Sortare topologică

Mirel Coşulschi
mirelc@central.ucv.ro

Mihai Gabroveanu
mihaiug@central.ucv.ro

July, 2023

1 Noţiuni introductive

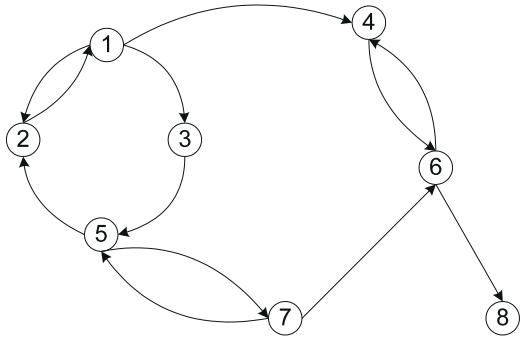

Fig. 1: Un exemplu de graf orientat

Fie V şi E două mulţimi finite astfel încât $V = \{x_1, x_2, \ldots, x_n\}$ şi $E \subseteq V \times V$.

Definiţia 1 *Un **graf orientat** (digraf) este o pereche ordonată $G = (V,\ E)$, unde V este o mulţime de vârfuri sau noduri, iar E este o mulţime de arce.*

În cazul unui graf orientat, noţiunea de *muchie* este înlocuită cu noţiunea de *arc*: o pereche de noduri (x, y) devine ordonată, adică $(x, y) \neq (y, x)$ dacă $x \neq y$.

Exemplul 1 *Fie graful orientat $G = (V, E)$, $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$,*
$E = \{(1, 2), (1, 3), (1, 4), (2, 1), (3, 5), (4, 6), (5, 2), (5, 7), (6, 4), (6, 8), (7, 5), (7, 6)\}$. *Acest graf se poate reprezenta ca în figura 1.*

Definiţia 2 *Fie $G = (V, E)$ un graf orientat, finit şi $e = (x, y) \in E$ un arc al lui G. Atunci spunem că*

1. *nodurile x şi y sunt **adiacente**;*
2. *nodul x este **extremitatea iniţială** (nod sursă) iar nodul y este **extremitatea finală** (nod destinaţie);*
3. *arcul e este **incident** din x **către** y.*

Într-un digraf nu putem avea mai multe arce care să aibă aceeaşi pereche de noduri ca extremitate iniţială şi finală, însă putem avea **bucle**.

Se numeşte **multigraf orientat** un graf orientat G în care putem avea şi arce paralele.

Definiţia 3 ***Gradul exterior*** *al unui vârf* $d_G^+(x)$ *este egal cu numărul arcelor ce au ca extremitate iniţială pe* x $(d_G^+(x) = |\{(x,u)|\ (x,u) \in E, u \in V\}|)$.

Gradul interior *al unui vârf* $d_G^-(x)$ *este egal cu numărul arcelor ce au ca extremitate finală pe* x $(d_G^-(x) = |\{(u,x)|(u,x) \in E, u \in V\}|)$.

2 Parcurgerea în adâncime (DFS-Depth First Search)

În cadrul acestei metode se va merge în adâncime ori de câte ori este posibil: *prelucrarea* unui vârf constă în prelucrarea primului dintre vecinii săi încă nevizitaţi.

- se vizitează vârful de pornire (notat cu k);
- urmează, primul vecin încă nevizitat al acestuia;
- se caută primul vecin, încă nevizitat, al primului vecin nevizitat al nodului de start, ş.a.m.d.;
- se merge în adâncime până când se ajunge la un vârf ce nu are vecini, sau pentru care toţi vecinii săi au fost vizitaţi. În acest caz, se revine în nodul său părinte (nodul din care a fost vizitat nodul curent), şi se continuă algoritmul, cu următorul vecin încă nevizitat al nodului curent.

Algoritm 1 Algoritm de vizitare în adâncime (varianta recursivă)

```
1: procedure DFS(k, n, Vecin)
Input: { k      - nodul curent ce se vizitează
       { n      - numărul de noduri din graf
       { Vecin  - matricea de adiacenţă a grafului
2:     vizitat_k ← 1                          ▷ marcarea nodului curent ca fiind vizitat
3:     call Vizitare(k)                       ▷ vizitarea nodului curent
4:     for i ← 1, n do
5:         if ((vizitat_i = 0) ∧ (vecin_{k,i} = 1)) then
6:             call DFS(i, n, vecin)          ▷ apelul recursiv al subrutinei DFS pentru nodul i
7:         end if
8:     end for
9: end procedure
```

În mod asemănător ca şi la algoritmul de parcurgere în lăţime, vectorul `vizitat` gestionează situaţia vizitării nodurilor grafului G:

$$vizitat_k = \begin{cases} 1 & \text{, dacă nodul } k \text{ a fost vizitat} \\ 0 & \text{, în caz contrar.} \end{cases}$$

Subrutina `DFS` (algoritmul 1) începe cu marcarea nodului curent ca fiind *vizitat* ($vizitat_k \leftarrow 1$) (linia 2) şi apelul procedurii `Vizitare()` (`call Vizitare(k)`) (linia 3). Se caută apoi primul *vecin nevizitat* i al vârfului curent k (nod ce verifică condiţia $(vizitat_i = 0) \wedge (vecin_{k,i} = 1)$) şi se reia secvenţa de operaţii pentru nodul i cu această proprietate, printr-un apel recursiv (`call DFS(i)`) (linia 6).

Observaţia 2 *Enunţul repetitiv* `for` $i \leftarrow 1, n$ *(linia 4) poate fi optimizat astfel încât să nu se mai verifice toate vârfurile grafului, ci numai nodurile adiacente cu nodul curent: în această situaţie reprezentarea cu liste de adiacenţă este optimă din punct de vedere al numărului de operaţii efectuate, fiind preferată reprezentării prin matricea de adiacenţă.*

În urma unei parcurgeri în adâncime a unui graf orientat, fiecare arc din mulţimea arcelor poate fi încadrat într-una dintre următoarele categorii:

- *arc al arborelui de acoperire* - arcul (u, v) este un arc al arborelui de acoperire dacă $dfs(u)$ apelează $dfs(v)$ ($dfs(u) \overset{call}{\rightsquigarrow} dfs(v)$).
- *arc de înaintare* - arcul (u, v) este un arc de înaintare dacă este paralel cu un drum de la u la v din arborele de acoperire ($u \rightsquigarrow v$) (nu face parte din arborele de acoperire).
- *arc de întoarcere* - arcul (u, v) se numeşte arc de întoarcere dacă are sensul contrar unui drum de la v la u din arborele de acoperire ($v \rightsquigarrow u$).
- *arc de traversare* - arcul (u, v) este un arc de traversare dacă $dfs(v)$ a fost apelat şi s-a terminat înainte de apelul lui $dfs(u)$.

Arborele de acoperire în adâncime corespunzător grafului din figura 1 ilustrează aceste tipuri de arce: *arc de înaintare* – $(1, 4)$, *arce de întoarcere* - $(2, 1)$, $(7, 5)$, $(4, 6)$ *arc de traversare* - $(5, 2)$, *arce ale arborelui de acoperire* - $(1, 2)$, $(1, 3)$, $(3, 5)$, $(5, 7)$, $(7, 6)$, $(6, 8)$, $(6, 4)$ (a se vedea figura 2).

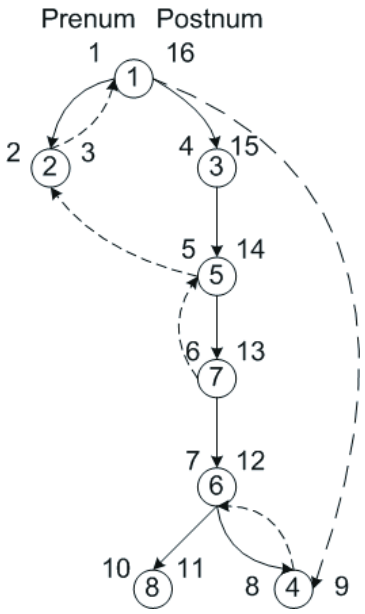

Fig. 2: Arbore de acoperire în adâncime pentru graful orientat din figura 1

Pentru fiecare nod v al unui graf vom introduce două numere, $prenum_v$ şi $postnum_v$, numere ce depind de ordinea în care sunt întâlnite nodurile acestuia în timpul vizitării în adâncime: $prenum_v$ marchează momentul când este întâlnit nodul v pentru prima oară iar $postnum_v$ momentul în care prelucrarea nodului v s-a încheiat. Variabila *counter* este iniţializată cu valoarea 1:

```
1: procedure PRENUM(v)
2:     prenum_v ← counter
3:     counter ← counter + 1
4: end procedure
```

şi

```
1: procedure POSTNUM(v)
2:     postnum_v ← counter
3:     counter ← counter + 1
4: end procedure
```

Un arc (u, v) va avea următoarele proprietăţi, în funcţie de una din cele patru categorii de arce introduse anterior în care se încadrează:

1. *arc al arborelui de acoperire*: $prenum_u < prenum_v$ şi $postnum_u > postnum_v$;
2. *arc de înaintare*: $prenum_u < prenum_v$ şi $postnum_u > postnum_v$;
3. *arc de întoarcere*: $prenum_u > prenum_v$ şi $postnum_u < postnum_v$;
4. *arc de traversare*: $prenum_u > prenum_v$ şi $postnum_u > postnum_v$.

