**Colecții de mulțimi disjuncte.**

O relație **R** este definită pe o mulțime **S**, dacă pentru orice pereche **(a, b), a, b ∈ S, a R b** este adevărat sau fals.

Dacă **a R b** este adevărat, atunci spunem că **a** este în relație cu **b**. De exemplu, dacă **a** şi **b** sunt vârfuri într-un graf, atunci o relație între vârfuri poate fi considerată adiacența lor (**a R b** ⇔ **a** şi **b** sunt adiacente printr-o muchie.

O *relație de echivalență* este o relație **R** care satisface proprietățile:

- *reflexivitate* **a R a** pentru ∀ **a ∈ S**
- *simetrie* **a R b ⇒ b R a**
- *tranzitivitate* **a R b && b R c ⇒ a R c**

Exemple:

- relația ≤ nu este o relație de echivalență deoarece nu este simetrică
- conectivitatea electrică este o relație de echivalență
- într-un graf neorientat relația „două vârfuri sunt legate printr-un drum" este o relație de echivalență

*Clasa de echivalență* a unui element **a ∈ S** este o submulțime din **S** care conține toate elementele aflate în relație cu **a**.

Clasele de echivalență formează o partiție pe **S** : fiecare element din **S** aparține unei clase de echivalență. Verificarea dacă două elemente sunt legate printr-o relație de echivalență revine la verificarea dacă cele două elemente aparțin aceleaşi clase de echivalență. Clasele de echivalență determinate de o relație de echivalență formează o *colecție de mulțimi disjuncte*.

*Operațiile specifice* unei colecții de mulțimi disjuncte sunt:

- **MakeSet** - *formează o mulțime* cu un singur element
- **Find** - *găseşte mulțimea* (clasa de echivalență) care conține un anumit element
- **Union** - *reuneşte* două *mulțimi* care conțin elemente aflate în relație, păstrând disjuncția față de celelalte mulțimi.

- Algoritmul UF (Union – Find) pentru colecții de mulțimi disjuncte este dinamic, întrucât mulțimile se se schimbă la execuție.
- Algoritmul UF nu lucrează cu valorile elementelor mulțimilor, motiv pentru care elementele mulțimilor ar putea fi identificate prin numere întregi
- Numele mulțimii la care aparține un element este de asemenea lipsit de semnificație, motiv pentru care va fi de asemenea precizat printr-un întreg. Fiecare mulțime este identificată printr-un *reprezentant*, care este unul dintre elementele mulțimii.
- Colecția de mulțimi disjuncte (CMD) este un tablou de mulțimi disjuncte. Fiecare mulțime este identificată prin poziția ei în colecție.

O mulțime disjunctă poate fi reprezentată prin

- listă înlănțuită
- arbore general.

O implementare naivă foloseşte o listă înlănțuită în care fiecare element este legat la reprezentantul mulțimii

Complexitățile operațiilor

- **MakeSet(): O(1)**
- **FindSet(): O(1)**
- **SetUnion(A,B):** "Copiază" elementele lui **A** în mulțimea **B** făcând ca toatele elementele din **A** să indice la reprezentantul mulțimii **B: O(A)**

O aplicație imediată a operațiilor cu mulțimi disjuncte este determinarea componentelor conexe ale unui graf neorientat , reprezentată prin algoritmul::

**Componente-Conexe(G)**

```
for  v∈V(G)
   MakeSet(v);
for (u,v) ∈ E(G)
   if FindSet(u)==FindSet(v)
      SetUnion(u, v)
```

**Implementarea cu listă înlănțuită**

- Fiecare mulțime disjunctă este o listă înlănțuită, cu acces la primul şi la ultimul element
- Fiecare nod al listei va conține: valoarea unui element al mulțimii şi pointeri la următorul element al mulțimii şi la reprezentantul mulțimii.

**Implementarea cu arbori**

- Fiecare mulțime disjunctă se reprezintă printr-un arbore general (multicăi) reprezentat printr-un tabel de predecesori. Colecția de mulțimi disjuncte va fi o pădure de mulțimi disjuncte.
- Rădăcina fiecărui arbore conține reprezentantul mulțimii, care este propriul său părinte

**FormeazăMulțime - MakeSet** – crează un arbore cu un singur nod
**GăseşteMulțime - FindSet** – determină rădăcina (reprezentantul mulțimii) pe calea predecesorilor (calea de căutare).
**Reuneşte -SetUnion)** – predecesorul nodului rădăcină a unuia dintre subarbori va fi rădăcina celuilalt arbore
Dacă **n** este numărul de mulțimi disjuncte, pot exista cel mult **n-1** operații **Reuneşte()**, obținându-se în acest caz o singură mulțime disjunctă (un singur arbore). Pentru îmbunătățirea timpului de execuție al acestei operații se utilizează următoarele euristici:

1) *Reuniunea după rang* – arborele cu mai puține noduri devine subarbore a celui cu mai multe noduri. Dimensiunea arborelui este păstrată prin **rang = ⌈log n⌉**, care este marginea superioară a înălțimii arborelui
2) *Comprimarea drumului* – în timpul execuției operației **FindSet** se modifică legătura la predecesor, astfel încât fiecare nod de pe calea de căutare să indice către rădăcină (reprezentantul mulțimii disjuncte). Comprimarea drumului nu modifică rangul.

Valorile elementelor mulțimii nu sunt semnificative pentru colecția de mulțimi disjuncte, aşa că le vom identifica prin întregi:

```
typedef int TipElem;
```

Reprezentantul unei mulțimi disjuncte este un element al mulțimii, deci tot un întreg:

```
typedef int Set;
```

Rangul unei mulțimi poate fi reprezentat în rădăcină printr-o valoare negativă sau zero (pentru a face distincția de predecesor).
Colecția de mulțimi disjuncte este reprezentată printr-un tablou de întregi:

```
typedef int *CMD;
```

```
//Formeaza n multimi initializate cu 1, 2,…
//specificarea multimii se face prin indice in S – colecția de mulțimi disjuncte
//S[i]=0 deoarece reprezentantul are rangul 0 (1 element)
CMD CMD_New(int n){
  CMD S;
  S = (CMD)calloc(n, sizeof(int));
  return S;
}
```

```
//Reuneste - in locul rangului unuia din arbori se pune radacina
celuilalt
// nu se foloseste inca nici o euristica
void SetUnion(Set R1, Set R2, CMD S){
  S[R2] = R1;
}

//GasesteMultimea la care apartine elementul
Set FindSet(TipElem x, CMD S){
  if(S[x]<=0) return x;
  return FindSet(S[x], S);
}
```

Dacă introducem euristica reuniunii după rang, vom compara rangurile (păstrate în rădăcinile arborilor ca valori negative sau 0). Se va actualiza rădăcina arborelui cu rang mai mic, care devine subarbore a arborelui cu rang mai mare. Dacă cei doi arbori au acelaşi rang se actualizează rădăcina unuia dintre ei (oricare); prin reuniunea lor numărul de noduri se dublează, deci rangul creşte cu o unitate (scade cu 1 pentru că se păstrează ca valoare negativă).

```
//Reuneste dupa rang
void SetUnion(Set R1, Set R2, CMD S){
  if(S[R2]<S[R1]) //R2 are rang mai mare
    S[R1]=R2;
  else
    {
      if(S[R1]==S[R2])
        S[R1]--; //creste rangul lui R1
      S[R2] = R1;
    }
}
```

Algoritmul UF are în cea mai nefavorabilă situație complexitatea **O(m log n)**, unde **m** este numărul de operații **Find**, iar **n** – numărul inițial de mulțimi disjuncte. Se preferă un algoritm **Find** rapid; compresia căii se face în operația **Find** şi este independentă de **Union**.

```
// Find cu compresia drumului
Set FindSet(Set x, CMD S){
  if(S[x]<=0)
    return x;
  S[x] = FindSet(S[x], S);
}
```

Funcția are două treceri:

1. un pas în sus pe drumul de căutare spre părinte
2. un pas în jos pe drumul de căutare pentru a actualiza fiecare nod care să indice rădăcina

Dacă ne aflăm în rădăcină se returnează reprezentantul. Apelul recursiv returnează un pointer la rădăcină, fiind actualizat nodul de pe cale care va indica direct rădăcina.

