Teoria grafurilor

Capitolul

5

- **Reprezentarea grafurilor**
- **Traversarea grafurilor**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

5.1. Reprezentarea grafurilor

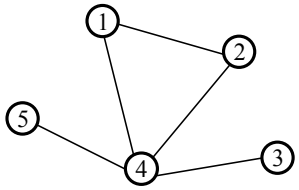
Există mai multe modalități de a reprezenta grafurile. Alegerea unei reprezentări se realizează în funcție de algoritmul folosit pentru rezolvarea unei probleme. Notăm cu $G = (X, U)$ un graf neorientat, având n noduri în mulțimea X (numerotate cu primele n numere naturale) şi cu m muchii în mulțimea U.

5.1.1. Matricea de adiacență

Matricea de adiacență este o matrice pătratică având n linii şi n coloane, câte o linie pentru fiecare nod din graf, în care elementele $A_{ij} = \begin{cases} 1 & \text{dacă } (i,j) \in U \\ 0 & \text{dacă } (i,j) \notin U \end{cases}$.

Având în vedere că într-un graf neorientat pentru o muchie $(i, j) \in U$ este adevărată relația $(i, j) = (j, i)$ rezultă că matricea de adiacență este simetrică față de diagonala principală.

Fie graful din figura alăturată. Pentru acest graf matricea de adiacență este vizualizată alăturat figurii:



$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Pentru o matrice de adiacență este nevoie de spațiu de memorie care nu depinde de numărul de muchii, ci doar de numărul de noduri. În unele probleme este folosită memorarea doar a elementelor situate deasupra diagonalei principale, având în vedere că matricea este simetrică şi pentru o valoare mare a lui n este folosită foarte multă memorie.

Muchia (x, y) există în graf, dacă elementul A_{xy} este egal cu 1, în caz contrar nu există astfel de muchie în graf. O altă proprietate a matricei de adiacență este că *suma tuturor elementelor matricei* este egală cu $2 \cdot m$, iar numărul elementelor de pe linia x este chiar *gradul vârfului x*.

Construirea matricei de adiacență se poate realiza direct dacă matricea se citeşte din fişierul de intrare, sau poate fi inițial completată cu elemente nule, urmând să se citească m perechi de numere corespunzătoare vârfurilor de la extremitățile muchiilor şi pentru fiecare pereche de numere x şi y se atribuie lui $A_{ij} \leftarrow 1$, respectiv $A_{ji} \leftarrow 1$.

Matricea de adiacență se poate folosi, de asemenea pentru a memora *grafurile orientate*. Diferența față de cele neorientate se datorează faptului că un arc (x, y) dintr-un graf orientat are următoarea proprietate (x, y) ≠ (y, x). Din acest motiv matricea de adiacență a unui graf orientat nu este simetrică, iar suma elementelor matricei este egală cu numărul arcelor din graf.

5.1.2. Lista vârfurilor adiacente

Lista vârfurilor adiacente este formată din n liste, unde a i-a listă conține vârfurile adiacente vârfului i. Memorarea se realizează într-o matrice, astfel încât linia i conține vârfurile adiacente cu vârful i. Pentru fiecare vârf i se memorează, de asemenea, numărul vârfurilor din listă. O altă variantă de utilizare a listei vârfurilor adiacente este aceea de a folosi listele liniare memorate static sau dinamic.

Pentru graful din figura dată anterior se formează următoarele liste de adiacență:

Număr ataşat vârfului	Liste de adiacență
1	2 4 5
2	1 4
3	4
4	1 2 3 5
5	1 4

Numărul vârfurilor adiacente ale unui vârf este chiar gradul acestuia şi, evident, suma lungimilor listelor este egală cu $2 \cdot m$.

Pentru a determina dacă o muchie aparține grafului, aceasta trebuie căutată în lista extremităților unuia dintre vârfuri, ceea ce implică un algoritm mai lung decât în cazul căutării într-o matrice de adiacență.

Construirea listelor de adiacență se poate realiza în momentul citirii datelor.

```
Subalgoritm Liste_de_adiacenţă:
  citeşte n                                  { citirea numărului de vârfuri }
  cât timp nu urmează marca de sfârşit de fişier execută:
    citeşte i,j                          { citirea extremităţilor a unei muchii }
    nvec[i] ← nvec[i] + 1          { creşte numărul vârfurilor adiacente lui i }
    nvec[j] ← nvec[j] + 1          { creşte numărul vârfurilor adiacente lui j }
    a[i,nvec[i]] ← j                     { memorăm vârful adiacent cu i }
    a[j,nvec[j]] ← i                     { memorăm vârful adiacent cu j }
sfârşit subalgoritm
```

De asemenea, listele de adiacenţă se pot utiliza la memorarea grafurilor orientate.

5.1.3. Lista extremităţilor muchiilor

Lista extremităţilor muchiilor este formată dintr-o matrice cu două coloane şi m linii în care pe o linie din matrice se află valorile celor două extremităţi ale unei muchii. Această matrice se memorează pe $2 \cdot m$ locaţii de memorie.

Pentru graful din figura din exemplu lista extremităţilor este:

1 2
1 4
1 5
2 4
3 4
4 5

Construirea acestei liste se realizează prin simpla citire a perechilor de numere. Această construcţie poate fi folosită şi la memorarea grafurilor orientate cu menţiunea că elementele din prima coloană sunt vârfurile iniţiale ale arcelor, iar cele din a doua coloană sunt vârfurile finale ale arcelor.

5.1.4. Matricea costurilor

Matricea costurilor este o matrice pătratică de ordin n, similară cu cea de adiacenţă. Diferenţa dintre ele este că matricea costurilor este utilizată atunci când se asociază grafului un cost, adică fiecărei muchii i se asociază un număr, reprezentând o lungime sau o cantitate, în funcţie de problemă.

Un element al matricei costurilor: $C_{ij} = \begin{cases} c & \text{dacă } (i,j) \in U \\ 0 & \text{în caz contrar} \end{cases}$, unde c este costul asociat muchiei (i, j).

Construirea matricei costurilor se realizează similar cu construirea matricei de adiacenţă. Matricea costurilor poate fi utilizată şi în cazul grafurilor orientate, dacă muchiilor li s-au asociat costuri.

5.1.5. Matricea de incidență

Matricea de incidență se mai numeşte *matrice vârfuri-arce* şi se utilizează în cazul grafurilor orientate. Această matrice are pentru fiecare vârf o linie şi pentru fiecare arc o coloană, deci matricea are *n* linii şi *m* coloane.

$$B_{ij} = \begin{cases} 1 & \text{dacă } (i,j) \in U \\ -1 & \text{dacă } (j,i) \in U \\ 0 & \text{dacă în rest} \end{cases}$$

Pe fiecare coloană a matricei de incidență se află exact două elemente nenule.

5.2. Traversarea grafurilor

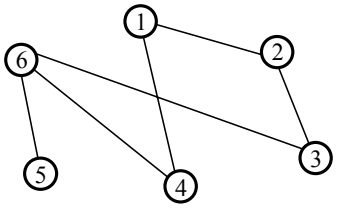
Traversarea grafurilor se referă la examinarea vârfurilor unui graf, astfel încât fiecare dintre ele să fie parcurs.

5.2.1. Traversarea în lățime

În traversarea în lățime (*Breadth First*) se porneşte de la un vârf şi se parcurg toate vârfurile adiacente lui, după care se parcurg vecinii acestora (care nu au fost parcurşi până la momentul respectiv) şi aşa mai departe, până se vizitează toate vârfurile grafului.

Această traversare determină cele mai scurte drumuri de la primul vârf considerat la celelalte şi este, de asemenea, folosită în algoritmul lui *Prim* pentru a determina arborele parțial de cost minim etc.

În parcurgerea în lățime se reține într-un şir valorile vârfurilor parcurse şi pentru fiecare dintre ele se adaugă la sfârşitul şirului vârfurile adiacente care nu au fost scrise încă. De exemplu, având graful din figura următoare şi pornind de la vârful 1 prin parcurgerea în lățime se obține următoarea succesiune de vizitare a vârfurilor: 1, 2, 4, 3, 6, 5.



Pentru început, pe prima poziție se reține primul vârf parcurs, şi anume 1. Apoi se adaugă toate nodurile adiancente cu el. Astfel şirul devine: 1, 2, 4. Urmează să se viziteze nodurile adiacente cu 2, care nu au fost scrise deja în şir – acesta devine: 1, 2, 4, 3. Se tratează apoi nodurile adiacente lui 4 şi se obține: 1, 2, 4, 3, 6. Apoi urmează vârfurile adiacente cu 3, dar vârful 2 şi vârful 6 sunt vizitate deja, deci nu vor fi scrise în şir. La sfârşit se parcurg vârfurile adiacente lui 6, adică doar 5 care este nevizitat şi cele adiacente lui 5 (care nu mai are vecini nevizitați). Se obține şirul final: 1, 2, 4, 3, 6, 5.

În algoritm am notat cu *c* şirul în care se vor reţine pe rând etichetele vârfurilor grafului, iar în *viz* ţinem evidenţa vizitărilor. Dacă valoarea lui viz_i este adevărat, înseamnă că nodul *i* a fost parcurs deja.

```
Subalgoritm BF:
  c[1] ← i { se determină vârfurile la care se poate ajunge pornind de la nodul i }
  k ← 2                                          { k este lungimea şirului c }
  z ← 1                                  { z parcurge şirul nodurilor memorate în c }
  cât timp z < k execută:                          { se iau toate vârfurile din şirul c }
    t ← c[z]                          { se caută vecinii nodului de pe poziţia z în şir }
    viz[t] ← adevărat          { se marchează faptul că acest nod a fost parcurs }
    pentru j=1,n execută:
                                        { dacă există noduri adiacente nevizitate încă}
      dacă (a[t,j] = 1) şi (nu viz[j]) atunci
        viz[j] ← adevărat                        { se marchează ca fiind vizitat }
        c[k] ← j                                          { şi se adaugă j în c }
        k ← k + 1                                { se măreşte lungimea lui c }
      sfârşit dacă
    sfârşit pentru
    z ← z + 1    { se trece la următorul vârf din c să i se caute vecinii nevizitaţi }
  sfârşit cât timp
sfârşit cât timp
```

5.2.2. Traversarea în adâncime

În cazul traversării în adâncime (*Depth First*) parcurgerea porneşte de la un vârf *a* şi se caută un vârf adiacent *b*, după care se caută un vecin a lui *b* care nu a fost parcurs şi aşa mai departe, până se parcurg toate vârfurile grafului. Dacă la un moment dat un vârf nu mai are vârfuri adiacente neparcurse se revine la vârful anterior şi se caută un alt vârf neparcurs.

Prin parcurgerea unui graf în adâncime se obţine o succesiune de vârfuri care se memorează într-un şir. Astfel, pentru graful din ultima figură succesiunea care se obţine pornind de la vârful 1 este: 1, 2, 3, 6, 5, 4.

Să parcurgem graful din figură. Pentru început pe prima poziţie se reţine primul vârf parcurs şi anume 1. Apoi se adaugă un nod adiacent cu el, în cazul de faţă 2, după care se caută un nod adiacent cu nodul 2 şi care nu a fost trecut deja în şir. Astfel şirul devine: 1, 2, 3. Urmează să se viziteze un nod adiacent cu 3 şi care nu a fost scris deja în şir – acesta devine: 1, 2, 3, 6. Se tratează apoi un nod adiacent cu 6 şi se obţine: 1, 2, 3, 6, 5. Apoi ar trebui parcurs un vârf adiacent cu 5, dar nu mai există un astfel de nod care să nu fi fost trecut deja în şir. În situaţia dată se caută un alt vârf adiacent cu 6 şi se găseşte vârful cu eticheta 4. Dacă nu mai sunt vârfuri adiacente cu un anumit

vârf, se caută vecini pentru vârfuri anterior memorate în şir. Astfel se obţine succesiunea de parcurgere a vârfurilor grafului din prima figură din acest capitol.

Pentru o astfel de parcurgere este recomandată memorarea grafului sub formă de liste ale vârfurilor adiacente. În cazul grafului din a doua figură din acest capitol listele de adiacenţă sunt:
(2, 4); (1, 3); (2, 6); (1, 6); (6); (3, 4, 5).

