrm=0;
[4] nComp=(numărul de membri ai lui N);
 /*iniţializează mulţimea de submulţimi, astfel încât
   fiecare componentă să conţină un singur nod din N*/
[5] pentru (toate nodurile x aparţinând lui N)
      {
[6]     compUrm=compUrm+1;           /*[11.2.3.1.a]*/
[7]     InitializeazaComp(compUrm,x,componente);
      } /*pentru*/
 /*iniţializează coada de priorităţi a arcelor*/
[8] pentru (toate arcele aparţinând mulţimii A)
[9]   Insereaza(a,CPArce);
 /*determină arborele de acoperire minim T*/
[10]cat timp ((nComp>1)&&(!Vid(CPArce)))
     {
       /*cercetează arcul următor*/
[11]   a=Extrage(CPArce);  /*se presupune că a=(x,y)*/
[12]   xComp=CautaComp(x,componente);
[13]   yComp=CautaComp(y,componente);
[14]   daca (xComp<>yComp)
         {
           /*a conectează două componente diferite*/
[15]       Uniune(xComp,yComp,componente);
[16]       nComp=nComp-1;
           /*adaugă arcul a arborelui  minim reprezentat de
             mulţimea T*/
[17]       Adauga(a,T);
         } /*daca*/
     } /*cat timp*/
/*KRUSKAL*/
-----------------------------------------------------------
```

- Referitor la această implementare se fac următoarele **precizări**:
- (1) Mulţimea arcelor este structurată ca şi o **coadă bazată pe priorităţi** denumită `CPArce`.
    - În coada `CPArce` se introduc iniţial toate arcele grafului cu ajutorul operatorului ***Insereaza*** (cea de-a doua buclă **`pentru`**, liniile `[8]` şi `[9]` din cadrul algoritmului).
    - Din coada `CPArce` se extrage în fiecare pas al algoritmului cu ajutorul operatorului ***Extrage***, arcul curent cu costul minim (linia `[11]`).
- (2) **Mulţimea componentelor conexe** este structurată ca şi o **mulţime de submulţimi** denumită `componente` pe care sunt definiţi următorii operatori specifici:

- (a) ***Uniune***`(componentă A, componentă B, Mulţime_De_Submulţimi C);` – reuneşte componentele `A` şi `B` ale mulţimii de submulţimi `C`. Rezultatul uniunii se va numi fie `A` fie `B` în mod arbitrar. Se observă diferenţa faţă de definiţia utilizată în capitolul 9, unde `C` nu apare ca şi parametru.

- (b) `componentă` ***CautaComp***`(nod x, Mulţime_De_Submulţimi C)` – returnează numele acelei componente a lui `C` al cărei membru este nodul `x`. Operaţia va fi utilizată pentru a stabili dacă cele două noduri care determină un arc aparţin unei aceleaşi componente conexe sau la componente diferite.

- (c) ***IniţializeazăComp***`(componentă A, nod x, Mulţime_De_Submulţimi C)` – crează componenta cu numele `A` a mulţimii de submulţimi `C`. Componenta `A` va conţine numai nodul `x`.

- (3) **Arborele minim** este reprezentat ca şi o mulţime de arce `T` care este structurată ca şi un **TDA Mulţime clasic** peste care sunt definiţi operatorii:

  - (a) ***Vid***`(Multime_De_Arce T)` care crează mulţimea vidă `T`.

  - (b) ***Adauga***`(Tip_Arc a, Multime_De_Arce T)` care adaugă arcul `a` mulţimii de arce `T` (linia `[17]`).

- **Algoritmul lui Kruskal** are ca rezultat construcţia mulţimii `T` care cuprinde arcele care alcătuiesc **arborele de acoperire minim al grafului**.

11.2.3.2. Analiza performanţei algoritmului lui Kruskal

- Analiza performanţei **algoritmului lui Kruskal** se realizează în baza următoarelor constatări.

