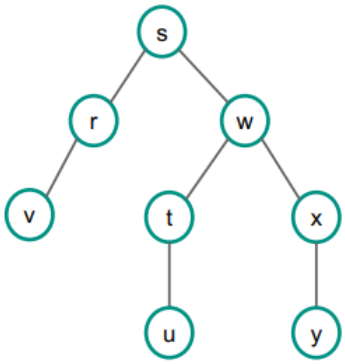
Cheat Sheet

Parcurgerea în lățime - Algoritm

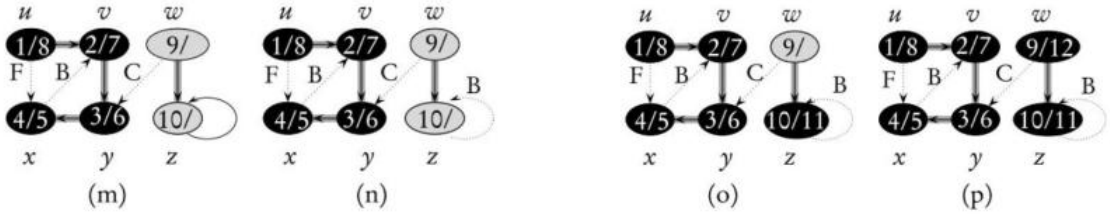
Algoritm Cormen:

Pi → tata



CL(G, s)
1: **pentru** fiecare vârf $u \in V[G] - \{s\}$ **execută**
2: $color[u] \leftarrow$ ALB
3: $d[u] \leftarrow \infty$
4: $\pi[u] \leftarrow$ NIL
5: $color[s] \leftarrow$ GRI
6: $d[s] \leftarrow 0$
7: $\pi[s] \leftarrow$ NIL
8: $Q \leftarrow \{s\}$
9: **cât timp** $Q \neq \emptyset$ **execută**
10: $u \leftarrow cap[Q]$
11: **pentru** fiecare vârf $v \in Adj[u]$ **execută**
12: **dacă** $color[v] =$ ALB **atunci**
13: $color[v] \leftarrow$ GRI
14: $d[v] \leftarrow d[u] + 1$
15: $\pi[v] \leftarrow u$
16: PUNE-ÎN-COADĂ(Q, v)
17: SCOATE-DIN-COADĂ(Q)
18: $color[u] \leftarrow$ NEGRU

Parcurgerea în adâncime - Exemplu



Parcurgerea în adâncime - Algoritm

Algoritm Cormen:

CA(G)
1: **pentru** fiecare vârf $u \in V[G]$ **execută**
2: $culoare[u] \leftarrow$ ALB
3: $\pi[u] \leftarrow$ NIL
4: $timp \leftarrow 0$
5: **pentru** fiecare vârf $u \in V[G]$ **execută**
6: **dacă** $culoare[u] =$ ALB **atunci**
7: CA-VIZITĂ(u)

CA-VIZITĂ(u)
1: $culoare[u] \leftarrow$ GRI ▷ Vârful alb u tocmai a fost descoperit.
2: $d[u] \leftarrow timp \leftarrow timp + 1$
3: **pentru** fiecare $v \in Adj[u]$ **execută** ▷ Explorează muchia (u, v).
4: **dacă** $culoare[v] =$ALB **atunci**
5: $\pi[v] \leftarrow u$
6: CA-VIZITĂ(v)
7: $culoare[u] \leftarrow$ NEGRU ▷ Vârful u este colorat în negru. El este terminat.
8: $f[u] \leftarrow timp \leftarrow timp + 1$

- Havel Hakimi:

Havel-Hakimi complexitate algoritm

—

- O soluție se folosește de heapuri. Această soluție are complexitatea O((N+M)*log(N)).
- O altă soluție, care nu ne oferă mulțimea muchiilor, se folosește de treapuri și de proprietatea de spargere a unui treap. Complexitatea acestei soluții este O(N*log(N)).

