

Notite examen - Algoritmi Fundamentali

Dinu Florin-Silviu
grupa 231

Contents

1	Notiuni fundamentale	1
2	Parcurgeri	1
3	Sortare topologica	2
4	Muchii critice	3
5	Puncte critice	4
6	Conexitate in graf orientat	5
7	APM - arbori partiali de cost minim	7
8	Drumuri minime de sursa unica s	11
9	Drumuri minime intre toate perechile de varfuri	17
10	Flow	18

1 Notiuni fundamentale

Grafuri izomorfe

Daca 2 grafuri:

- Au acelasi numar de noduri
- Au acelasi numar de muchii
- Nodurile formeaza o secventa cu acelasi grad
- Daca un graf are un ciclu de lungime k , si celalalt graf are la fel

Grafuri bipartite

Daca un graf:

- Neorientat
- Putem imparti nodurile in 2 multimi $V = V_1 \cup V_2$ cu $V_1 \cap V_2 = \emptyset$
- Fiecare muchie are o extremitate intr-o parte si cealalta in cealalta parte, adica $|e \cap V_1| = |e \cap V_2| = 1$

Construire graf cu secventa gradelor (Havel-Hakimi)

Ordonam descrescator gradele si incepem sa trasam muchii catre urmatoarele, scazand din gradele lor pana cand ajungem sa avem doar 0. Daca avem $d_i < 0$ sau $d_i > n - 1$ sau $(\sum_{i=0}^n d_i) \% 2 = 1$ atunci nu are solutie.

2 Parcurgeri

BFS

Se iau toti vecinii nevizitati, se pun in coada, se continua cu urmatorul din coada. Complexitate $O(V + E)$. Se foloseste pentru iesirea din labirint cu drum minim si calcul nivel.

```
void bfs(vector<vector<int>> &lista, vector<int> &pbfs, queue<int> &q)
{
    int s = q.front();
    q.pop();

    int nivel = pbfs[s] + 1;

    for (auto x: lista[s]) {
        if (pbfs[x] == -1) {
            pbfs[x] = nivel;
            q.push(x);
        }
    }

    if (q.empty()) return;

    bfs(lista, pbfs, q);
}
```

- Muchiile de arbore sunt muchii din pădurea de adâncime G_π . Muchia (u, v) este o muchie de arbore dacă v a fost descoperit explorând muchia (u, v) .

- Muchiile înapoi sunt acele muchii (u, v) care unesc un vârf u cu un strămoș v într-un arbore de adâncime. Buclele (muchii de la un vârf la el însuși) care pot apărea într-un graf orientat sunt considerate muchii înapoi.
- Muchiile înainte sunt acele muchii (u, v) ce nu sunt muchii de arbore și conectează un vârf u cu un descendent v într-un arbore de adâncime.
- Muchiile transversale sunt toate celelalte muchii. Ele pot uni vârfuri din același arbore de adâncime, cu condiția ca unul să nu fie strămoșul celuilalt, sau pot uni vârfuri din arbori, de adâncime, diferiți.

DFS

Complexitate $O(V + E)$.

```
void Graph::DFS(int v)
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}
```

3 Sortare topologica

Se face pe DAG (directed acyclic graph)

Cu BFS

Începe de la toate nodurile care au $indegree == 0$ (DAG garantează cel puțin 1 astfel de nod). După ce scoatem nodul din coada, îl punem în vector.

```
vector<int> topo(int N, vector<int> adj[]) {
    queue<int> q;
    vector<int> indegree(N, 0);
    for(int i = 0; i < N; i++) {
        for(auto it: adj[i]) {
            indegree[it]++;
        }
    }

    for(int i = 0; i < N; i++) {
        if(indegree[i] == 0) {
            q.push(i);
        }
    }
    vector<int> topo;
    while(!q.empty()) {
        int node = q.front();
        q.pop();
```

```

        topo.push_back(node);
        for(auto it : adj[node]) {
            indegree[it]--;
            if(indegree[it] == 0) {
                q.push(it);
            }
        }
    }
    return topo;
}