Algoritm 2 Algoritm de vizitare în adâncime pentru un graf orientat

```
1: procedure DFSNUM(k, n, Vecin)
Input: { k      - nodul curent vizitat
       { n      - numărul de noduri din graf
       { Vecin  - matricea de adiacenţă a grafului
2:     vizitat_k ← 1                ▷ marcarea nodului curent k ca fiind vizitat
3:     call Prenum(k)
4:     call Vizitare(k)             ▷ vizitarea nodului curent k
5:     for i ← 1, n do
6:         if ((vizitat_i = 0) ∧ (vecin_{k,i} = 1)) then
7:             call DFSNum(i, n, Vecin)   ▷ apelul recursiv al subrutinei DFSNum pt nodul i
8:         end if
9:     end for
10:    call Postnum(k)
11: end procedure
```

Lema 3 *Fiind dat un graf orientat G şi două noduri oarecare $u, v \in G$ ce aparţin aceluiaşi arbore de acoperire în adâncime rezultat în urma parcurgerii cu metoda DFS a grafului G avem:*

1. *dacă (u, v) este un arc al arborelui de acoperire sau un arc de înaintare sau un arc de traversare atunci avem relaţia $postnum_u > postnum_v$;*
2. *dacă (u, v) este un arc de întoarcere atunci avem relaţia $postnum_u < postnum_v$.*

Lema 4 *Fiind dat un graf orientat G pentru oricare două noduri $u, v \in G$ avem:*

1. *v este un descendent al lui u în pădurea de arbori de acoperire rezultaţi în urma vizitării în adâncime a grafului $G \Leftrightarrow$ intervalul $[prenum_v, postnum_v]$ este inclus în intervalul $[prenum_u, postnum_u]$;*

2. *nu există nici o legătură între u şi v în pădurea de arbori de acoperire rezultaţi în urma vizitării în adâncime a grafului G ⇔ intervalele* $[prenum_u, postnum_u]$ *şi* $[prenum_v, postnum_v]$ *sunt disjuncte;*

3. *nu sunt posibile situaţiile* $prenum_u < prenum_v < postnum_u < postnum_v$ *sau* $prenum_v < prenum_u < postnum_v < postnum_u$.

Lema 5 *Fiind dat un graf neorientat G, dacă pentru două noduri oarecare* $u, v \in G$ *avem relaţia* $prenum_u < prenum_v < postnum_u$ *atunci:*

- $prenum_u < prenum_v < postnum_v < postnum_u$;
- *există un drum de la u la v în G.*

3 Sortarea topologică

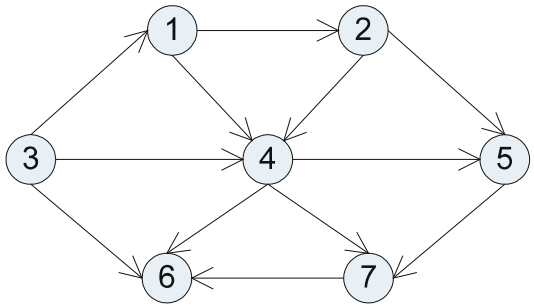

Fig. 3: Un graf orientat aciclic

Definiţia 4 *Un graf orientat şi care nu posedă circuite se numeşte* ***graf orientat aciclic*** *(directed acyclic graph - DAG).*

În practică, există mai multe situaţii în care o mulţime de activităţi sau sarcini, trebuie organizate într-o anumită ordine, între acestea existând, de obicei, o mulţime de restricţii sau dependenţe. În activitatea de construire a unei clădiri, anumite activităţi nu pot fi începute decât după finalizarea altor activităţi: spre exemplu, nu se poate începe ridicarea pereţilor unei construcţii decât după turnarea fundaţiei şi finalizarea structurii de rezistenţă, nu se poate realiza instalaţia electrică dacă nu au fost ridicate zidurile, ş.a.m.d.

Sau dacă un student doreşte să îşi alcătuiască un plan de studiu individual ce va conţine pe lângă cursurile obligatorii, şi o serie de cursuri opţionale, va trebui să ţină cont de anul de studiu în care se predă fiecare materie, precum şi de cerinţele obligatorii ale acestora: un curs nu poate fi inclus în planul de studiu individual al unui student decât dacă acesta a parcurs şi a obţinut creditele la toate materiile anterioare cerute explicit în programa cursului.

Dependenţa dintre două activităţi A şi B o putem modela prin introducerea a două noduri în graf x_i şi x_j, asociate celor două activităţi. Dacă activitatea $A(x_i)$ trebuie realizată înaintea activităţii $B(x_j)$, atunci se adaugă arcul (x_i, x_j).

Definiţia 5 *Se numeşte* ***sortare topologică*** *pentru un graf orientat $G = (V, E)$ o ordonare $\{x_1, x_2, \ldots, x_n\}$ a nodurilor grafului astfel încât pentru orice arc (x_i, x_j) să avem $i < j$.*

Prin urmare o *sortare topologică* presupune aranjarea liniară a vârfurilor unui graf astfel încât toate arcele sale să fie orientate de la stânga la dreapta.

Lema 6 *Dacă un graf orientat G admite o sortare topologică atunci G este aciclic.*

Lema 7 *Dacă un graf orientat G este aciclic atunci el admite o sortare topologică.*

Observaţia 8 *Într-un graf orientat aciclic există cel puţin un nod al cărui grad interior este 0 (graful nu posedă arce care să aibă nodul v drept extremitate finală).*

Algoritm 3 Algoritm de sortare topologică a unui graf orientat (algoritmul lui Kahn)

1: **procedure** SortTop1($n, Vecin$; L)
Input: $\begin{cases} n & \text{- numărul de noduri din graf} \\ Vecin & \text{- vector ce conţine listele cu vecini ai fiecărui nod} \end{cases}$
Output: L - lista nodurilor ordonate în urma sortării topologice
2: **for** $k \leftarrow 1, n$ **do**
3: $dminus_k \leftarrow 0$
4: **end for**
5: **for** $\forall (u, v) \in E$ **do**
6: $dminus_v = dminus_v + 1$
7: **end for**
8: $Q \leftarrow \emptyset$ ▷ se iniţializează coada
9: **for** $k \leftarrow 1, n$ **do**
10: **if** $(dminus_k = 0)$ **then**
11: $Q \Leftarrow k$ ▷ se inserează într–o coadă nodurile cu gradul interior 0
12: **end if**
13: **end for**
14: **while** $(Q \neq \emptyset)$ **do**
15: $Q \Rightarrow k$ ▷ se extrage din coadă un nod
16: $L \Leftarrow k$ ▷ se inserează nodul într-o listă
17: $w \leftarrow Vecin_k$ ▷ v ia valoarea capului listei de vecini a nodului k
18: **while** $(w \neq NULL)$ **do**
19: $dminus_{w.nodeIndex} \leftarrow dminus_{w.nodeIndex} - 1$
20: **if** $(dminus_{w.nodeIndex} = 0)$ **then**
21: $Q \Leftarrow w.nodeIndex$ ▷ se inserează în coadă nodul $w.nodeIndex$
22: **end if**
23: $w \leftarrow w.next$ ▷ se trece la următorul vecin
24: **end while**
25: **end while**
26: **end procedure**

Pornind de la această observaţie se schiţează următorul algoritm [4]: într-un graf G se determină un nod v astfel încât gradul său interior să fie zero ($d^-(v) = 0$); se adaugă acest nod la o listă ce va conţine ordonarea topologică, se şterge nodul din graf împreună cu toate arcele ce îl au ca extremitate iniţială, şi se reia algoritmul pentru graful $G' = (V', E')$, unde $V' = V \setminus \{v\}, E' = E|_{V' \times V'}$.

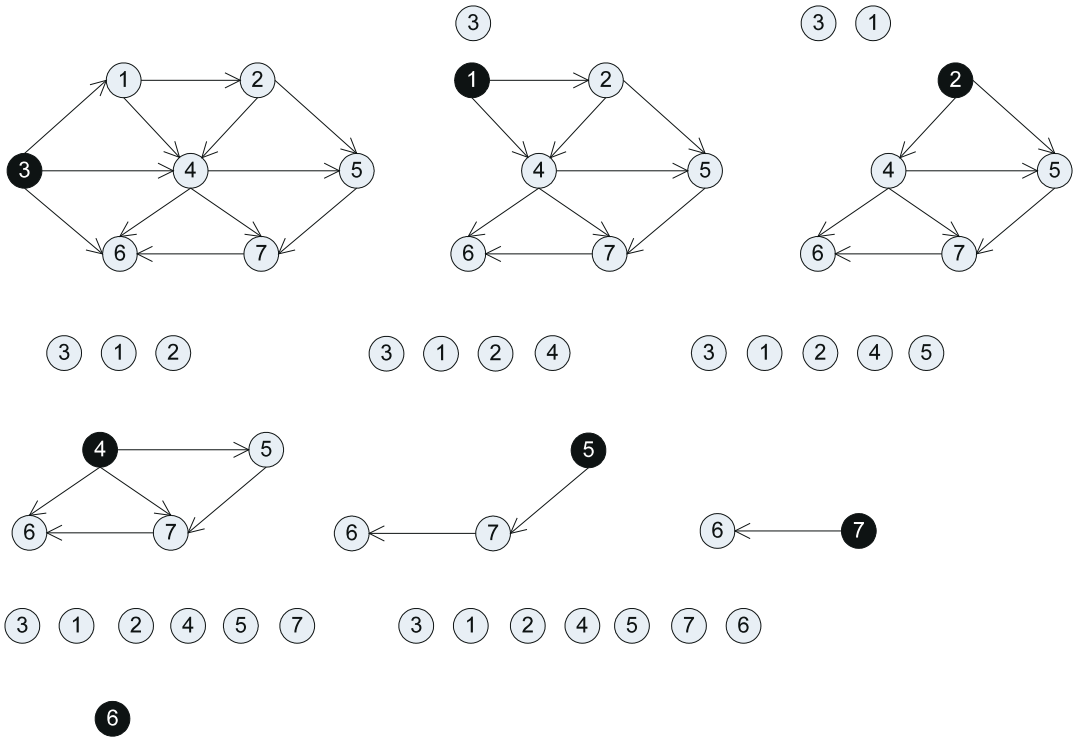

Fig. 4: Sortare topologică cu algoritmul 3 pentru graful orientat din figura 3

Algoritmul 3 se termină în cel mult n paşi ($|V| = n$). Dacă se termină mai devreme, atunci graful G nu este aciclic (la un moment dat nu mai există nici un nod v astfel încât $d^-(v) = 0$).