- O altă soluție se folosește de counting sort și poate atinge complexitatea O(N+M), oferind în același timp și mulțimea muchiilor. Această soluție sortează descrescător gradele și reține pentru fiecare grad ultima apariție. Scăderea cu unu a primelor x grade poate rezulta în pierderea proprietății de monotonie a șirului. Pentru a nu face asta solutia folosește ultima poziție a fiecărui grad pentru a verifica dacă poate scădea cu 1 din toate nodurile cu acel grad. Dacă nu, acesta va scădea cu 1 doar ultimele x poziții ale acelui grad. În timp ce scade gradele algoritmul va actualiza valorile corespunzătoare ultimelor poziții ale gradelor.

- Sortare topologica:

Kahn's algorithm [edit]

Not to be confused with Kuhn's algorithm.

One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort.[2] First, find a list of "start nodes" that have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty acyclic graph. Then:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Sortare topologică – Alt algoritm

Dacă **final[u]** = momentul la care a fost **finalizat** vârful u în parcurgerea DF, avem: uv ∈ E ⇒ f[u] > f[v]

- atunci sortarea topologică = sortare descrescătoare în raport cu **final**

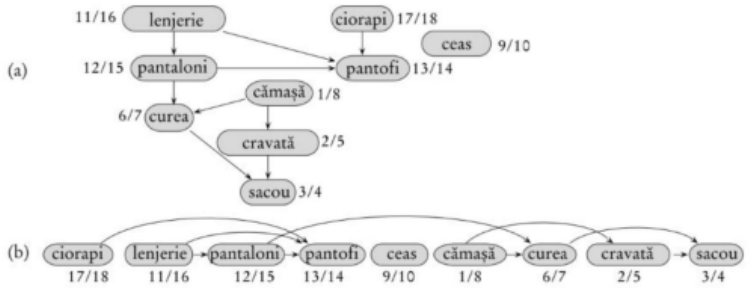


Figura 23.7 **(a)** Profesorul Bumstead îşi sortează topologic îmbrăcămintea când se îmbracă. Fiecare muchie (u, v) înseamnă că articolul u trebuie îmbrăcat înaintea articolului v. Timpii de descoperire şi de terminare dintr-o căutare în adâncime sunt prezentaţi alături de fiecare vârf. **(b)** Acelaşi graf sortat topologic. Vârfurile lui sunt aranjate de la stânga la dreapta în ordinea descrescătoare a timpului de terminare. Se observă că toate muchiile orientate merg de la stânga la dreapta.

- Muchii critice:

Muchii critice

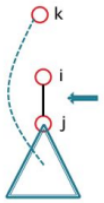
O muchie de avansare (i, j) este critică

⇔

nu este conținută într-un ciclu închis de o muchie de întoarcere

⇔

nu există nicio muchie de întoarcere cu



- o extremitate în **j sau într-un descendent al lui j**
- cealaltă extremitate în **i sau într-un ascendent al lui i** (într-un vârf de pe **un nivel mai mic sau egal** cu nivelul lui i)

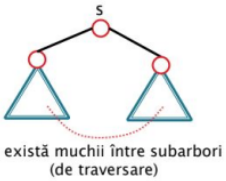
Puncte critice

Un vârf este **punct critic** ⇔ există două vârfuri x,y ≠ v astfel încât aparține oricărui x,y-lanț.

Arborele DF

- **rădăcina** este punct critic ⇔

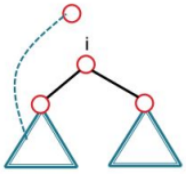
  are cel puțin 2 fii în arborele DF



nu există muchii între subarbori (de traversare)

- **un alt vârf i** din arbore este critic ⇔

  are cel puțin un fiu j cu niv_min[j] ≥ nivel[i]



- Conexitate

Componente tare conexe - Algoritm Kosaraju

Următorul algoritm de timp liniar (adică **θ(V+E)**) determină componentele tare conexe ale unui graf orientat G = (V, E), folosind două căutări în adâncime, una în G și una în G^T.