Reuniunea după rang conduce la un timp de execuție **O(m log n)**. Compresia drumului are complexitatea

$\Theta(m \log_{(1+m/n)} n)$ pentru $m \geq n$

$\Theta(n + m \log n)$ pentru $m < n$

Dacă se folosesc ambele euristici avem complexitatea $O(m.\alpha(m,n))$ cu $\alpha(m,n)$ – inversa funcției Ackermann, care creşte foarte încet şi în mod practic $\alpha(m,n) \leq 4$ deci complexitate $O(m)$

**Determinarea arborelui de acoperire minimal (minimal spanning tree).**

Considerăm un graf neorientat ponderat: **G = (V, EP)** în care **EP** este *matricea ponderată a costurilor*. Se cere să determinăm *arborele minimal de acoperire*. Acesta este un subgraf parțial de acoperire (conex şi aciclic) $G_A$**= (V, T)** în care $T \subseteq EP$ a.î.:

1. toate vârfurile din **V** să rămână conectate
2. **T** să nu conțină cicluri
3. suma costurilor muchiilor din **T** să fie minimă: $w(T) = \sum_{(u,v) \in T} w(u, v) = \min$

Algoritmii cei mai cunoscuți (Prim, Kruskal) utilizează o strategie Greedy. Aceştia au complexitatea $O(m.\log_2 n)$, care poate fi redusă la $O(m+n.\log_2 n)$, dacă se utilizează heapuri Fibonacci.

Un *algoritm generic* porneşte cu un subset **A** al arborelui parțial de cost minim **T**, $(A \subset T \subset EP)$ şi adaugă în fiecare pas câte o muchie **(u,v)** la arborele de acoperire de cost minim **A**, astfel încât $A \cup \{(u,v)\}$ să aparțină de asemenea arborelui de acoperire de cost minim. O astfel de muchie poartă numele de *muchie sigură*.

**Algoritm generic AAM(G, w)**

```
A ← ∅
while A nu formează AAM)
  găseşte muchia sigură (u,v) pentru A
  A ← A ∪ {(u,v)}
return A;
```

Pentru a găsi muchiile sigure vom introduce următoarele definiții:

- o *tăietură* **(S, V-S)** este o partiție a vârfurilor **V** (cele două submulțimi sunt disjuncte)
- o muchie **(u, v)** *traversează tăietura*, dacă una din extremitățile muchiei este în **S** şi cealaltă în **V-S**
- o tăietură *respectă* mulțimea de muchii **A**, dacă nici o muchie nu traversează tăietura
- o *muchie uşoară* este o muchie care traversează tăietura şi are cost minim în raport cu celelalte muchii care traversează tăietura.

*Teoremă*: Dacă:

- **G=(V, EP)** un graf neorientat conex ponderat
- **A** o submulțime a lui **EP**, inclusă în arborele de acoperire de cost minim a lui **G**
- o tăietură **(S, V-S)** care respectă mulțimea **A**
- **(u, v)** o muchie uşoară care traversează tăietura **(S, V-S)**

atunci **(u, v)** este muchie sigură pentru **A**.

**AACM** are **n-1** muchii, aşadar în algoritmul generic bucla în care se adaugă o muchie se va repeta de **n-1** ori.

**Algoritmul Kruskal.**

Urmăreşte algoritmul generic, pornind cu o pădure având **n** arbori de acoperire minimi, fiecare conținând un vârf al grafului. Se execută **n-1** paşi, într-un pas se conectează doi arbori diferiți printr-o muchie de cost minim (muchie uşoară, deci şi muchie sigură, care păstrează invarianța : arborele astfel format este de asemenea de cost minim).

Implementarea algoritmului foloseşte TDA *colecție de mulțimi disjuncte*, cu operațiile:

1. **MakeSet(v)** – crează o mulțime conținând elementul **v**
2. **FindSet(v)** – caută mulțimea căreia îi aparține **v**
3. **SetUnion (u,v)** – reuneşte mulțimile conținând elementele **u**, respectiv **v**

**Kruskal(V, E, A)**

```
A ← ∅
for v ∈ V
  MakeSet(v)
sortare crescatoare muchii din E nedescrescător, dupa ponderi
for (u,v) ∈ E
  if FindSet(u)≠FindSet(v)
    A ← A ∪ {(u,v)}
    SetUnion (u,v)
```

Algoritmul Kruskal :

- prelucrează muchiile în ordinea ponderilor
- adaugă o muchie, dacă aceasta nu formează un ciclu cu restul muchiilor
- se opreşte după **n-1** paşi

O implementare a algoritmului Kruskal care foloseşte o coadă prioritară **Q** pentru a sorta crescător muchiile grafului este dată mai jos. S-a folosit, de asemenea o colecție de mulțimi disjuncte **P**.

```
Graf Kruskal(Graf G){
   CMD  P = CMD_New (N);
   Graf T = G_New ();
   PQ   Q = PQ_New();
   Varf u, v;
   int ku, kv;
   Arc a;
   Int n = G_Size(G);
   for(v=primV(G); !dultimV(G); v=avansV(G,v))
      if(G_IsV(G,v))
         G_AddV(T, v);
   for(a=primA(G); !dultimA(G); a=avansA(G,a))
      PQ_Insert (Q, G_GetCost(G,a), a);
   while(G_Size(T) < n-1){
      a = (Arc)PQ_DelMin(Q);
      u = V1(a);
      v = V2(a);
      ku = G_GetKV(G,V1(w));
      kv = G_GetKV(G,V2(w));
      if(FindSet(ku, P)!=FindSet(kv, P)){
         G_AddA(T, u, v, A_GetCost(a),A_GetEt(a));
         SetUnion(u, v, P);
      }
   }
   return T;
}
```

**Algoritmul Prim.**

Spre deosebire de algoritmul Kruskal, în algoritmul Prim, muchiile din **A** formează un singur arbore.
Se pleacă dintr-un vârf arbitrar şi se creşte arborele, până când acesta include toate vârfurile. In fiecare pas se adaugă la arbore o muchie uşoară care uneşte un vârf din **A** cu un vârf din **V-A**. Strategia este greedy deoarece muchia adăugată este de cost minim.

Pentru o implementare eficientă, noua muchie care se adaugă la **A** trebuie să poată fi selectată cu uşurință. In acest scop, vârfurile care nu au fost ataşate arborelui sunt plasate într-o coadă prioritară. Cheia după care se face inserarea vârfului neselectat **v** în coadă este ponderea minimă a muchiei dintre **v** şi un vârf din arbore (dacă nu există o asemenea muchie, cheia se ia foarte mare).
Mulțimea **A** este păstrată prin subgraful predecesor: **(V, A)** cu