```

Cu DFS

Diferența față de DFS e că pun la sfârșit nodul pe stack. Diferențele față de anterior sunt că iau tot ce nu e vizitat și că e mai eficient din punct de vedere al memoriei, că timpul lanțului cel mai lung nu da stackoverflow. Complexitate $O(V + E)$

```

void Graph::topologicalSortUtil(int v, bool visited[],
                               stack<int>& Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack
    // which stores result
    Stack.push(v);
}

```

4 Muchii critice

Acele muchii care nu fac parte dintr-un ciclu. Facem un DFS și ținem 2 valori: low și disc. Pentru fiecare nod, mai întâi punem disc ca fiind $disc_{anterior} + 1$, trecem la copil, pentru el memorăm părintele, apoi facem DFS, apoi actualizăm low ca fiind $\min(low[copil], low[părinte])$. Dacă este critică, atunci $low[copil] > disc[părinte]$.

```

void Graph::bridgeUtil(int u, bool visited[], int disc[],
                      int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
}

```

```

// Go through all vertices adjacent to this
list<int>::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
    int v = *i; // v is current adjacent of u

    // If v is not visited yet, then recur for it
    if (!visited[v])
    {
        parent[v] = u;
        bridgeUtil(v, visited, disc, low, parent);

        // Check if the subtree rooted with v has a
        // connection to one of the ancestors of u
        low[u] = min(low[u], low[v]);

        // If the lowest vertex reachable from subtree
        // under v is below u in DFS tree, then u-v
        // is a bridge
        if (low[v] > disc[u])
            cout << u <<" " << v << endl;
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = min(low[u], disc[v]);
}
}

```

5 Puncte critice

V este punct critic, daca exista 2 noduri x si y , cu $x, y \neq v$, pentru care v apartine oricarui x, y -lant

```

void APUtil(vector<int> adj[], int u, bool visited[],
            int disc[], int low[], int& time, int parent,
            bool isAP[])
{
    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    for (auto v : adj[u]) {
        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v]) {
            children++;
            APUtil(adj, v, visited, disc, low, time, u, isAP);

            // Check if the subtree rooted with v has

```

```

        // a connection to one of the ancestors of u
        low[u] = min(low[u], low[v]);

        // If u is not root and low value of one of
        // its child is more than discovery value of u.
        if (parent != -1 && low[v] >= disc[u])
            isAP[u] = true;
    }

    // Update low value of u for parent function calls.
    else if (v != parent)
        low[u] = min(low[u], disc[v]);
}

// If u is root of DFS tree and has two or more children.
if (parent == -1 && children > 1)
    isAP[u] = true;
}

```

6 Conexitate in graf orientat

Exista 2 tipuri de conexitati intr-un graf orientat:

- Slab conex: Drum de intre \forall 2 noduri daca consideram graful neorientat
- Tare conex: Drum intre \forall 2 noduri

Kosaraju

Fac un DFS in care adaug pe stack nodurile. Transpun graful. Popuiesc stackul, fac DFS pe acel nod in care marchez elementele ca vizitate, apoi popuiesc urmatorul element si fac DFS daca e nevizitat. Complexitate $O(V + E)$.

```

#include <bits/stdc++.h>

#define MAX_N 20001
#define ll long long int
using namespace std;
int n, m;

struct Node {
    vector < int > adj;
    vector < int > rev_adj;
};

Node g[MAX_N];

stack < int > S;
bool visited[MAX_N];

int component[MAX_N];
vector < int > components[MAX_N];
int numComponents;

void dfs_1(int x) {
    visited[x] = true;