- Timpul de execuţie al implementări **algoritmului lui Kruskal** este dependent de doi factori:

  - (1) Implementarea **cozii bazate pe prioritate**.

  - (2) Implementarea **mulţimii de submulţimi**.

- Astfel, dacă există `a` arce:

  - Sunt necesare $O(a \log a)$ unităţi de timp pentru a insera toate arcele în coada bazată pe priorităţi (liniile `[8]` şi `[9]`), dacă aceasta este implementată cu ajutorul **ansamblelor** sau al **arborilor binari echilibraţi** spre exemplu.

- În fiecare iteraţie a buclei **`cat timp`** (linia `[10]`), determinarea arcului de cost minim (linia `[11]`) consumă în aceste condiţii $O(\log a)$ unităţi de timp.

  - În consecinţă dacă avem `a` arce **gestionarea cozii de priorităţi** consumă în cel mai rău caz $O(a \log_2 a)$ unităţi de timp.

- Timpul total necesar execuţiei operaţiilor ***Uniune*** (linia `[15]`) şi ***CautaComp*** (liniile `[12]` şi `[13]`) depinde de natura implementării structurii de date **mulţime de submulţimi**.

  - După cum s-a văzut în capitolul 9, există metode care obţin performanţe de ordinul $O(a \log_2 a)$ pentru un total de `a` elemente.

- În **concluzie** algoritmul lui Kruskal poate fi implementat astfel încât să fie executat în $O(a \log_2 a)$ unităţi de timp.

- **Cel mai defavorabil caz** este acela în care **graful nu este conex** şi în consecinţă trebuiesc examinate **toate arcele**.
- În situaţia în care **graful este conex**, **cel mai defavorabil caz** este acela în care arborele de acoperire minim constă din doi subarbori conectaţi prin **cel mai lung arc al grafului**, arc care va fi determinat doar după examinarea tuturor arcelor din coada bazată pe prioritate.
- Pentru grafurile tipice, este de aşteptat ca determinarea arborelui de acoperire minim (care conţine `n-1` arce) să se termine înaintea examinării tuturor arcelor, respectiv înainte de a ajunge la cel mai lung arc.
    - Cu toate acestea şi într-o astfel de situaţie timpul de execuţie este proporţional cu `a`, deoarece construcţia iniţială a cozii bazate pe prioritate presupune examinarea (inserţia) tuturor arcelor.

11.3. Drumul minim ("Shortest Path")

- Problema **drumului minim** se referă la a găsi acel drum într-un graf ponderat care conectează nodurile `x` şi `y` aparţinând grafului şi care se bucură de proprietatea că **suma ponderilor arcelor** care-l compun este **minimă**.
- Chiar şi în cazul **grafurilor neponderate**, situaţie în care **nu există ponderi**, problema prezintă încă interes.
    - Ea se referă la determinarea acelui drum care conţine **numărul minim de arce** care conectează pe `x` şi `y`.
    - În plus, se cunoaşte deja un algoritm care rezolvă elegant această problemă, şi anume **căutarea "prin cuprindere"**.
    - Se poate demonstra simplu prin **metoda inducţiei** că metoda **căutării "prin cuprindere"** care demarează cu nodul `x` va realiza în prima etapă vizitarea tuturor nodurilor la care se poate ajunge plecând de la `x` parcurgând un singur arc, apoi toate nodurile la care se poate ajunge de la `x` parcurgând două arce, ş.a.m.d.
        - Practic, la un moment dat, înainte de a trece la un nod `y` care este conectat la nodul `x` prin `k+1` arce, vor fi vizitate în prealabil toate nodurile conectate prin `k` arce.
        - Astfel, în momentul în care s-a ajuns la nodul `y`, a fost determinat şi **drumul minim**, deoarece nu există un alt drum mai scurt care conectează nodul `x` şi `y`.