**A = {(pred[v], v) : v ∈ V – {r}- QP}**

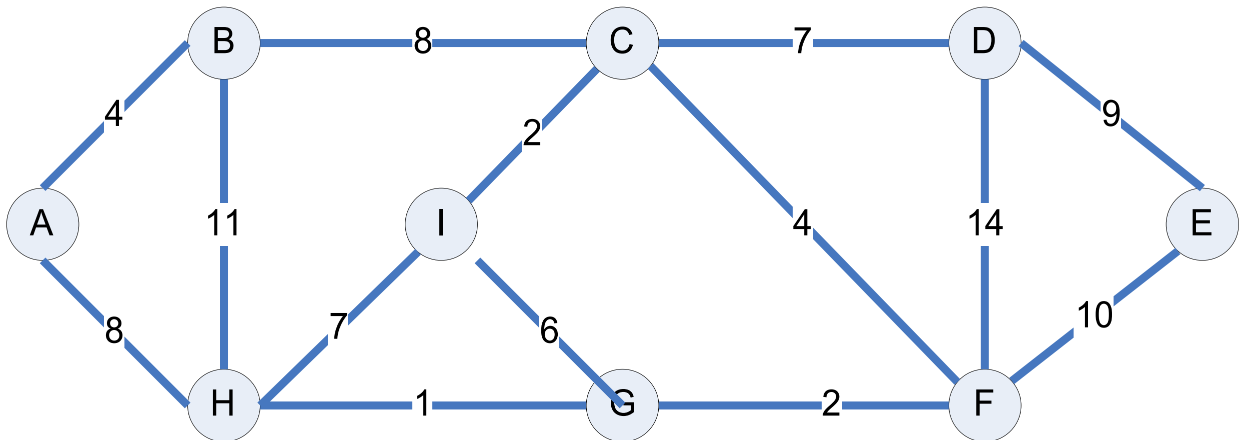
iar în final, când coada prioritară este vidă:

**A = {(pred[v], v) : v ∈ V – {r}}**

**Prim(V, E, r)**

```
for u ∈ V
  d[u] ← INF
  pred[u] ← 0
d[r] ← 0
for u ∈ V
  p ← Insert(QP, u)
```

```
  poz[u] ← p
while !Empty(QP)
  u ← ExtrageMin(QP)
  for e ∈ LA[u]
    v ← Opus(e,u)
    if v ∈ QP && w(u,v)<d[v]
      pred[v] ← u;
      d[v] ← w(u,v);
      ChPri(Q, poz[v],d[v])
```




| QP | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | ω | ω | ω | ω | ω | ω | ω | ω |
| pred | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| QP | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | ω | ω | ω | ω | ω | 8 | ω |
| pred | 0 | a | 0 | 0 | 0 | 0 | 0 | a | 0 |

| QP | a | b | h | c | d | e | f | g | i |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 8 | ω | ω | ω | ω | ω |
| pred | 0 | a | a | b | 0 | 0 | 0 | 0 | 0 |

| QP | a | b | c | h | d | e | f | g | i |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 8 | 7 | ω | 4 | ω | 2 |
| pred | 0 | a | b | a | c | 0 | c | 0 | c |

| QP | a | b | c | i | f | d | h | e | g |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 2 | 4 | 7 | 8 | ω | ω |
| pred | 0 | a | b | c | c | c | a | 0 | 0 |

| QP | a | b | c | i | f | d | h | e | g |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 2 | 4 | 7 | 7 | ω | 6 |
| pred | 0 | a | b | c | c | c | i | 0 | i |

| QP | a | b | c | i | f | g | d | h | e |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 2 | 4 | 2 | 7 | 7 | 10 |
| pred | 0 | a | b | c | c | f | c | i | f |

| QP | a | b | c | i | f | g | d | h | e |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 2 | 4 | 2 | 7 | 1 | 10 |
| pred | 0 | a | b | c | c | f | c | g | f |

| QP | a | b | c | i | f | g | h | | d | e |
|---|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 2 | 4 | 2 | 1 | | 7 | 10 |
| pred | 0 | a | b | c | c | f | g | | c | f |

| QP | a | b | c | i | f | g | h | d | e |
|---|---|---|---|---|---|---|---|---|---|
| cheie | 0 | 4 | 8 | 2 | 4 | 2 | 1 | 7 | 9 |
| pred | 0 | a | b | c | c | f | g | c | d |

Scăderea distanţei unui vârf, conduce la reaşezarea lui în coada prioritară. În acest scop se utilizează funcţia heapului binar (cozii prioritare) de schimbare a priorităţii, având semnătura: **`int ChPr(PQ Q, int Pozitie, int Pr)`**, care presupune cunoaşterea poziţiei în heap a elementului care va fi mutat. În acest scop se modifică funcţia de inserare în heap a unui element, astfel încât aceasta să întoarcă poziţia unde a fost inserat elementul în heap (noua semnătură **`int PQ_Insert(PQ Q, int v, int pr)`**). De asemenea, fiecărui vârf i se asociază suplimentar un câmp **`PozinQ`**, indicând poziţia vârfului în coada prioritară.

```
Graf Prim(Graf G, int s){
   PQ   Q = PQ_New ();
   Graf T = G_New ();
   Arc a;
   Varf v;
   int poz;
   for(v=primV(G); v!=dultimV(G); v=avansV(G,v))
      if(G_IsV(G, v)){
         G_AddV(T, v);
         G_SetDist(T, v, INF);
         G_SetCol(T, v, ALB);
         G_SetPred(T, v, 0);
         poz = PQ_Insert(Q, v, G_GetDist(T, v));
         G_SetPoz(T, v, poz);
      }
   G_SetDist(T, s, 0);
   poz = PQ_ChPr(Q, G_GetPoz(T,s), 0);
   G_SetPoz(T, s, poz);
   while(!Q_Empty(Q)){
      int u = PQ_RemoveMin(Q);
      G_SetCol(T, u, gri);
      for(a=primAinc(T,u); !dultimAinc(a,u); a=avansAinc(T, a, u){
         int w = Opus(a, u);
         int c = A_GetCost(a);
         if(G_GetCol(T, w)==alb && c < G_GetDist(T, w){
            G_SetDist(T, w, c);
            G_SetPred(T, w, u);
            poz=G_GetPoz(T,w);
            poz=PQ_ChPri(Q, poz, c);
            G_SetPoz(T, w, poz);
         }
      }
   }
   return T;
}
```

**Algoritmi pentru drumuri în grafuri.**

Se consideră graful orientat ponderat **G=(V, E, w)**, cu $\mathbf{w: E \rightarrow R}$ şi se defineşte costul drumului $d=(v_0, v_1, \ldots, v_p)$ :

$$w(d) = \sum_{i=1}^{p} w(v_{i-1}, v_i)$$

Costul optim de la **u** la **v** este:

$$\delta(u, v) = \begin{cases} \min \{w(d) : u \overset{d}{\dashrightarrow} v\} & \text{dacă } \exists\ u \to v \\ \infty & \text{altfel} \end{cases}$$

BFS calculează cea mai scurtă cale pentru grafuri neponderate la care muchiile au lungime 1.
Se pot considera următoarele variante de drumuri minime:

- drumuri minime de la o sursă **s** la celelalte vârfuri **v**
- drumuri minime la o destinație unică **t** de la celelalte vârfuri **v**
- drumul minim de la o sursă unică **u** la o destinație unică **v**
- drumuri minime între surse şi destinații multiple

Pentru păstrarea drumului minim se foloseşte subgraful predecesor $G_p=(V_p, E_p)$ în care:

$V_p = \{v \in V : pred[v] \neq 0\} \cup \{s\}$
$E_p = \{(pred[v], v) \in E : v \in V_p - \{s\}\}$

Un drum minim între două vârfuri conține alte subdrumuri optimale (subdrumurile drumului minim sunt drumuri minime). Fie $d = (v_1, v_2, \ldots, v_p)$ drumul minim de la $v_1$ la $v_p$. Atunci
$1 \le i \le j \le p$, $d_{ij} = (v_i, \ldots, v_j)$ este drum minim.