Componente-Tare-Conexe(G)

1. apelează CA(G) pentru a calcula timpii de terminare f[u] pentru fiecare vârf u
2. calculează G^T
3. apelează CA(G^T), dar, în bucla principală a lui CA, consideră vârfurile în ordinea **descrescătoare** a timpilor f[u] (calculați la linia 1)
4. afișează vârfurile fiecărui arbore din pădurea de adâncime din pasul 3 ca o componentă tare conexă separată

Kruskal

```
sorteaza(E)
for (v=1; v <= n; v++)
    Initializare(v);
nrmsel=0
for (uv ∈ E)
    if (Reprez(u) != Reprez(v)) {
        E(T) = E(T) ∪ {uv};
        Reuneste(u,v);
        nrmsel = nrmsel + 1;
        if (nrmsel == n-1)
            STOP;   // break
    }
```

Kruskal

Complexitate

- **Sortare** → O(m logm) = O(m logn)
- **n * Initializare**
- **2m * Reprez**
- **(n-1) * Reuneste**

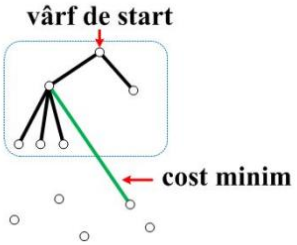
Depinde de modalitatea de memorare a componentelor conexe

- Disjoint-Set

```
int Solution::find(int node) {
    if (parent[node] == node) return node;

    // find the root node and do compression
    int result = find( node: parent[node]);
    parent[node] = result;
    return result;
}

void Solution::Union(int firstNode, int secondNode) {
    int firstNodeRoot = find( node: firstNode);
    int secondNodeRoot = find( node: secondNode);

    if (size[firstNode] < size[secondNode]) {
        parent[firstNodeRoot] = secondNodeRoot;
        return;
    }

    if (size[firstNode] > size[secondNode]) {
        parent[secondNodeRoot] = firstNodeRoot;
        return;
    }

    // if both sizes are equal then size goes up by one and it doesnt matter which ones the parent
    parent[firstNodeRoot] = secondNodeRoot;
    size[secondNodeRoot]++;
}
```

Algoritmul lui Prim

- Se pornește de la un vârf, care formează arborele inițial
- **La un pas**, este selectată o muchie de cost minim, de la un vârf deja adăugat în arbore, la un vârf neadăugat



KRUSKAL

- **Inițial: T = (V, ∅)**
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. u,v sunt în **componente conexe diferite** (T + uv aciclic)
  - E(T) = E(T) ∪ {uv}

PRIM

- **s** - vârful de start
- **Inițial, T = ({s}, ∅)**
- **pentru** i = 1, n-1
  - alege o muchie uv cu **cost minim din G** a.î. u ∈ V(T) și v ∉ V(T)
  - V(T) = V(T) ∪ {v}
  - E(T) = E(T) ∪ uv

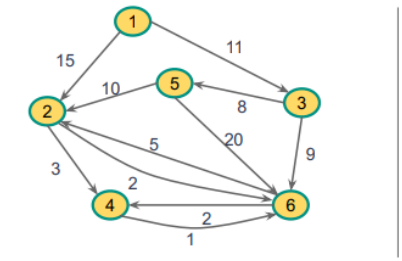
Prim - Algoritm

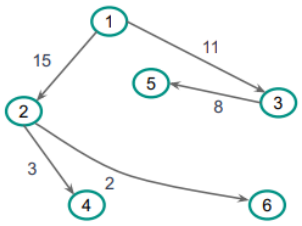
```
s - vârful de start
inițializează Q cu V
pentru fiecare u ∈ V execută
    d[u] = ∞
    tata[u] = 0
d[s] = 0
cât timp Q ≠ ∅ execută // pentru i=1,n (suficient n-1)
    extrage un vârf u ∈ Q cu eticheta d[u] minimă
    pentru fiecare uv ∈ E execută
        dacă v ∈ Q și w(u,v) < d[v] atunci
            d[v] = w(u,v)
            tata[v] = u
scrie (u, tata[u]) pentru u ≠ s
```

- Drumuri minime:

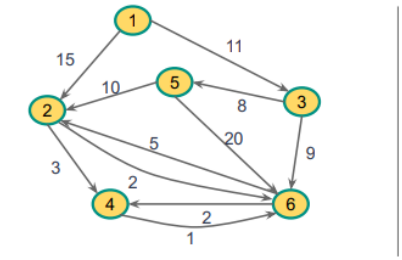
Algoritmul lui Dijkstra

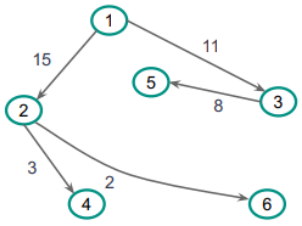
```
Dijkstra(G, w, s)
    inițializează mulțimea vârfurilor neselectate Q cu V
    pentru fiecare u∈V execută
        d[u] = ∞; tata[u] = 0
    d[s] = 0
    cât timp Q≠∅ execută
        u = extrage vârf cu eticheta d minimă din Q
        pentru fiecare uv∈E execută
            dacă d[u]+w(u,v) < d[v]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

    scrie d, tata
    // scrie drum minim de la s la un vârf t dat folosind tata
```