Parcurgerea în adâncime poate fi utilizată în determinarea componentelor conexe care apelează un subalgoritm recursiv pentru a putea parcurge toate vârfurile din graf.

```
Subalgoritm FD(i):
  viz[i] ← adevărat                    { se marchează vârful i ca fiind vizitat }
  pentru j=1,nvec_i execută:           { pentru toate vârfurile adiacente lui i }
    dacă nu viz[a[i,j]] atunci                  { dacă nu au fost vizitate }
      FD(a[i,j])                        { se caută vecinii vecinului lui i }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Această secvenţă se apelează atâta timp cât mai există vârfuri nevizitate.

5.3. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie rezolvate problemele din teoria grafurilor, vă sugerăm să încercaţi să implementaţi algoritmi pentru:

1. reprezentarea grafurilor prin matricea de adiacenţă;
2. reprezentarea grafurilor prin lista vârfurilor adiacente;
3. reprezentarea grafurilor prin lista extremităţilor muchiilor;
4. reprezentarea grafurilor prin matricea de incidenţă;
5. reprezentarea grafurilor prin matricea costurilor;
6. reprezentarea grafurilor prin matricea succesorilor vârfurilor;
7. reprezentarea grafurilor prin şirul succesorilor vârfurilor;
8. problema comis-voiajorului (Se consideră un graf neorientat complet cu *N* vârfuri. Fiecărei muchii i se atribuie un cost numit distanţă. Să se determine un drum hamiltonian de cost minim. Se va aplica metoda euristică greedy.)
9. colorarea grafurilor (Se consideră un graf neorientat având *N* vârfuri. Să se determine numărul minim de culori necesare pentru a colora vârfurile grafului, astfel încât oricare două vârfuri legate printr-o muchie să fie colorate diferit. Se va aplica metoda euristică greedy.)
10. stabilirea componentelor conexe ale unui graf (prin traversare în lăţime şi adâncime);
11. sortarea topologică;
12. determinarea unui ciclu eulerian.

5.4. Probleme propuse

5.4.1. Campanie electorală

În campania electorală un candidat doreşte să ţină discursuri în mai multe localităţi dintr-o anumită zonă a ţării. El doreşte să se întâlnească cu alegătorii din toate oraşele respective, astfel încât să nu treacă de două ori prin acelaşi oraş şi, în plus, să se întoarcă de unde a plecat.

Dându-se numărul localităţilor şi distanţele dintre ele, realizaţi planul călătoriei de lungime minimă a politicianului, ştiind că între oricare două oraşe există legătură directă.

Date de intrare

Pe prima linie a fişierului de intrare **`ORASE.IN`** este scris un număr întreg n, reprezentând numărul oraşelor care trebuie parcurse. Pe următoarele n linii se găsesc câte n numere pe fiecare, separate prin spaţii, reprezentând distanţele dintre oraşe.

Date de ieşire

Fişierul de ieşire **`ORASE.OUT`** va conţine pe prima linie un număr reprezentând lungimea celui mai scurt traseu, iar pe a doua linie va conţine $n + 1$ numere de ordine ale oraşelor în ordinea parcurgerii lor în traseu.

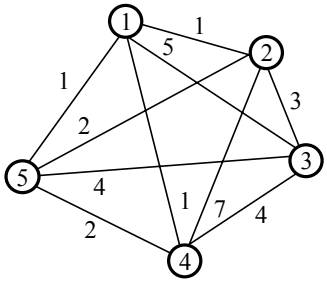
Restricţii şi precizări

- $1 \le n \le 100$;
- între oricare două oraşe există cel mult un drum;
- trebuie vizitate toate oraşele;
- traseul politicianului poate să înceapă în oricare oraş, cu condiţia să se termine în acelaşi oraş.

Exemplu

`ORASE.IN`
```
5
0 1 5 1 2
1 0 3 7 2
5 3 0 4 4
1 7 4 0 2
2 2 4 2 0
```

`ORASE.OUT`
```
11
2 1 4 5 3 2
```



5.4.2. Parcul colorat

Se deschide un nou parc de distracţii cu multe căsuţe în care pe copii îi aşteaptă tot felul de surprize. Pentru a oferi un ambient cât mai plăcut, organizatorii s-au gândit ca fiecare căsuţă să fie colorată, astfel încât toate căsuţele care „se văd" una din cealaltă să fie colorate diferit. În parc există unele perechi de căsuţe care nu „se văd" una din cealaltă, deoarece în parc se află o mulţime de arbori. Organizatorii îşi pun întrebarea câte culori trebuie achiziţionate, astfel încât să folosească un număr minim de culori.

Cunoscând numărul de căsuţe, precum şi perechile de căsuţe care se văd între ele, se cere să se determine numărul minim de culori necesare pentru a colora căsuţele, precum şi o colorare posibilă.

Date de intrare

Pe prima linie a fişierului de intrare **`PARC.IN`** se află un număr întreg *n*, reprezentând numărul căsuţelor. Pe următoarele linii se găsesc câte două numere *i* şi *j* separate prin spaţiu, reprezentând faptul că cele două căsuţe *i* şi *j* „se văd" una pe cealaltă şi, deci, trebuie colorate diferit.

Date de ieşire

Pe prima linie a fişierului de ieşire **`PARC.OUT`** se va scrie un număr natural, reprezentând numărul minim de culori necesare, iar a doua linie va conţine *n* numere naturale, despărţite prin câte un spaţiu, reprezentând numerele de ordine ale culorilor asociate fiecărei căsuţe în parte.

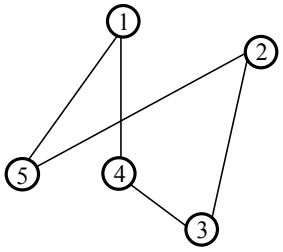
Restricţii şi precizări

- $1 \le n \le 100$.

Exemplu

`PARC.IN`	**`PARC.OUT`**

`5`
`1 5`
`1 4`
`2 5`
`2 3`
`4 3`
	`3`
	`1 1 2 3 2`



5.4.3. Cluburi maritime

De-a lungul coastei unui continent des vizitat de turişti există mai multe porturi pe lângă staţiuni. Pentru a câştiga câţi mai mulţi clienţi, porturile se asociază în cluburi maritime. Un club are reguli stricte de funcţionare, şi anume deţine şi acceptă doar curse ale navelor care aparţin porturilor care fac parte din acelaşi club maritim.

Un turist află care sunt porturile între care există curse directe ale navelor şi doreşte să stabilească cluburile maritime existente.

Să se scrie un algoritm care ajută turistul să determine numărul de cluburi şi componența lor.

Date de intrare

Pe prima linie a fişierului de intrare **CLUB.IN** se află un număr natural *n*, reprezentând numărul porturilor din zona respectivă. Pe următoarele linii se găsesc câte două numere *i* şi *j*, separate prin spațiu, reprezentând faptul că între porturile *i* şi *j* există cursă directă.

Date de ieşire

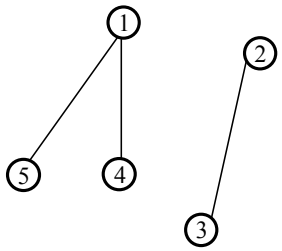
Pe prima linie a fişierului de ieşire **CLUB.OUT** se va scrie un număr natural *nc*, reprezentând numărul de cluburi existente, iar pe următoarele *nc* linii vor fi trecute numere naturale separate prin spațiu, reprezentând numerele de ordine ale porturilor care fac parte dintr-un acelaşi club.

Restricții şi precizări

- $1 \le n \le 100$.

Exemplu

CLUB.IN	CLUB.OUT
5	2
1 5	1 4 5
1 4	2 3
2 3	



5.4.4. Împărțirea fluturaşilor

În oraşul *X* se deschide un nou supermagazin. Patronii au decis să anunțe locuitorii oraşului despre eveniment cu ajutorul fluturaşilor şi au angajat în acest sens distribuitori.

Un astfel de distribuitor primeşte o listă de străzi şi harta acestora. Pentru a-şi termina treaba mai repede el îşi propune să-şi optimizeze deplasarea, astfel încât să nu treacă de două ori pe aceeaşi stradă, să nu parcurgă inutil alte străzi, să le parcurgă pe toate şi să se întoarcă de unde a plecat. Cu alte cuvinte el doreşte ca după ce a terminat de parcurs o stradă să continue cu o alta adiacentă.

Se consideră că fiecare stradă se află între două intersecții. O intersecție poate fi confluența uneia, a două, sau a mai multor străzi. De exemplu, dacă o stradă iese din

localitate, sau este un drum înfundat (adică nu are ieşire decât la un capăt), se consideră că acea stradă se termină cu o intersecție imaginară. Intersecțiile se numerotează cu primele numere naturale. O stradă se notează prin numerele de ordine ale celor două intersecții care o delimitează. Între două intersecții nu există decât o stradă care le leagă direct. Nu există nici o stradă care începe şi se sfârşeşte în aceeaşi intersecție.

Scrieți un algoritm care ajută distribuitorul să determine dacă există un astfel de traseu şi dacă există, să indice unul.

Date de intrare
Pe prima linie a fişierului de intrare **`STRAZI.IN`** se află două numere întregi *n* şi *m,* separate printr-un spațiu, reprezentând numărul intersecțiilor din oraş, respectiv numărul străzilor repartizate distribuitorului. Pe următoarele *m* linii se găsesc câte două numere *i* şi *j*, separate printr-un spațiu, reprezentând străzile, adică numerele de ordine ale celor două intersecții care delimitează o stradă.

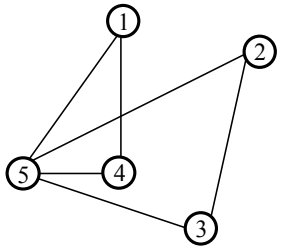
Date de ieşire
Fişierul de ieşire **`STRAZI.OUT`** va conține pe prima linie cuvântul `'DA'`, sau `'NU'`, exprimând răspunsul la întrebarea dacă există un traseu care trece pe toate străzile o singură dată. Pe următoarea linie vor fi trecute numerele de ordine ale intersecțiilor care delimitează câte o stradă prin care trece distribuitorul, separate prin câte un spațiu. Primul şi ultimul număr trebuie să coincidă deoarece distribuitorul trebuie să se întoarcă în punctul de plecare.

Restricții şi precizări
- $1 \le n \le 10$;
- parcurgerea unei străzi într-un sens implică distribuirea fluturaşilor la toate casele de pe strada respectivă (şi pe o parte şi pe alta a străzii) şi nu este necesară parcurgerea străzii în ambele sensuri.

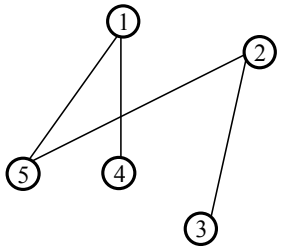
Exemple

`STRAZI.IN`
```
5 6
1 4
1 5
2 3
2 5
3 5
4 5
```

`STRAZI.OUT`
```
DA
1 4 5 2 3 5 1
```



STRAZI.IN	**STRAZI.OUT**
5 4	NU
1 4	
1 5	
2 3	
2 5	



5.4.5. Pregătirea mesei

Bubulina doreşte să facă o surpriză părinților ei şi să pregătească masa de prânz. S-a înarmat cu o carte de bucate, alimente şi condimente, un şorț mare şi e nedumerită de ordinea în care trebuie să realizeze operațiile pentru a pregăti o masă bună.

Ştie că anumite operații se realizează înaintea altora, de exemplu cartofii se spală înainte de a-i fierbe şi doar după ce au fiert se prepară.

Cunoscând numărul operațiilor pe care trebuie să le realizeze Bubulina şi prioritățile unor operații față de altele, realizați o ordonare liniară a operațiilor astfel încât masa să fie pregătită foarte bine.

Date de intrare

Pe prima linie a fişierului de intrare **PAPA.IN** se găseşte un număr n care reprezintă numărul de operații pe care Bubulina trebuie să le realizeze. Pe următoarele linii se găsesc câte două numere x şi y, separate printr-un spațiu, reprezentând faptul că operația x trebuie realizată înaintea operației y.

Date de ieşire

Fişierul de ieşire **PAPA.OUT** va conține pe o succesiune de n numere, separate prin câte un spațiu, reprezentând numerele de ordine ale operațiilor culinare în ordinea în care pot fi realizate, astfel încât masa să fie bine pregătită.

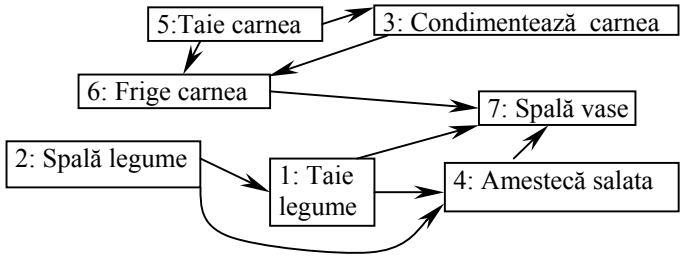
Restricții şi precizări

- $1 \le n \le 100$;
- Datele de intrare sunt corecte şi nu există operații care să determine o precedență circulară – de exemplu operația 1 să fie realizată înaintea operației 2, 2 înaintea lui 3 şi 3 înaintea lui 1.

Exemplu

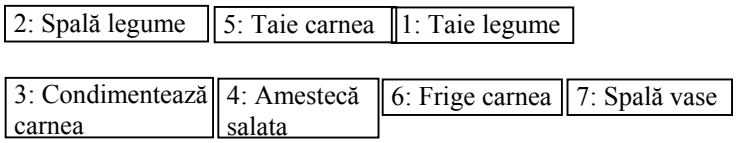
PAPA.IN