- În general, **drumul minim** poate implica oricare două noduri `x,y` aparţinând unui graf.
- În acest context prezintă interes şi **generalizarea** acestei **probleme** în sensul de a determina **drumurile minime care conectează un nod `x` cu toate celelalte noduri ale grafului.**
- Această problemă este cunoscută sub denumirea de "**problema drumurilor minime cu origine unică".**
    - Ea poate fi rezolvată în contextul **grafurilor ponderate neorientate** pe baza unui algoritm cunoscut şi anume parcurgând graful prin tehnica **"căutării bazate pe prioritate".**
    - O altă manieră de rezolvare a acestei probleme în contextul **grafurilor orientate**

va fi prezentată în capitolul următor.

- Dacă se desenează **drumurile minime** care conectează nodul `x` cu toate celelalte noduri ale grafului, cu siguranţă nu se vor obţine cicluri şi în consecinţă rezultă un **arbore de acoperire corespunzător drumului minim**.
    - Fiecare nod al grafului conduce la un alt arbore de acoperire.
    - Astfel, pentru graful ponderat neorientat din figura 11.3.a. (a), arborii de acoperire corespunzători drumului minim, pentru nodurile `A`, `G` şi `M` apar în aceeaşi figură, poziţiile (b), (c) respectiv (d).

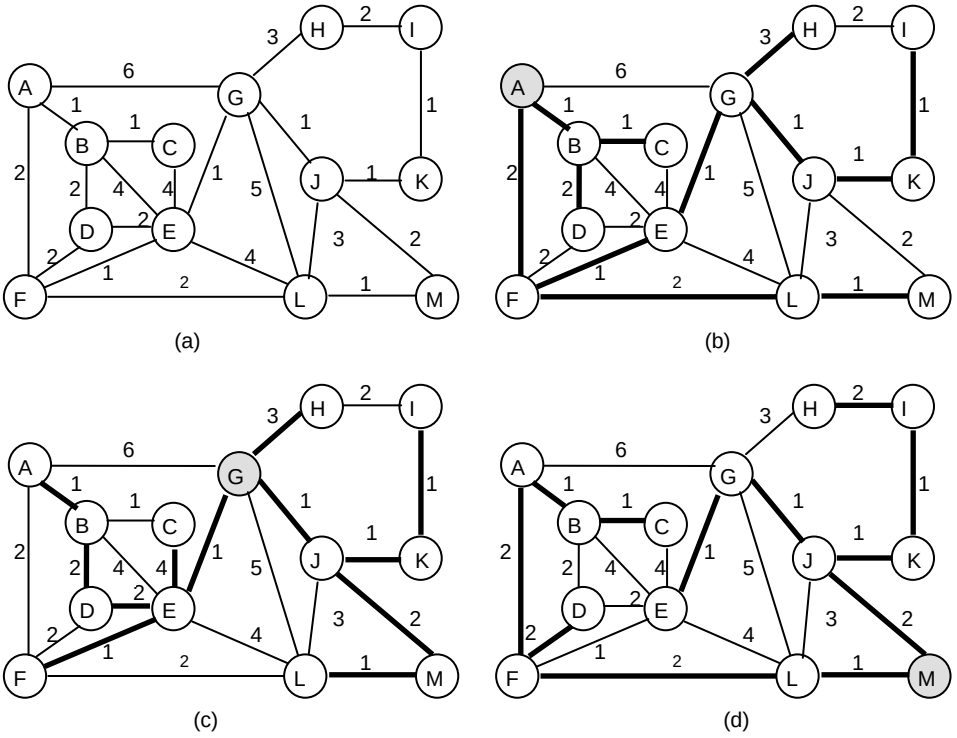


Fig.11.3.a. Arbori de acoperire corespunzători drumului minim

11.3.1. Determinarea drumurilor minime cu origine unică corespunzătoare unui nod al unui graf prin tehnica căutării "bazate pe prioritate"

- Problema determinării drumurilor minime corespunzătoare unui nod precizat al unui graf ponderat se reduce de fapt la determinarea **arborelui de acoperire corespunzător drumului minim** care are drept rădăcină **nodul** în cauză.