	1	2	3	4	5	6
d/tata	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Selectăm 1	[- ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Selectăm 3	[- ,	15/1,	- ,	∞/0,	19/3,	20/3]
Selectăm 2	[- ,	- ,	- ,	18/2,	19/3,	17/2]
Selectăm 6	[- ,	- ,	- ,	18/2,	19/3,	-]
Selectăm 4	[- ,	- ,	- ,	- ,	19/3,	-]
Selectăm 5	[- ,	- ,	- ,	- ,	- ,	-]





	1	2	3	4	5	6
d/tata SOLUȚIE	[0/0,	15/1,	11/1,	18/2,	19/3,	17/2]

Algoritmul lui BELLMAN – FORD

```
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
```

Complexitate: O(nm)

Detectarea de circuite negative

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 sursă unică în 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
//este suficient sa pastrăm vârfurile din sortare începând cu s
SortTop = sortare_topologica(G,s)

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
```

Algoritmi – G=(V, E) graf orientat

G – neponderat Parcurgere lățime BF	G – ponderat, ponderi >0 Algoritmul lui Dijkstra	G – ponderat fără circuite
BF(s)	**Dijkstra(s)**	**DAGS(s)**
coada C ← ∅; adauga(s, C)	(min-heap) Q ← V {se putea incepe doar cu Q ← {s} +vector viz; v∈Q ⇔ v nevizitat}	SortTop ← sortare_topologica(G)
pentru fiecare u∈V d[u]=∞;tata[u]=viz[u]=0 viz[s]← 1; d[s] ← 0	pentru fiecare u∈V d[u] = ∞; tata[u]=0 d[s] = 0	pentru fiecare u∈V d[u] = ∞; tata[u]=0 d[s] = 0
cat timp C ≠ ∅ u ← extrage(C); pentru fiecare uv∈E daca viz[v]=0 d[v] ← d[v]+1 tata[v] ← u adauga(v, C) viz[v] ← 1	cat timp Q ≠ ∅ u = extrage(Q) vârf cu eticheta d minimă pentru fiecare uv∈E daca ~~v∈Q~~ si d[u]+w(u,v)<d[v] d[v] = d[u]+w(u,v) tata[v] = u repara(v,Q)	pentru fiecare u ∈ SortTop pentru fiecare uv∈E daca d[u]+w(u,v)<d[v] d[v] = d[u]+w(u,v) tata[v] = u
scrie d, tata	scrie d, tata	scrie d, tata
O(n+m)	O(m log(n)) / O(n^2)	O(n+m)

Floyd(G, w)

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++){
        d[i][j]=w[i][j];
        if(w[i][j]== ∞)
             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]){
                        d[i][j]=d[i][k]+d[k][j];
                        p[i][j]=p[k][j];
                }
```

- Flux:

Implementare

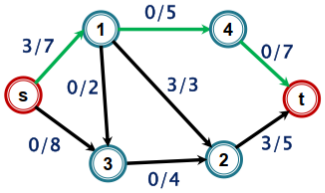
- Memorăm lanțurile (arborele BF) folosind vector **tata**
- **Convenție** – pentru arcele inverse (i,j) ținem minte tatăl cu semnul minus

  tata[j] = –i

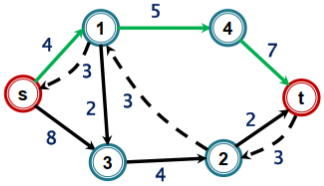
- Edmonds-Karp

```
construieste_s-t_lant_nesat_BF()
  pentru(v∈V) executa tata[v] ←0; viz[v] ←0
  coada C ← ∅
  adauga(s, C)
  viz[s]← 1
  cat timp C ≠ ∅ executa
     i ←  extrage(C)
     pentru (ij ∈ E) executa    arc direct
         dacă (viz[j]=0 și c(ij)-f(ij)>0) atunci
            adauga(j, C)
            viz[j] ← 1; tata[j] ← i
            daca (j=t) atunci STOP și returnează true(1)
     pentru (ji ∈ E) executa      arc invers
         daca (viz[j]=0 și f(ji)>0) atunci
            adauga(j, C)
            viz[j] ← 1; tata[j] ← -i
            daca (j=t) atunci STOP și returnează true(1)
  returnează false(0)
```

