**Grafuri**

**Notiuni:**

NG (u) = mulțimea vecinilor lui u

**Grad intern** - d-(v) - v este destinatia

**Grad extern** - d+(v) - v este sursa

**Drum simplu -** drum care nu contine aceeasi muchie x2

**Drum elementar -** drum care nu contine acelasi varf x2

**Circuit** - un drum simplu cu capetele identice

**Circuit simplu -** drumul asociat este simplu

**Circuit elementar-** drumul asociat este elementar

**Lant -** secventa de varfuri adiacente

**Graf partial -** graf cu aceleasi varfuri si muchii scoase

**Subgraf -** graf cu varfuri si muchii scoase

**Subgraf indus -** garf cu varfuri scoase si toate muchiile cu extremitatile in varfurile existente

**Graf conex -** intre oricare varfuri exista drum

**Componenta conexa -** subgraf indus conex maximal

**Graf bipartit -** doua multimi disjuncte

**Graf transpus -** se schimba sensul muchiilor

**Doua grafuri sunt izomorfe -** pastreaza adiacenta si neadiacenta

**Reprezentari:**

**Lista de adiacenta**

**Lista de muchii**

**Matricea de adiacenta**

**Parcurgeri:**

**BFS -** exploreaza in latime, parcurgand toti vecinii nodului curent

- se gaseste astfel si distanta de la un nod la toate celelalte

- se foloseste o coada la implementare

- se scoate un nod din coada, apoi se adauga toti vecinii lui

- **complexitate:** O(E + V)

**DFS -** exploreaza in adancime, pana la frunze

- se gasesc componentele conexe

- se poate implementa recursiv/stack

- se parcurge pana cand nu mai are fii nodul curent

- **complexitate:** O(E + V)

- **clasificare muchii arbore dfs:**

**-** muchii de arbore: basic

- muchii inapoi: unesc un stramos cu un varf, consecinta a ciclurilor in graf

- muchii inainte: conecteaaza un varf cu un fiu

- muchii traversale: celelalte

# **Algoritmul lui Kahn**

**Sortarea topologica -** parcurge graful in functie de dependenta muchiilor

- daca graful este aciclic -> are o sortarea topologica (daca este ciclic - cum alegi primul nod?)

- daca graful este aciclic -> are cel putin un varf cu gradul intern = 0

- ne folosim de o coada

- adaugam in coada toate nodurile cu gradul intern 0

- extragem un nod

- il scoatem din graf si scadem pentru fiecare vecin gradul intern cu 1

- adaugam nodurile vecine cand gradul intern este 0

- **complexitate:** O(V + E)

**Muchii de avansare** - ale arborelui DFS - muchii prin care se descopera noi varfuri

**Muchii de intoarcere -**  inchid ciclul

**Muchii critice -** muchie care nu este continuta intr-un ciclu

- nu este continuta intr-un ciclu inchis de o muchie de intoarcere

- consideram muchia i - j -> i - j este critica <=> nu exista nicio muchie cu o extremitate in j sau descendentii sai si cealalta extremitate in i sau ascendentii sai

- algoritmul retine timpul descoperirii si nivelul in arborele dfs

- muchie crtitice <=> timpul descoperirii > nivelul

- prin eliminarea muchiei critice se creaza componenta conexe

**Punct critic** - la fel ca sus

**Componentele tare conexe**

**Graf slab conex -** presupunem ca e graf neorientat si exsita drum de la oricare nod la oricare

**Graf tare conex -**  exista drum de la oricare nod la oricare

Componentele tare conexe ale lui G **sunt acelasi** cu cele ale grafului transpus.

# **Algoritmul lui Kosaraju**

**Determinarea componentelor conexe -** ne folosim de un stack

- apelam DFS pe graf si retinem pentru fiecare nod data cand a fost apelat

- adaugam in stack cand a fost parcurs un nod

- calculam graful transpus

- facem pop din stack si apelam DFS pe graful transpus

- se printeaza componentele in ordinea crescatoare a lungimii

# **Algoritmul Havel-Hakimi**

**Construcția de grafuri cu secvența gradelor dată**

**-** se verfica intai ca suma gradelor sa fie numar par

- se verifica ca gradul maxim <= # nodurilor - 1

- se sorteaza nodurilor descrescator dupa grade

- se alege nodul cu cel mai mare grad si se conecteaza cu urmatoarele noduri, luate descrescator dupa grade

- se elimina nodul ales si se actualizeaza lista

- se repeta procedeul pana cand fie se ajunge sa nu mai fie niciun nod, caz in care se pate construi graful

- fie raman noduri cu grade nenule, caz in care nu se poate construi graful

**Arbori partiali**

Orice graf neorientat **conex** contine un arbore partial.

