Notite examen - Algoritmi Fundamentali

Dinu Florin-Silviu
grupa 231

Contents

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 $V_1 \cap V_2 = \varnothing$
- 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 strămoş v într-un arbore de adâncime. Buclele (muchii de la un vârf la el însuşi) care pot apărea într-un graf orientat sunt considerate muchii înapoi.
- Muchiile înainte sunt acele muchii (u, v) ce nu sunt muchii de arbore şi conectează un vârf u cu un descendent v într-un arbore de adâncime.
- Muchiile transversale sunt toate celelalte muchii. Ele pot uni vârfuri din acelaşi arbore de adâncime, cu condiţia ca unul să nu fie strămoşul celuilalt, sau pot uni vârfuri din arbori, de adâncime, diferiţi.

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

Incepe de la toate nodurile care au $indegree == 0$ (DAG garanteaza cel putin 1 astfel de nod). Dupa ce scoatem nodul din coada, il punem in 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

Diferentea fata de DFS e ca pun la sfarsit nodul pe stack. Diferentele fata de anterior sunt ca iau tot ce nu e vizitat si ca e mai eficient dpdv al memoriei, cat timp lantul 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 si tinem 2 valori: low si disc. Pentru fiecare nod, mai intai punem disc ca fiind $disc_anterior + 1$, trecem la copil, pentru el memoram parintele, apoi facem DFS, apoi actualizam low ca fiind $min(low[copil], low[parinte])$. Daca este critica, atunci $low[copil] > disc[parinte]$.

```
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 ∀ 2 noduri daca consideram graful neorientat
- Tare conex: Drum intre ∀ 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 aceeaşi componentă tare conexă, atunci nici un drum între ele nu părăseşte, vreodată, componentă tare conexă.

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

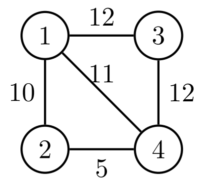
7 APM - arbori partiali de cost minim

Kruskal

$O(E * logE + E * logV)$ Algoritmul are 3 pasi:

- Sorteaza toate muchiile in oridinea crescatoare a greutatii
- Alege muchia cea mai mica, daca formeaza un ciclu, treci la urmatoarea, daca nu, adaug-o.
- Repeta pasul anterior pana ai $V - 1$ muchii.

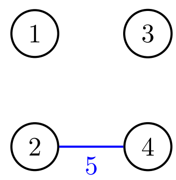
Exemplu Fie graful



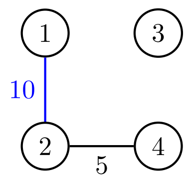
Pasul 1. Sortam ASC muchiile: $\{(2, 4, 5), (1, 2, 10), (1, 4, 11), (1, 3, 12), (3, 4, 12)\}$. Am folosit notatia $(nod1, nod2, greutate)$.

Pasii 2 & 3.

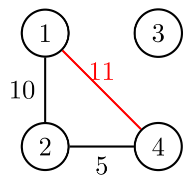
1. Incepem recursia alegand cea mai mica muchie $(2, 4, 5)$.



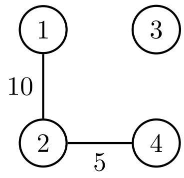
2. Continuam cu $(1, 2, 10)$.



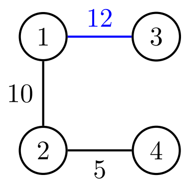
3. Continuam cu $(1, 4, 11)$.



Observam ca face ciclu, deci o ignoram



4. Continuam 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:**
	V= {o_1,…, o_n}, w($o_i o_j$) = d(o_i,o_j)
Inițial fiecare obiect (cuvânt) formează o clasă	Inițial fiecare vârf formează o componentă conexă (clasă): **T'=(V, ∅)**
pentru i = 1, n-k	pentru i = 1, n-k
• alege două obiecte o_r, o_t din clase diferite cu d(o_r, o_t) minimă	• 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 clasa lui o_r și clasa lui o_t	• reunește componenta lui u și componenta lui v: **E(T')=E(T') ∪ {uv}**
returneză cele k clase obținute	returnează cele k mulțimi formate cu vârfurile celor k componente conexe ale lui T'

Prim

Complexitate $O(V^2)$ sau $O(V * logV + E * logV)$ 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 scaotem 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∈V executa`

```
      d[u] = ∞; tata[u]=0
 d[s] = 0
```

- `cat timp Q ≠ ∅ executa`

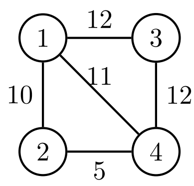
    extrage un vârf u∈Q cu eticheta d[u] minimă