```

```

    for (int i = 0; i < g[x].adj.size(); i++) {
        if (!visited[g[x].adj[i]]) dfs_1(g[x].adj[i]);
    }
    S.push(x);
}

void dfs_2(int x) {
    printf("%d ", x);
    component[x] = numComponents;
    components[numComponents].push_back(x);
    visited[x] = true;
    for (int i = 0; i < g[x].rev_adj.size(); i++) {
        if (!visited[g[x].rev_adj[i]]) dfs_2(g[x].rev_adj[i]);
    }
}

void Kosaraju() {
    for (int i = 0; i < n; i++)
        if (!visited[i]) dfs_1(i);

    for (int i = 0; i < n; i++)
        visited[i] = false;

    while (!S.empty()) {
        int v = S.top();
        S.pop();
        if (!visited[v]) {
            printf("Component %d: ", numComponents);
            dfs_2(v);
            numComponents++;
            printf("\n");
        }
    }
}

int main() {

    cin >> n >> m;
    int a, b;
    while (m--) {
        cin >> a >> b;
        g[a].adj.push_back(b);
        g[b].rev_adj.push_back(a);
    }

    Kosaraju();
    printf("Total number of components: %d\n", numComponents);

    return 0;
}

```

Lema

Dacă două vârfuri se află în aceeași componentă tare conexă, atunci nici un drum între ele nu pășește, vreodată, componentă tare conexă.

Demonstrație: Fie u și v două noduri din componenta tare conexă. Presupunem ca există w în afara componentei și există drum $u \rightarrow v$ prin w . Atunci avem drum de la u la w dar avem și drumul $w \rightarrow v \rightarrow u$ deci și drum de la w la u deci w este în componenta tare conexă.

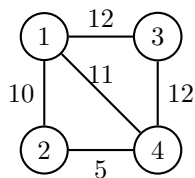
7 APM - arbori partiali de cost minim

Kruskal

$O(E * \log E + E * \log V)$ Algoritmul are 3 pași:

- Sorteaza toate muchiile în ordinea crescătoare a greutății
- Alege muchia cea mai mică, dacă formează un ciclu, treci la următoarea, dacă nu, adaug-o.
- Repeta pasul anterior până ai $V - 1$ muchii.

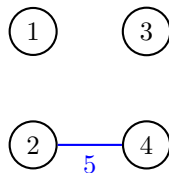
Exemplu Fie graful



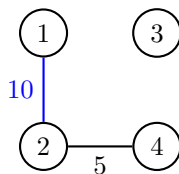
Pasul 1. Sortam ASC muchiile: $\{(2, 4, 5), (1, 2, 10), (1, 4, 11), (1, 3, 12), (3, 4, 12)\}$. Am folosit notația $(nod1, nod2, greutate)$.

Pasii 2 & 3.

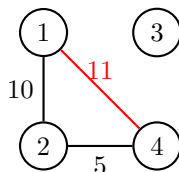
1. Începem recursia alegând cea mai mică muchie $(2, 4, 5)$.



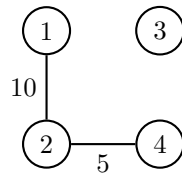
2. Continuăm cu $(1, 2, 10)$.



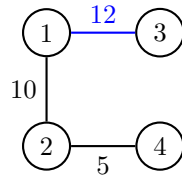
3. Continuăm cu $(1, 4, 11)$.



Observăm că face ciclu, deci o ignorăm



4. Continuum cu (1, 3, 12).



Observam ca avem $V - 1$ muchii, deci ramanem cu acest graf

K-clustering (aplicatie) Imparte in k clustere (are $n - k$ pasi)

Pseudocod:	Pseudocod – modelare cu graf complet G :
Inițial fiecare obiect (cuvânt) formează o clasă	$V = \{o_1, \dots, o_n\}$, $w(o_i, o_j) = d(o_i, o_j)$ Inițial fiecare vârf formează o componentă conexă (clasă): $T' = (V, \emptyset)$
pentru $i = 1, n-k$	pentru $i = 1, n-k$
<ul style="list-style-type: none"> alege două obiecte o_r, o_t din clase diferite cu $d(o_r, o_t)$ minimă reunește clasa lui o_r și clasa lui o_t 	<ul style="list-style-type: none"> alege o muchie $e_i = uv$ de cost minim din G astfel încât u și v sunt în componente conexe diferite ale lui T' reunește componenta lui u și componenta lui v: $E(T') = E(T') \cup \{uv\}$
returnează cele k clase obținute	returnează cele k mulțimi formate cu vârfurile celor k componente conexe ale lui T'