- Această problemă poate fi soluţionată într-o manieră principial identică cu determinarea **arborelui de acoperire minim.**
- Astfel, **arborele de acoperire corespunzător drumului minim** pentru nodul `x`, se poate construi adăugând în fiecare pas, nodul din clasa "vecinătate" care este **cel mai "apropiat" nodului origine** `x`.
    - Se reaminteşte faptul că procedura **`ArboreMinim`** (secvenţa [11.2.2.a]), care construieşte **arborele de acoperire minim (de cost minim)**, adăuga la fiecare pas arborelui minim, acel nod din clasa "vecinătate" care era **cel mai "apropiat" arborelui** la momentul respectiv, adică nodul cu costul curent minim.

- În situaţia de faţă, pentru a determina nodul curent cel mai "**apropiat**" de nodul `x`, se utilizează tot tabloul `marc`.
- Pentru fiecare nod `k` al arborelui de acoperire, `marc[k]` memorează **distanţa** de la nodul `k` la nodul `x`, parcurgând **drumul minim**, care de altfel este memorat de **arborele de acoperire.**
- În momentul în care nodul `k` este adăugat arborelui, se actualizează mulţimea "**vecinătate**" parcurgând lista de adiacenţe a lui `k`.
    - Pentru fiecare nod `t^.nume` aparţinând listei de adiacenţe a lui `k`, cea mai scurtă distanţă până la nodul `x`, trecând prin nodul `k`, este `marc[k]+t^.cost`.
    - Această remarcă conduce la implementarea imediată a algoritmului de determinare al **arborelui de acoperire corespunzător drumului minim**.
    - Pur şi simplu se utilizează procedura **`ArboreMinim`** (secvenţa [11.2.2.a]) bazată pe parcurgerea grafului prin **tehnica căutării "bazate pe prioritate"** în care se consideră prioritatea fiecărui nod `t` examinat, ca fiind egală cu `marc[k]+t^.cost`.
        - Aceasta presupune modificarea liniilor `[10]` şi `[11]` din procedura menţionată după cum urmează.

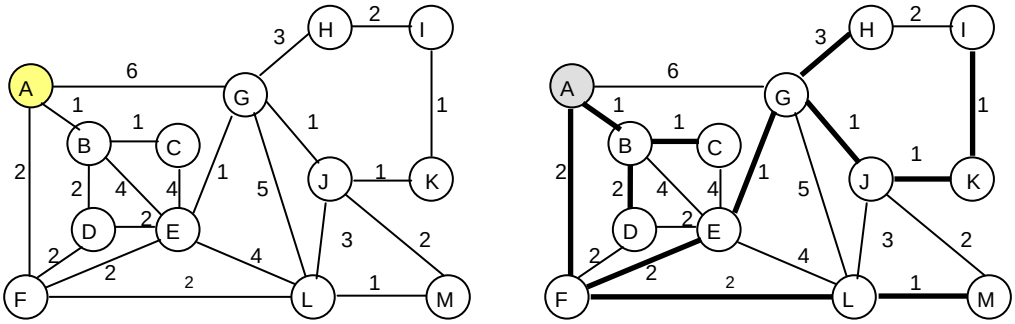
```
-------------------------------------------------------------
[10]     if Actualizeaza(q, t^.nume, mark[k]+t^.cost)
[11]          marc[t^.nume]=-(mark[k]+t^.cost); /*nodul
                        t^.nume trece în clasa "vecinătate"*/
-------------------------------------------------------------
```

- Noua formă a procedurii redenumită **`DrumMinim`** apare în secvenţa [11.3.1.a].