*Relaxarea* este o metodă care scade în mod repetat limita superioară a ponderii celei mai scurte căi a fiecărui vârf.
Inițial se face o estimare a drumului minim:

**Estimare(G,s)**

```
for(v∈V)
  d[v] ← INF
  pred[v] ← 0
d[s] ← 0;
```

Relaxarea muchiei **(u,v)** testează dacă se poate îmbunătăți calea minimă la **v**, trecînd prin **u**:

**Relaxare(u, v, w)**

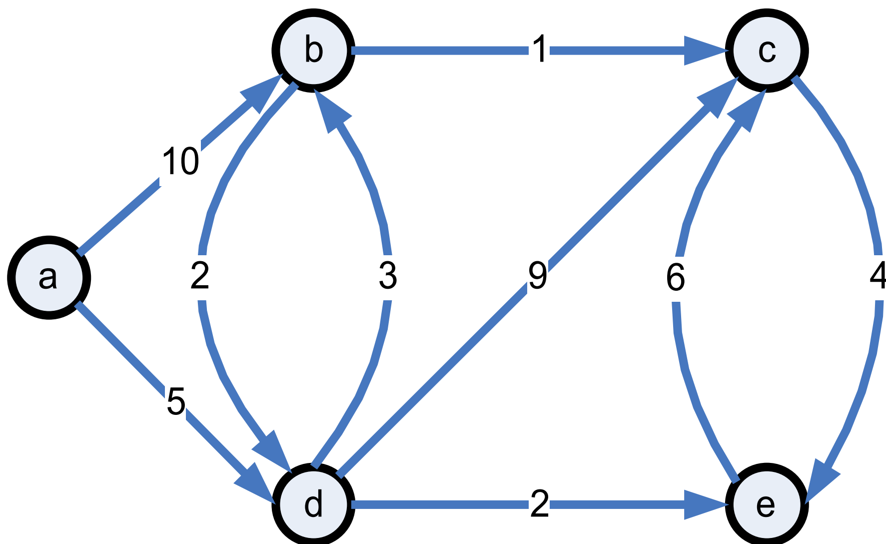
```
if(d[v] > d[u]+w(u,v))
  d[v] ← d[u] + w(u,v)
  pred[v] ← u
```

Prin relaxare estimările drumurilor minime scad monoton către ponderea drumului minim.
*Algoritmul Dijkstra* partiționează mulțimea vârfurilor **V** într-o mulțime **S** de vârfuri selectate, a căror drum minim a fost deja determinat, şi mulțimea **V-S** a vârfurilor rămase:
$S = \{v : d[v] = \delta(s, v)\}$
In pasul următor este selectat vârful $u \in V - S$ având cea mai mică estimare a căii minime: $S \leftarrow S + \{u\}$
Se relaxează toate muchiile care pleacă din **u**.
Vârfurile neselectate din **V-S** sunt păstrate într-o coadă prioritară **QP** ordonată după cheia **d**.

**Dijkstra(G,w,s)**

```
Estimare(G, s)
S←∅
QP←V
while !Empty(QP)
   u ← Extrage(QP)
   S ← S ∪ {u}
   for v ∈ LA[u]
      Relaxare(u,v,w)
```

Algoritmul Dijkstra foloseşte o strategie greedy în alegerea vârfului pe care-l pune în **S**. Operația **d[v]←d[u]+w(u,v)** presupune scăderea cheii asociate lui **v**, deci reanjarea lui în coada prioritară.




Formăm lista succesorilor:

| Virf | lista succesori | | |
|---|---|---|---|
| a | b(10) | d(5) | |
| b | c(1) | d(2) | |
| c | e(4) | | |
| d | b(3) | c(9) | e(2) |
| e | a(7) | c(6) | |

Estimarea drumurilor minime pornind din **a** este:

| QP | a | b | c | d | e |
|---|---|---|---|---|---|
| d | 0 | ω | ω | ω | ω |
| pred | 0 | 0 | 0 | 0 | 0 |

Se extrage **a** şi se relaxează muchiile incidente în **a** (se actualizează drumurile la vecinii lui **a**)

| QP | a | b | c | d | e |
|---|---|---|---|---|---|
| d | 0 | 0+10 | ω | 0+5 | ω |
| pred | 0 | a | 0 | a | 0 |

Se reordonează coada conform priorităților:

| QP | a | d | b | c | e |
|---|---|---|---|---|---|
| d | 0 | 5 | 10 | ω | ω |
| pred | 0 | a | a | 0 | 0 |

Se extrage **d** şi se relaxează muchiile incidente în **d**:

| QP | a | d | b | c | e |
|---|---|---|---|---|---|
| d | 0 | 5 | 5+3 | 5+9 | 5+2 |
| pred | 0 | a | d | d | d |

| QP | a | d | e | b | c |
|---|---|---|---|---|---|
| d | 0 | 5 | 7 | 8 | 14 |
| pred | 0 | a | d | d | d |

| QP | a | d | e | b | c |
|---|---|---|---|---|---|
| d | 0 | 5 | 7 | 8 | 7+6 |
| pred | 0 | a | d | d | e |

| QP | a | d | e | b | c |
|---|---|---|---|---|---|
| d | 0 | 5 | 7 | 8 | 8+1 |

| pred | 0 a d d | b |
|---|---|---|

```
Graf Dijkstra(Graf G, Varf s){
   int  poz;
   PQ   Q = PQ_New();
   Graf T = G_New ();
   Arc p;
   Varf v;
   // Estimare + Punere varfuri in coada prioritara
   for(v=primV(G); v!=dultimV(G); v=avansV(G,v))
      if(G_isV(G,v)){
         G_AddV(T, v);
         G_SetDist(T, v, INF);
         G_SetCol(T, v, alb);
         G_SetPred(T, v, 0);
         poz = PQ_Insert(Q, v, G_GetDist(T,v));
         G_SetPoz(T, v, poz);
      }
   G_SetDist(T, s, 0);
   poz=PQ_ChPr(Q, G_GetPoz(T,s), 0);
   G_SetPoz(T, s, poz);
   while(!PQ_Empty(Q)){
      int u = PQ_DelMin(Q);
      G_SetCol(T, u, gri);
      //relaxarea varfurilor adiacente
      for(p=primAinc(T,u); !dultimAinc(T,p,u); p=avansAinc(T,p,u)){
         Varf w = Opus(p, u);
         double c = A_GetCost(p);
         if(G_GetCol(w)==alb && G_GetDist(w) > G_GetDist(u)+c){
            G_SetDist(w, G_GetDist(T,u)+c);
            G_SetPred(w, u);
            poz = PQ_GetPoz(T, w);
            poz=PQ_ChPri(Q, poz, c);
            G_SetPoz(T, w, poz);
         }
      }
   }
   return T;
}
```

**Probleme propuse.**

**Traversări.**

1. Pentrul graful din Fig.1, marcați tipul arcelor la traversarea în adâncime, pornind din **A** şi reprezentați pădurea de acoperire în adâncime. Se consideră vârfurile în ordine lexicografică.

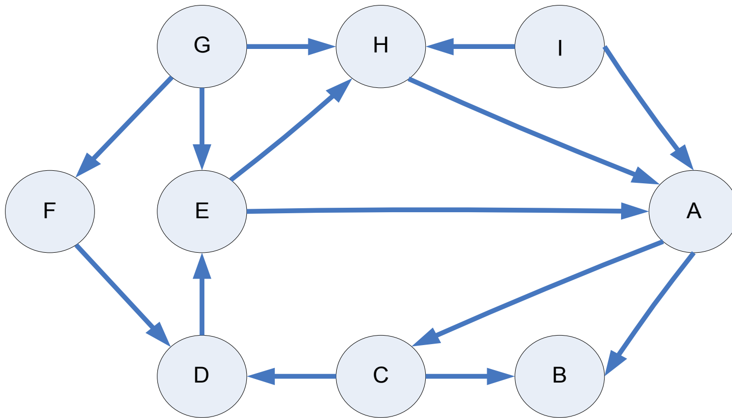



Fig. 1

2. Pentru graful dat mai jos, desenați arborii de acoperire în adâncime şi în lățime şi etichetați arcele în fiecare caz. Se pleacă din vârful **A** şi se consideră vârfurile în ordine lexicografică.