# Union Find

**Structura de date pentru multimi disjuncte**

- initializam un hash map in care tinem nodul si parintele acestuia

- la inceput fiecare nod se are pe le ca si parinte

- avem doua operatii: union si find

- find gaseste parintele noodului

- union uneste doua componente disjuncte

- operatiile sunt de **complexitate** logaritmica(cu optimizarea)

**Optimizare:** path compression

- cand unim doua componente mutam toti fii directi ai radacinii sa pointeze catrea cealalta radacina

**Optimizare:** union by rank

- tinem intr-un vector dimensiunea componentei si in functie de aceasta decidem sa unim componenta cu inaltimea mai mica in cea mare

# **Algoritmul lui Kruskal**

**Arbore partial de cost minim**

**-** se sorteaza muchiile in ordinea crescatoare a costurilor

- ne folosim de union find

- se alege pe rand muchia cu costul cel mai mic si care nu apartine unei componente deja existente

- reunim muchia cu componenta deja existenta

- repetam pana cand formam apm-ul

- **complexitate:** O(m logn)

**Clustering**

Un k-clustering este o partitionare a unei multimi in k submultimi nevide(numite clase sau clustere).

**Gradul de separe -** distanta minima intre doua obiecte din clase diferite

= distanta minima dintre doua clase ale multimii

- calculeaza distanta intre oricare doua clase

- gradul de separare este minimul dintre distante

Se aplica Kruskal de **n-k** ori si se obtine **k-clustering cu grad de separare maxim.**

# **Algoritmul lui Prim**

**Arbore partial de cost minim**

- ne alegem un nod random de start

- bagam intr-un heap toate muchiile

- tinem un vector/set de noduri vizitate

- luam pe rand muchiile si verificam daca cel putin un nod din muchie nu a fost vizitat(nu creem un ciclu)

- daca da, adaugam noua muchie si marcam noul nod ca vizitat

- daca nu, ignoram

- repetam pana cand am format apm-ul

- **complexitate(heap) - O(m logn)**

**Diferente Kruskal vs Prim:**

- kruskal construieste paduri pe care apoi le uneste

- prim construieste un arbore

# **Algoritmul lui Dijkstra**

**Drumul de distanta minima de la un nod la restul(costuri > 0)**

- tinem un vector cu nodurile vizitate

- tinem un vector cu distantele de la nodul de start la restul, initializandu-l cu infinit pentru toate nodurile, mai putin cel de start, pe care il initializam cu 0

- alegem la fiecare pas nodul cu distanta minima fata de cel curent

- actualizam pentru fiecare vecin de al sau distantele minime

- nodul curent este nodul cel mai apropiat de cel anterior si il marcam ca vizitat

- reconstructia drumului tinem minte si un vector de tati

- **complexitate:** O(m log n)

# **Algoritmul lui Bellman-Ford**

**Drumul de distanta minima de la un nod la restul(costuri reale)**

- tinem un vector cu distantele de la nodul de start la restul, initializandu-l cu infinit pentru toate nodurile, mai putin cel de start, pe care il initializam cu 0

- reconstructia drumului tinem minte si un vector de tati

- actualizam pentru fiecare muchie costul(daca este mai mic decat cel din tabel)

- repetam de maxim **#n - 1** ori

- **complexitate:** O(mn)

**Bellman-Ford vs Dijkstara**

- bellman-ford merge si pentru costuri negative, deoarece dijkstra este greedy si va da skip la un traseu initial mai scump, dar care pe viitor se scurteaza

- ambele pot contine **circuite,** dar sa nu fie cu pondere **negativa**

**Drumuri minime de sursa unica in grafuri aciclice**

- facem sortarea topologica

- pentru fiecare nod in sortare actualizam distantele pentru toate muchiile adiacente ( ca la dijkstra)

- **complexitate:** O(m + n)

# **Algoritmul lui Floyd-Warshall**

**Drumuri minime intre toate perechile**

- se initializeaza matricea distantelor cu infinit daca nu exista distanta directa, altfel cu distanta directa

- avem trei for-uri imbricate (k, i, j) si verificam daca

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];

}

- **complexitate:** O(n^3)

**Floyd-Warshall vs Dijkstra:**

- dijstra nu ruleaza pe costuri negative, floyd da

- plus determinare cicluri de cost negativ - floyd

# **Algoritmul lui Roy-Warshall**

**Inchiderea tranzitiva a unei relatii**

- se creeaza o matrice nxn in care se pune true daca exista relatie intre I si j si false in caz contrat

- se ia indicele i de la 0..<n si pe rand fiecare coloana si randul i

- se face setul coloanei cuprinzand indicii de pe acea coloana care au valoarea adevarata

- asemanator si pentru rand

- se face produs cartezian intre aceste doua multimi si se bifeaza in matrice true pentru toate perechile+

