**Notiuni introductive**

Grad in graf orientat: d-(x) = grad interior - muchii care merg in x, d+(x) = grad exterior - muchii care pleaca din x, d(x) = d+(x) + d-(x)

Suma d+ = Suma d- = nr. de muchii

Muchia este incidenta cu x ⬄ muchia are un capat in x

Muchiile e si f sunt adiacente ⬄ au un capat in comun

Drum (walk) = orice secventa de varfuri cu arce intre ele

Drum simplu (trail) = nu se repeta arce

Drum elementar (path) = nu se repeta varfuri (e si simplu)

Lungimea drumului = nr. de arce = cardinalul multisetului arcelor = nr. de noduri - 1

Circuit = drum cu identice capete (bucla)

Circuit poate fi simplu si elementar

Pentru un graf neorientat, avem lant si ciclu (similar cu drum si circuit din graful orientat)

Graf partial – are toate nodurile, dar posibil nu toate muchiile

Subgraf – nu are necesar nici toate muchiile nici toate nodurile

Subgraf indus – are toate muchiile pe care poate sa le aiba (exista muchie A->B si graful are A, B => are A->B)

Componenta conexa – subgraf indus conex maximal (i.e. subgrafurile componentei nu sunt componente)

In graf orientat – conex nu implica “exista drum intre orice doua noduri” – asta e **tare conexitate**

Grafuri izomorfe ⬄ G1 se poate transforma in G2 prin rearanjarea numerelor nodurilor

⬄ sunt identice daca nodurile nu sunt etichetate

⬄ au aceeasi structura ⬄ se pot reprezenta prin acelasi desen (ignorand numerele)

Graf bipartit ⬄ graf neorientat cu doua multimi a.i. orice muchie este intre cele doua multimi ⬄ bicolorabil

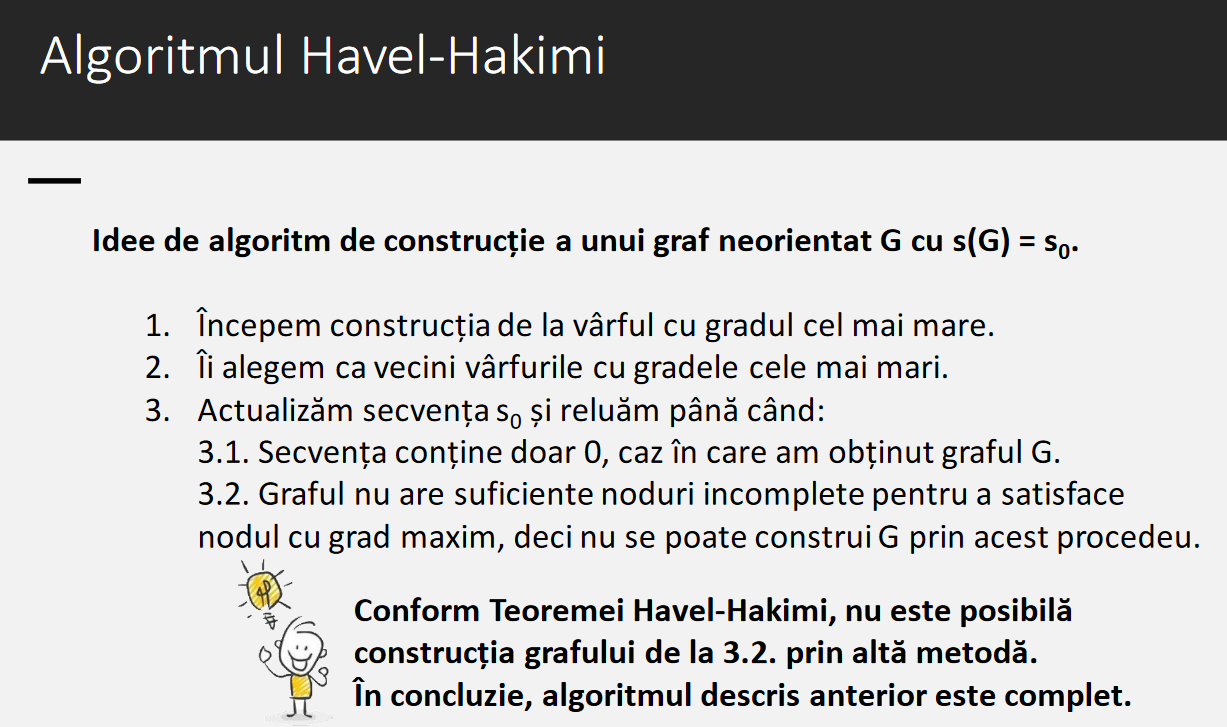
Pn ⬄ lant elemntar cu n noduri ⬄ 1 – 2 – 3 - … - n