```
7
2 1
2 4
1 7
4 7
6 7
5 6
3 6
5 3
1 4
```



PAPA.OUT

```
2 5 1 3 4 6 7
```



5.4.6. Expoziția de roboți

În cadrul unei expoziții se organizează un sector pentru prezentarea realizărilor unor studenți pasionați de construirea roboților. Fiecare exponat are nevoie de alimentare la rețeaua de curent electric pentru a putea demonstra publicului aplicabilitatea sa. În sectorul aferent există o sursă de curent şi se cunosc anumite distanțe dintre exponate şi distanțele dintre unele exponate şi sursă.

Deoarece expoziția este temporară se doreşte ca rețeaua de curent să fie realizată astfel încât să coste cât mai puțin şi fiecare exponat să fie alimentat direct sau indirect de la sursă cu curent electric.

Cunoscând numărul de exponate şi anumite distanțe dintre ele, precum şi distanțe dintre exponate şi sursă, şi ştiind totodată că prețul rețelei este direct proporțional cu lungimea firelor electrice folosite, să se determine o rețea electrică de cost minim.

Date de intrare

Pe prima linie a fişierului **ROBOTI.IN** se află numărul natural *n*, reprezentând numărul exponatelor. Pe următoarele linii se află câte trei numere separate prin câte un spațiu. Primele două reprezintă numerele de ordine ale exponatelor sau 0 pentru sursa de curent electric, iar al treilea reprezintă distanța în metri dintre cele două exponate.

Date de ieşire

Pe prima linie a fişierului **ROBOTI.OUT** se va afişa lungimea totală a rețelei de curent electric. Pe următoarele linii se vor scrie câte două numere, reprezentând numerele de ordine a două exponate legate între ele, sau 0 (zero) in situația sursei de curent electric.

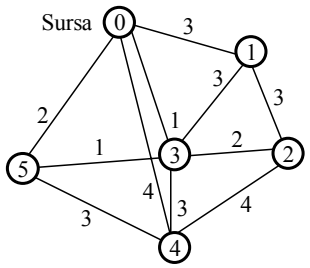
Restricţii şi precizări

- $1 \le n \le 100$;
- exponatele sigur pot fi conectate direct sau indirect la sursa de curent electric;
- un exponat poate fi legat la sursa de curent electric şi indirect, adică în serie cu un alt exponat.

Exemplu

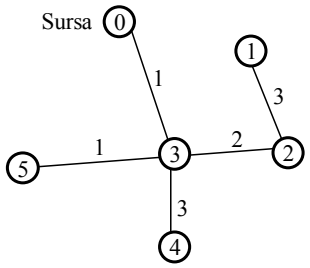
ROBOTI.IN

```
5
0 1 3
0 3 1
0 4 4
0 5 2
1 2 3
1 3 3
2 3 2
2 4 4
3 4 3
3 5 1
4 5 3
```



ROBOTI.OUT

```
10
0 3
1 2
2 3
3 4
3 5
```



5.4.7. Alianțe

Pe o planetă dintr-un sistem stelar se găsesc *n* triburi extraterestre. Între unele dintre ele există acorduri de pace. Un astfel de acord presupune că un trib nu poate ataca, sau nu se poate alia într-un atac împotriva altuia cu care are un acord. În plus un acord are această restricție şi asupra unui trib aliat şi asupra aliaților tribului aliat. Astfel triburile aliate direct sau indirect formează o comunitate. În schimb triburile din comunități diferite se pot ataca între ele.

Cunoscând numărul triburilor şi acordurile dintre ele, se cere să se afle numărul de comunități existente pe planetă. Se cere, de asemenea, să se determine acele perechi de triburi, care ar trebui să semneze acorduri de pace, pentru ca pe planetă să se formeze o singură comunitate.

Date de intrare
Pe prima linie a fişierului **STEA.IN** se află un număr natural *n*, reprezentând numărul de triburi. Pe următoarele linii se află câte două numere separate prin câte un spaţiu. Acestea reprezintă numerele de ordine ale triburilor între care există acorduri de pace.

Date de ieşire
Pe prima linie a fişierului de ieşire **STEA.OUT** se va afişa numărul *k* de comunităţi de pe planetă. Pe următoarele *k* linii se vor afla numerele de ordine ale triburilor care fac parte dintr-o comunitate. (Pe a *i*-a linie se va descrie a $(i-1)$-a comunitate.

Dacă pe planetă există mai mult de o comunitate, pe linia următoare se va afişa mesajul `'Acorduri necesare:'` şi pe următoarele linii se vor afişa perechi de numere de ordine ale triburilor care, prin semnarea unui acord vor determina unirea a cel puţin două comunităţi, astfel încât, în final, să fie unite toate comunităţile.

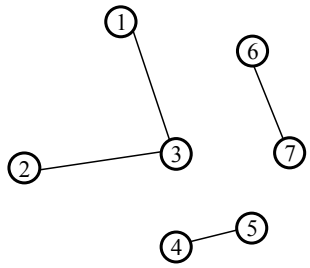
Dacă există o singură comunitate nu se mai scrie nimic în fişier.

Restricţii şi precizări
- $1 \le n \le 100$;
- Datele de intrare sunt corecte şi sunt date toate acordurile o singură dată.

Exemplu

STEA.IN
```
7
1 3
2 3
4 5
6 7
```

STEA.OUT
```
3
1 2 3
4 5
6 7
Acorduri necesare:
1 4
1 6
```



5.4.8. Pregătirea balului

Elevii clasei a X-a doresc să organizeze *Balul Primăverii* şi au format un nucleu de organizare care a decis să realizeze diferite surprize participanţilor, să orneze sala de bal şi să pregătească toate celelalte detalii. Ei doresc să repartizeze lucrările astfel încât să termine pregătirile în cel mai scurt timp. Se ştie că operaţiile trebuie să fie realizate într-o anumită succesiune. De exemplu invitaţiile nu pot fi împărţite până când nu au fost tipărite, sau ghirlandele nu pot fi întinse până nu au fost confecţionate.

Cunoscând operaţiile care trebuie efectuate, durata fiecărei astfel de operaţii şi succesiunile lor, determinaţi care sunt operaţiile care nu trebuie întârziate pentru ca balul să poată începe la timp.

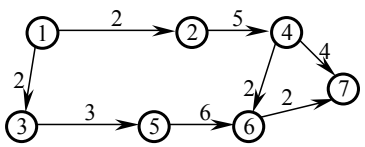
Date de intrare
Pe prima linie a fişierului **BAL.IN** se află un număr natural *n*, reprezentând numărul evenimentelor care se vor realiza. Pe următoarele linii se află triplete de numere *x y d* care reprezintă precedenţe între evenimente, adică operaţia *y* nu poate fi executată înainte ca operaţia *x* să se termine şi *d* durata activităţii dintre aceste două evenimente.

Date de ieşire
Pe prima linie a fişierului de ieşire **BAL.OUT** se va afişa durata totală a pregătirilor. Pe următoarea linie se vor afla perechi de numere de ordine, reprezentând evenimentele care nu pot întârzia, deoarece astfel s-ar întârzia întreaga pregătire a balului.

Restricţii şi precizări
- $1 \le n \le 100$;
- Datele de intrare sunt corecte.

Exemplu

BAL.IN	BAL.OUT
7	13
1 2 2	1 3
2 4 5	3 5
4 7 4	5 6
4 6 2	6 7
5 6 6	
1 3 2	
3 5 3	
6 7 2	



5.4.9. Pe drumuri de munte

Elevii clasei a X-a se află în expediţie într-o zonă montană. Ei doresc să ajungă la o anumită peşteră pentru a o vizita. Au harta zonei şi nu sunt decişi pe care drum să pornească ţinând cont că doresc să se bucure pe traseu de priveliştile zonei.

Cunoscând punctele de atracţie turistică din zonă şi traseele marcate între aceste puncte, determinaţi toate traseele posibile pe care se poate ajunge la peşteră pornind de la cabană, urmând ca ei să aleagă drumul pe care îl vor urma.

Date de intrare
Pe prima linie a fişierului **TRASEU.IN** se află un număr natural *n*, reprezentând numărul punctelor de atracţie legate prin trasee marcate. Pe cea de-a doua linie se găsesc două numere separate prin spaţiu, primul fiind numărul de ordine al cabanei, iar al doilea numărul de ordine al peşterii. Pe următoarele linii se află câte două numere separate prin câte un spaţiu, reprezentând puncte turistice din zonă legate direct printr-un drum marcat.

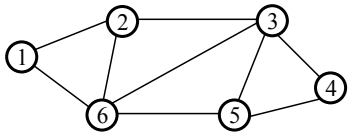
Date de ieşire
Pe fiecare linie a fişierului de ieşire **TRASEU.OUT** se va afişa câte un traseu. Un traseu începe la cabană şi se sfârşeşte la peşteră. Numerele de ordine ale punctelor dintr-un traseu sunt separate prin câte un spaţiu. Dacă nu există nici un traseu, în fişierul e ieşire se va scrie `'Nu exista traseu.'`.

Restricţii şi precizări
- $1 \le n \le 10$;
- Datele de intrare sunt corecte şi sunt date toate traseele marcate o singură dată.

Exemplu

TRASEU.IN
```
6
1 5
1 2
1 6
2 3
2 6
3 4
3 5
3 6
4 5
5 6
```

TRASEU.OUT
```
1 2 3 4 5
1 2 3 5
1 2 3 6 5
1 2 6 3 4 5
1 2 6 3 5
1 2 6 5
1 6 2 3 4 5
1 6 2 3 5
1 6 3 4 5
1 6 3 5
1 6 5
```



5.4.10. Mesaj telefonic

Dintr-un grup de colegi unii îşi cunosc numerele de telefon şi pot schimba mesaje. Din păcate nu oricare doi colegi îşi cunosc numerele de telefon. La un moment dat Ionel doreşte să îl anunţe pe Gigel despre noul film care rulează în oraş. Dacă Ionel nu cunoaşte numărul de telefon al lui Gigel, va anunţa un alt coleg al cărui număr îl cunoaşte, acesta pe altul, până când mesajul ajunge la Gigel.

Cunoscând numărul de colegi, precum şi perechile de colegi care pot comunica direct, determinaţi succesiunea de lungime minimă de telefoane care vor fi folosite pentru ca Gigel să fie anunţat despre film.

Date de intrare
Pe prima linie a fişierului **MESAJ.IN** se află un număr natural *n*, reprezentând numărul total de prieteni. Pe cea de-a doua linie se găsesc numerele corespunzătoare lui Ionel, respectiv al lui Gigel. Pe următoarele linii se află câte două numere separate prin câte un spaţiu. Acestea reprezintă numerele de ordine ale copiilor care îşi cunosc reciproc numerele de telefon şi pot comunica direct.

Date de ieşire
Pe prima linie a fişierului de ieşire **MESAJ.OUT** se va afişa numărul *k* de telefoane care se vor folosi, astfel încât Gigel să afle mesajul printr-un număr minim de intermediari. Pe următoarea linie se vor afişa numerele de ordine ale colegilor care trebuie să telefoneze, în ordinea în care, la rândul lor, primesc mesajul, primul fiind numărul de ordine al lui Ionel, iar ultimul al lui Gigel. Dacă nu există soluție, în fişierul de ieşire se va scrie mesajul `'Nu se poate transmite mesajul.'`.

Restricții şi precizări
- $1 \le n \le 100$;
- dacă există mai multe posibilități de a comunica mesajul, se va afişa una dintre ele;
- datele de intrare sunt corecte.

Exemplu

MESAJ.IN	**MESAJ.OUT**
6 1 4 1 2 1 6 2 6 3 6 3 4 5 4	3 1 6 3 4 *Explicație* 1 telefonează lui 6; 6 telefonează lui 3; 3 îl anunță pe 4.

5.4.11. Distanțe rutiere

O companie de transport persoane cu autobuzul a introdus în rutele sale o destinație nouă şi vrea să stabilească prețul călătoriei. Prețul este direct proporțional cu distanța parcursă între două localități pe cel mai scurt drum posibil.

Cunoscând numărul de destinații pentru care există transporturi şi distanțele dintre localitățile care sunt legate de câte un drum direct pe care există transport, determinați cel mai scurt drum între localitatea din care porneşte autobuzul şi localitatea nou introdusă.

Date de intrare
Pe prima linie a fişierului **AUTO.IN** se află un număr natural *n*, reprezentând numărul de localități în care autobuzul are stații. Pe linia a doua se află două numere reprezentând numerele de ordine ale localităților între care se cere determinarea drumului de lungime minimă. Pe următoarele linii se află câte trei numere separate prin câte un spațiu. Acestea reprezintă numerele de ordine ale localităților direct legate printr-un drum, iar al treilea reprezintă distanța dintre cele două localități.

Date de ieşire
Pe prima linie a fişierului de ieşire **AUTO.OUT** se va afişa distanţa minimă dintre cele două localităţi. Pe următoarea linie se vor afla numerele de ordine ale localităţilor prin care se va călători între cele două localităţi date în fişierul de intrare.

Restricţii şi precizări
- $1 \le n \le 100$;
- datele de intrare sunt corecte.

Exemplu

AUTO.IN
```
5
1 3
1 2 20
1 4 7
2 4 3
2 3 5
4 3 10
4 5 3
3 5 6
```

AUTO.OUT
```
15
1 4 2 3
```

5.5. Soluţiile problemelor propuse

5.5.1. Campanie electorală

Se observă că putem asocia hărţii oraşelor un graf neorientat cu n vârfuri, reprezentând oraşele. Drumurile dintre acestea sunt muchiile grafului. Între fiecare două oraşe există drum direct, deci graful este *complet*. În plus se asociază muchiilor grafului un cost strict pozitiv, reprezentând lungimea drumului dintre cele două oraşe. Graful (harta) va fi reprezentat în memorie cu ajutorul *matricei costurilor*, dată în fişierul de intrare.

Rezolvarea problemei revine la a găsi în graful asociat un ciclu hamiltonian de lungime minimă. Reamintim că un ciclu hamiltonian trece prin toate vârfurile grafului o singură dată.

Dacă se va folosi o rezolvare bazată pe metoda *backtracking*, se pot obţine toate ciclurile grafului şi dintre ele se poate determina ciclul de lungime minimă. Acest algoritm însă are nevoie de un timp de rezolvare de ordin exponenţial, deoarece într-un graf complet cu n vârfuri există $(n-1)!/2$ cicluri hamiltoniene.

Pentru a rezolva problema într-un timp rezonabil se poate folosi un algoritm bazat pe *metoda greedy* (metoda optimului local), dar care nu va furniza pentru orice date de intrare rezultat optim. Conform principiului metodei *greedy*, în fiecare moment vom alege oraşul cel mai apropiat de cel în care se află candidatul.

Vom nota soluția corectă ca fiind o succesiune de noduri (oraşe) ale grafului parcurse într-un traseu de lungime minimă [c_1, c_2, …, c_n, c_1]. Pentru a determina o astfel de soluție, să considerăm că la un moment dat al construirii ciclului sunt determinate primele k oraşe în succesiunea de parcurgere a lor ([c_1, c_2, …, c_k]) şi trebuie să determinăm al (k + 1)-lea nod. Acesta se va alege dintre vârfurile adiacente lui c_k, vârfuri care nu au fost deja alese între primele k. Dintre nodurile având această proprietate vom alege pe acela care este cel mai „aproape" de nodul c_k. Deci îl alegem pe $x \in \{1, 2, \ldots, n\}$, astfel încât $x \notin \{c_1, c_2, \ldots, c_k\}$ şi (c_k, x) are lungime minimă.

Exemplu

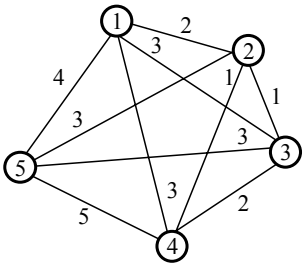


Figura 1

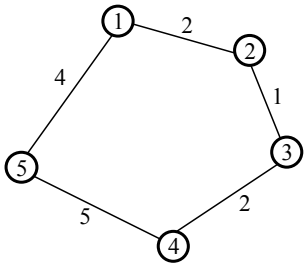


Figura 2

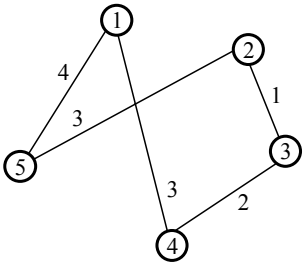


Figura 3

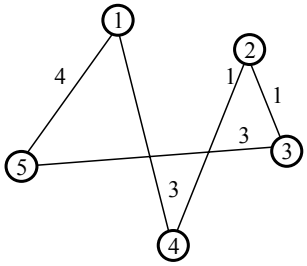


Figura 4

În graful din exemplu avem n = 5. Dacă primul oraş din traseu este 1, atunci conform principiului enunțat, vom alege muchia de lungime minimă dintre cele incidente cu vârful 1. Aceasta este (1, 2) de lungime 2. Apoi căutăm muchia având cel mai mic cost (lungime de drum) printre muchiile incidente cu oraşul (vârful) 2. Alegem muchia (2, 3) având costul 1. Muchia având cel mai mic cost printre cele incidente cu vârful 3 este (3, 1), dar oraşul 1 a fost deja vizitat. Astfel continuăm drumul prin nodul 4, apoi 5, după care ne întoarcem în oraşul 1.

Din păcate acest traseu nu este de lungime minimă. Având în vedere că prin fiecare oraş trebuie să trecem exact o singură dată, putem privi ciclul căutat ca pe un cerc care poate (temporar) să înceapă din oricare alt nod. Astfel, dacă se schimbă oraşul de pornire şi se aplică aceeaşi strategie *greedy*, obținem soluții mai bune, după cum se poate vedea în figura 3, respectiv figura 4, unde s-au obținut lungimile 13 respectiv 12 ale ciclurilor hamiltoniene.