```
    pentru fiecare uv∈E executa
             daca v∈Q si w(u,v)<d[v] atunci
                  d[v] = w(u,v)
                  tata[v] = u
```

- scrie (u, tata[u]), pentru u≠ s

↑ Prim prin vector vizitat ↑

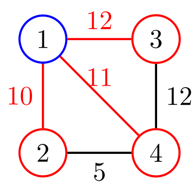
Exemplu Fie graful



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

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

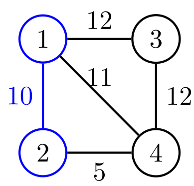
Pasul 2.



Pasul 3.

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

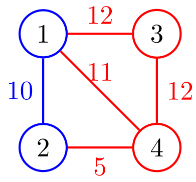
Pasul 4.



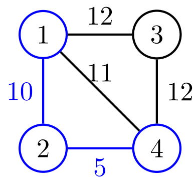
Pasul 5.

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

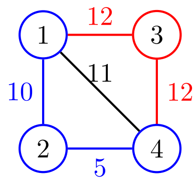
Pasul 6. Repetam



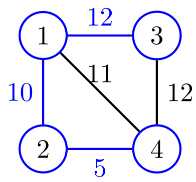
$$[(d/tata)] = \{(0,0),(10,1),(12,0),(5,0)\}$$



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

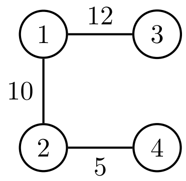


$$[(d/tata)] = \{(0,0),(10,1),(12,0),(5,2)\}$$



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

APM final



Prim(G, w, s)

```
pentru fiecare u∈V executa
    d[u] = ∞; tata[u]=0
d[s] = 0
inițializează Q cu V
cat timp Q ≠ ∅ executa
    u=extrage vârf cu eticheta d minimă din Q
    pentru fiecare v adiacent cu u executa
        daca v∈Q si w(u,v)<d[v] atunci
            d[v] = w(u,v)
            tata[v] = u
            //actualizeaza Q - pentru Q heap

scrie (u, tata[u]), pentru u≠ s
```

↑ **Prim prin minheap** ↑

8 Drumuri minime de sursa unica s

Dijkstra

Se poate folosi doar pentru drumuri de cost pozitiv

Complexitate

- Cu heap $O(E * logV)$
  - 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: $O(m^2)$

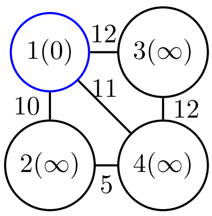
```
pentru fiecare u∈V executa
    d[u] = ∞; tata[u]=0
d[s] = 0
Q = V //creare heap cu cheile din d
cat timp Q ≠ ∅ executa
    u = extrage_min(Q)
    pentru fiecare uv∈E executa
        daca d[u]+w(u,v)<d[v] atunci
            d[v] = d[u]+w(u,v)
            repara(Q,v)
            tata[v] = u
scrie d, tata
```

↑ **Dijskstra cu min heap** ↑

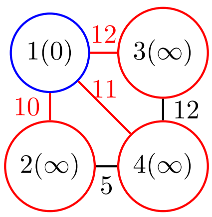
```
inițializează mulțimea vârfurilor neselectate Q cu V
pentru fiecare u∈V executa
    d[u] = ∞; tata[u]=0
d[s] = 0
cat timp Q ≠ ∅ executa
    u = extrage vârf cu eticheta d minimă din Q
    pentru fiecare uv∈E executa
        daca d[u]+w(u,v)<d[v] atunci
            d[v] = d[u]+w(u,v)
            tata[v] = u
scrie d, tata

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

↑ **Dijskstra cu vector** ↑

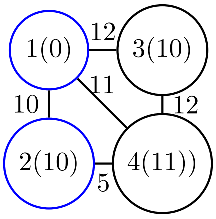
Exemplu Fie graful in care incepem de la 1.



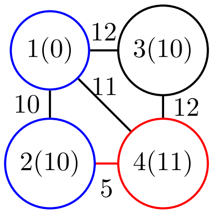
Consideram vecinii lui 1



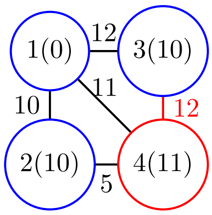
Actualizam distantele vecinilor lui 1 si il alegem pe minimul nevizitat (2)



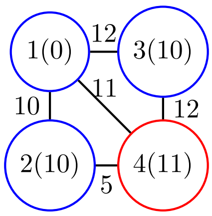
Actualizam distantele vecinilor lui 2 si il luam pe minimul nevizitat (3)



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 $w : E \to \mathbb{R}_+$ și $s \in V$ fixat.
La finalul algoritmul 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