Cn ⬄ ciclu elementar cu n noduri ⬄ 1 – 2 – 3 - … - n - 1 (Pn cu o muchia de la n la 1)

Kn ⬄ graf complet de grad n

Kp,q ⬄ graf bipartit complet cu multimi de marime p si q

Graf complementar ⬄ inversezi existenta muchiilor ⬄ orice muchie in G nu e in G̅ si vice versa

Graf transpus ⬄ graf orientat cu directiile inversate

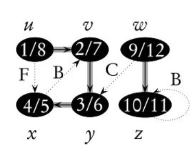
BFS – O(n + m)

DFS – O(n + m)

**Parcurgerea in DFS**

Se poate tine cont de timpurile parcurgerii (open/close)

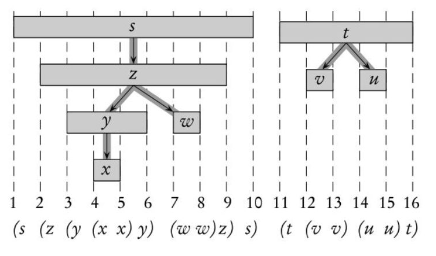
Cand ajungem intr-un nod sau cand terminam de vizitat vecinii unui nod incrementam un contor



Teorema parantezelor

Pt. cautare DFS a unui graf, pt. u sau v, exact o conditie e adevarata:

1. Intervalele [d[u],f[u]] si [d[v],f[v]] sunt disjunce
2. Intervalul [d[u],f[u]] e continut in [d[v],f[v]], iar u e descendent al lui v in arborele DFS
3. Conditia b, invers

Cororal – v e descendent al lui u ⬄ d[u] < d[v] < f[v] < f[u]

Muchiile in arbore DFS:

1. Muchiile de arbore – (u, v) muchie de arbore ⬄ v a fost descoperit explorand muchia (u,v)
2. Muchiile inapoi – unesc u cu un stramos v in arborele DFS
3. Muchiile inainte – unesc u cu un descendent v
4. Muchiile transversale – toate celelate muchii

Intr-un graf neorientat avem doar primele 2 categorii

**Sortarea topologica** – are logica doar in graf orientat (altfel deadlock imediat)

Nu este unica, exista daca graful este aciclic

Este sortarea in ordinea dependentelor ⬄ exista muchia u->v implica u se afla inaintea lui v

Algoritm pentru gasirae unei sortari topologice, simplu, similar BFS:

Initializam coada Q cu toate varfurile fara dependente (grad intern 0), while (Q not empty) { extragem varf din coada, eliminam varful (scadem gradele interne ale vecinilor), adaugam toti vecinii al caror grad devine 0}

Exista algoritm bazat pe parcurgere DFS – sortarea descrescatoare in raport cu f[v] (de la open/close)

Muchie critica ⬄ stergerea muchiei ar deconecta o componenta conexa ⬄ nu este intr-un ciclu

Nod critic ⬄ stergerea nodului ar cresce nr. de comp. conexe (un nod izolat sau frunza nu e critic)

**Gasirea muchiilor si nodurilor critice folosind arborele DFS in graf neorientat**

niv\_min[i] = nivelul minim al unui varf la care se duce o muchie de intoarcere din i sau dintr-un descendent ar lui i