Prim

Complexitate $O(V^2)$ sau $O(V * \log V + E * \log V)$ daca folosim minheap. Algoritmul are 6 pasi:

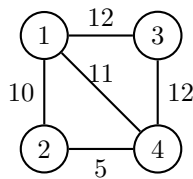
- Alegem un nod n aleatoriu si il scoatem din lista nodurilor de ales
- Adaugam in structura de date toti vecinii sai impreuna cu greutatea muchiilor
- Actualizam eticheta pentru fiecare varf cu distanta minima fata de cele incluse in graf
- Adaugam nodul cu distanta minima cea mai mica, il scoatem din lista nodurilor de ales si ii facem parintele nodul de care se leaga
- Adaugam in structura de date toti vecinii sai impreuna cu greutatea muchiilor
- Repetam al 4-lea pas.

► Prim

- s – vârful de start
- inițializează Q cu V
- pentru fiecare $u \in V$ executa
 $d[u] = \infty$; $tata[u] = 0$
 $d[s] = 0$
- cat timp $Q \neq \emptyset$ executa
 extrage un vârf $u \in Q$ cu eticheta $d[u]$ minimă
 pentru fiecare $uv \in E$ executa
 daca $v \in Q$ si $w(u,v) < d[v]$ atunci
 $d[v] = w(u,v)$
 $tata[v] = u$
- scrie $(u, tata[u])$, pentru $u \neq s$

↑ Prim prin vector vizitat ↑

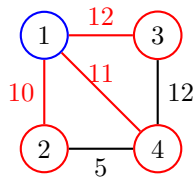
Exemplu Fie graful



Pasul 1. Plecam de la nodul 1 si facem urmatoarea structura

$$[(d/tata)] = \{(0, 0), (\infty, 0), (\infty, 0), (\infty, 0)\}$$

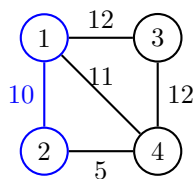
Pasul 2.



Pasul 3.

$$[(d/tata)] = \{(0, 0), (10, 0), (12, 0), (11, 0)\}$$

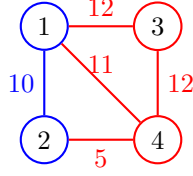
Pasul 4.



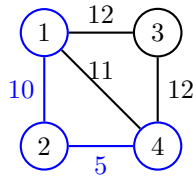
Pasul 5.

$$[(d/tata)] = \{(0, 0), (10, 1), (12, 0), (5, 0)\}$$

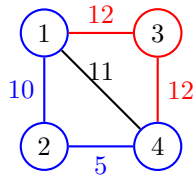
Pasul 6. Repetam



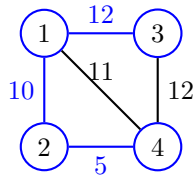
$$[(d/tata)] = \{(0, 0), (10, 1), (12, 0), (5, 0)\}$$



$$[(d/tata)] = \{(0, 0), (10, 1), (12, 0), (5, 2)\}$$

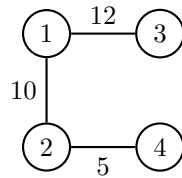


$$[(d/tata)] = \{(0, 0), (10, 1), (12, 0), (5, 2)\}$$



$$[(d/tata)] = \{(0, 0), (10, 1), (12, 1), (5, 2)\}$$

APM final



Prim(G, w, s)

```
pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
inițializează  $Q$  cu  $V$ 
cat timp  $Q \neq \emptyset$  executa
     $u = \text{extrage vârf cu eticheta } d \text{ minimă din } Q$ 
    pentru fiecare  $v$  adiacent cu  $u$  executa
        dacă  $v \in Q$  și  $w(u, v) < d[v]$  atunci
             $d[v] = w(u, v)$ 
             $tata[v] = u$ 
            //actualizează  $Q$  - pentru  $Q$  heap
scrie  $(u, tata[u])$ , pentru  $u \neq s$ 
```