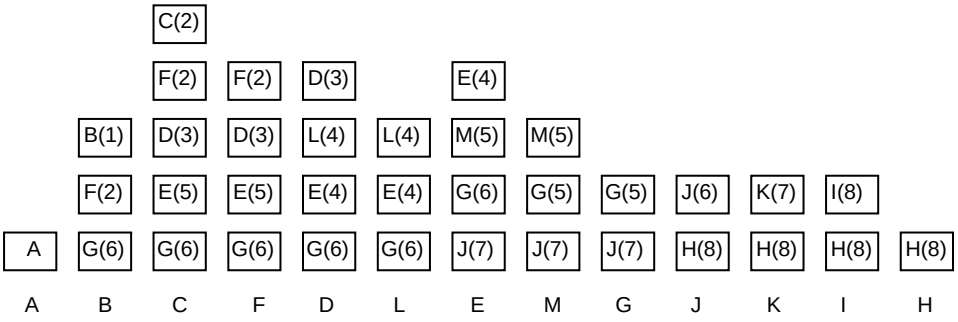
```
-------------------------------------------------------------
/*Determinarea unui arbore de acoperire corespunzător
drumului minim al unui graf prin metoda căutării bazate pe
prioritate - varianta pseudocod - se utilizează TDA Coadă
Bazată pe Prioritate */

/*structuri de date*/
typedef struct Tip_Nod* Ref_Tip_Nod;

/*structura unui nod al listei de adiacenţe*/
typedef struct Tip_Nod
{
   int nume;
   int cost;
   Ref_Tip_Nod urm;
} Nod;

Ref_Tip_Nod StrAdj[MaxNod]; /*structura de adiacenţe*/
int id,k;
int marc[MaxNod]; /*tablou pentru evidenţa nodurilor*/
int parinte[MaxNod]; /*arborele de acoperirie minim*/
Coada_Bazata_Pe_Prioritate q;


  procedura CautaPrioritar(int k)
    Ref_Tip_Nod t;
```

```
[1]  daca Actualizeaza(q,k,nevazut)
        parinte[k]=0;  /*amorsare proces de căutare*/
[2]  repeta
[3]    id=id+1;
[4]    k=Extrage(q);
[5]    marc[k]=-marc[k]; /*k trece în clasa "arbore"*/
[6]    daca (marc[k]==nevazut)
[7]       marc[k]=0; /*nodul origine*/
[8]    t=StrAdj[k];
[9]    cat timp (t<>null)                  /*[11.3.1.a]*/
[10]     daca (marc[t^.nume]<0) /*nod nevizitat sau în
                                  coadă*/
[11]       daca Actualizeaza(q,t^.nume,mark[k]+t^.cost)
[12]         marc[t^.nume]=-(mark[k]+t^.cost); /*nodul
                    t^.nume trece în clasa "vecinatate"*/
[13]         parinte[t^.nume]=k; /*înregistrare părinte*/
           □ /*daca*/
[14]       t=t^.urm;
         □
[15]   pana cand Vid(q);
       □
 /*CautaPrioritar*/

void DrumMinim;
/*programul principal pentru construcţia unui arbore de
acoperire corespunzător drumului minim*/

  id= 0;
  Initializeaza(q);
  pentru (k= 1 la N)
     marc[k]= -nevazut;
  pentru (k= 1 la N)
    daca (marc[k]== -nevazut)
      CautaPrioritar(k);
/*DrumMinim*/
-----------------------------------------------------------
```

- Utilizând procedura **DrumMinim** modificată pentru construcţia arborelui de acoperire corespunzător drumului minim pentru graful din figura 11.3.1.a (a) considerând nodul A drept origine, se obţine arborele de acoperire (b) iar evoluţia conţinutului cozii bazate pe prioritate apare în aceeaşi figură (c).
- Reprezentarea "**indicator spre părinte**" a arborelui construit apare în figura 11.3.1.b.