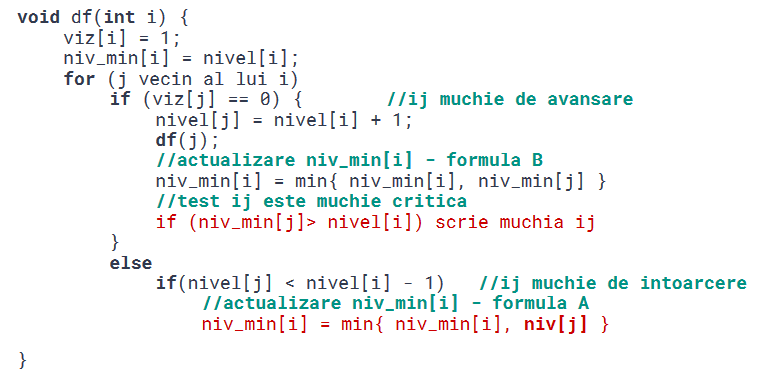
nivel[i] = nivelul lui i in arborele DFS

niv\_min[i] = min(nivel[i], min(nivel[k] | j-k muchie de intoarcere, j = i sau j descendent al lui i))

O muchie de avansare i-j este cricita ⬄ niv\_min[j] > nivel[i]

O muchie de intoarcere nu poate fi critica (inchide un ciclu)

niv\_min se poate calcula eficient recursiv: Cand descoperim o muchie de intoarcere, actualizam niv\_min daca nivelul nou descoperit e mai mic. Cand facem return dintr-un nod x intr-un nod y, test muchie critica: niv\_min[x] > nivel[y] inseamna x-y critica, actualizam niv\_min[y] daca niv\_min[x] e mai mic.



Noduri critice:

Radacina e critica ⬄ are cel putin 2 fii in arborele DFS (nu exista muchii intre cei 2 sau mai multi subarbori, altfel radacina ar avea un singur fiu)

Alt nod i e critic ⬄ are cel putin un fiu j cu niv\_min[j] >= nivel[i] in arborele DFS (i.e. nu e alta metoda decat nodul i de a accesa nodul j)

**Componente tare conexe in graf orientat folosind DFS (Algoritm Kosaraju)**

1. Calculeaza timpii de terminare f[u] pentru fiecare varf u folosind DFS
2. Calculeaza G transpus (directiile muchiilor inversate)
3. DFS pe G transpus, dar in bucla principala a DFS, considera varfurile in ordinea descrescatoare a timpilor calculati la pasul 1
4. Fiecare arbore din padurea DFS din pasul 3 e o componenta tare conexa separata

Arbore partial ⬄ graf partial care e arbore

Orice graf neorientat conex are arbore partial

APM (arbore partial de cost minim) in graf ponderat

**Algoritmul lui Kruskal**

La fiecare pas adaugam o muchie de cost minim din G care nu formeaza un ciclu cu muchiile deja selectate. Ordonam muchiile crescator dupa cost, se foloseste o structura pt. multimi disjuncte (union find). Complexitate O(m log n).

**Algoritmul lui Prim**

Se porneste dintr-un varf arbitrar. La fiecare pas e selectata o muchie de cost minim de la un varf deja adugat la un varf neadaugat.

Pentru fiecare nod neadaugat tinem cont de costul minum curent pt. adaugare si o muchie cu acel cost. Se foloseste priority-queue pentru a selecta un nod cost minim la fiecare pas (se poate folosi trucul lazy delete/lazy Djikstra pt. a nu fi nevoie de update in priority queue). O(m log n)

Daca graful e dens (m ~ n2) atunci e mai eficient sa cautam in O(n) la fiecare pas => O(n2)

**Implementarea de paduri disjuncte**

struct union\_find {

    std::vector<int> parent;

    std::vector<int> size;

    int find(int x) {

        assert(x < parent.size());

        int answer = x;

        while(parent[answer] != answer) {

            answer = parent[answer];

        }

        int second\_pass = x;

        while(parent[second\_pass] != second\_pass) {

            int next = parent[second\_pass];

            parent[second\_pass] = answer;

            second\_pass = next;

        }

        return answer;

    }

    void merge(int a, int b) {

        assert(a < parent.size());

        assert(b < parent.size());

        a = find(a);

        b = find(b);

        if(a == b) {

            return;

        }

        if(size[a] < size[b]) {

            std::swap(a, b);

        }

        parent[b] = a;

        size[a] += size[b];

    }

    void init(int n) {

        parent.resize(n);

        size.resize(n);

        for(int x = 0;x < n;x++) {

            parent[x] = x;

            size[x] = 1;

        }

    }

};

Optimizari importante:

1. Path Compression – find face doua parcurgeri, a doua optimizeaza calea la parinte
2. Union By Size – merge ataseaza arborele mai mic la arborele mai mare, pentru a nu crea arbori excesiv de adanci

**Drumuri minime in grafuri ponderate**

Problema drumurilor minime de sursa unica ⬄ determinarea arborelui distantelor fata de s (nu e echivalent cu APM)

**Algoritmul Dijkstra**

Presupunem ca arcele au cost pozitiv (graful poate contine circuite)

Daca toate arcele au cost egal Dijkstra ⬄ BFS

Pentru fiecare varf retinem: d[u] costul minim descoperit pana in acest moment, tata[u] predecesorul pt. drumul de cost minim (necesar pt. a reconstrui drumul de cost minim la final)

La un pas: selectam un varf u cu d[u] minim, actualizam d[v] si tata[v] pt. vecinii v ai lui u (relaxare)

Priority queue pt. selectarea de varf la fiecare pas (tehnica lazy Djikstra evita nevoia de update a costului in priority queue, deoarece deobicei libraria limbajului nu are o operatie de update cost)

Complexitate O(m log n), sau O(n2) daca nu folosim priority queue

Daca ne intereseaza o singura muchie x->y putem termina algoritmul odata ce a fost finalizat varful y (**nu** putem termina imediat ce a fost intalnit y, deoarece s-ar putea relaxa mai tarziu, trebuie sa astemptam pana ce alegem y pentru a fi siguri ca am gasit costul minim)

**Algoritmul Bellman-Ford**

Permite arce de cost negativ

La fiecare pas nu relaxam arcele dintr-un varf selecat, ci relaxam toate arcele

**function** BellmanFord(*list* vertices, *list* edges, *vertex* source) **is**

**for each** vertex v **in** vertices **do**

distance[v] := **inf**

predecessor[v] := **null**

distance[source] := 0

**repeat** |V|−1 **times**:

**for each** edge (u, v) **with** weight w **in** edges **do**

**if** distance[u] + w < distance[v] **then**

distance[v] := distance[u] + w

predecessor[v] := u

**for each** edge (u, v) **with** weight w **in** edges **do**

**if** distance[u] + w < distance[v] **then**

**error** "Graph contains a negative-weight cycle"

**return** distance, predecessor

Dupa pasul i, d[u] = costul minim la u cu max. i arce

Doar circuite accesibile din nodul de start ales sunt detectabile.

Complexitate O(nm)

Cu optimizarea de terminare daca nu se mai relaxeaza, se reduce la O(l \* m) unde l e lungimea maxima a unui drum de cost minim in graf

**Drumuri de sursa unica in grafuri aciclice**

Parcurgem nodurile in ordinea sortarii topologice, relaxam. O(m + n)

**Drumurile minime intre toate perechile de varuri – Algoritul Floyd-Warshall – O(n3)**

Fie W matricea costuilor – w(i,j) = 0 daca i = j, infinit daca nu exista muchie, altfel costul muchiei i->j

Vrem sa calculam matricea distantelor D. Se poate retine si matricea predecesor P pt. reconstruire.

**let** dist be

|

V

|

×

|

V

|

array of minimum distances initialized to

∞

infinity

**let** prev be

|

V

|

×

|

V

|

array of vertex indices initialized to **null**

**procedure** *FloydWarshallWithPathReconstruction*() **is**

**for each** edge (u, v) **do**

dist[u][v] ← w(u, v) *// The weight of the edge (u, v)*

prev[u][v] ← u

**for each** vertex v **do**

dist[v][v] ← 0

prev[v][v] ← v

**for** k **from** 1 **to** |V| **do** *// standard Floyd-Warshall implementation*

**for** i **from** 1 **to** |V|

**for** j **from** 1 **to** |V|

**if** dist[i][j] > dist[i][k] + dist[k][j] **then**

dist[i][j] ← dist[i][k] + dist[k][j]

prev[i][j] ← prev[k][j]

**procedure** Path(u, v)

**if** prev[u][v] = null **then**

**return** []

path ← [v]