↑ Prim prin minheap ↑

8 Drumuri minime de sursa unica s

Dijkstra

Se poate folosi doar pentru drumuri de cost pozitiv

Complexitate

- Cu heap $O(E * \log V)$
 - Initializare Q : $O(n)$
 - n * extragere varf minim: $O(n * \log n)$
 - Actualizare etichete vecini: $O(n * \log n)$
- Cu vector $O(V^2)$
 - Initializare Q : $O(n)$
 - n * extragere varf minim: $O(n^2)$
 - Actualizare etichete vecini: $O(m^2)$

```

pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
 $Q = V$  //creare heap cu cheile din  $d$ 
cat timp  $Q \neq \emptyset$  executa
     $u = \text{extrage\_min}(Q)$ 
    pentru fiecare  $uv \in E$  executa
        daca  $d[u] + w(u,v) < d[v]$  atunci
             $d[v] = d[u] + w(u,v)$ 
            repara( $Q, v$ )
             $tata[v] = u$ 
scrie  $d, tata$ 

```

↑ Dijkstra cu min heap ↑

```

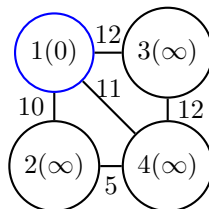
inițializează mulțimea vârfurilor neselectate  $Q$  cu  $V$ 
pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 
cat timp  $Q \neq \emptyset$  executa
     $u = \text{extrage vârf cu eticheta } d \text{ minimă din } Q$ 
    pentru fiecare  $uv \in E$  executa
        daca  $d[u] + w(u,v) < d[v]$  atunci
             $d[v] = d[u] + w(u,v)$ 
             $tata[v] = u$ 
scrie  $d, tata$ 

//scrie drum minim de la  $s$  la  $t$  un varf  $t$  dat folosind  $tata$ 

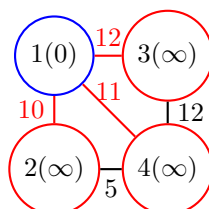
```

↑ Dijkstra cu vector ↑

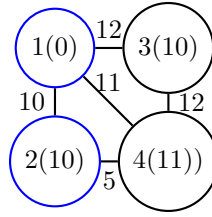
Exemplu Fie graful in care incepem de la 1.



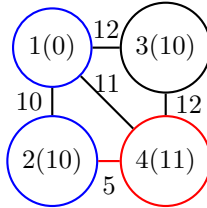
Consideram vecinii lui 1



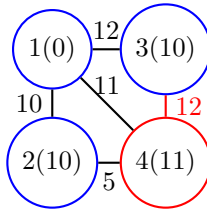
Actualizam distantele vecinilor lui 1 si il alegem pe minimul nevizitat (2)



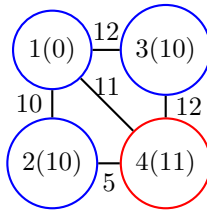
Actualizam distantele vecinilor lui 2 si il luam pe minimul nevizitat (3)



Actualizam distantele vecinilor lui 2 si il luam pe minimul nevizitat (4)



Luam minimul nevizitat si vedem ca nu mai are vecini



Lema Pentru $\forall u \in V$, la orice pas al algoritmului lui Dijkstra avem:

- dacă $d[u] < \infty$, există un drum de la s la u în G de cost $d[u]$ și acesta se poate determina din vectorul $tata$:
 $tata[u] =$ predecesorul lui u pe un drum de la s la u de cost $d[u]$
- $d[u] \leq \delta(s, u)$

Consecinta Dacă la un pas al algoritmului avem pentru un vârf u relația $d[u] = \delta(s, u)$, atunci $d[u]$ nu se mai modifică până la final.

Teorema Fie $G=(V, E, w)$ un graf orientat ponderat cu $w : E \rightarrow \mathbb{R}_+$ și $s \in V$ fixat. La finalul algoritmului lui Dijkstra avem: $d[u] = \delta(s, u)$ pentru orice $u \in V$ și $tata$ memorează un arbore al distanțelor față de s .