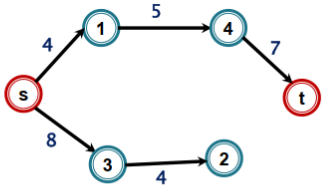
- **Complexitate**
  - Algoritm generic Ford Fulkerson O(mL)/O(nmC)
  - Implementare Edmonds Karp $O(nm^2)$



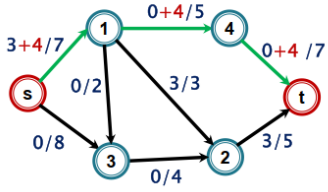
Graful rezidual G_f



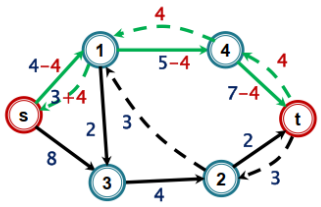
BF(s) – în graful rezidual

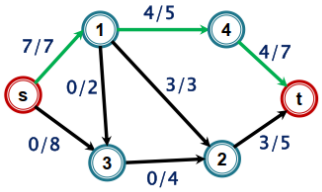


Drumul de creștere [s, 1, 4, t] – capacitate rezidulă 4

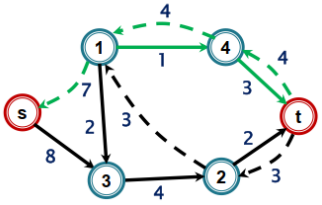


Graful rezidual G_f





Graful rezidual G_f



Aplicații p –colorări

Exemplu – De câte săli este nevoie minim pentru programarea într-o zi a n conferințe cu intervale de desfășurare date?

Conf. 1: interval (1,4)

Conf. 2: interval (2,3)

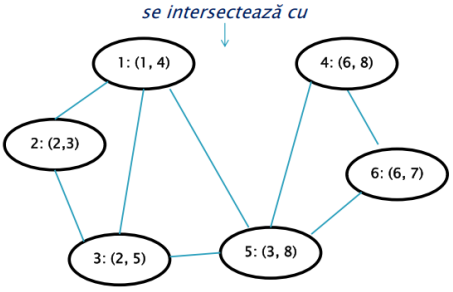
Conf. 3: interval (2,5)

Conf. 4: interval (6,8)

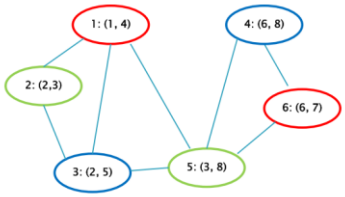
Conf. 5: interval (3,8)

Conf. 6: interval (6,7)

se intersectează cu



Graful intersecției intervalelor este 3-colorabil:



Sunt necesare minim 3 săli (corespunzătoare celor 3 culori):

Sala 1: (1,4), (6,7)

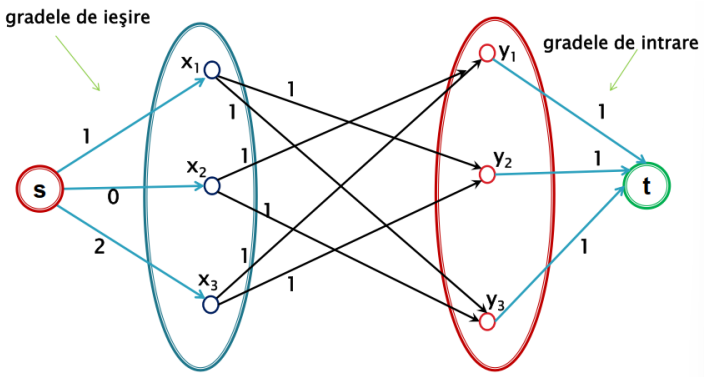
Sala 2: (2,3), (3,8)