Exemplul 9 *Fie graful orientat din figura 4. Vârful* 3 *are* $d^-(v) = 0$. *Se adaugă acest nod la lista finală ce va conţine nodurile ordonate, se şterge nodul împreună cu arcele care îl au drept extremitate iniţială. În graful rezultat, vârful* 1 *are proprietatea că* $d^-(v) = 0$. *Se adaugă la lista de rezultate, şi se elimină din graf împreună cu arcele ce pleacă din el. Se continuă procedeul până când graful devine vid (întregul proces poate fi urmărit în figura 4).*

Lema 10 *Un graf orientat G este aciclic dacă şi numai dacă în urma unei vizitări în adâncime a acestuia nu este întâlnit nici un arc de întoarcere.*

Ideea algoritmului 4 o reprezintă lema 10[5].

Exemplul 11 *Să considerăm ca date de intrare pentru algoritmul 4 graful orientat din figura 3. După etapa de iniţializare (liniile 2 - 5) avem:*

	1	*2*	*3*	*4*	*5*	*6*	*7*
prenum	*0*	*0*	*0*	*0*	*0*	*0*	*0*
postnum	*0*	*0*	*0*	*0*	*0*	*0*	*0*
vizitat	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Se apelează mai întâi `DFSNum(1, 7, Vecin)`. *Secvenţa rezultată de apeluri recursive este următoarea:* `DFSNum(1, 7, Vecin)` → `DFSNum(2, 7, Vecin)` → `DFSNum(4, 7, Vecin)` → `DFSNum(5, 7, Vecin)` → `DFSNum(7, 7, Vecin)` → `DFSNum(6, 7, Vecin)`.

Algoritm 4 Algoritm de sortare topologică a unui graf orientat (a doua variantă)

1: **procedure** SortTop2($n, Vecin$; L)
Input: $\begin{cases} n & \text{- numărul de noduri din graf} \\ Vecin & \text{- matricea de adiacenţă a grafului} \end{cases}$
Output: L - lista nodurilor ordonate în urma sortării topologice
2: **for** $k \leftarrow 1, n$ **do**
3: $prenum_k \leftarrow 0, postnum_k \leftarrow 0$
4: $vizitat_k \leftarrow 0$
5: **end for**
6: **for** $k \leftarrow 1, n$ **do**
7: **if** $(vizitat_k = 0)$ **then**
8: **call** $DFSNum(k, n, Vecin)$
9: **end if**
10: **end for**
11: se ordonează descrescător nodurile după $postnum_k$ şi se păstrează astfel ordonate în lista L
12: **end procedure**

În urma acestei secvenţe valorile vectorilor `prenum` *şi* `postnum` *sunt următoarele:*

	1	2	3	4	5	6	7
prenum	1	2	0	3	4	6	5
postnum	12	11	0	10	9	7	8
vizitat	1	1	0	1	1	1	1

Mai rămâne nevizitat un singur nod, 3*, drept pentru care vom mai avea un apel* `DFSNum(3, 7, Vecin)` *din procedura principală* `SortTop2()`*:*

	1	2	3	4	5	6	7
prenum	1	2	13	3	4	6	5
postnum	12	11	14	10	9	7	8
vizitat	1	1	1	1	1	1	1

Ordonarea descrescătoare a nodurilor mulţimii V *după valorile vectorului* `postnum`*, conduce la următoarea configuraţie:*

postnum	14	12	11	10	9	8	7
	3	1	2	4	5	7	6

Astfel, sortarea topologică a nodurilor grafului obţinută în urma aplicării algoritmului 4 este: $3, 1, 2, 4, 5, 7, 6$.

Observaţia 12 *O optimizare a algoritmului 4 se referă la adăugarea nodului* v *într-o stivă atunci când se termină vizitarea acestui nod* v *şi se calculează* $postnum_v$*. Această stivă va conţine la final nodurile grafului în ordinea descrescătoare a valorilor* $postnum_v$*.*

Prin urmare, nu mai este nevoie să ordonăm descrescător valorile vectorului postnum pentru a obţine o sortare topologică a nodurilor grafului G*, iar complexitatea algoritmului 4 devine* $O(|V| + |E|)$*.*

4 Probleme

1. Se dă un graf orientat aciclic cu n noduri numerotate de la 1 la n. Să se realizeze o sortare topologică a nodurilor.

   Date de intrare

   Fişierul de intrare `topsort.in` conţine pe prima linie numerele `n m`, reprezentând numărul de noduri şi numărul de arce din graf, $1 \leq n \leq 100000$, $1 \leq m \leq 400000$, iar pe următoarele m linii câte o pereche de noduri `i j`, cu semnificaţia că în graf există arcul (i, j).

   Date de ieşire

   Fişierul de ieşire `topsort.out` va conţine pe prima linie cele n noduri ale grafului, separate prin exact un spaţiu, în ordinea dată de sortarea topologică.

   Exemplu

topsort.in
7 10 1 2 2 5 3 4 3 5 4 5 7 1 7 2 7 4 7 5 7 6

topsort.out
7 6 3 4 1 2 5

   (Topsort, `https://www.pbinfo.ro/probleme/1861/topsort`)

   Rezolvare:
   Prima variantă de rezolvare reprezintă implementarea algoritmului lui Kahn [4]. Graful orientat este reprezentat prin intermediul listelor de vecini. Acestea sunt implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică.

   Listing 1: `topsort_v1.c`

```
#include <stdio.h>

#define NMAX 100001
#define MMAX 400001

typedef struct queue {
           int f;        // indicele primului element din coada
           int l;        // indicele ultimului element din coada
           int t[NMAX];  // tabloul ce pastreaza elementele cozii
        } QUEUE;

int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
```

```
                    //           al nodului k in tabloul vecin
int last;           // numarul total de elemente

char visit[NMAX];   // visit[k] = 1 daca nodul k a fost vizitat
                    //          = 0 in caz contrar
int dminus[NMAX];   // dminus[i] - gradul interior al nodului i
QUEUE q;            // coada liniara

/*
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  q->f = 0;
  q->l = -1;
}

/*
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return ((q->l + 1) == q->f);
}

/*
*   Functia adauga in coada q elementul a carui valoare este
*  pastrata in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->l++;
  q->t[q->l] = x;
}

/*
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
  q->f++;
}

/*
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->t[q->f];
}

/*
*  Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
```

```
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  FILE *fin, *fout;
  int n, m, i, u, v, p;

  fin = fopen("topsort.in", "r");
  fout = fopen("topsort.out", "w");

  fscanf(fin, "%d %d", &n, &m);

  last = 1;
  for (i = 0; i < m; i++) {
    fscanf(fin, "%d %d", &u, &v);

    addEdge(u, v);            // se adauga v in lista de vecini a lui u

    dminus[v]++;              // incrementeaza gradul interior al nodului v
  }

  init(&q);                   // initializeaza coada
  for (u = 1; u <= n; u++) {
    if (dminus[u] == 0) {     // se adauga in coada toate nodurile care
                              // nu sunt extremitate finala pt vreun arc
      visit[u] = 1;           // marchez nodul u ca fiind adaugat in coada

      enqueue(&q, u);         // se adauga nodul u in coada
    }
  }

  while (!isempty(&q)) {      // cat timp mai exista elemente in coada
    u = front(&q);            // se extrage elementul aflat la inceputul
       cozii
    dequeue(&q);

    fprintf(fout, "%d ", u);

    // se parcurge lista de vecini a nodului u
    for (p = head[u]; p != 0; p = next[p]) {
      v = vecin[p];           // se obtine eticheta nodului vecin
      if (!visit[v]) {        // daca nodul v nu se afla in coada
        dminus[v]--;          // stergem arcul (u, v)
        if (dminus[v] == 0) { // daca nodul v nu mai are arce ce intra in v
          visit[v] = 1;       // marchez nodul v ca fiind adaugat in coada

          enqueue(&q, v);     // adaug nodul v in coada
        }
      }
    }
  }
```

```
  fclose(fin);
  fclose(fout);

  return 0;
}
```

Cea de-a doua variantă de rezolvare are la bază algoritmul de vizitare în adâncime al unui graf. Graful orientat este reprezentat cu ajutorul listelor de vecini implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică.