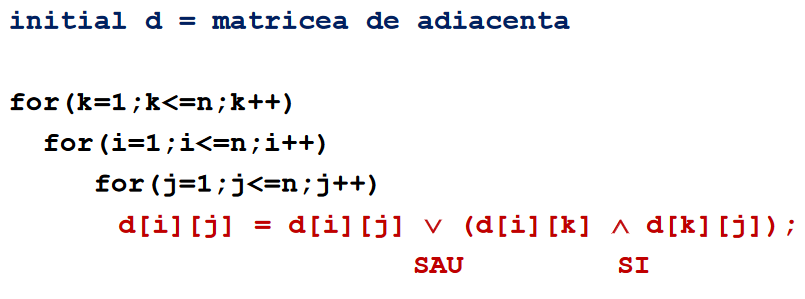
**while** *u* ≠ *v*

v ← prev[u][v]

path.prepend(v)

**return** path

**Inchiderea tranzitiva a grafului orientat neponderat – Algoritmul Roy-Warshall**

****

Pt. A matricea de adiacenta, Ak puterea k a matricii, pe pozitia i,j = numarul de drumuri distincte de lungime k de la i la j (nu neaparat elementare)

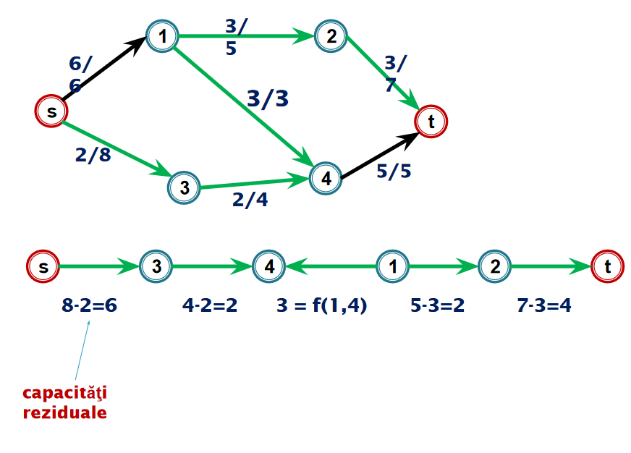
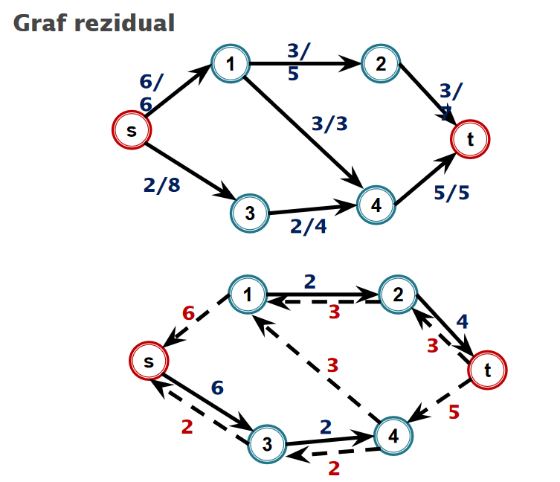
**Flux**

G – graf orientat se numeste retea de transport. Fiecare arc are capacitate

Daca sunt multiple surse sau destinatii, se poate transforma in singura sursa si destinatie

**Algoritmul Ford-Fulkersor de flux maxim (si taietura minima)**

Definitie s-t lant f-nesaturat – un lant din arc direct si/sau arc invers care creste fluxul

Capacitatea reziduala a lantului e min{6,2,3,2,4}

O taietura e o bipartitie(X,Y) a retelei a.i. sursa in X si destinatie in Y

Capacitatea taieturii e suma cap. arcelor din X in Y

Cap. taieturii minime = fluxul maxim

**Fork-Fulkerson**

Cat timp exista un s-t lant f-nesaturat P in G: revizuieste fluxul

Pt. taietura minima, la finalul algoritmului, consideram X = multimea varfurilor accesibile din s, taietura = (X, V-X)

O(m\*f) unde f este fluxul maxim

Se gaseste lant prin parcurgerea din sursa, ignorand arce de capacitate 0

Ordinea nu e specificata in Ford-Fulkerson

**Algoritmul Edmonds-Karp = Ford-Fulkerson in care se alege lantul cu lungime minima (prin BFS)**

O(nm2)