```
Subalgoritm CicluHam(costmin,trmin):
  costmin ← ∞
  pentru i=1,n execută:
                { se determină câte un traseu pentru fiecare oraş ca punct de plecare }
    cost ← tur(i)
              { se calculează costul minim pentru traseul care pleacă din oraşul i }
    dacă cost < costmin atunci
      costmin ← cost                          { se reţine traseul de cost minim }
      trmin ← traseu_i
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

În secvenţa de mai sus am folosit notaţiile:
- *costmin* reprezintă valoarea costului traseului de lungime minimă;
- *cost* este costul unui traseu;
- *trmin* este traseul de lungime minimă.

Subalgoritmul `tur(i,co)` (apelat în subalgoritmul `CicluHam(costmin)`) determină costul şi traseul care porneşte din oraşul *i*. Pentru a şti în fiecare clipă dacă un oraş a fost vizitat sau nu, se va folosi un vector de valori booleene *viz*. Dacă oraşul *i* a fost vizitat, viz_i este *adevărat* în caz contrar este *fals*.

```
Subalgoritm tur(i,co):
  co ← 0                                  { iniţial costul traseului este zero }
  pentru k=1,n excută:
    viz[k] ← fals                  { la început nici un oraş nu a fost vizitat }
  sfârşit pentru
  viz[i] ← adevărat                                { se vizitează oraşul i }
  tr[1] ←i                                  { în traseu primul oraş este i }
  pentru t=1,n-1 execută:
    y ← tr[t]
    min ← ∞
                        { se caută oraşul vecin nevizitat, aflat la distanţă minimă }
    pentru j=1,n execută:
                      { se caută oraşul cel mai apropiat care nu a fost încă vizitat }
      dacă  (nu viz[j]) şi (cost[y,j] < min) atunci
        min ← cost[y,j]
        nod ← j
      sfârşit dacă
    sfârşit pentru
```

```
                                { oraşul care se află la distanţă minimă este marcat ca vizitat }
    viz[nod] ← adevărat
    tr[t+1] ← nod                                  { şi este trecut în traseu }
    co ← co + min                    { se calculează costul în acest moment }
  sfârşit pentru
  tr[n+1] ← i                        { călătorul se întoarce de unde a plecat }
  co ← co + cost[tr[n],i]             { se calculează costul total al traseului }
sfârşit subalgoritm
```

Ordinul de mărime a timpului de execuţie este polinomial în cazul aplicării acestui algoritm, dar reamintim faptul că nu întotdeauna furnizează rezultatul optim.

5.5.2. Parcul colorat

Se observă că se poate asocia hărţii căsuţelor un graf neorientat cu n vârfuri. Căsuţele sunt reprezentate prin vârfurile grafului şi faptul că ele „se văd" se reprezintă prin muchiile grafului. Două căsuţe care „se văd" sunt legate printr-o muchie. Problema revine astfel la a asocia vârfurilor grafului culori (coduri de culoare) astfel încât două noduri adiacente să fie colorate diferit.

Pentru problema de faţă reprezentarea cu ajutorul matricei de adiacenţă a grafului este bine venită, astfel se poate verifica uşor dacă două căsuţe „se văd" sau nu.

Construirea matricei de adiacenţă se realizează în paralel cu citirea datelor. Matricea de adiacenţă iniţial conţine doar elemente nule. Cu fiecare citire a câte două numere i şi j atât a_{ij} cât şi a_{ji} devin 1, deoarece, dacă din i se vede j, atunci şi din j „se vede" i.

Se poate observa că dacă graful este complet, sunt necesare n culori, iar dacă graful conţine doar vârfuri izolate se poate folosi o singură culoare. Un caz particular îl reprezintă *graful planar* (cel care admite o reprezentare grafică astfel încât muchiile sale nu se intersectează), în cazul cărora teorema celor patru culori afirmă că orice graf planar poate fi colorat doar cu patru culori.

Dacă se va folosi o rezolvare bazată pe metoda *backtracking*, se pot obţine toate posibilităţile de colorare a căsuţelor şi apoi se poate alege varianta cu număr de culori minim. Acest algoritm este bun pentru valori mici ale lui n, însă, deoarece are nevoie de un timp de rezolvare de ordin exponenţial, pentru valori mari ale lui n este inacceptabil.

Pentru a rezolva problema se va folosi un algoritm bazat pe metoda *greedy*, astfel încât vârfurile se colorează pe rând cu cel mai mic cod de culoare posibil. Acest algoritm nu furnizează soluţia optimă pentru orice date de intrare, dar îl prezentăm, deoarece, faţă de rezolvarea cu metoda backtracking are un timp de execuţie incomparabil mai bun.

Prima căsuţă se colorează întotdeauna cu culoarea având codul 1. Culoarea cu care se colorează căsuţa i o vom nota cu cul_i.