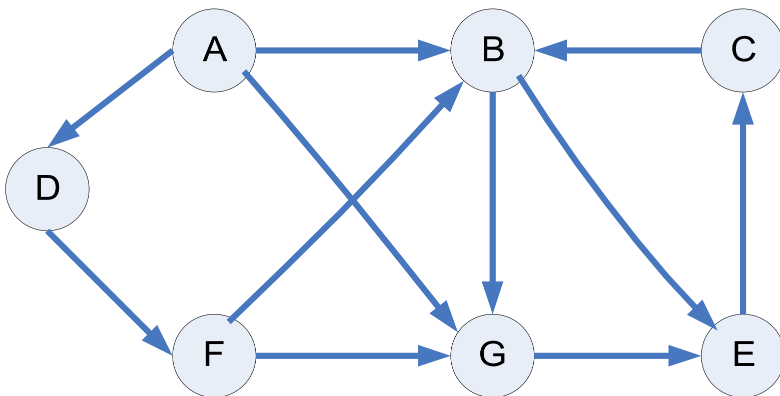



Fig. 2

3. Pentru graful de mai jos se cere să se efectueze o traversare în adâncime şi o traversare în lățime pornind din vârful A. Se vor eticheta muchiile în fiecare din cele două situații.

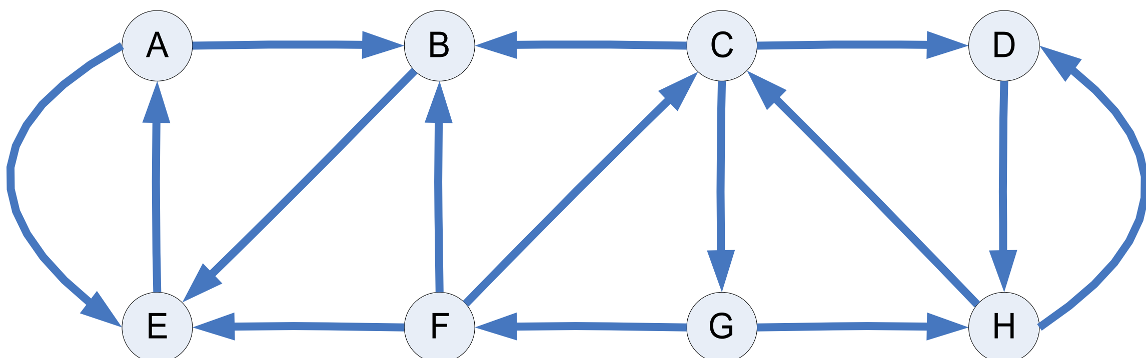



Fig. 3

4. Modificați funcția de traversare în adâncime astfel încât aceasta să să stabilească dacă un graf este sau nu aciclic. Funcția aresemnătura **`int Aciclic(Graf G);`** și să afişeze vârfurile din fiecare componentă conexă, în cursul procesului de traversare.

5. Un graf orientat este specificat prin perechi reprezentând arce, de forma:
**`et_vf_orig  et_vf_dest`**
Etichetele sunt şiruri de caractere. Ultima linie reprezintă vârful de start **x**.
Scrieți un program care efectuează o traversare în adâncime pornind din **`x`** şi marchează:

- momentele descoperirii şi terminării prelucrării fiecărui vârf
- tipul arcelor grafului (arbore, revenire, înaintare, traversare).Eticheta arcului va fi setată la tipul arcului grafului.

Programul va citi arcele, şi vârful de start, va crea graful, va marca vârfurile şi va eticheta arcele. Pentru afişarea tipului arcelor se vor crea 4 liste de arce. De exemplu pentru graful precizat prin:

```
A    B
B    D    se vor afişa rezultatele:
A    D    Arbore:        A-B   B-D
D    A    Revenire:  D-A
C    D    Înaintare:     A-D
C    A    Traversare:    C-A   C-B   C-D
C    B
A
```

6. Rețeaua de străzi a unui oras este reprezentată printr-un graf neorientat, având ca vârfuri - intersecții de străzi (piețe). Străzile se consideră de lungimi egale. Din piețele **`A`** şi **`B`** pleacă în acelaşi moment, doi pietoni, care se deplasează cu viteze egale în piețele destinații **`X`** şi **`Y`**. Ei cunosc oraşul şi merg spre destinații pe trasee minime. Afişați traseele parcurse. Stabiliți dacă traseele au trecut prin piațe comune, care sunt acestea şi dacă s-au întâlnit. Graful este dat prin lista muchiilor **`vârf - vârf`** şi este terminat prin 0-0.

7. Scrieți o funcție care verifică pentru un graf conex neorientat, dacă succesiunea vârfurilor la parcurgerea în adâncime şi lățime este aceeaşi, pornind dintr-un vârf dat.

8. Noul primar al oraşului Bucureşti a decis reîmpărțirea oraşului în sectoare concentrice, pornind dintr-o intersecție, considerată centru al oraşului. Astfel în sectorul 1 vor intra toate intersecțiile (şi străzile dintre ele) situate la distanța de cel mult **`p`** străzi de centru; sectorul 2 va cuprinde intersecțiile (şi străzile dintre ele) aflate la distanțe între **`p+1`** şi **`2*p`** străzi de centru, etc. Determinați numărul de sectoare, şi pentru fiecare sector – străzile pe care le conține. Străzile sunt considerate toate de aceeaşi lungime.
Datele, citite dintr-un fişier sunt şiruri de caractere, câte un şir pe o linie şi reprezintă:
- numărul de străzi şi **`p`**
- numele intersecției centru
- pentru fiecare stradă:
  - numele intersecției capăt 1 al străzii
  - numele intersecției capăt 2 al străzii

`- numele străzii`

**`9.`** **`n`** oraşe dintr-o țară sunt legate între ele prin autostrăzi, specificate sub forma: **`i j`** cu semnificația: oraşul **`i`** este legat direct cu oraşul **`j.`** Toate autostrăzile sunt cu taxă, care este aceeaşi, indiferent de lungimea autostrăzii. Stabiliți care este costul unui abonament de circulație pe autostrăzi, ştiind că acesta se ia de 100 ori numărul maxim de autostrăzi parcurse între două perechi de oraşe.

10. Într-un graf neorientat, să se determine toate vârfurile situate la distanta **`d`** (exprimată în număr de muchii) de un vârf **`s`**, şi toate traseele de la **`s`** la aceste vârfuri. Se vor defini structurile folosite şi funcțiile apelate. (Indicație: se face o traversare BFS).
Cele `n` piete din Veneția sunt legate printr-un sistem de canale. În piata San-Marco, o gondolă cu motor, cu rezervorul spart a poluat pietele situate la distanta `<= r` de San-Marco. Stabiliți numărul de piete

poluate. Pietele sunt identificate prin numerele 1 la `n`, iar canalele prin perechi de piete `(i,j)`. Graful este reprezentat prin liste de adiacențe.

11. Între **n** orase, identificate prin întregi de la **1** la **n** există comunicații directe, precizate prin perechi de forma **a-b**. Relația este comutativă şi tranzitivă (**a-b** => **b-a, a-b** şi **b-c** => **a-c**).
Stabiliți dacă în sistemul de comunicație există legaturi redundante (în plus) şi în caz afirmativ, afişați-le.

12. Harta rutieră a unei țări este precizată prin autostrăzile care leagă perechi de oraşe **a – b**. Afişați toate oraşele la care se poate ajunge dintr-un oraş **x**, folosind **n** autostrăzi.

13. Diametrul unui graf se defineşte ca **max d(u,v)** cu **u,v** ∈ **V**, în care **d(u,v)** este cel mai scurt drum între **u** şi **v**. Considerând distanța între două vârfuri egală cu numărul de muchii parcurse între ele, aflați diametrul unui graf neorientat.