Listing 2: `topsort_v2.c`

```
#include <stdio.h>

#define NMAX 100001
#define MMAX 400001

int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost vizitat
                   //          = 0 in caz contrar
int trace[NMAX];   // lista nodurilor in ordinea in care au fost parasite
                   // la parcurgerea in adancime
int kth;

/*
*  Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

void dfs(int u) {
  int v, p;

  visit[u] = 1;         // marchez nodul u ca fiind vizitat

  // se parcurge lista de vecini a nodului u
  for (p = head[u]; p != 0; p = next[p]) {
    v = vecin[p];       // se obtine eticheta nodului vecin
    if (!visit[v]) {    // daca nodul v nu a fost vizitat
      dfs(v);           // apelez recursiv DFS pentru nodul v
    }
  }
```

```
  trace[kth++] = u;      // se adauga nodul u la vectorul trace;
                         // se marcheaza ultima vizitare a nodului u
}

int main() {
  FILE *fin, *fout;
  int n, m, i, u, v;

  fin = fopen("topsort.in", "r");
  fout = fopen("topsort.out", "w");

  fscanf(fin, "%d %d", &n, &m);

  last = 1;
  for (i = 0; i < m; i++) {
    fscanf(fin, "%d %d", &u, &v);

    addEdge(u, v);              // se adauga v in lista de vecini a lui u
  }

  for (i = 1; i <= n; i++) {    // pentru toate nodurile grafului
    if (!visit[i]) {            // daca nodul i nu a fost vizitat
      dfs(i);                   // se initiaza o parcurgere in adancime din i
    }
  }

  // se afiseaza continutul vectorului trace de la ultima la prima pozitie
  for (i = kth - 1; i > -1; i--) {
    fprintf(fout, "%d ", trace[i]);
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

2. La o firmă de software se lucrează la un mare proiect. Proiectul constă în executarea a n (număr natural) faze de dezvoltare, numerotate cu numerele $1, 2, \ldots, n$. Unele faze pot fi executate în paralel (în acelaşi timp), însă executarea altor faze nu poate fi începută până când nu se finalizează executarea anumitor faze.

   Să se scrie un program care să determine:

   a) timpul minim t în care se poate finaliza executarea proiectului.

   b) pentru fiecare fază k (k din $\{1, 2, \ldots, n\}$), momentul de timp c_k la care poate începe faza k cel mai devreme, respectiv momentul de timp d_k la care poate începe faza k cel mai târziu, fără a influenţa durata totală de executare a proiectului.

   Date de intrare

   Fişierul de intrare `pm.in` conţine:

- pe prima linie, un număr natural n ($0 \leq n \leq 100$), reprezentând numărul fazelor proiectului;
- pe a doua linie, n numere naturale, separate prin câte un spaţiu, reprezentând timpul necesar finalizării fiecărei faze;
- pe fiecare linie k dintre următoarele n linii, un număr natural m_k şi un şir a format din m_k numere naturale: $a_1, a_2, \ldots, a_{m_k}$, cele $m + 1$ numere din linie fiind separate prin câte un spaţiu, m_k reprezentând numărul de faze ce trebuie finalizate înaintea începerii fazei k, iar numerele din şirul a reprezentând numerele de ordine ale fazelor ce trebuie finalizate înaintea începerii fazei k.

Date de ieşire

Fişierul de ieşire `pm.out` va conţine $n + 1$ linii. Pe prima linie se va scrie numărul natural t, iar pe fiecare linie k dintre următoarele n linii, se vor scrie cele două numere naturale c_k şi d_k, separate prin câte un spaţiu.

Restricţii şi precizări

Timpul necesar finalizării executării oricărei faze nu va depăşi 1.000.000.
Se consideră că executarea proiectului începe la momentul de timp 0.
Nu vor exista dependenţe circulare (proiectul întotdeauna se poate finaliza).

Exemplu

pm.in
7
2 3 5 3 3 3 2
0
0
1 2
1 1
1 1
3 3 4 5
1 3

pm.out
11
0 3
0 0
3 3
2 5
2 5
8 8
8 9

(Project Management, `https://www.pbinfo.ro/probleme/1101/project-management`)

Rezolvare:

Pentru rezolvarea acestei probleme am utilizat un algoritm pentru sortarea topologică a nodurilor unui graf orientat aciclic.

Graful orientat este reprezentat prin intermediul listelor de vecini, implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică. Un nod al grafului corespunde unei faze a proiectului. Arcul (u, v) corespunde situaţiei în care faza u a proiectului trebuie finalizată înaintea începerii fazei v a proiectului.

Listing 3: `project-management.c`

```
#include <stdio.h>

#define NMAX 301
#define MMAX 9901

typedef struct queue {
           int f;        // indicele primului element din coada
```

```
           int l;        // indicele ultimului element din coada
           int t[NMAX];  // tabloul ce pastreaza elementele cozii
        } QUEUE;

int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int headGT[NMAX];  // headGT[k] - adresa (pozitia) primului vecin
                   //             al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost prelucrat
                   //          = 0 in caz contrar
int dminus[NMAX];  // dminus[i] - gradul interior al nodului i
QUEUE q;           // coada liniara

int topsort[NMAX]; // lista ordonata a nodurilor obtinuta in urma sortarii
                   // topologice
int kth;

int t[NMAX];    // t[k] - timpul necesar finalizarii fazei k
int est[NMAX];  // est[k] - momentul de timp la care poate incepe faza k
                //          cel mai devreme
int lst[NMAX];  // lst[k] - momentul de timp la care poate incepe faza k
                //          cel mai tarziu, fara a influenta durata totala
                //          de executare a proiectului


int min(int x, int y) {
  return (x > y) ? y : x;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

/**
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  q->f = 0;
  q->l = -1;
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return ((q->l + 1) == q->f);
```

```
}

/**
*   Functia adauga in coada q elementul a carui valoare este
*  pastrata in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->l++;
  q->t[q->l] = x;
}

/**
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
  q->f++;
}

/**
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->t[q->f];
}

/*
*  Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v, int head[]) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  FILE *fin, *fout;
  int n, m, i, u, v, p, pft;

  fin = fopen("pm.in", "r");
  fout = fopen("pm.out", "w");

  fscanf(fin, "%d", &n);      // numarul fazelor proiectului

  for (i = 1; i <= n; i++) {
    fscanf(fin, "%d", &t[i]); // timpul necesar finalizarii fazei i
  }

  last = 1;
  for (u = 1; u <= n; u++) {
    fscanf(fin, "%d", &m);   // numarul de faze ce trebuie finalizate
                             // inaintea inceperii fazei u
    dminus[u] = m;           // gradul interior al nodului u
```

```
  for (i = 0; i < m; i++) {
    fscanf(fin, "%d", &v);

    addEdge(v, u, head);   // se adauga nodul u in lista de vecini a lui v
                           // in graful normal
    addEdge(u, v, headGT); // se adauga nodul v in lista de vecini a lui u
                           // in graful transpus
  }
}

init(&q);                  // initializeaza coada
for (u = 1; u <= n; u++) {
  if (dminus[u] == 0) {    // adauga in coada toate fazele care
                           // nu depind de finalizarea altei faze
    visit[u] = 1;          // marchez nodul u ca fiind prelucrat

    enqueue(&q, u);        // adaug nodul u la coada
    est[u] = 0;            // momentul de timp la care poate incepe faza u
                           // cel mai devreme
  }
}

while (!isempty(&q)) {      // cat timp mai sunt elemente in coada
  u = front(&q);            // extrag elementul aflat la inceputul cozii
  dequeue(&q);

  topsort[kth++] = u;       // adaug nodul u in tabloul topsort

  // se parcurge lista de vecini a nodului u
  for (p = head[u]; p != 0; p = next[p]) {
    v = vecin[p];           // se obtine eticheta nodului vecin
    est[v] = max(est[v], est[u] + t[u]); // momentul de timp la care poate
                           // incepe faza v cel mai devreme este maximul
                           // dintre estimarea curenta si momentul de timp
                           // la care poate incepe faza u cel mai devreme
                           // + durata fazei u
    if (!visit[v]) {        // daca nodul v nu a fost prelucrat
      dminus[v]--;          // stergem arcul (u, v)
      if (dminus[v] == 0) { // daca nodul v nu mai are arce ce intra in v
        visit[v] = 1;       // marchez nodul v ca fiind prelucrat
        enqueue(&q, v);     // adaug nodul v in coada
      }
    }
  }
}

// pft - timpul minim in care se poate finaliza executarea proiectului
// este dat de maximul sumelor dintre momentul de timp la care poate incepe
// faza i cel mai devreme + durata fazei i
for (i = 1, pft = 0; i <= n; i++) {
  pft = max(pft, est[i] + t[i]);
}
```

```
  // initializam momentul de timp la care poate incepe faza i cel mai tarziu
  // cu timpul minim in care se poate finaliza executarea proiectului
  for (i = 1; i <= n; i++) {
    lst[i] = pft;
  }

  // parcurgem de la sfarsit la inceput lista nodurilor rezultata in urma
  // sortarii topologice
  for (i = kth; i > 0; i--) {
    v = topsort[i - 1];
    lst[v] -= t[v];            // scadem durata fazei v
    // parcurgem lista de vecini a nodului v in graful transpus
    // practic parcurgem lista de faze ce trebuie incheiate inainte de
    // inceperea fazei v
    for (p = headGT[v]; p != 0; p = next[p]) {
      u = vecin[p];            // se obtine eticheta nodului vecin u

      lst[u] = min(lst[u], lst[v]); // momentul de timp la care poate incepe
                                    // faza u cel mai tarziu este minimul
                                    // dintre valoarea curenta si momentul
                                    // de timp la care poate incepe faza v
    }
  }

  fprintf(fout, "%d\n", pft);
  for (i = 1; i <= n; i++) {
    fprintf(fout, "%d %d\n", est[i], lst[i]);
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

3. Problema `https://open.kattis.com/problems/pickupsticks`

   (Pick up sticks, Waterloo Programming Contest September 27, 2009, `https://open.kattis.com/problems/pickupsticks`)

   Rezolvare:

   Pentru rezolvarea acestei probleme vom utilizat algoritmul lui Kahn [4] pentru sortarea topologică a nodurilor unui graf orientat aciclic.

   Graful orientat este reprezentat prin intermediul listelor de vecini, implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică. Un nod al grafului corespunde unui băţ colorat. Arcul (u, v) corespunde situaţiei în care băţul u se află, în cel puţin un punct, amplasat peste băţul v.