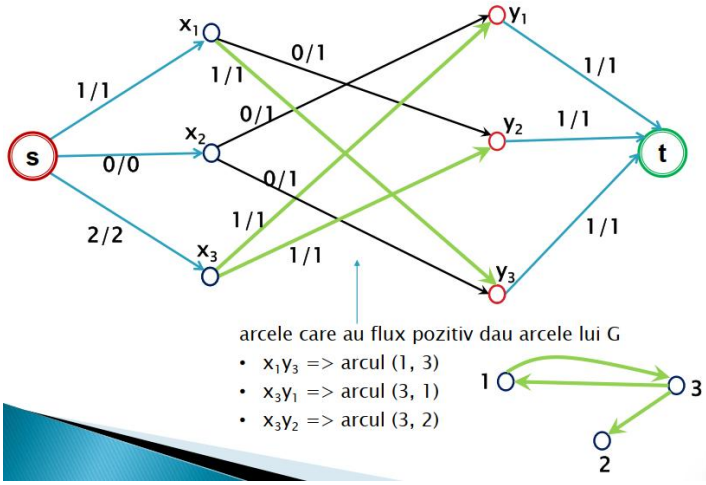
Sala 3: (2,5), (6,8)

Aplicaţie
Construcţia unui graf orientat din secvenţele de grade

Exemplu

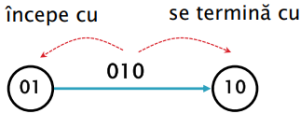
- $s_0^+ = \{1, 0, 2\}$
- $s_0^- = \{1, 1, 1\}$



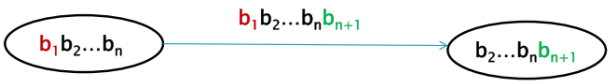


Grafuri de Bruijn

- Etichete pe arce:



- În general:



$b_i \in \{0,1\}$

Algoritmul lui Hierholzer

- **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**

Complexitate – O(m)

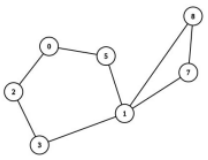
- Grafuri Hamiltoniene:

Conectivitatea κ(G) unui graf G este este marimea minima a unei mulțimi de tăiere a lui G.
Cum o calculăm ? -> **Flux maxim = taietura minima.**

Definiții ajutătoare

Conectivitatea κ(G) unui graf G este este marimea minima a unei mulțimi de tăiere a lui G.
O mulțime de noduri a unui graf G este independentă dacă nu contine noduri adiacente. Numărul de independență α(G) al unui graf G este marimea cea mai mare posibila a unei mulțimi independente a lui G.

3 (una din solutii este {0,3,8})



- Teorema lui Chvatal si Erdos

Fie G un graf conectat cu ordinal n ≥ 3, conectivitatea κ(G), și numărul de independență α(G). Daca κ(G) ≥ α(G), atunci G este hamiltonian

Cum găsim un ciclu hamiltonian ?

O a doua soluție este sa folosim un algoritm exponential mai eficient:
Vom considera matricea `C` având următoarea semnificaţie: `C[j][k]` este costul minim al unui lanţ ce începe în nodul `1`, se termină în nodul `k` şi conţine toate nodurile identificate cu `1` în configuraţia binară a lui `j` exact o singură dată. De exemplu, pentru graful din enunţ, starea caracterizată de tripletul `(4, 23, 0)` va avea valoarea `7` şi va reprezenta costul minim al unui lanţ ce începe în nodul `4`, se termină în nodul `0` şi conţine exact o singură dată nodurile `{0, 1, 2, 4}`, deoarece `23 = (10111)`$_2$ (ordinea biţilor este considerată cea inversă scrierii în baza `2`).

Solutia are complexitatea O(2^n *n)

Graf planar

- **Teorema celor 6 culori**

  Orice graf planar conex este 6 -colorabil.

- **Algoritm de colorare a unui graf planar cu 6 culori**

```
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)

Distanța Levenshtein

- Distanța Levenshtein este un număr care ne spune cât de diferite sunt două cuvinte. Considerăm cuvintele indexate de la 1 în momentul în care le reținem în memorie.

- $$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & , dacă \min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & , altfel \end{cases}$$