14. Sistemul rutier al unei țări este format din autostrăzi care leagă perechi de oraşe. Pentru a folosi o autostradă, un automobilist trebuie să cumpere o rovinietă. Vasilică locuieşte în oraşul `a` şi dispune de `n` roviniete. Pornind din `a`, V. vrea să cunoască toate oraşele în care poate ajunge folosind o rovinietă, fără să treacă de două ori prin acelaşi oraş, , oraşele în care ajunge cu două roviniete, etc. Dacă există oraşe la care poate ajunge cu `n` roviniete, precizați toate traseele din `a` la aceste oraşe. Menționăm că se foloseşte întotdeauna traseul cel mai economic, adică dacă se poate ajunge în oraşul `b` folosind două sau trei roviniete se va folosi prima soluție. Nu foloseşte backtracking.
Se dau `a`, `n` şi autostrăzile, exprimate prin perechi de oraşe pe care le leagă. Autostrăzile sunt bidirecționale. Oraşele sunt identificate prin numere întregi, începând cu 0.

15. În sistemul hidrografic din România, pentru fiecare râu (identificat printr-un număr) se cunoaşte debitul la izvor.
Un râu poate avea mai mulți afluenți (dar nu se poate vărsa în mai multe râuri), astfel încât la vărsarea lui în alt râu (sau în mare) el va avea un debit la vărsare constituit din debitul lui la izvor, la care se adaugă debitele afluenților la vărsarea lor în acel râu.
Se dau: `n` – numărul râurilor, debitul fiecărui râu la izvor, şi perechile `(a, b)` cu semnificația „`b` este afluent a lui `a` (`b` se varsă în `a`)". Graful hidrografic va fi reprezentat prin liste de adiacențe. Scrieți un program care calculează şi afişează pentru fiecare râu debitul la vărsare.

16. Prin ştergerea unei muchii critice dintr-un graf neorientat conex, graful îşi pierde conexitatea. Un algoritm cu complexitate liniară pentru determinarea muchiilor critice face o traversare în adâncime a grafului, marcând nivelul fiecărui vârf în arborele de acoperire în adâncime. Acesta conține muchii de arbore şi muchii de revenire. O muchie critică este o muchie de arbore, care nu aparține nici unui ciclu din arbore. O muchie de arbore `(a,b)` nu este critică dacă din `b` se poate ajunge într-un vârf situat pe un nivel mai mic sau egal cu nivelul lui `a`. Definiți o funcție care verifică dacă o muchie este sau nu critică.
Dându-se un graf prin lista de muchii, creați reprezentarea lui prin liste de adiacențe şi determinați care sunt muchiile critice ale grafului.

## Conexitate

17. O peşteră are **n** încăperi, fiecare la o înălțime $h_i$.Configurația peşterii este precizată prin cele **m** culoare care leagă încăperile între ele. În încăperea **s** se află un izvor. Se dau **n, m, s**, şi cele **m** perechi (**i,j**) legate prin culoare de comunicație.. Care încăperi sunt inundate, care sunt acestea şi ce culoare sunt umezite.

18. Între persoanele dintr-o instituție, identificate prin nume, există relații de rudenie, precizate prin perechi de forma **ana ion** cu semnificația "**ana este rudă cu ion**". Datele se termină printr-o linie vidă..
- Stabiliți numărul persoanelor, ştiind că fiecare are cel puțin o rudă.
- Stabiliți numărul de clanuri existente şi dați componența acestora. Relația de rudenie este simetrică şi tranzitivă

19. Se citesc linii cu perechi de cuvinte înrudite, separate prin spații albe. Să se găsească grupurile de cuvinte înrudite şi să se afişeze, câte un grup pe o linie, separând cuvintele dintr-un grup, între ele prin liniuță (cratimă).

*Exemplu:*

| Intrări | Rezultate |
|---|---|
| **copac pom** | **alun-arbore-cires-copac-mar-par-pom** |
| **animal cal** | **animal-caine-cal-crocodil-leu-pisica** |
| **arbore alun** | **calm-serenitate** |
| **pom arbore** | |
| **serenitate calm** | |
| **mar par** | |
| **cires copac** | |
| **pom par** | |
| **caine pisica** | |
| **crocodil leu** | |
| **pisica leu** | |
| **leu animal** | |

20. Determinați componentele tari conexe şi graful condensat pentru graful:

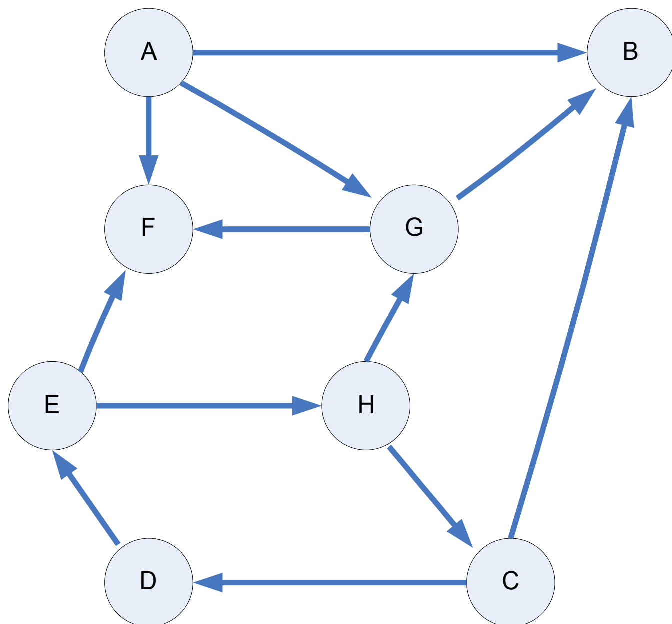


Fig. 4

21. Determinați componentele tari conexe şi grafurile condensate pentru graful din Fig. 2.:

## Sortare topologică

22. O sortare topologică a unui graf orientat aciclic reprezintă o enumerare de vârfuri, în care vârfurile **u** şi **v** apar în această ordine, dacă există un arc orientat de la u la v sau dacă u şi v sunt independente şi nu există un drum de la v la u.

Să se realizeze sortarea topologică a unui graf dat prin matrice de adiacențe, folosind următorul algoritm:

1. se găseşte un vârf u cu gradul de intrare 0 şi se afişează
2. se înlătură vârful şi muchiile adiacente şi se repetă pasul 1

Dacă rămân vârfuri, înseamnă că avem un ciclu.

23. Vasilică a pus `n` peşti într-un acvariu. Din nefericire peştii erau carnivori, între ei existând numai relaţii `a→b` cu semnificaţia "`a` mănâncă pe `b`". Relaţiile nu sunt simetrice. Un peşte mănâncă numai pe un altul mai mic decât el.
Evaluaţi dezastrul, adică stabiliţi ordinea în care trebuie să se mămânce peştii astfel încât în acvariu să rămână un număr minim de peşti şi care este acest număr.
Se dau: `n` – numărul iniţial de peşti şi relaţiile `a→b` (`a` mănâncă pe `b`) terminate prin `0 0` şi se cer: numărul rămas de peşti şi ordinea în care se mănâncă.

24. Între cursurile unei programe analitice, notate secvenţial cu 1,2,… există condiţionari de forma `a-b`, cu semnificaţia "cursul **a** trebuie să preceadă cursul **b**". Într-un semestru se vor plasa toate cursurile independente, care fac apel la cunoştinţe din cursurile din semestrele precedente.
Scrieţi un program care citeşte perechi `a - b`, terminate prin `0 - 0`, reprezentând condiţionări de cursuri şi determinaţi:
- numărul de cursuri
- numărul minim de semestre
- repartizarea cursurilor pe semestre 2.5 p

Exemplu: Pentru condiţionările:
`1-4,1-7,4-8,2-6,3-5,3-11,6-9,9-12,6-7,5-8,7-11,8-10,3-6,5-12`
Se obţin rezultatele: număr de cursuri: `n=12,` Număr condiţionări: `m=14`
Număr minim de semestre: `4`
sem.1 - cursuri `1,2,3`
sem.2 - cursuri `4,5,6`
sem.3 - cursuri `7,8,9`
sem.4 - cursuri `10,11,12`

25. Într-un dicţionar în care apar **n** noţiuni, există explicaţii de forma: **noţiune1 - noţiune2**, cu semnificaţia că noţiunea 1 este definită în funcţie de noţiunea 2. Noţiunile sunt reprezentate prin întregi între **1** şi **n**.
Să se stabilească dacă în definirea acestor noţiuni se comit tautologii.
În caz că nu există tautologii, să se stabilească în ce ordine trebuie date explicaţiile, astfel încât în definirea unei noţiuni să nu se folosească decât noţiuni deja definite.
Noţiunile primare (cele care nu trebuie definite) sunt specificate sub forma **n - 0**.