Să presupunem că avem colorate primele $k - 1$ căsuțe cu cul_1, cul_2, …, cul_{k-1}, în care intervin *nrcul* culori şi se pune problema colorării căsuței k. Aceasta va fi colorată cu culoarea c, unde $c \in \{1, 2, \ldots, nrcul\}$, dacă pentru $\forall\, j \in \{1, 2, \ldots, k - 1\}$ având proprietatea că dacă $a_{kj} = 1$, atunci $cul_j \neq c$, iar c este valoarea cea mai mică având această proprietate. Dacă nu există o astfel de culoare c, atunci căsuța va fi colorată cu culoarea *nrcul* + 1.

```
Subalgoritm Colorare(n,a,culmin,cul):
  c ← 1                          { prima căsuță se colorează cu culoarea 1 }
  pentru k=2,n execută:              { se colorează şi celelalte n – 1 căsuțe }
    c ← 1                          { se încearcă colorarea cu prima culoare }
    repetă                                 { se repetă încercarea de a colora }
      ok ← adevărat            { presupunem că se poate colora cu culoarea c }
      pentru j=1,k-1 execută:
                { dacă o căsuță care „se vede" a fost colorată cu aceeaşi culoare }
        dacă (a[j,k] ≠ 0) şi (cul[j] = c) atunci
          ok ← fals
        sfârşit dacă
      sfârşit pentru
      dacă nu ok atunci          { verificăm dacă culoarea c este bună sau nu }
        c ← c + 1                { dacă nu, se încearcă culoarea următoare }
      sfârşit dacă
    până când ok
    cul[k] ← c                          { se colorează căsuța k cu culoarea c }
    dacă cul[k] > culmin atunci
      culmin ← cul[k]                { se memorează codul culorilor folosite }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

5.5.3. Cluburi maritime

Vom asocia hărții porturilor un graf neorientat cu n vârfuri, în care vârfurile reprezintă porturi. Legăturile directe între porturi vor fi muchiile grafului. Memorăm graful cu ajutorul matricei de adiacență pe care o construim odată cu citirea datelor. Matricea de adiacență inițial conține doar elemente nule. Corespunzător fiecărei perechi de numere i şi j, a_{ij} şi a_{ji} devin egale cu 1.

Observăm că dacă graful este conex, toate porturile fac parte din acelaşi club maritim, iar dacă graful conține doar vârfuri izolate, fiecare port este parte din clubul său exclusivist şi probabil că turiştii folosesc alte mijloace de transport.

Problema revine la a determina componentele conexe ale grafului. O componentă conexă este un subgraf conex maximal.

Pentru a determina componentele conexe ale grafului se pot utiliza două metode.

I. Parcurgerea în lățime.

Folosind această metodă se vor asocia porturilor numere care reprezintă clubul din care fac parte.

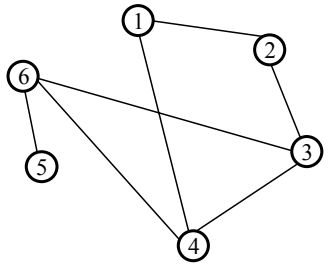
Prin parcurgerea în lățime se obține o listă de vârfuri care sunt legate direct sau indirect de un anumit nod al grafului.

Pentru început se porneşte de la un vârf (port) oarecare, de exemplu 1, şi se determină toate vârfurile la care se poate ajunge de la el. Toate aceste noduri fac parte din componenta conexă numărul 1.

Apoi, se determină dacă mai există vârfuri care nu au fost parcurse. Dacă există astfel de vârfuri se repetă o parcurgere ca cea descrisă mai sus pentru unul dintre aceste noduri ca făcând parte dintr-o altă componentă conexă. Aceşti ultimi paşi se repetă cât timp există vârfuri neparcurse.

```
Subalgoritm ComponenteConexe(n,a,ncc,club)
  ncc ← 0
  pentru j=1,n execută:
    viz[j] ← fals { la început nici un port nu a fost adăugat în vreo componentă }
  sfârşit pentru
  i ← 1
  repetă
    ncc ← ncc + 1                                  { o nouă componentă conexă }
    club[i] ← ncc                  { portul i face parte din această componentă }
    c[1] ← i
                        { determinăm toți vecinii lui i şi îi trecem în c şi memorăm }
                                                   { componenta conexă curentă }
    Parcurgere(i,c,club)
    i ← 0                        { verificăm dacă mai există noduri neparcurse }
    pentru j=1,n execută:
      dacă nu viz[j] atunci                { dacă mai există vârfuri nevizitate }
        i ← j                 { se reține unul dintre ele în variabila i urmând să se }
                                          { reia construcția pentru i cu o nouă valoare }
      sfârşit dacă
    sfârşit pentru
  până când i=0          { repetiția se opreşte când nu există vârfuri neparcurse }
sfârşit subalgoritm
```

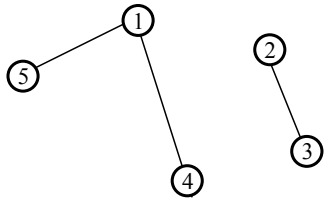
Parcurgerea în lățime se realizează reținând într-un şir valorile vârfurilor parcurse şi pentru fiecare dintre ele adăugând la sfârşitul şirului vârfurile adiacente care nu au fost trecute deja.



De exemplu, având graful din figura alăturată şi pornind de la vârful 1, prin parcurgerea în lățime se obține succesiunea de vizitare a vârfurilor grafului: 1, 2, 4, 3, 6, 5.

Vom reține în şirul *c* etichetele vârfurilor grafului. Pe prima poziție se reține primul vârf parcurs şi anume 1. Apoi se adaugă toate nodurile adiacente cu el. Astfel şirul devine: 1, 2, 4. Urmează să se viziteze nodurile adiacente cu 2, care nu au fost trecute deja în şir. Şirul devine 1, 2, 4, 3. Se tratează apoi nodurile adiacente lui 4 şi se obține: 1, 2, 4, 3, 6. Urmează vârfurile adiacente cu 3, dar vârful 2 şi vârful 6 sunt deja vizitate, deci nu vor fi trecute în şir. La sfârşit se parcurg vârfurile adiacente lui 6 şi nevizitate încă, adică vârful 5 şi cele adiacente lui 5 (care nu mai are vecini nevizitați). Se obține şirul final: 1, 2, 4, 3, 6, 5.

```
Subalgoritm Parcurgere(i,c,club):
  k ← 1                                  { k este lungimea curentă a şirului c }
  z ← 1                                  { cu z parcurgem şirul nodurilor memorate în c }
  cât timp z ≤ k execută:                { se iau toate vârfurile din şirul c }
    t ← c[z]                             { se caută vecinii nodului de pe poziția z în şir }
    viz[t] ← adevărat                    { se marchează faptul că acest nod a fost parcurs }
    pentru j=1,n execută:
                                         { dacă există noduri adiacente şi neparcurse }
      dacă (a[t,j] = 1) şi (nu viz[j]) atunci
        viz[j] ← adevărat
        k ← k + 1                        { se măreşte lungimea lui c }
        c[k] ← j                         { se adaugă în c şi vârful j }
        club[j]← ncc                     { se adaugă la componenta conexă ncc }
      sfârşit dacă
    sfârşit pentru
    z ← z + 1     { trecem la următorul vârf din c şi îi căutăm vecinii nevizitați }
  sfârşit cât timp
sfârşit subalgoritm
```



În graful din figura precedentă, toate vârfurile au făcut parte din aceeaşi componentă conexă. Dacă este vorba însă de un graf neconex ca în figura alăturată, afişarea componentelor conexe este ceva mai dificilă.

Pentru graful din această figură şirul *club* va conține valorile 1, 2, 2, 1, 1.

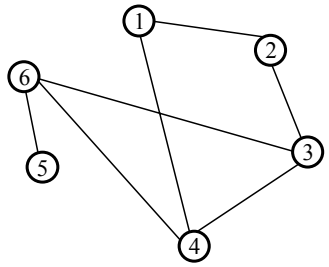
Afişarea vârfurilor pe componente conexe se poate realiza astfel:

```
Subalgoritm Scrie(club,n):
  pentru i=1,ncc execută:                { pentru fiecare componentă conexă }
    pentru j=1,n execută:                  { se verifică toate vârfurile }
      dacă club[j] = i atunci          { dacă face parte din a i-a componentă }
        scrie j                        { se afişează eticheta vârfului respectiv }
      sfârşit dacă
    sfârşit pentru
    trecem la linie nouă
  sfârşit pentru
sfârşit subalgoritm
```

II. Parcurgerea în adâncime

Prin parcurgerea în adâncime a unui graf se obține tot o succesiune de vârfuri memorate într-un şir.

Astfel pentru graful din figura alăturată succesiunea de vârfuri va fi 1, 2, 3, 6, 5, 4.



În şirul *c* se reține etichetele vârfurilor grafului. Pe prima poziție se reține primul vârf parcurs şi anume 1. Apoi se adaugă un nod adiacent cu el, în cazul de față 2, după care se caută un nod adiacent cu 2 şi care nu a fost trecut deja în şir. Astfel şirul devine: 1, 2, 3. Urmează să se viziteze nodul adiacent cu 3 şi care nu a fost trecute deja în şir. Şirul devine 1, 2, 3, 6. Se introduce în şir nodul 5 (adiacente cu 6). Apoi ar trebui parcurs un vârf adiacent cu 5, dar nu mai există un astfel de nod care să nu fi fost trecut în şir. În situația dată se caută un alt vârf adiacent cu 6 şi se obține vârful cu eticheta 4. Dacă nu mai sunt vârfuri adiacente cu un anumit vârf se caută vecini pentru vârfuri anterior memorate în şir.

Pentru o astfel de parcurgere este recomandată memorarea grafului sub formă de liste ale vârfurilor adiacente. În cazul grafului din prima figură listele de adiacență sunt: (2, 4); (1, 3); (2, 6); (1, 6); (6); (3, 4, 5).

Construirea listelor vârfurilor adiacente se realizează concomitent cu citirea datelor:

```
Subalgoritm ListaVârfurilorAdiacente(n,a,nvec):
  citeşte n
  cât timp nu urmează marca de sfârşit de fişier execută:
    citeşte i,j
    nvec_i ← nvec_i + 1
    nvec_j ← nvec_j + 1
    a[i,nvec_i] ← j
    a[j,nvec_j] ← i
  sfârşit cât timp
sfârşit subalgoritm
```

Determinarea componentelor conexe poate folosi o structură recursivă pentru a putea parcurge elegant toate vârfurile grafului.

```
Subalgoritm Compcon(i,c):
  club[i] ← c                    { vârful i face parte din componenta conexă c }
  viz[i] ← adevărat              { se marchează vârful i ca fiind vizitat }
  pentru j=1,nvec_i execută:     { pentru toate vârfurile adiacente lui i }
    dacă nu viz[a[i,j]] atunci   { dacă nu au fost vizitate }
      Compcon(a[i,j],c)          { căutăm vecinii vecinului lui i }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Această secvență se apelează din programul principal atâta timp cât mai există vârfuri nevizitate.

```
Algoritm Club:
  pentru i=1,n execută:
    viz[i] ← fals                { la început nici un nod nu a fost vizitat }
    nvec[i] ← 0                  { înaintea citirii toate nodurile au 0 vecini }
  sfârşit pentru
  citire(n,a)                    { subalgoritm de citire }
  ncc ← 0

  pentru i=1,n execută:
    dacă nu viz[i] atunci        { dacă există un vârf nevizitat }
      ncc ← ncc+1                { trecem la următoarea componentă conexă }
      Compcon(i,ncc)             { determinăm elementele care fac parte din componentă }
    sfârşit dacă
  sfârşit pentru
  scrie(ncc,cc)                  { subalgoritm de afişare }
sfârşit algoritm
```

Afişarea componentelor conexe se realizează ca în programul prezentat în cazul parcurgerii în lățime, având acelaşi tip de memorare a rezultatului.

5.5.4. Împărțirea fluturaşilor

Se observă că se poate asocia hărții străzilor un graf neorientat cu *n* vârfuri şi *m* muchii, unde vârfurilor li se asociază intersecțiile şi muchiilor străzile.