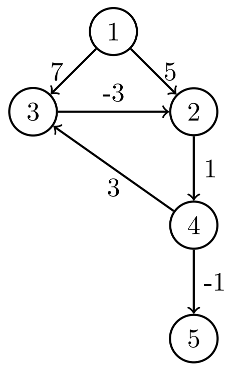
Se poate folosi si pentru drumuri negative

Complexitate $O(V * E)$

```
pentru fiecare u∈V executa
    d[u] = ∞; tata[u]=0
d[s] = 0

pentru i = 1,n-1 executa
   pentru fiecare uv∈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

Etapa 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 => nu se mai fac actualizări la pasul n (din corectitudine)
- Dacă nu se mai fac actualizări la pasul n, pentru orice ciclu $C=[v_0,\ldots,v_p, v_0]$ => $d[v_i]<=d[v_i]+w(v_iv_{i+1})$

Însumând pe ciclu:

$d[v_o]+\ldots+d[v_p] \leq d[v_o]+\ldots+d[v_p] +w(v_ov_1)+\ldots+w(v_pv_o)$

$\Rightarrow\ 0 \leq w(v_ov_1)+\ldots+w(v_pv_o)= w(C)$

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∈V executa
       d[u] = ∞; tata[u]=0
d[s] = 0

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

pentru fiecare u ∈ SortTop
       pentru fiecare uv∈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: $O(m)$

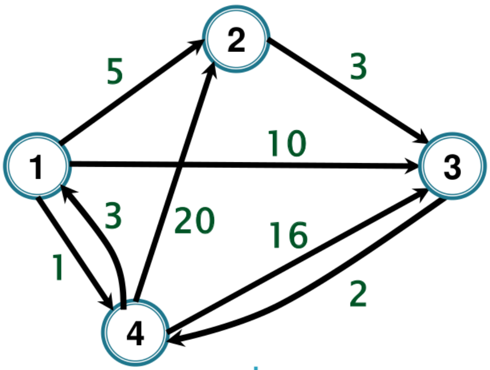
9 Drumuri minime intre toate perechile de varfuri

Floyd-Warshall

Complexitate $O(n^3)$