## Arbore de cost minim

**16. n** oraşe sunt date prin nume şi coordonate într-un sistem de referinţă cartezian. Proiectaţi un sistem de autostrăzi care să lege toate oraşele între ele cu cost minim. O autostradă leagă două oraşe şi nu au cotituri. Se va defini şi folosi algoritmul Prim, folosind cozi prioritare. Datele, introduse de la tastatură sunt: **n** şi **n** linii formate din tripleţi: **nume oraş**(şir de caractere), **x** şi **y** (întregi), separate prin spaţii liber

## Drumuri minime

**27. n** magazine dintr-un oraş sunt legate între ele prin străzi cu sens unic. Stabiliţi în care din ele pot fi amplasate posturi de poliţie, din care să se poată ajunge la toate magazinele.
Se dau: n şi n perechi (i, j) cu semnificaţia – de la magazinul I se poate ajunge la magazinul j printr-o stradă cu sens unic.

*Indicaţie*: Se construieşte matricea drumurilor între toate perechile de vârfuri.

**28. n** oraşe dintr-o ţară sunt legate între ele prin şosele, specificate prin triplete de forma: **o1 o2 d** cu semnificaţia: oraşul **o1** este legat direct cu oraşul **o2** printr-o şosea de lungime **d**. (**o1** şi **o2** sunt şiruri de caractere, iar **d** este un întreg). Nu toate oraşele sunt legate direct între ele, dar există comunicaţii între toate oraşele.
Stabiliţi care oraş poate fi ales capitală, ştiind că acesta satisface proprietatea că suma distanţelor drumurilor la celelalte oraşe este minimă.

*Indicaţie*: Pentru fiecare oraş se calculează suma distanţelor minime la celelalte oraşe, folosind algoritmul Dijkstra şi se alege cel pentru care suma distanţelor este minimă.

29. Diametrul unui graf ponderat neorientat se defineşte ca cel mai lung drum minim între două vârfuri ale grafului.
Graful este introdus prin lista muchiilor, specificate prin triplete **`etich_vârf etich_vârf pondere`**. , în care etichetele vârfurilor sunt şiruri de caractere, iar ponderea este un număr real. Datele se termină printr-un triplet cu pondere **`0`**. Determinaţi diametrul grafului şi scrieţi succesiunea vârfurilor pe care le cuprinde. Se va defini şi folosi algoritmul Dijkstra, implementat cu coadă prioritară, cu semnătura: **`Graf Dijkstra(Graf G, Varf s);`**

30. Pentru graful din Fig.5, determinaţi drumurile minime pornind din **`A`**

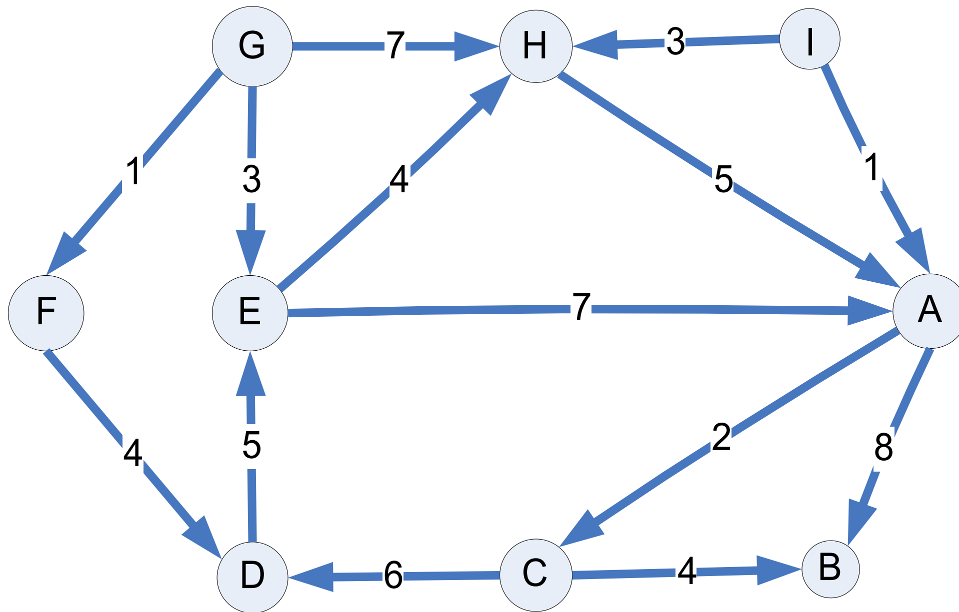



Fig.5

31. Aplicaţi simbolic algoritmul Dijkstra pentru a calcula drumurile minime pornind din **`F`**,pentru graful de mai jos:

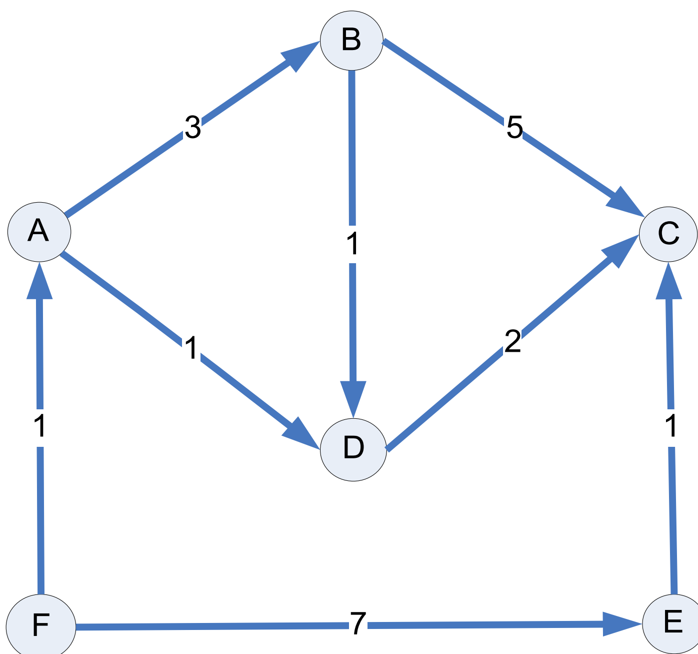



Fig. 6

Trasaţi evoluţia cozii prioritare.

32. Noul primar al oraşului Bucureşti a decis reîmpărțirea oraşului în sectoare concentrice, pornind dintr-o intersecție, considerată centru al oraşului. Astfel în sectorul 1 vor intra toate intersecțiile (şi străzile dintre ele) situate la distanța de cel mult **d** de centru; sectorul 2 va cuprinde intersecțiile (şi străzile dintre ele) aflate la distanțe între **d+1** şi **2*d** de centru, etc. Determinați numărul de sectoare, şi pentru fiecare sector – străzile pe care le conține.
Datele, citite dintr-un fişier sunt şiruri de caractere, câte un şir pe o linie şi reprezintă:
- numărul de străzi şi **d**
- numele intersecției centru
- pentru fiecare stradă:
    - numele intersecției capăt 1 al străzii
    - numele intersecției capăt 2 al străzii
    - lungimea străzii
    - numele străzii

Se va defini şi folosi o funcție **Graf Dijkstra(Graf G, Varf v)**, care calculează distanțele de la vîrful **v** la celelalte vârfuri din graf.