- Astfel, în graful menţionat din figura 11.3.a. (a) spre exemplu, cel mai scurt drum de la nodul A la nodul H are lungimea totală 8 (memorată în mark[8] care este intrarea pentru nodul H) şi porneşte de la nodul A trece prin nodurile F,E,G şi ajunge la nodul H (tabloul nume(parinte[k])).
    - Acest drum este efectiv memorat în tabloul părinte şi poate fi reconstituit în sens invers urmărind traseul precizat de părintele fiecărui nod începând cu H (poziţia 8) obţinându-se succesiunea H,G,E,F,A.



(a) (b)



(c)

Fig.11.3.1.a. Conţinutul cozii bazate pe prioritate pentru construcţia arborelui de acoperire corespunzător drumului minim

nume	A	B	C	D	E	F	G	H	I	J	K	L	M
k	1	2	3	4	5	6	7	8	9	10	11	12	13
nume(parinte[k])	0	A	B	B	F	A	E	G	K	G	I	F	L
marc[k]	0	1	2	3	4	2	5	8	8	6	7	4	5

Fig.11.3.1.b. Reprezentarea arborelui de acoperire corespunzător drumului minim

- Această metodă de determinare a **drumurilor minime cu origine unică** corespunzătoare unui nod al unui graf, este aplicabilă **grafurilor rare** şi ea se încadrează în performanţa precizată deja pentru algoritmul **`ArboreMinim`**, respectiv O*((n+a) log n)*.
- Algoritmul prezentat în acest paragraf este cunoscut şi sub numele de **algoritmul lui Dijkstra** pentru determinarea drumului minim, algoritm care va fi reluat într-o manieră mai aprofundată în capitolul destinat grafurilor orientate.

11.4. Arbori de acoperire şi drumuri minime în grafuri dense. Generalizarea algoritmului căutării bazate pe prioritate

- Este cunoscut faptul că în marea majoritate a aplicaţiilor, în cazul **grafurilor dense**, reprezentarea cea mai eficientă este cea bazată pe **matrice de adiacenţe**.
- Pentru un graf ponderat implementat printr-o matrice de adiacenţe, o modalitate uzuală de a reprezenta **coada bazată pe prioritate**, este aceea de a utiliza în acest scop un **tablou neordonat** [Se88].
    - Această reprezentare permite obţinerea unei performanţe de ordinul $O(n^2)$ pentru orice algoritm de traversare al grafului prin tehnica **căutării bazate pe prioritate**.
    - Performanţa este posibilă dacă se combină bucla de **actualizare a priorităţilor** cu bucla de **determinare a elementului minim**.
        - De fiecare dată când se extrage un nod din clasa "vecinătate", se procesează toate nodurile adiacente actualizându-li-se, dacă este cazul, priorităţile şi căutând **ponderea minimă**.
    - În consecinţă determinarea **arborilor de acoperire minimi** şi a **drumurilor minime** în **grafuri dense** utilizând **căutarea "bazată pe prioritate"** se poate realiza într-un interval de timp proporţional cu $O(n^2)$ (`n` numărul de noduri).
- În secvenţa [11.4.a] apare **un exemplu generalizat** de implementare a **tehnicii de căutare "bazată pe prioritate"**.
    - Se precizează faptul că această procedură **`CautPrioritar`** a fost dezvoltată pornind de la procedura **`ArboreMinim`** (secvenţa [11.2.2.a]) cu următoarele particularităţi:
        - (1) Graful se consideră reprezentat prin **matricea de adiacenţe** `A` care memorează ponderile arcelor, adică priorităţile nodurilor vizate.
        - (2) **Coada bazată pe priorităţi** se păstrează în tabloul `marc` iar operatorii definiţi pe această structură se implementează **direct** în cadrul algoritmului.
        - (3) Ca şi în cazul secvenţei [11.2.2.a] semnul unui element aparţinând tabloului precizează dacă nodul corespunzător (precizat prin indicele de intrare în tabloul `marc`) este în arbore dacă are semnul plus respectiv în coada bazată pe priorităţi, dacă are semnul minus.