   Listing 4: `pickupsticks.c`

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

```
#define NMAX 1000000
#define MMAX 1000000

typedef struct queue {
           int f;        // indicele primului element din coada
           int l;        // indicele ultimului element din coada
           int t[NMAX];  // tabloul ce pastreaza elementele cozii
        } QUEUE;

int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost vizitat
                   //          = 0 in caz contrar
int dminus[NMAX];  // dminus[i] - gradul interior al nodului i
QUEUE q;           // coada liniara
int trace[NMAX];
int kth;

/**
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  q->f = 0;
  q->l = -1;
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return ((q->l + 1) == q->f);
}

/**
*   Functia adauga in coada q elementul a carui valoare este
*  pastrata in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->l++;
  q->t[q->l] = x;
}

/**
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
```

```
  q->f++;
}

/**
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->t[q->f];
}

/*
*   Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  int n, m, i, u, v, p;

  scanf("%d %d", &n, &m);

  last = 1;
  for (i = 0; i < m; i++) {
    scanf("%d %d", &u, &v);

    // adauga v in lista de vecini a lui u
    addEdge(u, v);
    // incrementeaza gradul interior al nodului v
    dminus[v]++;
  }

  // initializeaza coada
  init(&q);
  for (u = 1; u <= n; u++) {
    // adauga in coada toate nodurile care nu sunt
    // extremitate finala pt vreun arc
    if (dminus[u] == 0) {
      // marchez nodul u ca fiind adaugat in coada
      visit[u] = 1;

      // adaug nodul u in coada
      enqueue(&q, u);
    }
  }

  kth = 0;
  // cat timp mai exista elemente in coada
  while (!isempty(&q)) {
    // se extrage elementul aflat la inceputul cozii
```

```
    u = front(&q);
    dequeue(&q);

    //fprintf(fout, "%d ", u);
    trace[kth++] = u;

    // se parcurge lista de vecini a nodului u
    for (p = head[u]; p != 0; p = next[p]) {
      // se obtine eticheta nodului vecin
      v = vecin[p];
      // daca nodul v nu se afla in coada
      if (!visit[v]) {
        // stergem arcul (u, v)
        dminus[v]--;
        // daca nodul v nu mai are arce ce intra in v
        if (dminus[v] == 0) {
          // marchez nodul v ca fiind adaugat in coada
          visit[v] = 1;
          // adaug nodul v in coada
          enqueue(&q, v);
        }
      }
    }
  }

  if (kth < n) {
    printf("IMPOSSIBLE\n");
  } else {
    for (i = 0; i < kth; i++) {
      printf("%d\n", trace[i]);
    }
  }

  return 0;
}
```

4. Problema `https://open.kattis.com/problems/brexitnegotiations`

   (Brexit Negotiations, Northwestern Europe Regional Contest (NWERC) 2018, `https://open.kattis.com/problems/brexitnegotiations`)

   Rezolvare:

   Pentru rezolvarea acestei probleme se va utiliza algoritmul lui Kahn pentru sortarea topologică a nodurilor unui graf orientat aciclic.

   Se construieşte o sortare topologică $a[1]a[2], \ldots, a[n]$ astfel încât valoarea

   $$\max\{e[a[i]] + i - 1\}$$

   să fie minimă dintre toate variantele de sortare topologică.

   Se va folosi o coadă de priorităţi în loc de coada obişnuită, coada fiind un *min-heap* după valoarea lui $e[i]$.

Spre deosebire de algoritmii prezentaţi în secţiunea 3, sortarea topologică se construieşte de la sfârşit la început.

Listing 5: `brexitnegotiations.c`

```
#include <stdio.h>

#define NMAX 400001
#define MMAX 800001

typedef struct heap {
          int n;        // numarul de elemente din heap
          int a[NMAX];  // valorile nodurilor aflate in heap
        } HEAP;


int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost vizitat
                   //          = 0 in caz contrar
int dplus[NMAX];   // dplus[i] - gradul exterior al nodului i

int e[NMAX];       // e[i] -
HEAP h;            // min-heap
int kth;


int max(int a, int b) {
  return ((a > b) ? a : b);
}

/*
*  Initializeaza heap-ul vid.
*/
void init(HEAP* h) {
  h->n = 0;
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(HEAP* h) {
  return (h->n == 0);
}

/*
*   Se reorganizeaza datele din structura de heap h pentru nodul start.
```

```
* Pornind de la elementul aflat pe pozitia start, se verifica conditia
* de min-heap, si in cazul neindeplinirii, se cauta o pozitie pentru
* acesta, coborandu-se in cadrul subarborelui de radacina start.
*  @param h - pointer catre o zona de memorie ce pastreaza informatii de
*             tip HEAP
*  @param start - indicele nodului reprezentand radacina unui subarbore
*  @param m - numarul de elemente din heap
*/
void push(HEAP* h, int start, int m, int d[]) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (d[h->a[j]] > d[h->a[j + 1]])) {
      j++;
    }

    if (d[h->a[i]] > d[h->a[j]]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

/*
*  Construieste un heap din multimea de elemente 1..n.
*/
void buildheap(HEAP* h, int d[]) {
  int i;

  for (i = h->n / 2; i > 0; i--) {
    push(h, i, h->n, d);
  }
}

/*
*   Se sterge elementul aflat in varful min-heap-ului h: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de min-heap.
*   Functia returneaza valoarea initiala pastrata in varful min-heap-ului
* (valoarea memorata in nodul radacina inainte de stergere).
*/
int deletemin(HEAP* h, int d[]) {
  int k = h->a[1];

  h->a[1] = h->a[h->n--];
```

```
  push(h, 1, h->n, d);

  return k;
}

/*
*   Promoveaza in heap nodul i pentru a se restabili proprietatea
* de min-heap. Elementul de pe pozitia i va fi promovat in heap
* atata timp cat nu se verifica proprietatea de min-heap, putand ajunge
* pana in radacina.
*/
void liftup(HEAP* h, int i, int d[]) {
  int j, aux;

  j = i >> 1;
  while ((0 < j) && (d[h->a[j]] > d[h->a[i]])) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

    i = j;
    j = i >> 1;
  }
}

/*
*  Adauga un element in heap.
*/
void addheap(HEAP* h, int d[], int u) {
  // adaugam nodul u pe ultima pozitie din heap
  h->a[++h->n] = u;

  liftup(h, h->n, d);
}

/*
*   Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  int n, m, i, j, u, v, p, cost;

  scanf("%d", &n);

  last = 1;
  for (i = 1; i <= n; i++) {
    scanf("%d %d", &e[i], &m);
```

```
  for (j = 0; j < m; j++) {
    scanf("%d", &v);

    // adauga v in lista de vecini a lui i
    // vecinii unui nod u inseamna aici, nodurile care au un arc catre u
    addEdge(i, v);
    // incrementeaza gradul exterior al nodului v
    dplus[v]++;
  }
}

for (u = 1; u <= n; u++) {
  // adauga in coada toate nodurile care nu au
  // arce care pleaca din ele
  if (dplus[u] == 0) {
    // marchez nodul u ca fiind adaugat in coada
    visit[u] = 1;

    // adaug nodul u in coada
    // adaugam nodul u pe ultima pozitie din heap
    h.a[++h.n] = u;
  }
}

// initializeaza coada - organizam structura de min-heap
buildheap(&h, e);

cost = 0;
kth = n - 1;
// cat timp mai exista elemente in coada
while (!isempty(&h)) {
  // se extrage elementul aflat la inceputul cozii
  u = deletemin(&h, e);

  //printf("%d ", u);
  cost = max(cost, e[u] + kth);
  kth--;

  // se parcurge lista de vecini a nodului u
  for (p = head[u]; p != 0; p = next[p]) {
    // se obtine eticheta nodului vecin
    v = vecin[p];
    // daca nodul v nu se afla in coada
    if (!visit[v]) {
      // stergem arcul (v, u)
      dplus[v]--;
      // daca nodul v nu mai are arce ce il parasesc
      if (dplus[v] == 0) {
        // marchez nodul v ca fiind adaugat in coada
        visit[v] = 1;
        // adaug nodul v in coada
        addheap(&h, e, v);
```

```
        }
      }
    }
  }

  printf("%d\n", cost);

  return 0;
}
```

5. Problema `https://www.pbinfo.ro/probleme/2768/sorttopminlex`

   (SortTopMinLex, `https://www.pbinfo.ro/probleme/2768/sorttopminlex`)

   Rezolvare:

   Graful orientat este reprezentat prin intermediul listelor de vecini, implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică.

   Vom utiliza *algoritmul lui Kahn* modificat: în loc de coada obişnuită se va folosi o coadă de priorităţi, implementată printr-un *min-heap* după valoarea etichetei unui nod.