```
for(i=1;i<=n;i++)
    for(j=1;j<=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<=n;k++)
        for(i=1;i<=n;i++)
            for(j=1;j<=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];
            }
```



W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

k=1

d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	8	13	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

k=2

d=

0	5	8	1
∞	0	3	∞
∞	∞	0	2
3	8	11	0

p=

0	1	2	1
0	0	2	0
0	0	0	3
4	1	2	0

k=3

d=

0	5	8	1
∞	0	3	5
∞	∞	0	2
3	8	11	0

p=

0	1	2	1
0	0	2	3
0	0	0	3
4	1	2	0

k=4

d=

0	5	8	1
8	0	3	5
5	10	0	2
3	8	11	0

p=

0	1	2	1
4	0	2	3
4	1	0	3
4	1	2	0

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

Definitii

Fie N = (G, s, t, I, c) o rețea.'

Lant. Un s-t lanț este o succesiune de vârfuri distincte și arce din G

$$P = [s = v_0, e_1, v_1, ..., v_{k-1}, e_k, v_k = t]$$

Capacitate reziduala a arcului. Asociem fiecărui arc e din P o pondere, numită capacitate reziduală în P = cu cat mai poate fi modificat fluxul pe arcul e de-a lungul lantului P

$$i_P(e) = \begin{cases} c(e) - f(e), e\ este\ arc\ direct\ in\ P \\ f(e), e\ este\ arc\ invers\ in\ P \end{cases}$$

Capacitatea reziduala a lantului. Cu cat poate fi modificat fluxul de-a lungul lantului P.

$$i(P) = min\{i_P(e)|e \in E(P)\}$$

- **f-saturat** daca $i(P) = 0$
- **f-nesaturat** daca $i(P) \neq 0$

Flux revizuit de-a lungul lantului P. Fie P un lant f-nesaturat, definim fluxul revizuit ca $f_P : E \to \mathbb{N}$

$$f_P(e) = \begin{cases} f(e) + i(P), e\ este\ arc\ direct\ in\ P \\ f(e) - i(P), e\ este\ arc\ invers\ in\ P \\ f(e), altfel \end{cases}$$

Proprietate $val(f_P) = val(f) + i(P) \geq val(f) + 1$

Taietura in retea. Fie N = (G, s, t, I, c) o rețea.
O tăietură K = (X, Y) în rețea este o (bi)partiție (X, Y) a mulțimii vârfurilor V, astfel încât $s \in X$ și $t \in Y$.

Fie K = (X, Y) o tăietură. Capacitaeta taieturii = suma arcelor care ies din X catre Y.

$$c(K) = c(X, Y) = \sum_{x \in X, y \in Y, xy \in E} c(xy)$$

Taietura minima. Fie N o retea. O taietura $\widetilde{K}$ se numeste **taietura minima in N** daca

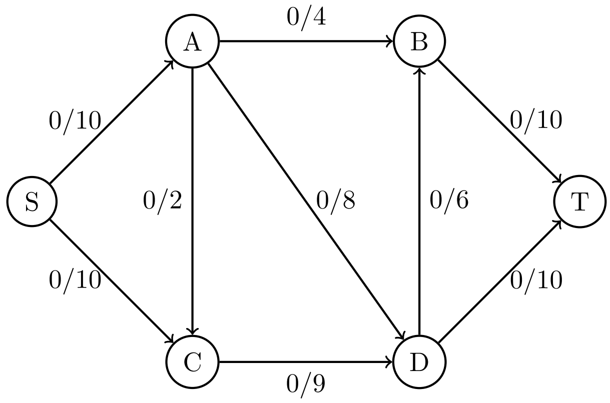
$$c(\widetilde{K}) = min\{c(K)|K\ este\ taietura\ in\ N\}$$