Bellman-Ford

Se poate folosi si pentru drumuri negative

Complexitate $O(V * E)$

```

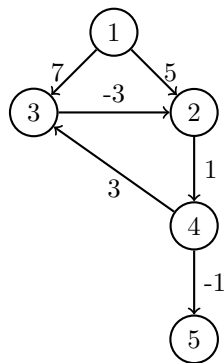
pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u] = 0$ 
 $d[s] = 0$ 

pentru  $i = 1, n-1$  executa
    pentru fiecare  $uv \in E$  executa
        daca  $d[u] + w(u, v) < d[v]$  atunci
             $d[v] = d[u] + w(u, v)$ 
             $tata[v] = u$ 

```

Optimizare Se pot relaxa doar arcele din varfurile ale caror etichete s-au modificat anterior

Exemplu



Etapa 1

Relaxam

- 1 2
- 1 3

	1	2	3	4	5
d	0	5	7	∞	∞

Relaxam

- 1 2
- 1 3
- 2 4

	1	2	3	4	5
d	0	5	7	6	∞

Relaxam

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

	1	2	3	4	5
d	0	5	7	6	5

Relaxam

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

	1	2	3	4	5
d	0	4	7	6	5

Etapă 2

Relaxam

- 2 4

	1	2	3	4	5
d	0	5	7	6	5

Relaxam

- 2 4
- 4 3

	1	2	3	4	5
d	0	5	7	6	5

Relaxam

- 2 4
- 4 3
- 4 5

	1	2	3	4	5
d	0	5	7	6	4

Relaxam

- 2 4
- 4 3
- 4 5
- 3 2

	1	2	3	4	5
d	0	5	7	6	4

Etapa 3

Relaxam

- 2 4
- 4 3
- 4 5
- 3 2

	1	2	3	4	5
d	0	5	7	6	4

Nu se mai actualizeaza nimic. Ne oprim

Lema Pentru $\forall u \in V$, la orice pas al algoritmului lui Bellman-Ford avem:

- dacă $d[u] < \infty$, există un drum de la s la u în G de cost $d[u]$ și acesta se poate determina din vectorul tata:
tata[u] = predecesorul lui u pe un drum de la s la u de cost $d[u]$
- $d[u] \leq \delta(s, u)$

Detectarea de circuite negative

Demonstrație: Arătăm că

nu există cicluri negative \Leftrightarrow

nu se mai fac actualizări la pasul n

- ▶ Dacă nu există cicluri negative \Rightarrow nu se mai fac actualizări la pasul n (din corectitudine)
- ▶ Dacă nu se mai fac actualizări la pasul n , pentru orice ciclu $C = [v_0, \dots, v_p, v_0] \Rightarrow d[v_i] \leq d[v_i] + w(v_i v_{i+1})$

Însumând pe ciclu:

$$d[v_0] + \dots + d[v_p] \leq d[v_0] + \dots + d[v_p] + w(v_0 v_1) + \dots + w(v_p v_0)$$

$$\Rightarrow 0 \leq w(v_0 v_1) + \dots + w(v_p v_0) = w(C)$$

Algoritm Afișarea ciclului negative detectat - folosind tata:

- Fie v un vârf al cărei etichetă s-a actualizat la pasul k
- Facem n pași înapoi din v folosind vectorul $tata$ (către s) ; fie x vârful în care am ajuns
- Afișăm ciclul care conține pe x folosind $tata$ (din x până ajungem iar în x)

Drumuri minime de sursa unica in grafuri aciclice

s - vârful de start

```
//initializam distante - ca la Dijkstra
pentru fiecare  $u \in V$  executa
     $d[u] = \infty$ ;  $tata[u]=0$ 
 $d[s] = 0$ 

//determinăm o sortare topologică a vârfurilor
SortTop = sortare_topologica(G)

pentru fiecare  $u \in \text{SortTop}$ 
    pentru fiecare  $uv \in E$  executa
        daca  $d[u]+w(u,v) < d[v]$  atunci //relaxam uv
             $d[v] = d[u]+w(u,v)$ 
             $tata[v] = u$ 

scrie  $d$ ,  $tata$ 
```