            - Iniţial, toate nodurile aparţin clasei "neîntâlnite" şi ele se plasează în coada bazată pe priorităţi respectiv li se alocă valoarea `"-nevăzut"` în tabloul `marc`.
            - Pentru a modifica prioritatea unui nod, pur şi simplu se introduce noua prioritate în intrarea corespunzătoare nodului din tabloul `marc`.
            - Pentru a extrage nodul cu cea mai înaltă prioritate, se balează tabloul `marc` şi se caută poziţia care memorează cea mai mică valoare negativă, valoare care se complementează.
            - Nodul în cauză este extras astfel din coada bazată pe prioritate şi introdus în arbore (`părinte[t]=k`).
        - (4) Se precizează faptul că valoarea `"nevăzut"` trebuie să fie cu ceva mai mică decât valoarea maximă întreagă reprezentabilă, deoarece un

număr cu 1 mai mare decât valoarea "nevăzut" este folosit ca şi fanion pentru determinarea minimului, iar valoarea negativă a acestui număr trebuie să fie reprezentabilă.

- Tabloul marc se prelungeşte spre stânga cu componenta marc[0] utilizată pe post de **fanion**.

```
-------------------------------------------------------------
/*Exemplu generalizat de implementare a tehnicii de căutare
"bazată pe prioritate" - varianta pseudocod*/

/*structuri de date*/
float A[DimMax,DimMax]; /*matricea de adiacenţe (costuri)*/
float marc[DimMax]; /*tabloul marc - coada bazată pe
                      prioritate*/
int parinte[DimMax]; /*tabloul părinte - arborele de căutare
                       bazată pe prioritate*/
int N; /*numărul curent de noduri*/

procedura CautPrioritar;
  int k,min,t;

  /*iniţializare structuri de date*/
  pentru (k=1 la N)
    marc[k]=-nevazut;                    /*[11.4.a]*/
    parinte[k]=0;
  □
  marc[0]=-(nevazut+1); /*fanion*/
  min=1;
  repeta
    k=min; marc[k]=-marc[k]; min=0; /*trece nodul k
                                      în clasa arbore*/
    daca marc[k]=nevazut
      marc[k]=0; /*nodul origine*/
    pentru (t=1 la N)
      daca (marc[t]<0)
        daca ((A[k,t]<>0) && (marc[t]<(-prioritate)))
            marc[t]=-prioritate; /*nodul în coada BP*/
            parinte[t]=k; /*înregistrare părinte*/
        □ /*daca*/
        daca (marc[t]>marc[min])
          min=t;
      □ /*daca*/
    □ /*pentru*/
  pana cand (min==0);
  □ /*repeta*/
/*CautPrioritar*/
-------------------------------------------------------------
```

- Caracterul de **generalitate** al acestui algoritm rezultă din interpretarea valorii "prioritate":
  - (1) Dacă în matricea de adiacenţe se memorează ponderi şi dacă pe post de prioritate se utilizează valoarea A[k,t] se obţine **algoritmul lui PRIM** de determinare a **arborelui de acoperire minim** (de cost minim).
  - (2) Dacă se utilizează ca şi prioritate valoarea marc[k]+A[k,t] se obţine **algoritmul lui Dijkstra** pentru **problema drumului minim**.

- Presupunând că se utilizează variabila `id` care memorează numărul nodului curent vizitat:
    - (3) Dacă pe post de `prioritate` se utilizează valoarea `n-id` (`n` este numărul de noduri), se obţine **căutarea "în adâncime"**.
    - (4) Dacă pe post de `prioritate` se utilizează chiar `id` se obţine **căutarea "prin cuprindere"**.