Se poate demonstra: $val(f) \leq c(\widetilde{K})$

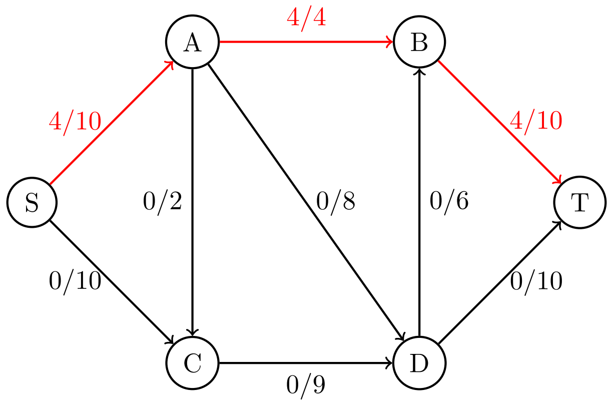
Ford Fulkerson

Complexitate $O(f * E)$

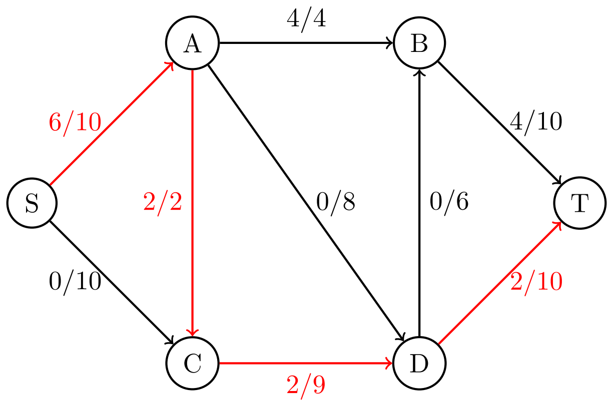
Exemplu Fie graful



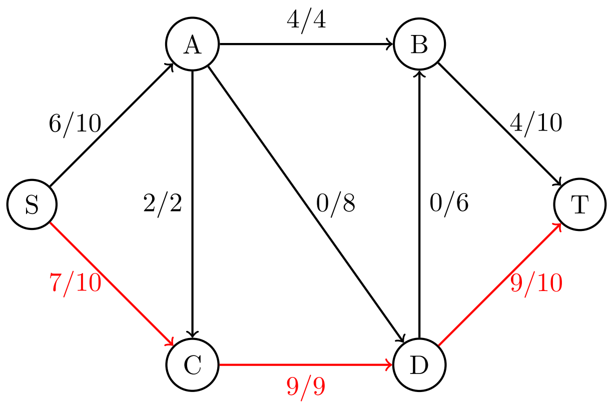
Fie lantul S - A - B - T (flow = 4, total = 4)



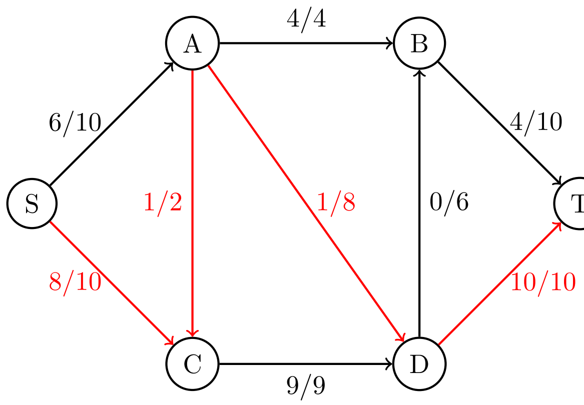
Fie lantul S - A - C - D - T (flow = 2, total = 6)



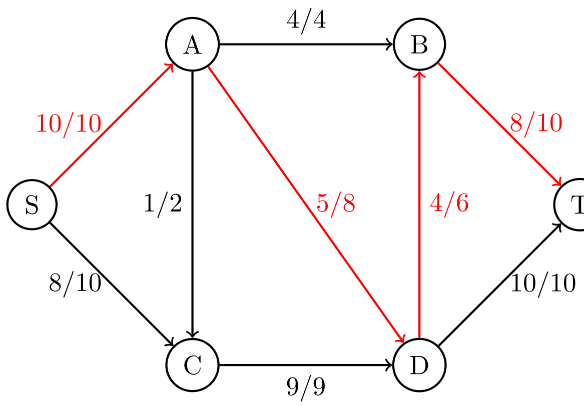
Fie lantul S - C - D - T (flow = 7, total = 13)



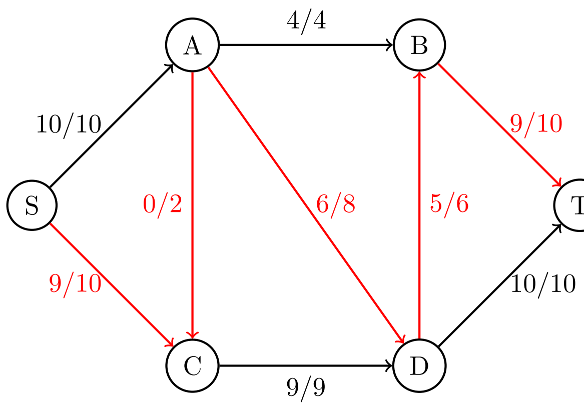
Fie lantul S - C - A - D - T (flow = 1, total = 14)



Fie lantul S - A - D - B - T (flow = 4, total = 18)



Fie lantul S - C - A - D - B - T (flow = 1, total = 19)

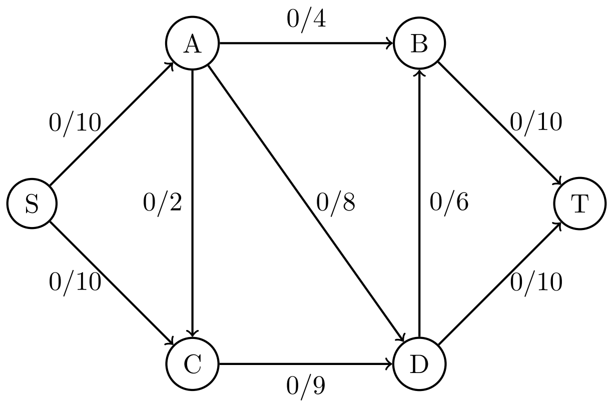


Nu mai avem drumuri de la S la T, deci flow total = 19

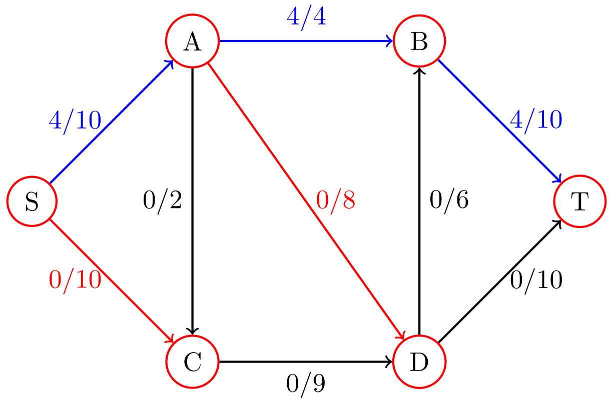
Edmonds-Karp

Complexitate $O(V * E^2)$

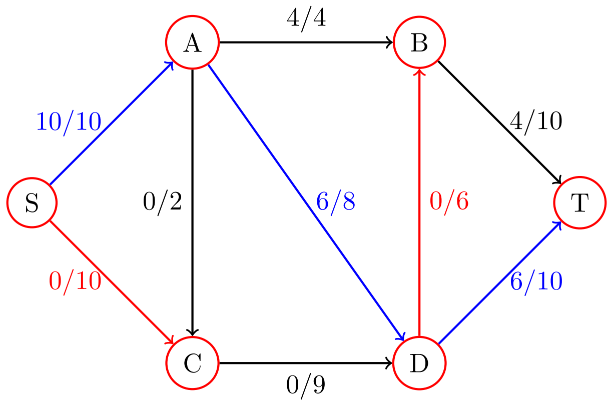
Exemplu Fie graful



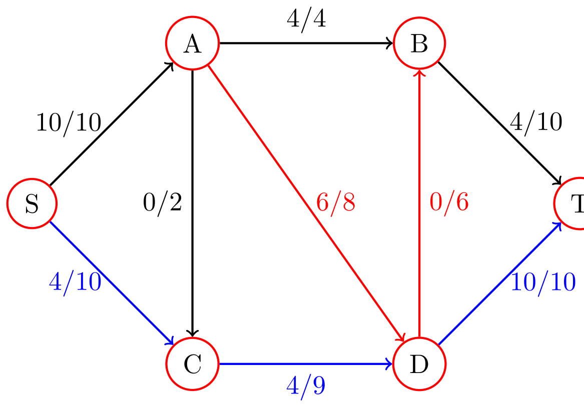
Parcurgerea BFS Flow = 4, Total = 4



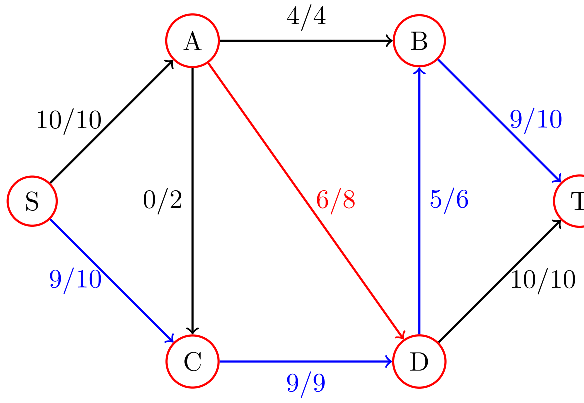
Parcurgerea BFS Flow = 6, Total = 10



Parcurgerea BFS Flow = 4, Total = 14



Parcurgerea BFS Flow = 5, Total = 19



Nu mai putem parcurge BFS, asa ca avem flow total 19

11 Grafuri bipartite

Definitie. G = (V, E) graf neorientat s.n. bipartit ⇔ există o partiție a lui V în două submulțimi V1, V2 (bipartiție):

$$V = V_1 \cup V_2$$
$$V_1 \cap V_2 = \varnothing$$

astfel încât orice muchie $e \in E$ are o extremitate în V_1 și cealaltă în V_2

$$\text{Notăm } G = (V1 \cup V2, E)$$

Bipartit complet. $G = (V, E)$ este bipartit complet ⇔ este bipartit si $E = \{xy | x \in V_1, y \in V_2\}$

$$\text{Notam cu } K_{p,q} \text{ daca } p = |V_1| \text{ si } q = |V_2|$$

Observatie. G = (V, E) bipartit ⇔ există o 2-colorare proprie a vârfurilor (bicolorare): $c : V \to \{1, 2\}$ (adica pentru orice muchie $e = xy \in E$, avem $c(x) \neq c(y)$)

Teorema König. Fie $G = (V, E)$ un graf cu $n \geq 2$ vârfuri. Avem G este bipartit ⇔ toate ciclurile elementare din G sunt pare.

Algoritm pentru testare.

- Colorăm cu 2 culori un arbore parțial al său printr-o parcurgere (colorăm orice vecin j nevizitat al vârfului curent i cu o culoare diferită de cea a lui i)