Listing 6: `sorttopminlex.c`

```
#include <stdio.h>

#define NMAX 100001
#define MMAX 500001

typedef struct heap {
          int n;        // numarul de elemente din heap
          int a[NMAX];  // valorile nodurilor aflate in heap
        } HEAP;

typedef struct queue {
           HEAP h;
        } QUEUE;

int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost vizitat
                   //          = 0 in caz contrar
int dminus[NMAX];  // dminus[i] - gradul interior al nodului i
QUEUE q;           // coada cu prioritati


/*
*  Initializeaza heap-ul vid.
*/
void initheap(HEAP* h) {
```

```
  h->n = 0;
}

/*
*   Se reorganizeaza datele din structura de heap h pentru nodul start.
* Pornind de la elementul aflat pe pozitia start, se verifica conditia
* de min-heap, si in cazul neindeplinirii, se cauta o pozitie pentru
* acesta, coborandu-se in cadrul subarborelui de radacina start.
*  @param h - pointer catre o zona de memorie ce pastreaza informatii de
*             tip HEAP
*  @param start - indicele nodului reprezentand radacina unui subarbore
*  @param m - numarul de elemente din heap
*/
void push(HEAP* h, int start, int m) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (h->a[j] > h->a[j + 1])) {
      j++;
    }

    if (h->a[i] > h->a[j]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

/*
*   Se sterge elementul aflat in varful min-heap-ului h: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de min-heap.
*   Functia returneaza valoarea initiala pastrata in varful min-heap-ului
* (valoarea memorata in nodul radacina inainte de stergere).
*/
int deletemin(HEAP* h) {
  int k = h->a[1];

  h->a[1] = h->a[h->n];
  h->n--;

  push(h, 1, h->n);

  return k;
}
```

```
/*
*   Promoveaza in heap nodul i pentru a se restabili proprietatea
* de min-heap. Elementul de pe pozitia i va fi promovat in heap
* atata timp cat nu se verifica proprietatea de min-heap, putand ajunge
* pana in radacina.
*/
void liftup(HEAP* h, int i) {
  int j, aux;

  j = i >> 1;
  while ((0 < j) && (h->a[j] > h->a[i])) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

    i = j;
    j = i >> 1;
  }
}

/**
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  initheap(&q->h);
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return (q->h.n == 0);
}

/**
*   Functia adauga in coada q elementul a carui valoare este
*  pastrata in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->h.n++;
  q->h.a[q->h.n] = x;

  liftup(&q->h, q->h.n);
}

/**
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
  deletemin(&q->h);
}
```

```
/**
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->h.a[1];
}

/*
*   Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  int n, m, i, u, v, p;

  scanf("%d %d", &n, &m);

  last = 1;
  for (i = 0; i < m; i++) {
    scanf("%d %d", &u, &v);

    addEdge(u, v);            // adauga v in lista de vecini a lui u

    dminus[v]++;              // incrementeaza gradul interior al nodului v
  }

  init(&q);                   // initializeaza coada
  for (u = 1; u <= n; u++) {
    if (dminus[u] == 0) {     // adauga in coada toate nodurile care
                              // nu sunt extremitate finala pt vreun arc
      visit[u] = 1;           // marchez nodul u ca fiind adaugat in coada

      enqueue(&q, u);         // adaug nodul u in coada
    }
  }

  while (!isempty(&q)) {      // cat timp mai exista elemente in coada
    u = front(&q);            // se extrage elementul aflat la inceputul
        cozii
    dequeue(&q);

    printf("%d ", u);

    // se parcurge lista de vecini a nodului u
    for (p = head[u]; p != 0; p = next[p]) {
      v = vecin[p];           // se obtine eticheta nodului vecin
      if (!visit[v]) {        // daca nodul v nu se afla in coada
        dminus[v]--;          // stergem arcul (u, v)
```

```
        if (dminus[v] == 0) { // daca nodul v nu mai are arce ce intra in v
          visit[v] = 1;       // marchez nodul v ca fiind adaugat in coada

          enqueue(&q, v);     // adaug nodul v in coada
        }
      }
    }
  }

  return 0;
}
```

6. Problema `https://www.pbinfo.ro/probleme/2309/competitie`

(Competitie, ONI 2001, clasa a IX-a, `https://www.pbinfo.ro/probleme/2309/competitie`)

Rezolvare:

Graful orientat este reprezentat prin intermediul listelor de vecini, implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică. Un nod u al grafului semnifică concurentul cu numărul de concurs u. Arcul (u, v) corespunde situaţiei în care concurentul cu numărul de concurs u a fost situat în clasamentul final înaintea concurentului cu numărul de concurs v.

Vom utiliza *algoritmul lui Kahn* modificat: se va folosi o coadă de priorităţi în loc de coada obişnuită, coada fiind un *min-heap* după valoarea etichetei unui nod.

Listing 7: `competitie.c`

```
#include <stdio.h>

#define NMAX 1001
#define MMAX 1001

typedef struct heap {
          int n;        // numarul de elemente din heap
          int a[NMAX];  // valorile nodurilor aflate in heap
        } HEAP;

typedef struct queue {
           HEAP h;
        } QUEUE;

int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost vizitat
                   //          = 0 in caz contrar
int dminus[NMAX];  // dminus[i] - gradul interior al nodului i
QUEUE q;           // coada cu prioritati
```

```
/*
*  Initializeaza heap-ul vid.
*/
void initheap(HEAP* h) {
  h->n = 0;
}

/*
*   Se reorganizeaza datele din structura de heap h pentru nodul start.
* Pornind de la elementul aflat pe pozitia start, se verifica conditia
* de min-heap, si in cazul neindeplinirii, se cauta o pozitie pentru
* acesta, coborandu-se in cadrul subarborelui de radacina start.
*  @param h - pointer catre o zona de memorie ce pastreaza informatii de
*             tip HEAP
*  @param start - indicele nodului reprezentand radacina unui subarbore
*  @param m - numarul de elemente din heap
*/
void push(HEAP* h, int start, int m) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (h->a[j] > h->a[j + 1])) {
      j++;
    }

    if (h->a[i] > h->a[j]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

/*
*   Se sterge elementul aflat in varful min-heap-ului h: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de min-heap.
*   Functia returneaza valoarea initiala pastrata in varful min-heap-ului
* (valoarea memorata in nodul radacina inainte de stergere).
*/
int deletemin(HEAP* h) {
  int k = h->a[1];

  h->a[1] = h->a[h->n];
  h->n--;
```

```
  push(h, 1, h->n);

  return k;
}

/*
*   Promoveaza in heap nodul i pentru a se restabili proprietatea
* de min-heap. Elementul de pe pozitia i va fi promovat in heap
* atata timp cat nu se verifica proprietatea de min-heap, putand ajunge
* pana in radacina.
*/
void liftup(HEAP* h, int i) {
  int j, aux;

  j = i >> 1;
  while ((0 < j) && (h->a[j] > h->a[i])) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

    i = j;
    j = i >> 1;
  }
}

/**
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  initheap(&q->h);
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return (q->h.n == 0);
}

/**
*   Functia adauga in coada q elementul a carui valoare este
*  pastrata in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->h.n++;
  q->h.a[q->h.n] = x;

  liftup(&q->h, q->h.n);
}

/**
*  Functia sterge din coada q elementul situat la inceputul cozii.
```

```
*/
void dequeue(QUEUE* q) {
  deletemin(&q->h);
}

/**
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->h.a[1];
}

/*
*  Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  FILE *fin, *fout;
  int n, m, i, u, v, p;

  fin = fopen("competitie.in", "r");
  fout = fopen("competitie.out", "w");

  fscanf(fin, "%d %d", &n, &m);

  last = 1;
  for (i = 0; i < m; i++) {
    fscanf(fin, "%d %d", &u, &v);

    addEdge(u, v);            // adauga v in lista de vecini a lui u

    dminus[v]++;              // incrementeaza gradul interior al nodului v
  }

  init(&q);                   // initializeaza coada
  for (u = 1; u <= n; u++) {
    if (dminus[u] == 0) {     // adauga in coada toate nodurile care
                              // nu sunt extremitate finala pt vreun arc
      visit[u] = 1;           // marchez nodul u ca fiind adaugat in coada

      enqueue(&q, u);         // adaug nodul u in coada
    }
  }

  while (!isempty(&q)) {      // cat timp mai exista elemente in coada
    u = front(&q);            // extrag elementul aflat la inceputul cozii
    dequeue(&q);
```

```
    fprintf(fout, "%d ", u);

    // se parcurge lista de vecini a nodului u
    for (p = head[u]; p != 0; p = next[p]) {
      v = vecin[p];           // se obtine eticheta nodului vecin
      if (!visit[v]) {        // daca nodul v nu se afla in coada
        dminus[v]--;          // stergem arcul (u, v)
        if (dminus[v] == 0) { // daca nodul v nu mai are arce ce intra in v
          visit[v] = 1;       // marchez nodul v ca fiind adaugat in coada

          enqueue(&q, v);     // adaug nodul v in coada
        }
      }
    }
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

7. Problema `https://www.pbinfo.ro/probleme/2123/relatii`

   (Relatii, Olimpiada Municipala de Informatica, Iasi, 2015, Clasa a X-a,`https://www.pbinfo.ro/probleme/2123/relatii`)

   Rezolvare:

   Graful orientat este reprezentat prin intermediul listelor de vecini, implementate sub forma unor liste liniare simplu înlănţuite, alocarea elementelor fiind statică. Un nod u al grafului reprezintă o variabilă. Arcul (u, v) corespunde situaţiei în care avem $u < v$ (variabila u este mai mică decât variabila v).

   Vom utiliza *algoritmul lui Kahn* modificat: se va folosi o coadă de priorităţi în loc de coada obişnuită, coada fiind un *min-heap* după valoarea etichetei unui nod.

   Listing 8: `relatii.c`

```
#include <stdio.h>

#define ALPHA 27
#define NMAX ALPHA+1
#define MMAX 201

typedef struct heap {
          int n;         // numarul de elemente din heap
          int a[NMAX];   // valorile nodurilor aflate in heap
        } HEAP;

typedef struct queue {
           HEAP h;
        } QUEUE;
```

```
int vecin[MMAX];   // vecin[i] - eticheta nodului situat pe pozitia i
                   //            in lista de vecini
int next[MMAX];    // next[i] - adresa urmatorului vecin
int head[NMAX];    // head[k] - adresa (pozitia) primului vecin
                   //           al nodului k in tabloul vecin
int last;          // numarul total de elemente

char visit[NMAX];  // visit[k] = 1 daca nodul k a fost vizitat
                   //          = 0 in caz contrar
int dminus[NMAX];  // dminus[i] - gradul interior al nodului i
QUEUE q;           // coada cu prioritati