Memorarea grafului se realizează cu ajutorul matricei de adiacență. Construirea matricei de adiacență se realizează odată cu citirea datelor. Astfel matricea de adiacență inițial conține doar elemente nule. Cu fiecare citire a câte două numere *i* şi *j* se ada-

ugă matricei câte două elemente având valorile egale cu 1. *i* şi *j* reprezintă numerele de ordine a două intersecţii între care există stradă.

Distribuitorul trebuie să parcurgă toate străzile care i-au fost repartizate şi dacă se poate în mod optim. Prin optim înţelegem că el nu doreşte să treacă de două ori pe aceeaşi stradă, de asemenea, nu va parcurge inutil alte străzi, dar trebuie să parcurgă toate străzile care i s-au repartizat şi trebuie să se întoarcă de unde a plecat.

Problema revine la a determina dacă graful este eulerian şi dacă este, la a determina un ciclu eulerian al grafului.

Un graf este eulerian dacă este conex şi dacă are gradul vârfurilor par.

Pentru început trebuie să răspundem la întrebarea dacă poate fi găsit un traseu cu proprietăţile cerute (toate străzile să fie parcurse o singură dată şi să se revină la locul de plecare). Pentru această verificare vom folosi o funcţie în care la început presupunem că graful asociat străzilor este eulerian. Pentru orice condiţie neîndeplinită (conex sau grade pare) valoarea de adevăr a funcţiei de verificare devine falsă.

Gradul unui vârf se calculează simplu, dacă graful este memorat cu ajutorul matricei de adiacenţă, prin însumarea elementelor de pe linia corespunzătoare vârfului.

```
Subalgoritm verificare(n,m,a):
  verificare ← adevărat                { presupunem că graful este eulerian }
  pentru i=1,n execută:
    s ← 0
    pentru j=1,n execută:                    { calculul gradului vârfului i }
      s ← s + a[i,j]
      dacă s este număr impar atunci
        verificare ← fals                      { graful nu este eulerian }
        ieşire forţată din subalgoritm
      altfel
        gr[i] ← s                        { se memorează gradele vârfurilor }
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
...
```

Graful este conex dacă pentru oricare două vârfuri ale sale există un lanţ care le leagă. Astfel, pornind de la un vârf oarecare din graf şi parcurgând graful (de exemplu în lăţime) ar trebui să putem ajunge în toate vârfurile sale. Dacă nu se ajunge într-un vârf, atunci graful sigur nu este conex, iar dacă se ajunge în toate, graful este conex.

Pentru a parcurge graful în lăţime vom păstra într-un şir *dr* vârfurile care se parcurg. La început trecem în şir un vârf, de exemplu vârful 1, apoi îi trecem pe toţi vecinii săi (vârfurile adiacente cu el), apoi ajungem la vecinii vecinilor şi aşa mai departe. În consecinţă, dacă subalgoritmul nu s-a întrerupt din cauza că graful nu avea suma gradelor vârfurilor număr par, urmează să verificăm dacă acesta este conex:

```
...
  pentru i=1,n execută:
    viz[i] ← 0                          { la început toate vârfurile sunt nevizitate }
  sfârşit pentru
  k ← 1                          { k este numărul de vârfuri vizitate, iar p este numărul }
  p ← 1                                { de ordine al vârfului căruia i se caută vecinii }
  dr[1] ← 1                                                { se porneşte din vârful 1 }
  viz[1] ← 1                        { se marchează faptul că vârful 1 a fost vizitat }
  cât timp p ≤ k execută: { câtă vreme mai sunt vârfuri care au fost vizitate }
    pentru j=1,n execută:                                    { li se caută vecinii }
      dacă (a[dr[p],j] = 1) şi (viz[j] = 0) atunci
        k ← k+1                  { dacă aceşti vecini nu au fost vizitați, se vizitează }
        dr[k] ← j
        viz[j] ← 1                          { şi se reține faptul că au fost parcurse }
      sfârşit dacă
    sfârşit pentru
    p ← p + 1                    { se trece la următorul vârf dintre cele vizitate deja }
  sfârşit cât timp
  dacă k < n atunci
    verif ← fals       { dacă nu s-au vizitat toate vârfurile, graful nu este conex }
  sfârşit dacă
sfârşit subalgoritm
```

Subalgoritmul descris determină răspunsul la prima întrebare din problemă.

Cea de-a doua cerință a fost aceea de a determina un traseu pentru distribuitorul de fluturaşi. Această cerință revine la a determina un ciclu eulerian. Se va determina un ciclu eulerian prin construirea acestuia.

Se porneşte dintr-un vârf al grafului şi se încearcă construirea unui ciclu $C1$. Construirea este posibilă datorită faptului că gradul fiecărui vârf este par. Astfel dacă un vârf are gradul $2 \cdot k$, un ciclu care trece prin acest nod parcurge două dintre muchii şi rămân alte $2 \cdot (k - 1)$ muchii care trebuie parcurse. În construirea ciclului $C1$ ajungându-se într-un nod, de acolo se poate continua traseul spre un alt nod printr-o muchie neparcursă.

Dacă ciclul $C1$ este de lungime maximă, atunci el cuprinde toate muchiile grafului, deci este ciclu eulerian. Dacă nu este de lungime maximă, atunci există muchii neparcurse şi în plus, cel puțin una dintre ele este adiacentă cu un vârf x din ciclul determinat $C1$. Pornind pe această muchie se ajunge în alte noduri care la rândul lor au gradul par şi deci se poate continua drumul. În plus, acest nod x mai are cel puțin o muchie incidentă cu el şi nepacursă, deci în x se va închide un alt ciclu $C2$. Aceste două cicluri pot fi concatenate într-unul singur. Astfel se poate construi orice alt ciclu în graf, până când prin concatenări succesive se obține un ciclu eulerian. Programul care rezolvă problema ține seama de această construcție.

```
Subalgoritm construire(n,m,a,ce):
  ce[1] ← 1          { se construieşte ciclul C1 în graf pornind de la vârful 1 }
  k ← 1
  repetă
    pentru i=1,n execută:
      dacă a[ce[k],i] = 1 atunci              { se caută o stradă în graf }
        a[ce[k],i] ← 0      { se marchează în matricea de adiacenţă faptul }
        a[i,ce[k]] ← 0                  { că această stradă a fost parcursă }
                                  { se scade gradul vârfurilor incidente străzii }
        gr[ce[k]] ← gr[ce[k]] - 1
        gr[i] ← gr[i] -1
        k ← k + 1
        ce[k] ← i          { se adaugă vârful găsit în şirul vârfurilor ciclului }
                                                  { eulerian care se construieşte }
        ieşire forţată din pentru
      sfârşit dacă
    sfârşit pentru
                    { se repetă găsirea unui nou vârf până când ciclul este complet }
  până când ce[k] = ce[1]

  cât timp k < m+1 execută:          { dacă nu au fost parcurse toate străzile }
    pentru i=1,k execută: { se trece la construirea ciclului C2 care porneşte }
      dacă gr[ce[i]] > 0 atunci  { dintr-un vârf v al cărui grad este nenul }
        v ← i
        ce1[1] ← ce[i]
        k1 ← 1
        ieşire forţată din pentru
      sfârşit dacă
    sfârşit pentru
    se repetă construirea unui ciclu C2 analog cu construirea lui C1
                                    { ciclul nou găsit este concatenat cu primul }
    pentru i=k,v+1,-1 execută:
      ce[i+k1-1] ← ce[i]
    sfârşit pentru
    pentru i=2,k1 execută:
      ce[v+i-1] ← ce1[i]
    sfârşit pentru
    k ← k + k1 - 1
  sfârşit cât timp
sfârşit subalgoritm
```

5.5.5. Pregătirea mesei

Din enunțul problemei rezultă clar că Bubulina trebuie să execute anumite operații înaintea altora, astfel încât preparatele pe care le realizează să fie bine pregătite.

Ordinea de executare a operațiilor este impusă astfel încât o anume operație nu poate să fie realizată înaintea ei însăşi. Ținând cont de aceste considerente datele problemei se pot reprezenta cu ajutorul unui graf orientat fără cicluri. Proprietatea de aciclicitate impune ca activitățile gospodinei să nu fie posibil să se realizeze înaintea lor însele.

Operațiile culinare le vom asocia vârfurilor grafului, iar precedențele lor le vom nota cu ajutorul arcelor (ca în figura din enunț).

Ordonarea operațiilor trebuie realizată astfel încât vârfurile grafului să fie ordonate de-a lungul unei linii imaginare în aşa fel încât pentru orice arc (x, y) al grafului, vârful x să se afle înaintea lui y pe această axă. O astfel de ordonare se numeşte *ordonare topologică* şi poate fi realizată doar într-un graf aciclic.

Rezolvarea problemei se realizează căutând acele vârfuri care nu au predecesor, adică acțiunile care pot fi realizate fără a fi nevoie de alte acțiuni preliminare – de exemplu, „spală legumele" sau „taie carnea" care nu au evenimente anterioare. Aceste vârfuri se şterg din listă şi se trec „pe linia imaginară". În continuare, dintre celelalte vârfuri se aleg cele care nu mai au predecesor în lista de vârfuri.

Pentru a realiza algoritmul şi a regăsi uşor elementele care nu au predecesor în listă la un moment dat, vom folosi n liste liniare şi fiecare va conține succesorii unui vârf. În plus, pentru fiecare vârf, se va memora numărul predecesorilor proprii şi de fiecare dată, când se adaugă un vârf pe „axă", toți succesorii săi vor avea cu un predecesor mai puțin.

În subalgoritm *p* conține şirul listelor, şirul *np* reține lungimile fiecărei astfel de liste. Lista ordonată o construim în *l*.

```
Subalgoritm Ordonare(n,np,l,p):
  pentru i=1,n execută:
    viz[i] ← fals
  sfârşit pentru
  k ← 0
  cât timp k < n execută:          { cât timp mai sunt elemente neordonate }
    pentru i=1,n execută: { caută elementele fără predecesori şi neordonate }
      dacă (np_i = 0) şi (nu viz_i) atunci
        k ← k + 1                                  { le adăugăm listei ordonate }
        l_k ← i
        viz_i ← adevărat                        { marcăm faptul că s-au ordonat }
        pentru j=1,n execută:                    { îl şterg pe i din fiecare listă }
          q ← 1
```

```
          cât timp (q ≤ np[j]) şi (p[j][q] ≠ i) execută:
            q ← q + 1
          sfârşit cât timp
          dacă q ≤ np[j] atunci          { dacă l-am găsit, îl „şstergem" }
            pentru z=q,np[j]-1 execută:
              p[j][z] ← p[j][z+1]                  { suprascriindu-l }
            sfârşit pentru
            np[j] ← np[j] - 1                      { scade lungimea listei }
          sfârşit dacă
        sfârşit pentru
      sfârşit dacă
    sfârşit pentru
  sfârşit cât timp
sfârşit subalgoritm.
```

5.5.6. Expoziția de roboți

Roboții sunt aşezați într-o sală de expoziție şi trebuie legați la sursa de curent electric direct sau indirect. Reprezentăm datele cu un graf căruia îi asociem o funcție de cost care va fi distanța între cele două exponate sau distanța între un exponat şi sursa de curent electric. Rezolvarea problemei revine la a lega exponatele şi sursa între ele astfel încât să existe un lanț între sursa de curent electric şi fiecare robot.

Graful trebuie să fie conex deoarece fiecare exponat trebuie să fie legat de sursa de curent. Într-un graf conex între oricare două vârfuri există un lanț. Astfel fie x şi y două vârfuri din graf unde există lanțul [0, v_{l1}, …, x] care leagă nodul 0 (sursa de curent electric) de nodul x şi lanțul [0, v_{k1}, …, y] care leagă vârful 0 de vârful y. Se poate deduce că prin concatenarea celor două lanțuri, x şi y sunt legate prin lanțul [x, …, v_{l1}, 0 , v_{k1}, …, y].

Problema cere determinarea modului de conectare a exponatelor cu cost minim. Ştim că un arbore este un graf conex şi minimal cu această proprietate – adică dacă se elimină orice muchie el nu mai este conex. Astfel problema revine la a determina un arbore care are costul total minim.