- Testăm dacă celelalte muchii – de la i la vecini j deja vizitați (colorați) au extremitățile i și j colorate diferit

Cuplaj maxim in grafuri bipartite

Definitii. Fie $G = (V, E)$ un graf şi $M \subseteq E$.

- M s.n cuplaj dacă orice două muchii din M sunt neadiacente
- V(M) = mulţimea vârfurilor M-saturate
- V(G) - V(M) = mulţimea vârfurilor M-nesaturate

Un cuplaj M^* s.n cuplaj de cardinal maxim (cuplaj maxim):

$$|M^*| \geq |M|, \forall M \subseteq E \ cuplaj$$

Algoritm de determinare.

- Reducem problema determinării unui cuplaj maxim într-un cuplaj bipartit G la determinarea unui flux maxim într-o rețea de transport asociată lui G
- Construim rețeaua de transport N_G asociată lui G
- Adăugăm două noduri noi s şi t
- Adăugăm arce (s, x_i), pentru $x_i \in X$ şi (y_j, t), $y_j \in Y$
- Transformăm muchiile $x_i y_j$ în arce (de la X la Y)
- Asociem fiecărui arc capacitatea 1
- Cuplaj M în G $\Leftrightarrow$ flux f în rețea
- $|M| = val(f)$

Proprietati

Proprietatea 1. Fie $G = (X \cup Y, E)$ un graf bipartit şi M un cuplaj în G. Atunci există un flux f în reţeaua de transport asociată N_G cu

$$val(f) = |M|$$

Proprietatea 2. Fie $G = (X \cup Y, E)$ un graf bipartit şi f un flux în reţeaua de transport N_G asociată. Atunci există M un cuplaj în G cu

$$val(f) = |M|$$

Consecinta. f^* flux maxim în N $\Rightarrow$ cuplajul corespunzător M^* este cuplaj maxim în G
A determina un cuplaj maxim într-un graf bipartit $\Leftrightarrow$ a determina un flux maxim în reţeaua asociată

Algoritm. Complexitate: C=1 (sau $L \leq c^+(s) \leq n$) $\Rightarrow$ O(mn). Fie $G = (X \cup Y, E)$

- Construim N reţeaua de transport asociată
- Determinăm f^* flux maxim în N
- Considerăm $M = \{xy | f^*(xy) = 1, x \in X, y \in Y, xy \in N\}$ (pentru fiecare arc cu flux nenul xy din N care nu este incident în s sau t, muchia xy corespunzătoare din G se adaugă la M)
- return M

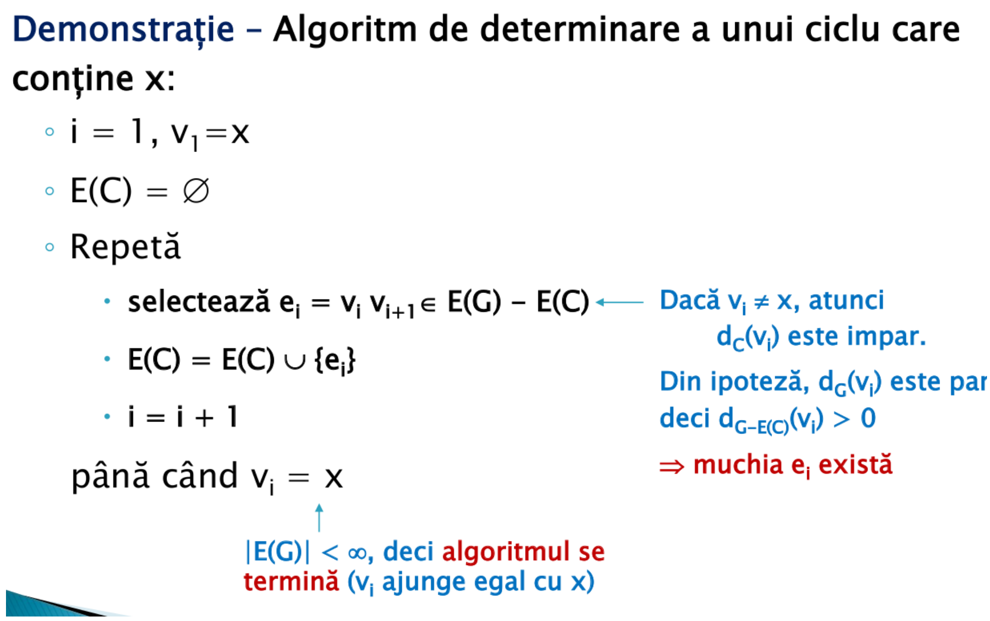
12 Grafuri Euleriene

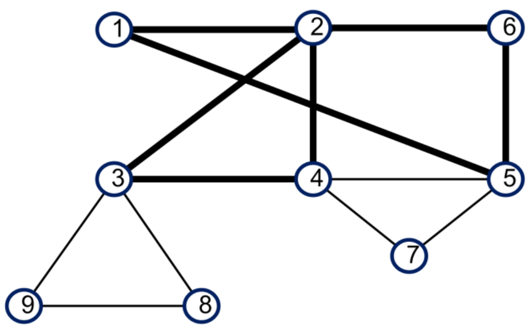
Ciclu eulerian: traseu închis care trece o singură dată prin toate muchiile

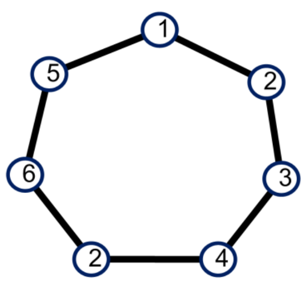
Graf eulerian: conține un ciclu eulerian