/*
*  Initializeaza heap-ul vid.
*/
void initheap(HEAP* h) {
  h->n = 0;
}

/*
*   Se reorganizeaza datele din structura de heap h pentru nodul start.
* Pornind de la elementul aflat pe pozitia start, se verifica conditia
* de min-heap, si in cazul neindeplinirii, se cauta o pozitie pentru
* acesta, coborandu-se in cadrul subarborelui de radacina start.
*  @param h - pointer catre o zona de memorie ce pastreaza informatii de
*             tip HEAP
*  @param start - indicele nodului reprezentand radacina unui subarbore
*  @param m - numarul de elemente din heap
*/
void push(HEAP* h, int start, int m) {
  int i, j, aux;

  i = start;
  j = i << 1;
  while (j <= m) {
    if ((j < m) && (h->a[j] > h->a[j + 1])) {
      j++;
    }

    if (h->a[i] > h->a[j]) {
      aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

      i = j;
      j = i << 1;
    } else {
        j = m + 1;
    }
  }
}

/*
```

```
*   Se sterge elementul aflat in varful min-heap-ului h: pe prima pozitie
* este adus ultimul element, se micsoreaza numarul de elemente cu 1 si
* se reorganizeaza structura de date a.i. sa se pastreze proprietatea
* de min-heap.
*   Functia returneaza cea mai mica valoare din min-heap (valoarea aflata
* in radacina inainte de stergere).
*/
int deletemin(HEAP* h) {
  int k = h->a[1];

  h->a[1] = h->a[h->n];
  h->n--;

  push(h, 1, h->n);

  return k;
}

/*
*   Promoveaza in heap nodul i pentru a se restabili proprietatea
* de min-heap. Elementul de pe pozitia i va fi promovat in heap
* atata timp cat nu se verifica proprietatea de min-heap, putand ajunge
* pana in radacina.
*/
void liftup(HEAP* h, int i) {
  int j, aux;

  j = i >> 1;
  while ((0 < j) && (h->a[j] > h->a[i])) {
    aux = h->a[i]; h->a[i] = h->a[j]; h->a[j] = aux;

    i = j;
    j = i >> 1;
  }
}

/**
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  initheap(&q->h);
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return (q->h.n == 0);
}

/**
```

```
*   Functia adauga in coada q elementul a carui valoare este
*  pastrata in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->h.n++;
  q->h.a[q->h.n] = x;

  liftup(&q->h, q->h.n);
}

/**
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
  deletemin(&q->h);
}

/**
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->h.a[1];
}

/*
*  Functia adauga nodul v la lista de vecini a nodului u.
*/
void addEdge(int u, int v) {
  vecin[last] = v;
  next[last] = head[u];
  head[u] = last++;
}

int main() {
  FILE *fin, *fout;
  int n, m, i, u, v, p;
  char txt[4];

  fin = fopen("relatii.in", "r");
  fout = fopen("relatii.out", "w");

  fscanf(fin, "%d %d", &n, &m);

  last = 1;
  for (i = 0; i < m; i++) {
    fscanf(fin, "%s", txt);
    u = txt[0] - 'a' + 1;
    v = txt[2] - 'a' + 1;

    if (txt[1] == '<') {
      addEdge(u, v);          // adauga v in lista de vecini a lui u
```

```
      dminus[v]++;            // incrementeaza gradul interior al nodului v
    } else {
        addEdge(v, u);        // adauga u in lista de vecini a lui v

        dminus[u]++;          // incrementeaza gradul interior al nodului u
    }
  }

  init(&q);                   // initializeaza coada
  for (u = 1; u <= n; u++) {
    if (dminus[u] == 0) {     // adauga in coada toate nodurile care
                              // nu sunt extremitate finala pt vreun arc
      visit[u] = 1;           // marchez nodul u ca fiind adaugat in coada

      enqueue(&q, u);         // adaug nodul u in coada
    }
  }

  while (!isempty(&q)) {      // cat timp mai exista elemente in coada
    u = front(&q);            // extrag elementul aflat la inceputul cozii
    dequeue(&q);

    fprintf(fout, "%c", (char)(u - 1 + 'a'));

    // se parcurge lista de vecini a nodului u
    for (p = head[u]; p != 0; p = next[p]) {
      v = vecin[p];           // se obtine eticheta nodului vecin
      if (!visit[v]) {        // daca nodul v nu se afla in coada
        dminus[v]--;          // stergem arcul (u, v)
        if (dminus[v] == 0) { // daca nodul v nu mai are arce ce intra in v
          visit[v] = 1;       // marchez nodul v ca fiind adaugat in coada

          enqueue(&q, v);     // adaug nodul v in coada
        }
      }
    }
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

8. Pacman s-a apucat să înveţe Excel(sior), un program open-source de calcul tabelar, similar (ca denumire şi funcţionalitate) cu un alt program de calcul tabelar implementat de o corporaţie malefică cu numele de M... .

   Dar pe Pacman nu-l preocupă aceste lucruri. El s-a apucat să completeze primele N celule ale primei coloane. O celulă poate fi completată fie cu o valoare (un număr întreg), fie cu o sumă de alte celule completate.

   După ce a completat toate celulele, Pacman se întreabă care va fi valoarea celulei de

pe linia L.

Dându-se N, conţinutul celor N celule şi L, să se calculeze valoarea celulei de pe linia L.

Date de intrare

Fişierul de intrare `pacman.in` va conţine:

- pe prima linie un număr natural N, reprezentând numărul de celule completate;
- pe următoarele N linii vor fi descrise cele N celule astfel:
  - caracterul `N`, urmat de întregul val: pentru o celulă ce conţine un număr întreg val;
  - caracterul `S`, urmat de întregii K l_1 l_2 ... l_K: pentru o celulă ce conţine o sumă de K alte celule aflate pe liniile l_1, l_2, ... l_K;
  - pe ultima linie un număr natural L, reprezentând linia celulei de interes pentru Pacman.

Date de ieşire

În fişierul de ieşire `pacman.out` se va găsi valoarea celulei de pe linia L.

Restricţii şi precizări

- $1 \leq L \leq N \leq 100000$
- $1 \leq K \leq 10$, pentru toate celulele care conţin o sumă.
- $1 \leq l_1, l_2, \ldots, l_K \leq N$, $1 \leq val \leq 100000000$, pentru orice valoare a unei celule - fie dată direct de Pacman, fie obţinută ca sumă.
- Pentru o celulă completată cu o sumă, valorile sale $l_1, l_2, \ldots l_K$ nu sunt neapărat distincte două câte două. Altfel spus, într-o sumă termenii se pot repeta.
- O celulă nu depinde de ea însăşi direct sau indirect. Altfel spus, orice celulă poate fi calculată pe baza valorilor altor celule calculate anterior.

Exemplu

pacman.in
5 N 1 N 1 S 2 1 2 S 2 5 3 S 2 2 3 4

pacman.out
5

(Pacman, Olimpiada pe scoala 2016, `https://www.nerdarena.ro/problema/pacman`)

Rezolvare:
Avem un graf orientat aciclic $G = (V, E)$, unde V este alcătuită din mulţimea de celule, iar E conţine arcele (u, v) dacă celula v se află în lista de celule de care depinde calcularea valorii celulei u.

Se iniţiază o parcurgere în adâncime (DFS) a grafului de dependenţe pornind din celula situată la intersecţia dintre linia L şi prima coloană.

Se evită parcurgerea de două ori a unei celule deja vizitată, prin marcarea faptului că s-a calculat, la un pas anterior, valoarea celulei respective.

Listing 9: pacman_v1.c

```
#include <stdio.h>

#define NMAX 100000
#define MMAX 1000000

int head[NMAX + 1];  // Lista liniara simplu inlantuita in care se pastreaza
                     // celulele de ale caror valori depinde calcularea
                     // valorii din celula curenta.
                     // head[k] - adresa (pozitia) primului vecin al celulei
                     //           k in tabloul vecin
int vecin[MMAX + 1]; // vecin[i] - eticheta celulei situata pe pozitia i
                     //            in lista de vecini
int next[MMAX + 1];  // next[i] - adresa urmatorului vecin
int m;

int cell[NMAX + 1];        // valorile celulelor de pe prima coloana
char isComputed[NMAX + 1]; // isComputed[i] = 0 daca valoarea celulei i
                           //        depinde de alte celule si nu a fost
                           //        inca calculata.
                           //        = 1, daca avem valoarea celulei i
char command[2];

/*
*  Functia adauga celula v la lista de vecini a celulei u.
*/
void add(int u, int v) {
  vecin[m] = v;
  next[m] = head[u];

  head[u] = m++;
}

/*
*  Functia intoarce valoarea calculata a celului u.
*/
int getCellValue(int u) {
  int p, v;

  // daca valoarea celulei u nu a fost inca calculata
  if (!isComputed[u]) {
    // parcurgem toate celulele ale caror valori contribuie la valoarea
    // celulei curente
    for (p = head[u]; p; p = next[p]) {
      v = vecin[p];               // se obtine indicele celulei v

      cell[u] += getCellValue(v); // se adauga valoarea celulei v
                                  // la valoarea celulei curente
    }
```

```
    isComputed[u] = 1;              // se marcheaza faptul ca s-a calculat
                                    // valoarea celulei curente
  }

  return cell[u];                   // se returneaza valoarea celulei curente
}

int main() {
  FILE *fin, *fout;
  int n, i, j, k, l;

  fin = fopen("pacman.in", "r");
  fout = fopen("pacman.out", "w");

  m = 1;
  fscanf(fin, "%d", &n);
  for (i = 1; i <= n; i++) {
    fscanf(fin, "%s", command); // se citeste un sir de caractere

    if (command[0] == 'N') {
      fscanf(fin, "%d", &cell[i]); // se citeste valoarea dintr-o celula
    } else {
        fscanf(fin, "%d", &k);      // numarul de celule de care depinde
        for (j = 0; j < k; j++) {
          fscanf(fin, "%d", &l);    // se citeste numarul unei celule

          add(i, l);                // adauga l in lista de vecini a celulei i
        }
    }
  }

  fscanf(fin, "%d", &l);
  fprintf(fout, "%d\n", getCellValue(l));

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Pentru cea de-a doua variantă de rezolvare vom utiliza tot un graf orientat aciclic $G = (V, E)$, unde V este alcătuită din mulţimea de celule, iar E conţine arcele (u, v) dacă celula u se află în lista de celule de care depinde calcularea valorii celulei v. Acest graf corespunde grafului transpus de la prima variantă de rezolvare.

Soluţia de bazează pe algoritmul lui Kahn pentru determinarea unei sortări topologice [3].

Listing 10: `pacman_v2.c`