- De fapt procedurile **`ArboreMinim`** (secvenţa [11.2.2.a]) şi **`CautPrioritar`** (secvenţa [11.4.a]) realizează în principiu acelaşi lucru cu următoarele diferenţe:
    - (1) Procedura **`ArboreMinim`** se aplică mai eficient **grafurilor rare** (implementate prin structuri de adiacenţe) în timp ce **`CautPrioritar`** este mai eficientă în cazul **grafurilor dense** (implementate ca matrici de adiacenţe).
    - (2) **Coada bazată pe prioritate** este implementată în primul caz printr-o metodă avansată (de exemplu structura ansamblu) iar în cel de-al doilea caz printr-un tablou neordonat.

11.4.1. Analiza algoritmului generalizat de căutare bazată pe prioritate

- Performanţa algoritmului generalizat de căutare bazată pe prioritate este $O(n^2)$.
- Această afirmaţie rezultă imediat din inspecţia secvenţei [11.4.a].
    - De fiecare dată când este vizitat un nod, se realizează o trecere prin toate cele `n` intrări ale rândului corespunzător nodului din matricea de adiacenţe cu scop dublu:
        - (1) De a actualiza ponderile tuturor nodurilor învecinate.
        - (2) De a găsi elementul cu prioritate maximă din coada bazată pe prioritate.
- Din punctul de vedere al numărului de arce, algoritmul este liniar, adică performanţa sa este $O(a)$.

11.5. Considerente referitoare la performanţele comparate ale algoritmilor de determinare a arborilor de acoperire minimi

- În cadrul acestui capitol au fost dezvoltaţi mai mulţi algoritmi care rezolvă **problema arborelui de acoperire minim** şi corelat cu aceasta şi **problema drumului minim**.
- Este vorba despre:
    - Algoritmul lui Prim
    - Metoda căutării "bazate pe prioritate"
    - Algoritmul lui Kruskal.
- Fiecare din aceşti algoritmi este mai potrivit pentru o anumită categorie de grafuri.
- Astfel, referitor la cel mai defavorabil caz se pot preciza următoarele:
    - **Căutarea "bazată pe prioritate"** obţine în cel mai defavorabil caz o performanţă de ordinul $O((a+n) \log n)$.

- Performanţa **algoritmului lui Prim** în cel mai defavorabil caz este O*(n^2)*.
- Performanţa **algoritmului lui Kruskal** în cel mai defavorabil caz este O*(a log a)*.

- În practică însă **nu** este recomandabil ca selecţia algoritmului utilizat să se realizeze după criteriul "**cel mai defavorabil caz**" din cel puţin două motive:
  - (1) Cazul cel mai defavorabil apare de regulă în practică cu o probabilitate redusă.
  - (2) Funcţie de situaţia reală, algoritmii care în cazul cel mai defavorabil obţin performanţe mai slabe, pot conduce în exploatarea curentă la performanţe superioare algoritmilor care în situaţii extreme obţin performanţe mai bune.

- În final se pot formula următoarele **concluzii**:
  - (1) **Căutarea "bazată pe prioritate"** şi **metoda lui Kruskal** rulează în intervale de timp proporţionale cu **numărul de arce**, pentru marea majoritate a grafurilor care apar în practică.
    - În primul rând, deoarece multe din arcele cercetate **nu** necesită ajustarea cozii bazate pe priorităţi, ajustare care necesită $\log_2 n$ paşi.
    - În al doilea rând, deoarece cel mai lung arc al arborelui de acoperire minim este probabil suficient de scurt pentru ca până la determinarea lui să nu fie extrase prea multe arce din coada bazată pe priorităţi.
  - (2) Pentru **grafuri rare**, este de aşteptat ca metoda **căutării "bazate pe prioritate"** să fie mai rapidă deoarece este de presupus că ea va gestiona cozi bazate pe prioritate scurte.
  - (3) **Algoritmul lui Prim**, rulează într-un timp proporţional cu **numărul de arce** în cazul **grafurilor dense**, **nu** însă şi în cazul **grafurilor rare**.