**Cuplaj maxim intr-un graf bipartit folosind flux – complexitate O(mn)**

**Constructia unui graf orientat din secvente de grade folosind flux – O(m2)**

**Grafuri Euleriene**

Ciclu eulerian ⬄ciclu cu toate muchiile

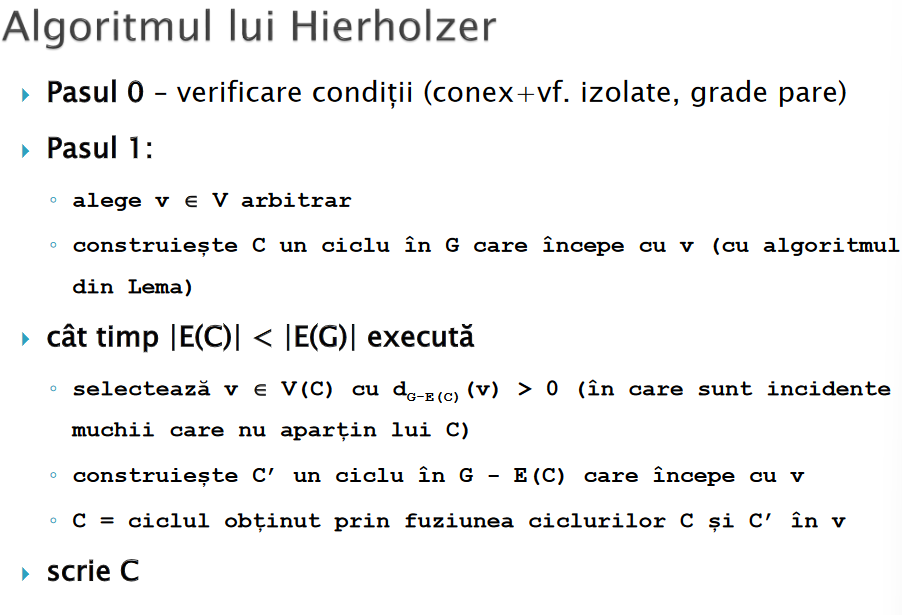
Lant eulerian ⬄lant simplu cu toate muchiile

Graf eulerian ⬄ contine un ciclu eulerian

G (multi)graf neorientat este eulerian ⬄ Orice nod din G are grad par

G graf neorientat conex are lant eulerian ⬄ G are cel mult doua varfuri de grad impar

**Algoritmul lui Hierholzer – ciclu eulerian intr-un graf conex (+ varfuri izolate) - O(m)**



**Graf planar**

G este planar ⬄ muchiile se pot reprezenta in plan a.i. muchiile nu se intersecteaza

Fie M o harta a G

M induce o impartile intr-o multile F de fete a planului (inclusiv fata exterioara infinita)

Pentru o fata f definim d(f) = gradul fetei f = numarul muchiilor lantului care delimiteaza f

Avem suma d(f) = 2m

Teorema poliedrala a lui Euler: N – E + F = 2 (pentru orice harta)

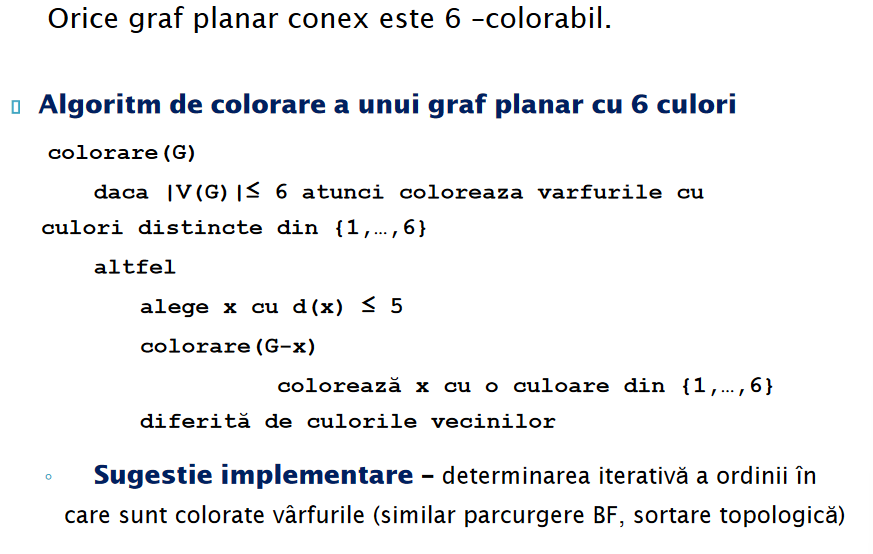
Proprietati:

1. m <= 3n – 6
2. Exista nod x cu d(x) <= 5

Pentru G bipartit cu n > 2

1. m <= 2n – 4
2. Exista nod x cu d(x) <= 3

**Teorema celor 6 culori: orice graf planar conex este 6-colorabil**

****

**Teorema celor 5 culori: orice graf planar conex este 5-colorabil**

**Graf hamiltonian**

Ciclu hamiltonian ⬄ contine toate nodurile

Grafurile hamiltoniene => nu are noduri critice (conditie necesara dar nu suficienta)

Fiecare nod are grad par (si conex except. izolate) => Graf hamiltonian (conditie necesara si suficienta)

Teorema lui Dirac: Fie G cu n >= 3. Daca d(x) >= n/2 for all x atunci G Hamiltonian

Teorema lui Ore: Fie G cu n >= 3. Daca pentru orice x,y neadiactente d(x) + d(y) >= n atunci Ham.

Conectivitatea K(G) marimea minima a unei multimi de taiere a lui G (calculam prin flux)

O multime independenta nu contine noduri adiacente. Nr. de indepedenta α(G) al unui graf G este

marimea cea mai mare posibila a independente

Teorema lui Chvatal si Erdos: Fie G conectat cu n>=3, daca K(G) >= α(G) atunci G Hamiltonian

Teorema lui Goodman si Hedetniemi

Daca G este 2-conectat si liber de {K1,3 Z1} atunci G este hamiltonian

**Algoritmul Held-Karp**

Gaseste ciclu Hamiltonian in timp exponential. Merge pe graf ponderat (TSP)

**function** algorithm TSP (G, n) **is**

**for** k := 2 to n **do**

g({k}, k) := d(1, k)

**end for**

**for** s := 2 **to** n−1 **do**

**for all** S ⊆ {2, ..., n}, |S| = s **do**

**for all** k ∈ S **do**

g(S, k) := minm≠k,m∈S [g(S\{k}, m) + d(m, k)]

**end for**

**end for**

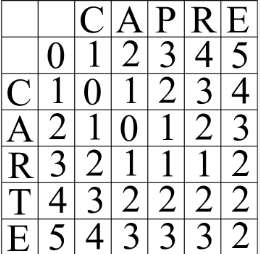
**end for**

opt := mink≠1 [g({2, 3, ..., n}, k) + d(k, 1)]

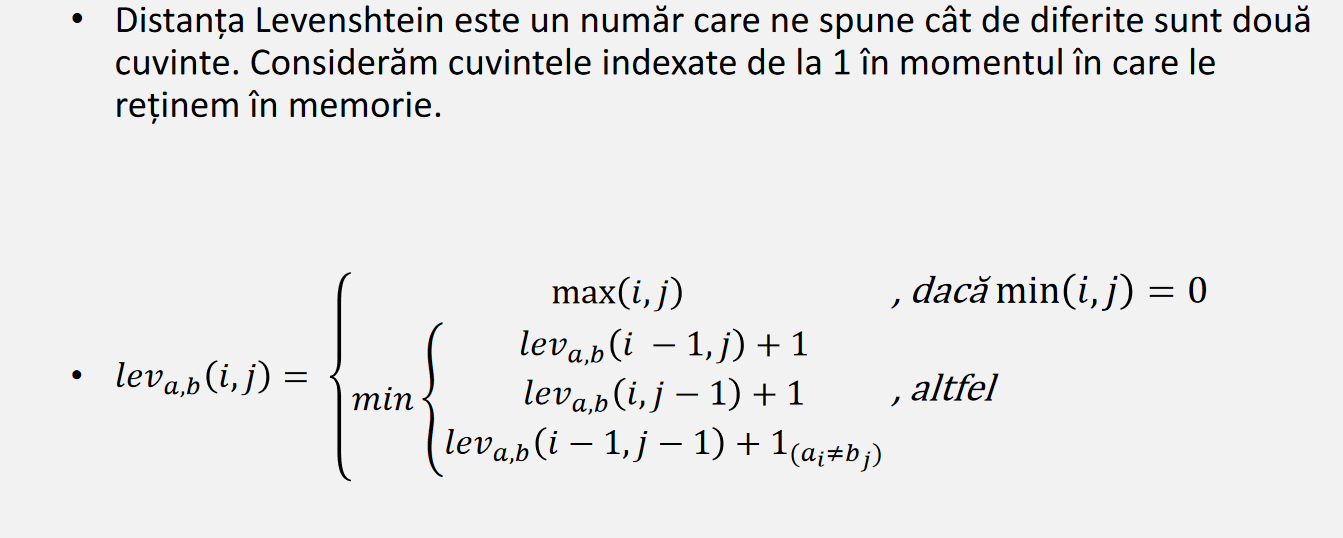
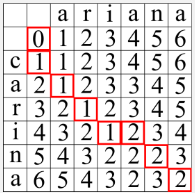
**return** (opt)