```
#include <stdio.h>

#define NMAX 100000
```

```
#define MMAX 1000000

typedef struct queue {
          int h;           // indicele primului element din coada
          int t;           // indicele ultimului element din coada
          int a[NMAX + 1]; // tabloul ce pastreaza elementele cozii
        } QUEUE;

int head[NMAX + 1];  // Lista liniara simplu inlantuita in care se pastreaza
                     // celulele ale caror valori depind de calcularea
                     // valorii din celula curenta.
                     // head[k] - adresa (pozitia) primului vecin al celulei
                     //           k in tabloul vecin
int vecin[MMAX + 1]; // vecin[i] - eticheta celulei situata pe pozitia i
                     //            in lista de vecini
int next[MMAX + 1];  // next[i] - adresa urmatorului vecin
int m;

int inDegree[NMAX + 1]; // inDegree[u] - gradul interior al nodului u
QUEUE q;                // coada
int cell[NMAX + 1];     // valorile celulelor de pe prima coloana
char command;

/*
*  Functia adauga celula v la lista de vecini a celulei u.
*/
void add(int u, int v) {
  vecin[m] = v;
  next[m] = head[u];

  head[u] = m++;
  inDegree[v]++;
}

/*
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  q->h = 0;
  q->t = -1;
}

/*
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return (q->t + 1 == q->h);
}

/*
*   Functia adauga in coada q elementul a carui valoare este pastrata
```

```
*  in variabila x.
*/
void enqueue(QUEUE* q, int x) {
  q->a[++q->t] = x;
}

/*
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
  q->h++;
}

/*
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
int front(QUEUE* q) {
  return q->a[q->h];
}

int main() {
  FILE *fin, *fout;
  int n, i, j, k, l, u, v, p;

  fin = fopen("pacman.in", "r");
  fout = fopen("pacman.out", "w");

  m = 1;
  fscanf(fin, "%d ", &n);
  for (i = 1; i <= n; i++) {
    fscanf(fin, "%c ", &command); // se citeste un sir de caractere

    if (command == 'N') {
      fscanf(fin, "%d ", &cell[i]); // se citeste valoarea dintr-o celula
    } else {
        fscanf(fin, "%d ", &k);    // numarul de celule de care depinde
        for (j = 0; j < k; j++) {
          fscanf(fin, "%d ", &l);  // se citeste numarul unei celule

          add(l, i);               // adauga i in lista de vecini a celulei l
        }
    }
  }

  init(&q);                    // se initializeaza coada
  for (i = 1; i <= n; i++) {   // parcurgem toate celulele
    if (inDegree[i] == 0) {    // daca celula curenta nu depinde de alte
                               // celule
      enqueue(&q, i);          // atunci adaugam celula i in coada Q
    }
  }
```

```
  while (!isempty(&q)) {       // cat timp coada Q nu este vida
    u = front(&q);             // extragem o celula aflata in varful cozii
    dequeue(&q);

    // parcurgem toate celulele ale caror valori depind de valoarea
    // celulei curente u
    for (p = head[u]; p; p = next[p]) {
      v = vecin[p];            // se obtine indicele celulei v

      cell[v] += cell[u];      // se adauga valoarea celulei u la valoarea
                               // celulei v
      inDegree[v]--;           // stergem dependenta (u,v)
      if (inDegree[v] == 0) {  // daca toate celulele de care depinde v
                               // au fost calculate
        enqueue(&q, v);        // adaugam celula v in coada Q
      }
    }
  }
  fscanf(fin, "%d", &l);
  fprintf(fout, "%d\n", cell[l]);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

9. Problema `https://www.pbinfo.ro/probleme/2155/facebook-fmi`

(Facebook_FMI, Admitere FMI Bucuresti, 2016, `https://www.pbinfo.ro/probleme/2155/facebook-fmi`)

Rezolvare:

Pentru rezolvarea acestei probleme vom utiliza un algoritm obţinut prin modificarea *algoritmului lui Kahn*.

Listing 11: `facebook-fmi.c`

```
#include <stdio.h>

#define NMAX 1000

typedef struct queue {
           short f;        // indicele primului element din coada
           short l;        // indicele ultimului element din coada
           short t[NMAX + 1];  // tabloul ce pastreaza elementele cozii
         } QUEUE;

char vecin[NMAX + 1][NMAX + 1]; // matricea de adiacenta
char visit[NMAX + 1];           // visit[k] = 1 daca nodul k a fost prelucrat
                                //          = 0 in caz contrar
int d[NMAX + 1];                // d[u] - gradul nodului u
QUEUE q;                        // coada liniara
```

```
/**
*  Functia initializeaza coada q ca fiind vida.
*/
void init(QUEUE* q) {
  q->f = 0;
  q->l = -1;
}

/**
*   Functia verifica daca coada q este vida. Intoarce valoarea 1
*  daca coada este vida sau 0 daca coada contine cel putin un element.
*/
int isempty(QUEUE* q) {
  return ((q->l + 1) == q->f);
}

/**
*   Functia adauga in coada q elementul a carui valoare este pastrata
*  in variabila x.
*/
void enqueue(QUEUE* q, short x) {
  q->l++;
  q->t[q->l] = x;
}

/**
*  Functia sterge din coada q elementul situat la inceputul cozii.
*/
void dequeue(QUEUE* q) {
  q->f++;
}

/**
*  Functia intoarce valoarea elementului situat la inceputul cozii q.
*/
short front(QUEUE* q) {
  return q->t[q->f];
}

int main() {
  FILE *fin, *fout;
  int n, m, k, i, u, v;

  fin = fopen("fb_fmi.in", "r");
  fout = fopen("fb_fmi.out", "w");

  fscanf(fin, "%d %d %d", &n, &m, &k);
  for (i = 0; i < m; i++) {
    fscanf(fin, "%d %d", &u, &v);

    vecin[u][v] = vecin[v][u] = 1; // relatia de prietenie este simetrica
```

```
    d[u]++;       // se incrementeaza gradul nodului u
    d[v]++;       // se incrementeaza gradul nodului v
  }

  init(&q);       // se initializeaza coada

  m = n;                      // refolosim variabila m; va pastra
                              // numarul de persoane in multimea initiala S
  for (u = 1; u <= n; u++) { // pentru toate persoanele din grup
    if (d[u] < k) {           // daca persoana u are mai putin de k prieteni
      enqueue(&q, u);         // se adauga persoana u intr-o coada
      visit[u] = 1;           // marcam ca u este prelucrata
      m--;                    // decrementam numarul de persoane din
                              // multimea S
    }
  }

  while (!isempty(&q)) {      // cat timp mai sunt elemente in coada
    u = front(&q);            // se extrage elementul aflat la inceputul cozii
    dequeue(&q);

    // pentru toate persoanele v, cunoscute de Ionut
    for (v = 1; v <= n; v++) {
      // daca persoana v este prietena cu persoana u
      if (vecin[u][v] == 1) {
        vecin[u][v] = vecin[v][u] = 0;  // stergem relatia de prietenie
        d[u]--;                         // decrementam numarul de prieteni
        d[v]--;                         // pentru persoanele u si v

        // daca persoana v are mai putin de k prieteni in S si
        // daca persoana v nu a fost prelucrata
        if ((d[v] < k) && (visit[v] == 0)) {
          enqueue(&q, v);   // se adauga persoana v la coada
          visit[v] = 1;     // se marcheaza faptul ca v este prelucrata
          m--;              // decrementam numarul de persoane din multimea S
        }
      }
    }
  }

  fprintf(fout, "%d\n", m);     // se afiseaza valoarea lui m
  for (i = 1; i <= n; i++) {
    if (visit[i] == 0) {
      fprintf(fout, "%d ", i);  // se afiseaza ID-ul fiecarei persoane din S
    }
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introducere în Algoritmi*, Computer Libris Agora, Cluj-Napoca, 1999.

[2] M. Coşulschi, M. Gabroveanu, *Practica programării în C*, Editura Universitaria, Craiova, 2014.

[3] M. Coşulschi, M. Gabroveanu, *Algoritmica grafurilor. Aplicaţii în limbajul C*, Editura Sitech, Craiova, 2021.

[4] A. B. Kahn, *Topological sorting of large networks*, Communications of the ACM, vol. 5(11), pp. 558-562, 1962.

[5] R. E. Tarjan, *Edge-disjoint spanning trees and depth-first search*, Algorithmica, vol. 6(2), pp. 171-185, 1976.