Lant eulerian: Lanț eulerian al lui G = lanț simplu P în G cu $E(P) = E(G)$

Lema Fie $G = (V, E)$ un graf neorientat, conex, cu toate vârfurile de grad par și $E \neq \varnothing$. Atunci pentru orice $x \in V$ există un ciclu C în G cu $x \in V(C)$ (ciclu care conține x, nu neapărat eulerian, nici neapărat elementar).



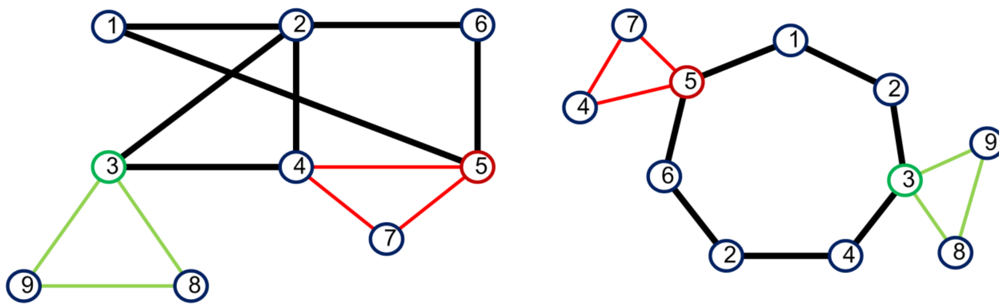




Teorema lui Euler. Fie $G = (V, E)$ un (multi)graf neorientat, conex, cu $E \neq \varnothing$. Atunci G este eulerian $\Leftrightarrow$ orice varf din G are grad par.

Hierholzer. Complexitate $O(m)$. Determinarea unui ciclu eulerian într-un graf conex (sau un graf conex+ vârfuri izolate) cu toate vârfurile de grad par

- **Pasul 0** – verificare condiții (conex+vf. izolate, grade pare)
- **Pasul 1:**
  - `alege v ∈ V arbitrar`
  - `construiește C un ciclu în G care începe cu v (cu algoritmul din Lema)`
- **cât timp |E(C)| < |E(G)| execută**
  - `selectează v ∈ V(C) cu` $d_{G-E(C)}$`(v) > 0 (în care sunt incidente muchii care nu aparțin lui C)`
  - `construiește C′ un ciclu în G - E(C) care începe cu v`
  - `C = ciclul obținut prin fuziunea ciclurilor C și C′ în v`
- scrie C



C= [1, 2, 3, 8, 9, 3, 4, 2, 6, 5, 4, 7, 5, 1]

Ciclul conține toate muchiile

$\Rightarrow$ este eulerian

13 Grafuri planare

Definitie. $G = (V, E)$ graf neorientat s.n. planar $\Leftrightarrow$ admite o reprezentare în plan a.î. muchiilor le corespund segmente de curbe continue care nu se intersectează în interior unele pe altele

Harta. Fie $G = (V, E)$ graf planar, M o hartă a sa. M induce o împărțire a planului într-o mulțime F de părți convexe numite fețe. Una dintre acestea este fața infinită (exterioară).

- $M = (V, E, F)$ hartă
- Pentru o față $f \in F$ definim: $d_M(f)$ = gradul feței f = numărul muchiilor lanțului închis (frontierei) care delimitează f (câte muchii sunt parcurse atunci când traversăm frontiera)
- $\sum_{f \in F} d_M(f) = 2 * |E|$

Teorema poliedrală a lui Euler. Fie $G = (V, E)$ un graf planar conex și $M = (V, E, F)$ o harta a lui. Are loc relația $|V| - |E| + |F| = 2$

Consecinta: Orice hartă M a lui G are $2 - |V| + |E|$ fețe

Teorema celor 6 culori. Orice graf planar conex este 6 – colorabil.