Determinarea unui arbore parțial de cost minim se poate realiza folosind doi algoritmi: algoritmul lui *Kruskal* şi algoritmul lui *Prim*. În cele ce urmează prezentăm aceşti doi algoritmi.

I. Algoritmul lui Kruskal

Pornim de la graful în care considerăm că toate vârfurile au gradul 0 (zero). Pe parcursul algoritmului vom selecta câte o muchie pe care o adăugăm grafului până când acesta devine conex. În procesul de selectare luăm în considerare muchiile pe rând, ordonate crescător după cost, şi le selectăm dacă prin adăugarea lor nu se formează ciclu. Strategia de bază în algoritmul lui *Kruskal* este de tip *greedy*.

Vom asocia fiecărui nod un număr de ordine care reprezintă numărul componentei conexe din care face parte. Inițial fiecare nod *k* face parte din componenta conexă având numărul de ordine *k*. În momentul în care dorim să stabilim dacă o muchie poate fi selectată sau nu, vom verifica dacă extremitățile lor fac parte din componente conexe diferite sau nu. În caz afirmativ muchia nu poate fi selectată, deoarece astfel s-ar forma un ciclu. Dacă extremitățile aparțin unor componente conexe diferite, prin selectarea muchiei, aceştia se unesc. Reamintim că un arbore are exact $n - 1$ muchii, deci algoritmul se opreşte când s-au selectat $n - 1$ muchii.

Memorăm graful prin lista muchiilor, şi pentru fiecare muchie reținem extremitățile sale (*i* şi *j*) şi costul său (*c*).

```
Subalgoritm ArborePartialMinimKruskal(n,a,ct):
  much ← 0
  pentru i=0,n execută
    cc[i] ← i            { fiecare nod este într-o componentă conexă separată }
  sfârşit pentru
  k ← 1
  cât timp much < n - 1 execută:    { cât timp nu am selectat n – 1 muchii }
    dacă cc[a[k].i]≤cc[a[k].j] atunci  { dacă muchia nu formează ciclu }
      a[k].d ← adevărat                                    { o selectăm }
      dacă cc[a[k].i] < cc[a[k].j] atunci
        min ← cc[a[k].i]
        max ← cc[a[k].j]
      altfel
        min ← cc[a[k].i]
        max ← cc[a[k].j]
      sfârşit dacă
      pentru i=1,n execută:          { unim cele două componente conexe }
        dacă cc[i] = max atunci
          cc[i] ← min
        sfârşit dacă
      sfârşit pentru
      much ← much + 1                 { numărul muchiilor selectate creşte }
      ct ← ct+a[k].c                        { calculăm costul total parțial }
    sfârşit dacă
    k ← k + 1                        { avansăm în şirul muchiilor ordonate }
  sfârşit cât timp
sfârşit subalgoritm
```

II. Algoritmul lui Prim

Acest algoritm, asemănător celui gândit de *Kruskal* foloseşte metoda *greedy*. Pornim cu nodul 1 şi la fiecare pas alegem muchia de cost minim dintre muchiile adiacente

vârfului în care suntem. Deoarece vârfurile neselectate sunt considerate vârfuri izolate, în acest algoritm nu este necesar să verificăm dacă muchia care urmează să fie adăugată formează ciclu sau nu.

De data aceasta reprezentarea avantajoasă este prin matricea costurilor, în care corespunzător muchiilor inexistente valoarea va fi, nu 0, ci *infinit*, astfel facilitând selectarea muchiei de cost minim. Această inițializare o realizăm anterior citirii matricei costurilor.

```
Subalgoritm ArborePartialMinimPrim():
  ct ← 0
  pentru i=1,n execută:
    sel[i] ← fals                              { încă nu am selectat nici un vârf }
  sel[1] ← adevărat                            { selectăm primul vârf }
  e1[1] ← 1                                    { extremitatea 1 a muchiei selectate }
  pentru i=1,n execută:
    d[i] ← a[1,i]                              { distanța dintre vârful 1 şi i }
    e[i] ← 1                                   { vârful de plecare este 1 }
  sfârşit pentru
  k ← 1
  pentru i=1,n-1 execută:                      { trebuie să selectăm n – 1 muchii }
    min ← inf
    pentru j=2,n execută:
          { dacă vârful j nu este selectat şi distanța la el este mai mică decât min }
      dacă nu sel[j] şi (d[j] < min) atunci
        min ← d[j]                             { căutăm distanța minimă }
        care ← j                 { extremitatea de sosire a muchiei de cost minim }
      sfârşit dacă
    sfârşit pentru
    sel[care] ← adevărat                       { selectam vârful care }
    e1[i] ← e[care]                            { şi muchia corespunzătoare }
    e2[i] ← care
    ct ← ct + min                { adunăm costul muchiei selectate la costul total }
                                   { actualizăm şirul distanțelor minime }
    pentru j=2,n execută:
      dacă nu sel[j] şi (d[j]>a[care,j]) atunci
                        { dacă găsim distanța mai scurtă între care şi un vârf }
        d[j] ← a[care,j]
        e[j] ← care                { o schimbăm şi reținem vârful de pornire }
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

5.5.7. Alianțe

Din nou, reprezentarea datelor sub forma unui graf neorientat este binevenită, astfel triburile le asociem vârfurilor, iar acordurile dintre triburi, muchiilor grafului.

Dacă între două triburi există un acord, înseamnă că cele două vârfuri corespunzătoare fac parte din aceeaşi componentă conexă. Deci, dacă ar exista acorduri între triburi astfel încât toate vârfurile să facă parte din aceeaşi componentă conexă, am putea considera că triburile şi-au atins scopul.

Vom porni de la graful în care toate vârfurile sunt de grad zero. Acestui graf îi vom adăuga muchii, una după alta, unind componentele conexe existente la un moment dat. Inițial există *n* componente conexe, corespunzătoare celor *n* vârfuri izolate.

Pentru fiecare vârf vom memora în şirul *cc* numărul de ordine al componentei conexe din care face parte. Inițial, vârful *k* face parte din componenta conexă având numărul de ordine *k*. Numărul de ordine al componentei conexe căreia îi aparține un anumit vârf se modifică dacă vârful este unit printr-o muchie de o altă componentă conexă.

Reprezentarea datelor se va realiza sub forma listei muchiilor.

```
Subalgoritm DeterminareaComponentelorConexe(n,m,a,ncc,cc):
  k ← 1
  ncc ← n
  pentru i=1,n execută:                          { fiecare nod este izolat }
    cc[i] ← i
  sfârşit pentru
  cât timp k ≤ m execută:                        { se adaugă toate muchiile }
    dacă cc[a[k].i] ≠ cc[a[k].j] atunci          { dacă extremitățile unei }
      min ← cc[a[k].i]                  { muchii fac parte din componente diferite }
      max ← cc[a[k].j]
      ncc ← ncc-1
    sfârşit dacă
    pentru i=1,n execută:                  { unim cele două componente conexe }
      dacă cc[i] = max atunci
        cc[i] ← min
      sfârşit dacă
    sfârşit pentru
    k ← k + 1
  sfârşit cât timp
sfârşit subalgoritm
```

După determinarea componentelor conexe acestea trebuie afişate câte una pe o linie. Numărul componentelor conexe este memorat în variabila *ncc*.

```
Subalgoritm Afişare(n,ncc,cc):
  scrie ncc                                  { afişează numărul de comunităţi }
  pentru i=1,n execută:
    dacă cc[i] > 0 atunci
      scrie i
      pentru j=i+1,n execută:
        dacă cc[j] = cc[i] atunci  { se afişează membrii unei comunităţi }
          scrie j
          cc[j] ← 0                          { marcăm vârfurile afişate }
        sfârşit dacă
      sfârşit pentru
    sfârşit dacă
  sfârşit pentru
...
```

O a doua problemă ridicată este posibilitatea de a aduce pace pe întreaga planetă prin semnarea de acorduri acolo unde este nevoie. Altfel spus, ar fi bine ca pe întreaga planetă să fie o singură comunitate. Acest lucru se realizează prin transformarea grafului într-un graf conex.

Pentru a obține un graf conex din cel existent, o posibilitate ar fi aceea de a lega primul vârf de câte un vârf din fiecare componentă conexă diferită de cea a primului vârf. Răspundem la întrebarea din enunț, afișând în continuare aceste muchii.

```
...
  dacă ncc > 1 atunci
    scrie 'Acorduri necesare:'      { se afişează muchiile posibile pentru }
    pentru i=2,n execută:           { a transforma graful într-unul conex }
      dacă (cc[i] > 0) şi (cc[i] ≠ cc[1]) atunci
        scrie 1, i
      sfârşit dacă
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

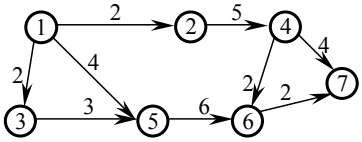
5.5.8. Pregătirea balului

Pentru a putea rezolva problema propusă se vor prezenta câteva considerente teoretice. În cazul unei probleme care propune tratarea unei lucrări complexe care presupune desfăşurarea unor evenimente, vom construi un graf orientat numit graf de activități.

În acest graf vârfurile sunt evenimente – obiective parțiale ale lucrării, iar arcele sunt activități cărora li se asociază lungimile lor – timpul de execuție.

O regulă importantă în realizarea unui graf de activități este aceea că dacă (x, y) este o activitate în graf, atunci această activitate se poate desfăşura doar după ce toate activitățile care au extremitatea finală în x s-au terminat.

În orice graf de activităţi există un vârf care reprezintă începerea lucrării şi nu este extremitate finală a nici unui arc. De asemenea, există un vârf care reprezintă încheierea lucrării şi nu este extremitate iniţială a nici unui arc. Cu alte cuvinte, din primul vârf pleacă arce şi în cel de-al doilea sosesc arce. În plus, graful nu are circuite.

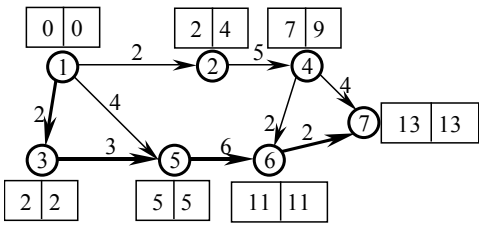


În graful din figura de mai sus vârful 1 reprezintă începerea activităţii, iar vârful 7 reprezintă încheierea activităţii. În cazul unei lucrări executantul îşi pune problema necesarului de timp pentru ca întreaga lucrare să poată fi realizată. Din exemplu reiese că lucrarea respectivă se poate executa în 13 unităţi de timp. Acest număr este dat de lungimea celui mai lung drum din graf (de la vârful 1 la vârful 7). Practic, pot exista anumite activităţi care dispun de mai mult timp decât au nevoie. În exemplul de mai sus există trei drumuri de la vârful 1 la vârful 6. Lungimile lor sunt 9, 11 respectiv 10. Activitatea (6, 7) nu se poate realiza decât după ce activităţile (4, 6) şi (5, 6) au fost încheiate şi evident, cele care le preced pe acestea. Din acest punct de vedere activităţile (1, 2) sau (2, 4) sau (4, 6) au o posibilitate ca pe ansamblu să întârzie câte 2 unităţi faţă de durata lor prestabilită, deoarece în aceste condiţii nu întârzie terminarea întregii lucrări. Cât despre activităţile (1, 3), (3, 5) şi (5, 6), dacă acestea întârzie, determină întârzierea întregii lucrări. Cu alte cuvinte aceste muchii se numesc critice. Activităţile critice într-un graf de activităţi determină un drum care leagă primul vârf de ultimul care se numeşte *drum critic*.

În problema dată se cere determinarea lungimii drumului critic şi a arcelor care îl compun, din graful de activităţi asociat pregătirilor balului.

Pentru a determina drumul critic vom construi două şiruri *lung* şi *tr*.