33. Se consideră **n** oraşe, legate prin **m** şosele, oraşele **i** şi **j** fiind legate printr-o şosea de lungime $L_{ij}$. Dintre cele **n** oraşe, **p** oraşe, precizate suplimentar reprezintă obiective turistice. Scrieți un program pentru o agenție turistică, cuprinzând toate traseele minimale între oraşe care nu sunt obiective turistice, şi care conțin cel puțin 3 obiective turistice..

*Indicație*: Matricea celor mai scurte drumuri **D** între perechile (**i,j**) de oraşe este inițializată cu matricea de adiacență a costurilor **L**. Se fac **n** iterații, pentru a include pe drumul $D_{ij}$ oraşele **1,2,...,n**. După iterația **k** , **D** conține cele mai scurte drumuri care pot folosi vârfurile intermediare **1, 2, ..., k**. Criteriul este ca prin includerea vârfului **k**, drumul între **i** şi **j** să se scurteze, adică:
$D_k$**{i][j] = min (**$D_{k-1}$**[i][j],** $D_{k-1}$**{i][k]+**$D_{k-1}$**[k][j]).**

Păstrarea vârfurilor de pe drumurile minime se face într-o matrice **P**. După creerea acestor matrice se inspectează toate traseele posibile, memorând într-o listă pe cele care satisfac condițiile impuse.

34. Într-o stațiune turistică punctele de interes sunt unite prin cărări cu sens unic. Harta precizează cotele, lungimea fiecarei cărări şi sensul acesteia.
Un taximetrist a primit o comandă pentru deplasarea între **A** şi **B,** date ca şiruri de caractere. Pentru a accepta comanda, el trebuie să stabilească dacă există un traseu realizabil, ştiind că maşina sa nu poate urca o pantă > **0.2**.
Dacă există traseul, va stabili suma pretinsă, ştiind că pentru 1 km ia **5** lei.
Se dau: **n** = numărul punctelor de interes, cotele acestora, şi cărările existente, precizate prin tripleți **i j d** de punctede interes între care există comunicație (**i** şi **j** –punctele de interes sunt şiruri de caractere, **d**- este distanța între ele). Datele se termină printr-un triplet cu **i=j**. Fiecărui punct de interes i se precizează şi o cotă h. Drumurile minime vor fi stabilite cu algoritmul lui Dijkstra, care va fi scris folosind numai primitivele din interfețele Graf şi Coadă Prioritară.
Panta între 2 puncte **i** şi **j** se calculează ca $(h_i-h_j)/d_{ij}$

35. În oraşul Sinaia există `n` obiective turistice; fiecare obiectiv `i` se află situat la cota $h_i$. Între unele din aceste obiective există trasee. Un traseu este precizat prin capetele traseului **i** şi **j** şi lungimea traseului $l_{ij}$. Datele se termină prin `0 0 0`.
Un biciclist bătrân doreşte să se plimbe între obiectivele **a** şi **b**, dar nefiind prea antrenat alege traseul minim, în care nu se depăşeşte panta `p`, la urcare sau coborâre.
Se ştie că panta între două obiective **i** şi **j** se calculează ca: $(h_j - h_i)/l_{ij}$.
Ajutați biciclistul să-şi alcătuiască traseul sau comunicați-i că nu-l poate parcurge.
*Indicație*: Drumurile minime vor fi stabilite cu algoritmul lui Dijkstra, care va fi scris folosind numai primitivele din interfețele Graf şi Coadă Prioritară.

**Grafuri bipartite.**

36. Într-un grup de femei există relații de antipatie, exprimate prin perechi `(a, b)`, cu semnificația „`a` antipatizează pe `b`". Relația este simetrică.
Un moderator (bărbat) decide separarea grupului în două subgrupuri, astfel încât în fiecare subgrup să nu existe două persoane care să se antipatizeze.
Scrieți un program care să decidă rapid (fără a folosi backtracking), dacă acest lucru este posibil, şi în caz afirmativ, să precizeze componența fiecărui grup. Pentru a nu da naştere la diferite interpretări, persoanele sunt identificate prin numere.

37. Într-un graf bipartit, mulțimea vârfurilor poate fi partiționată în alte două submulțimi, astfel încât două vârfuri adiacentenu vor aparține aceleaşi submulțimi de vârfuri. Scrieți o funcție care verifică dacă un graf este sau nu bipartit.

38. Într-un graf bipartit, mulțimea vârfurilor poate fi partiționată în alte două submulțimi, astfel încât două vârfuri adiacente nu vor aparține aceleaşi submulțimi de vârfuri (vârfurile pot fi colorate folosind numai două culori).
Scrieți un program, care pentru un graf dat prin muchiile sale (perechi de vîrfuri terminate cu 0 0) încearcă colorarea vârfurilor cu două culori (verifică dacă graful este sau nu bipartit). În caz de reuşită, se afişează două linii: pe prima apar vârfurile grafului colorate cu prima culoare, iar pe a doua linie, vârfurile colorate cu cea de a doua culoare. Se vor folosi numai primitivele date în interfața TAD Graf

**Diverse**

39. Între cele **`n`** lucrări care se efectuează pentru construirea unei clădiri există **`c`** condiționări de forma **`a`** – **`b`**, cu semnificația "**`a`** `precede` **`b`**".
Fiecărei lucrări i se asociază o durată maximă de execuție
Stabiliți numărul de lucrări din fiecare etapă şi care sunt acestea, precum şi numărul minim de etape necesar pentru ridicarea construcției, într-o etapă fiind plasate toate lucrările independente ce se pot efectua ținând seama de condiționări. Determinați timpul maxim de execuție al lucrării.

40. Între cele **`n`** lucrări care se efectuează pentru construirea unei clădiri există **`c`** condiționări de forma **`a`** – **`b`**, cu semnificația "**`a`** `precede` **`b`**".
Fiecărei lucrări i se asociază o durată maximă de execuție
Stabiliți numărul de lucrări din fiecare etapă şi care sunt acestea, precum şi numărul minim de etape necesar pentru ridicarea construcției, într-o etapă fiind plasate toate lucrările independente ce se pot efectua ținând seama de condiționări.
Determinați timpul maxim de execuție al lucrării.

41. Se consideră 2 grafuri: `G1 = (V, E1)` şi `G2 =(V, E2)`, repezentate prin Liste de Adiacențe. Se defineşte compunerea grafurilor: `G3 = G1 o G2 = (V, E3)`
`E3 = {(x,y):` există `z ∈ V, (x,z) ∈ E1` şi `(z,y) ∈ E2}`
Scrieți o funcție având ca parametri două grafuri `G1` şi `G2`, care le compune şi obține graful `G3`, dat tot ca parametru.

42. Se defineşte reuniunea a două grafuri `G1=(V,E1)` şi `G2=(V,E2) ca G3=(V, E1 ∪ E2).`
Scrieți o funcție având ca parametri cele două grafuri `G1` şi `G2` şi graful reuniune, care folosind funcțiile specifice Tipului de Date Abstract Graf obține graful reuniune.

43. Definiți o funcție care compară 2 grafuri reprezentate ambele prin Liste de Adiacențe, stabilind dacă sunt sau nu egale.

44. Scrieți o funcție care verifică dacă 2 grafuri orientate neponderate sunt sau nu egale.

45. Un ciclu eulerian conține toate muchiile grafului. Un graf conex este eulerian dacă gradele tuturor vârfurilor sunt numere pare. Pentru construirea unui ciclu eulerian se pleacă dintr-un vârf (1) şi se

construieşte un ciclu, scăzând gradele vârfurilor prin care trece. Se construieşte apoi un nou ciclu, concatenându-l cu primul, până când nu mai sunt vârfuri.
Pentru un graf dat prin matrice de adiacențe, se verifică dacă este eulerian şi în caz afirmativ se construieşte un ciclu eulerian.