Complexitate $O(V + E)$

- Initializare: $O(n)$
- Sortare topologica: $O(m + n)$
- m * relaxare uv: $O(m)$

9 Drumuri minime intre toate perechile de varfuri

Floyd-Warshall

Complexitate $O(n^3)$

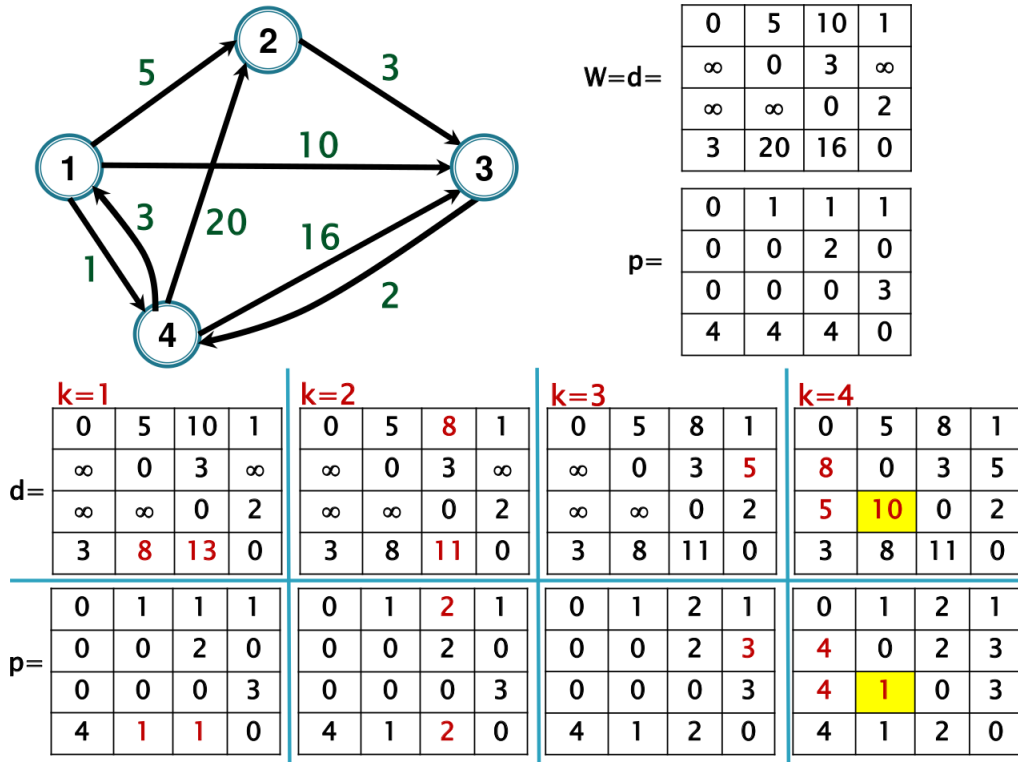
```
for( $i=1$ ;  $i \leq n$ ;  $i++$ )
    for( $j=1$ ;  $j \leq n$ ;  $j++$ ){
         $d[i][j]=w[i][j]$ ; // adaugam distanta ca fiind greutatea muchiei
        // daca e INF, nu e parinte, altfel il adaugam
        if( $w[i][j] == INF$ )
             $p[i][j]=0$ ;
        else
             $p[i][j]=i$ ;
    }

for( $k=1$ ;  $k \leq n$ ;  $k++$ )
    for( $i=1$ ;  $i \leq n$ ;  $i++$ )
        for( $j=1$ ;  $j \leq n$ ;  $j++$ )
            if( $d[i][j] > d[i][k]+d[k][j]$ ){
```

```

// Daca ce avem e mai mare decat suma distantelor,
// actualizam
d[i][j]=d[i][k]+d[k][j];
p[i][j]=p[k][j];
}

```



Inchiderea tranzitiva

```

for(k=1;k<=n;k++)
  for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
      d[i][j] = d[i][j] || (d[i][k] && d[k][j]);

```

10 Flow

Ford Fulkerson

Exemplu