- *lung* este pentru fiecare vârf v_i lungimea celui mai lung drum de la primul vârf la v_i;
- *tr* reprezintă pentru fiecare vârf v_i diferenţa dintre lungimea totală a drumului critic şi lungimea minimă a drumului de la v_i la ultimul vârf din graf.
- diferenţa $tr_k - lung_k$ reprezintă rezerva de timp pe care o au evenimentele care nu fac parte din drumul critic şi ajung în vârful k.
- pentru evenimentele (x, y) care fac parte din drumul critic avem:
  $tr_x = lung_x$ şi $tr_y = lung_y$ şi $lung_y = lung_x + d_{x,y}$.



```
Subalgoritm DrumCritic(n,a,lung,tr):
  pentru i=1,n execută:                    { iniţializarea datelor de lucru }
    lung_i ← -1
    lung_1 ← 0
    viz_1 ← adevărat
    i ← 1
    v[i] ← 1
    pentru k=1,n execută:                  { parcurgem toate vârfurile }
      pentru j=1,n execută:          { arcele care pornesc din al k-lea vârf }
        dacă a[v[k],j] > 0 atunci
                                           { calculăm lungimea maximă }
          lung_j ← max(lung_j,lung_v[k] + a[v[k],j])
          dacă nu viz[j] atunci
            i ← i+1
            v[i] ← j
          sfârşit dacă
        viz_v[k] ← adevărat
      sfârşit pentru
    sfârşit pentru
  sfârşit pentru
  v_1 ← n                                  { calculăm lungimea drumului critic }
  tr_n ← lung_n
  i ← 1
  pentru k=1,n execută:
    pentru j=1,n execută:
      dacă (a[j,v[k]] > 0) şi viz_j atunci
        tr_j ← tr_v[k] - a[j,v[k]]       { calculăm timpul maxim de care }
        i ← i + 1                          { poate dispune evenimentul j }
        v[i] ← j
        viz_v[k] ← fals
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

5.5.9. Pe drumuri de munte

Reprezentăm harta zonei cu un graf neorientat având *n* vârfuri, în care vârfurilor le asociem punctele de pe traseu şi muchiilor traseele marcate din zonă.

Memorăm graful cu ajutorul matricei de adiacenţă şi o construim odată cu citirea datelor. Matricea de adiacenţă iniţial conţine doar elemente nule. Cu fiecare citire a câte două vârfuri *i* şi *j* adăugăm în matrice câte două elemente egale cu unu.

Pentru a răspunde cerinţelor problemei trebuie să folosim o metodă care generează *toate* soluţiile posibile. Vom recurge la *backtracking*, cu toate că este o metodă mare consumatoare de timp.

În şirul x vom păstra numerele de ordine ale punctelor vizitate pe traseu. Primul punct este cel al cabanei – punctul de plecare. În acest şir se vor adăuga puncte care sunt legate succesiv, până când punctul curent este cel final (al peşterii). La momentul k există în şir $k-1$ puncte care pot fi atinse succesiv pe un traseu. Trebuie ales pasul următor şi anume punctul de indice k. Punctul x_k se alege din toate punctele posibile, dacă există traseu marcat de la punctul x_{k-1} la acest punct şi traseul nu a mai trecut prin x_k.

```
Subalgoritm Drum(k):
  dacă x[k-1] = s atunci                    { s-a găsit punctul de sosire }
    afişăm traseul având k-1 puncte
  altfel
    pentru j=1,n execută
      x[k] ← j                  { alegem un punct pentru continuarea traseului }
      dacă a(x[k-1],x[k])=1 şi x[k] nu a fost în traseu până acum atunci
        Drum(k+1)                      { se continuă drumul cu punctul următor }
      sfârşit dacă
    sfârşit pentru
  sfârşit dacă
sfârşit subalgoritm
```

5.5.10. Mesaj telefonic

Vom construi un graf neorientat în care copii vor fi codificaţi prin vârfurile grafului, iar faptul că îşi cunosc reciproc numerele de telefon se codifică muchii.

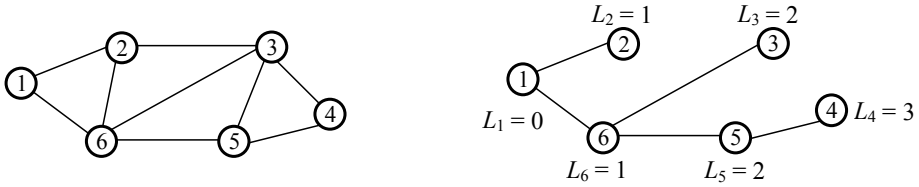
Se poate observa că dacă în graful asociat există muchie între vârful corespondent lui Ionuţ şi cel corespondent lui Gigel, este suficient un singur apel telefonic pentru ca acesta să afle mesajul. Dacă însă între cele două vârfuri corespunzătoare celor doi copii nu există muchie, atunci este nevoie de intermediari. Este evident, că dacă graful nu este conex, transmiterea mesajului este imposibilă.

Pentru a rezolva problema vom parcurge graful pornind de la vârful p asociat lui Ionuţ care vrea să transmită mesajul şi vom determina un lanţ de lungime minimă între p şi s (unde s este vârful asociat lui Gigel care urmează să fie anunţat). Lungimea acestui lanţ este chiar numărul minim de telefoane care trebuie date pentru ca mesajul să ajungă la s.

Există mai multe metode cu care un astfel de lanţ se poate determina: greedy, programare dinamică, traversarea grafului în lăţime etc. Noi vom rezolva problemă cu cea din urmă metodă.

Pornim din *p* şi marcăm toate vârfurile adiacente cu vârful *p*. Lungimea lanţului care porneşte din *p* şi se termină într-unul dintre vecinii săi este 1. Dacă vreunul dintre aceste vârfuri este *s* atunci algoritmul se opreşte. Dacă nici unul dintre vecinii lui *p* nu este *s*, se caută *s* printre vecinii vecinilor lui *p*. Dacă este printre aceştia lungimea lanţului este 2. Practic, printr-o parcurgere în lăţime a grafului se pot construi lanţurile de lungime minimă, care pornesc din *p* şi astfel se poate obţine şi lanţul de lungime minimă de la *p* la *s*.

Fie graful din figura din stânga. Făcând abstracţie de *p* şi *s*, construirea lanţurilor de lungime minimă se realizează ca în figura din dreapta.



Memorăm graful cu ajutorul listelor vecinilor. Construirea lor se realizează deodată cu citirea datelor.

```
Subalgoritm Citire(n,p,s,nv,ve):
  citeşte n                                          { numărul de copii }
  citeşte p,s                                        { p vrea să transmită mesaj lui s }
  cât timp sunt muchii de citit execută:
                          { se citesc perechile care îşi cunosc numerele de telefon }
    citeşte i,j
    nv_i ← nv_i + 1                  { creşte numărul de vecini ai lui i şi ai lui j }
    nv_j ← nv_j + 1
    ve[i,nv_i] ← j
    ve[j,nv_j] ← i
  sfârşit cât timp
sfârşit subalgoritm
```

Pentru a rezolva problema este nevoie de memorarea predecesorilor fiecărui vârf în lanţul de lungime minimă, după ce a fost calculată lungimea sa. În subalgoritmul următor avem următoarele semnificaţii ale variabilelor:

- *n*: numărul copiilor;
- *s*, *p*: *p* vrea să transmită lui *s*;
- *prec*: lista predecesorilor fiecărui vârf în lanţ;
- *nv*: numărul de vecini ai fiecărui vârf;
- *lung*: distanţele minime de la primul vârf la celelalte;
- *ve*: listele vecinilor.

```
Subalgoritm LanțMinim(n,p,s,nv,ve,prec,lung):
  pentru i=1,n execută:                          { se inițializează distanțele }
    lung[i] ← -1
  sfârşit pentru
  lung[p] ← 0                          { distanța de la nodul p la el însuşi este zero }
  tel[1] ← p
  k ← 1
  u ← 1
  cât timp k ≤ u execută:                          { toți copiii se vor vizita }
                          { pentru fiecare din listă se iau în considerare vecinii săi }
    pentru i=1,nv[tel[k]] execută:
      x ← ve[tel[k],i]                                              { un vecin }
      dacă lung[x] < 0 atunci { dacă nu a fost prelucrat, se adaugă în listă }
        lung[x] ← lung[tel[k]]+1  { se calculează distanța față de primul }
        u ← u + 1
        tel[u] ← x
        prec[x] ← tel[k]                    { i se memorează predecesorul }
        dacă x = s atunci
          ieşire forțată din subalgoritm
        sfârşit dacă
      sfârşit dacă
    sfârşit pentru
    k ← k + 1                          { se trece la următorul element din listă }
  sfârşit cât timp
sfârşit subalgoritm
```

Afişarea se realizează cu ajutorul unui subalgoritm recursiv, deoarece dacă am afişa pur şi simplu lista predecesorilor, am obține o secvență inversă de la *s* la *p* în loc să fie de la *p* la *s*.

```
Subalgoritm Afişare(k):
  dacă k ≠ p atunci
    Afişare(prec[k])
  sfârşit dacă
  scrie k
sfârşit subalgoritm
```

5.5.11. Distanțe rutiere

Se poate asocia localităților şi drumurilor dintre ele un graf cu costuri.

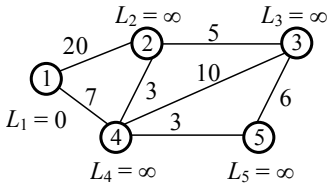
Problema noastră cere să determinăm lungimea minimă a unui drum între două localități. Ştim că aceasta este egală cu lungimea lanțului de lungime minimă între cele două localități.

Pentru a putea determina distanța minimă între două localități vom folosi algoritmul lui *Dijkstra*. Acesta determină lungimea minimă a lanțurilor care pornesc dintr-un vârf al grafului şi pleacă spre celelalte vârfuri cu o strategie *greedy*.

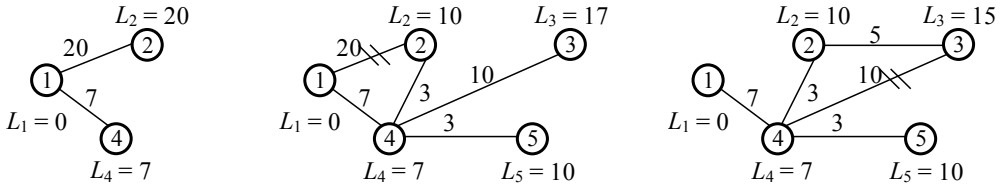
Algoritmul se poate aplica atât în cazul unor grafuri orientate cât şi neorientate. Memorarea grafurilor se va realiza cu ajutorul matricei costurilor. Memorarea predecesorilor fiecărui vârf va fi utilă în afişarea lanțurilor (drumurilor).

Fiecărui vârf din graf i se va asocia o lungime față de vârful x_0 care inițial este considerată foarte mare (*infinit*). Vârful x_0 are asociată lungimea 0 (zero). Acest vârf se trece în mulțimea vârfurilor vizitate. În mulțimea vârfurilor vizitate se adaugă pe rând vârfurile care sunt adiacente lui x_0 determinând lungimea minimă a drumului către ele.

În figura următoare exemplificăm modalitatea de construire a drumurilor pentru graful:



Pornind de la $x_0 = 1$ se construiesc succesiv lanțurile:



Se poate observa că o muchie directă între două vârfuri nu este neapărat cel mai scurt drum între cele două localități. Astfel la calculul distanțelor vom ține cont de valoarea minimă a tuturor drumurilor între cele două localități.

La fiecare moment alegem un vârf neparcurs, care se situează la distanță minimă față de x_0, luând în considerare toate drumurile care îl leagă pe acesta de x_0. Pentru toți succesorii săi (vârfurile adiacente) se recalculează lungimile drumurilor.

```
Subalgorim Dijkstra(n,a,lung):
  W ← V                          { W este inițial mulțimea tuturor vârfurilor }
  pentru i=1,n execută:
    lung_i ← infinit
  sfârşit pentru
  lung_p ← 0                     { se calculează distanțele pornind de la vârful p }
```

```
  cât timp W ≠ ∅ execută:                    { cât timp există noduri nevizitate }
    vm ← vârful din W pentru care lung_vm este minimă
    W ← W \ {vm}                             { vârful se marchează ca fiind vizitat }
    pentru j=1,n execută:
      dacă a[vm,j] ≠ 0 atunci
                    { se calculează distanţele minime la care se află vecinii săi }
        lung_j ← min(lung_j,lung_vm+a[vm,j])
      sfârşit dacă
    sfârşit pentru
  sfârşit cât timp
sfârşit subalgoritm
```