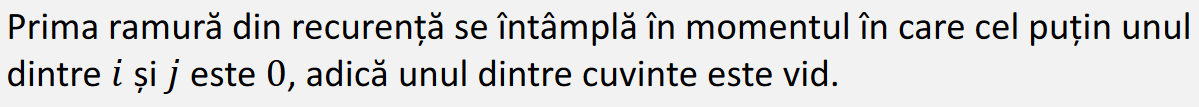
**end function**

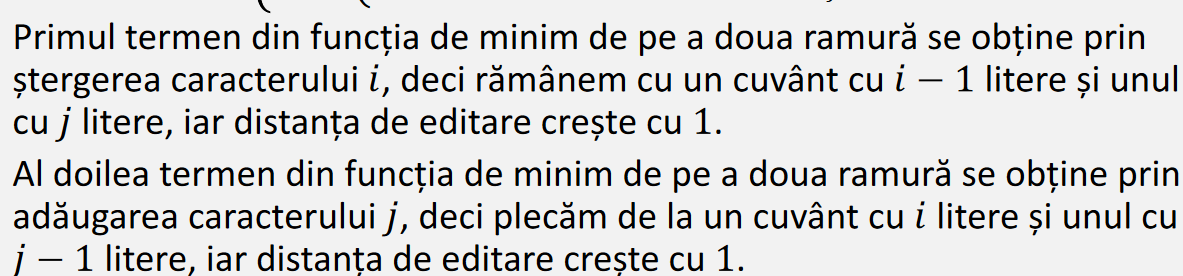
O(2nn2)

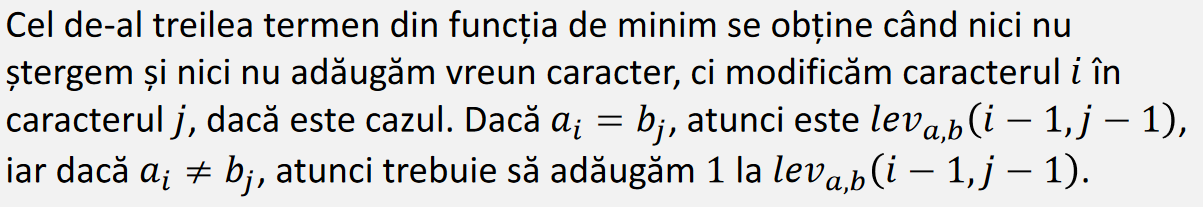
**Distanta Levenshtein**

Operatii: inserare caracter, stergere caracter, editare caracter









function LevenshteinDistance(char s[1..m], char t[1..n]):

// for all i and j, d[i,j] will hold the Levenshtein distance between

// the first i characters of s and the first j characters of t

declare int d[0..m, 0..n]

set each element in d to zero

// source prefixes can be transformed into empty string by

// dropping all characters

for i from 1 to m:

d[i, 0] := i

// target prefixes can be reached from empty source prefix

// by inserting every character

for j from 1 to n:

d[0, j] := j

for j from 1 to n:

for i from 1 to m:

if s[i] = t[j]:

substitutionCost := 0

else:

substitutionCost := 1

d[i, j] := minimum(d[i-1, j] + 1, // deletion

d[i, j-1] + 1, // insertion

d[i-1, j-1] + substitutionCost) // substitution

return d[m, n]