- s-a obtinut matricea inchiderii tranzitive

# **Algoritmul lui Ford-Fulkerson**

**Fluxul maxim**

**-** aplicam un algoritm care sa ne gaseasca un drum de la nodul de start(s) la cel terminal(t)

- gasim de-alungul drumului bottleneck-ul(capacitatea maxima pe care o putem adauga la orice nod) si o adaugam tuturor nodurilor

- reluam cautarea pana cand fie nu mai exista path-uri augmentate

- pot aparea si situatii cand adaugam flux in directia opusa arcului(arc rezidual => graf rezidual)

- se poate folosi orice algoritm de gasire a drumului, cel predefinit fiind DFS

- **complexitate:** O(mF(luxul))

**Drum augmentat -** un drum in care putem adauga capacitate

# **Algoritmul lui Edmonds-Karp**

**Fluxul maxim**

- este Ford-Fulkerson ce utilizeaaza BFS

- **complexitate:** O(nm^2)

**Edmonds-Karp vs Ford-Fulkerson  
 -** edmonds ruleaza **independent** de capacitatile muchiilor, in timp de ford se poate adauga cate un flux/cautare, facandu-l ineficient, teoretic

**Graf bipartit**

**Problema p-colorarii -** colorarea varfurilor grafului in p culori astfel incat oricare doua varfuri adiacente nu au aceeasi culoare

Un graf bipartit - este un graf **2-**colorabil

= graf cu toate ciclucrile elementare pare

**Verificare graf bipartit** - folosim BFS

- coloram pe rand cate un nivel cu o culoare

- daca se intampla sa coloram un parinte si un copil cu aceeasi culoare, graful nu este biparit

- daca nu, este

# **Cuplax maxim in graf bipartit**

- adaugam doua noduri s(start) si t(terminal) pe care le conectam cu toate nodurile din cele doua multimi ale grafului

- setam capacitatile tuturor muchiilor 1 si fluxul 0

- rulam allgoritmul de flux maxim

- modificarea muchiilor capacitatilor din s sau t reprezinta cate/de cate ori se poate folosi/este folosit ceva

- **complexitate:** O(mn)

**Aplicatie - Constructia unui graf cu gradele date**

- se creaza doua multimi cu toate nodurile repetate

- se adauga s si se conecteaza cu toate nodurile dar capacitatea acum reprezinta gradul de iesire al nodului

- se adauga t, dar acum capacitatea reprezinta gradele de intrare

# **Taietura minima**

Taietura minima intr-un graf este egala cu fluxul maxim dintre cele doua noduri

Muchiile din taietura sunt saturate

**Grafuri euleriene**

**Ciclu eulerian -** traseu închis care trece o singură dată

prin toate muchiile

**Graf eulerian -** graf care are un ciclu eulerian

Un graf neorientat este eulerian daca orice varf are grad **par**

Un graf orientat este eulerian daca cel mult doua varfuri au grad **impar**

# **Algoritmul lui Hierholzer**

**Gasirea unui ciclu eulerian**

- verificam conditiile de existenta, cel mult un nod are grad interior - grad exterior = 1 si cel mult un nod are grad exterior - grad interior = 1 si restul au gradele egale

- numaram pentru fiecare nod gradul intern si extern

- alegem nodul care are grad exterior - grad interior = 1, in caz de multiplicitate, gradele sunt egeale, alegem orice nod

- aplicam DFS si substragem din tabele cu gradele externa cand trecem printr-un nod

- cand nodul curent are gradul extern 0 il adaugam la solutie

- daca am parcurs toate nodurile si gradele interne sunt 0 pentru toate nodurile, am gasit un ciclu eulerian

- solutia trebuie inversata

- **complexitate:** O(m)

**Grafuri hamiltoniene**

**Ciclu hamiltonian -** ciclu care contine toate nodurile grafului

**Graf hamiltonian -** graf care contine un ciclu hamiltonian

Graful este hamiltonian daca este **biconex**(nu are noduri critice)

**Conditie necesara si suficienta -** fiecare nod sa aibe grad par si graful sa fie conex

**Conditii suficiente**

**Teorema lui Dirac -** Fiecare nod are gradul >= n/2, pentru n >= 3

**Teorema lui Ore -** orice doua perechi de noduri neadiacente x si y au grad(x) + grad(y) >= n

**Teorema lui Goodman si Hedetniemi**

- Conectivitatea κ(G) unui graf G este este marimea minima a unei mulțimi de

tăiere a lui G.

- 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

**Gasirea unui ciclu hamiltonian**

- este o problema NP completa

- se poate face brute-force => **complexitate** - O(n! \* n)

- se folosesc bitmask-uri care marcheaza ce noduri am vizitat si ordinea in care am facut-o => **complexitate -** O(2^n \* n)