```
colorare(G)
    daca |V(G)|≤ 6 atunci coloreaza varfurile cu
culori distincte din {1,…,6}
    altfel
        alege x cu d(x) ≤ 5
        colorare(G-x)
                    colorează x cu o culoare din {1,…,6}
        diferită de culorile vecinilor
```

- **Sugestie implementare** – determinarea iterativă a ordinii în care sunt colorate vârfurile (similar parcurgere BF, sortare topologică)

14 Grafuri Hamiltoniene

Definitie. Un graf este hamiltonian dacă admite un ciclu hamiltonian adică un ciclu care contine toate nodurile grafului.

Conditie necesara. Grafurile hamiltoniene sunt biconexe(nu are noduri critice). Inversa nu este adevarata.

Teorema lui Dirac. Fie G un graf cu ordinul $n \geq 3$. Dacă $\delta(G) \geq n/2$ atunci G este hamiltonian.

Teorema lui Ore. Fie G un graf cu ordinul $n \geq 3$, dacă avem pentru oricare pereche de noduri neadiacente $deg(x) + deg(y) \geq n \rightarrow$ atunci graful este Hamiltonian.

Definitii ajutatoare. Conectivitatea $K(G)$ unui graf G este este marimea minima a unei mulțimi de tăiere a lui G.
Se calculeaza prin: Flux maxim = taietura minima

O mulțime de noduri a unui graf G este independentă dacă nu contine noduri adiacente. Numărul de independență $\alpha(G)$ al unui graf G este marimea cea mai mare posibila a unei mulțimi independente a lui G

Teorema lui Chvatal si Erdos. Fie G un graf conectat cu ordinal $n \geq 3$, conectivitatea $K(G)$, și numărul de independență $\alpha(G)$. Daca $K(G) \geq \alpha(G)$, atunci G este hamiltonian.

Teorema lui Goodman si Hedetniemi. Daca G este un graf 2-conectat ¸si liber de $\{K1, 3, Z1\}$ atunci G este hamiltonian