PA

- Algoritmi Grafuri Rezumat -

I. Grafuri

0) Itro .....

1) BFS .....
2) DFS ..... PARCURGERI

3) Sortare topologica cu DFS .....
4) Kahn ..... TOPOSORT

5) Kosaraju .....
6) Tarjan pentru CTC ..... COMPONENTE TARE CONEXE

7) Tarjan ..... PUNCTE CRITICE

8) Tarjan cu conditia modificata ..... PUNTI

9) Dijkstra .....
10) Bellman-Ford .....
11) Floyd-Warshall (Roy-Floyd) ..... DRUMURI MINIME
12) Johnson .....

13) Floyd-Warshall modificat ..... INCHIDEREA TRANZITIVA

13) Kruskal .....
14) Prim ..... ARBORI MINIMI DE ACOPERIRE

15) Ford-Fulkerson + Edmons-Karp ..... FLUX MAXIM

I. Intro

Tipuri:
- orientate
- neorientate

In graful G = (V, E), V – noduri.
E – muchii/ arce. u..v – drumul de la u la v.
R(u) = multimea nodurilor la care se poate ajunge din u (reachable).
succs(u) = succesorii lui u, pt grafuri orientate si nodurile adicacente lui u, pentru grafuri neorientate.
c(u) = culoarea nodului u.

Reprezentari:
- liste de adiacenta
- matrice de adiacenta (matricea este rara pentru grafuri rare) (sparce)
- vectori de adiacenta

Noduri:
- alb – neexplorat
- gri – in curs de explorare
- negru - explorat complet

Arce:
gri ----> alb (direct / de arbore)
gri ----> gri (invers / de ciclu)
gri ----> negru (inainte, cand d(u) < d(v) <=> cand primul a fost descoperit inaintea celui de-al 2 lea)
gri ----> negru (traversal) (inainte, cand d(v) < d(u) <=> cand primul a fost descoperit dupa cel de-al 2 lea)

II. BFS

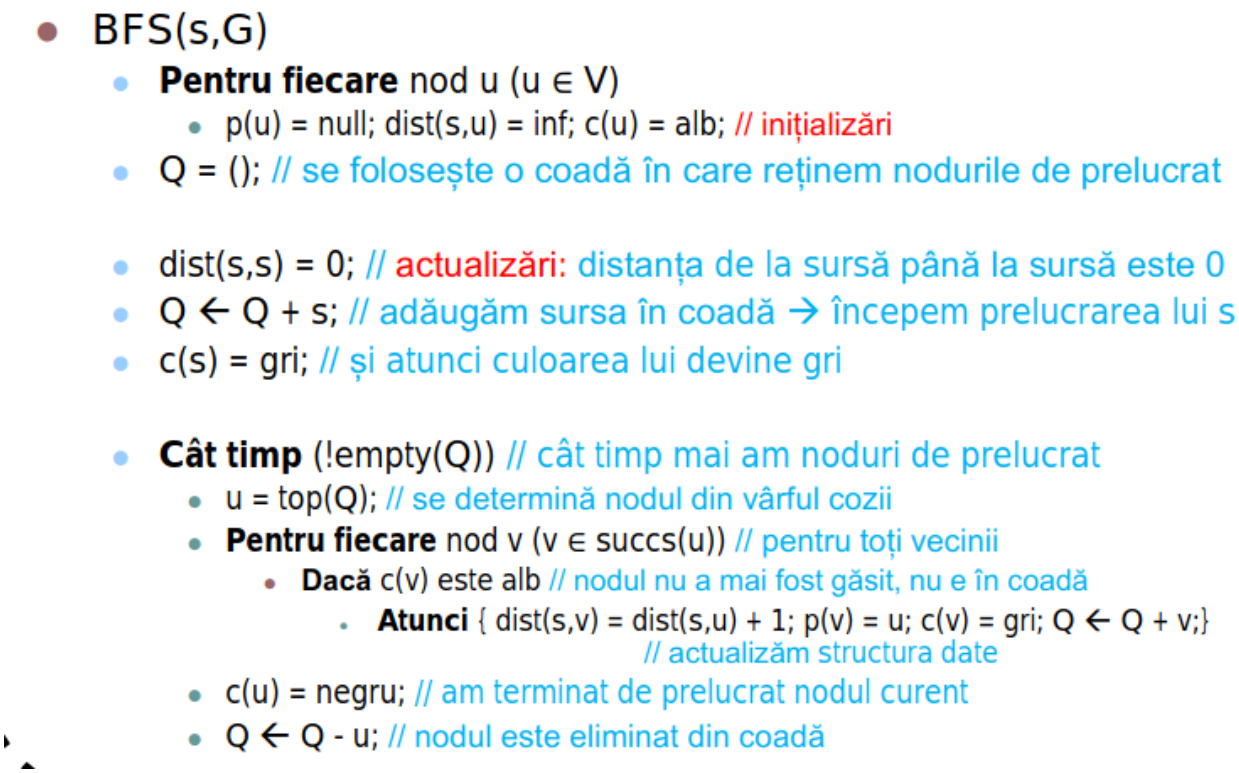
Proprietati:

- Pentru orice muchie (u,v) ∈ E, costul optim pana la v <= costul optim pana la u + 1
- dupa executia ciclului princial, coada contine nodurile in ordinea costului minim
- este corect, iar dupa derminare distanta de la sursa la nodul v este optima (drumul minim), pe grafuri neorientat, fara ponderi

Observatii:

- BFS gaseste drumuri minime intr-un graf neorientat, fara ponderi.
- BFS nu exploreaz anodurile care nu sunt reachable din sursa.

Complexitate: O(V+ E)



Step by step:

Functioneaza cu coada (FIFO) si o multime de vizitati.
Se pune sursa in coada si in multimea de vizitati.
Se pun toate nodurile reachable din sursa si in vizitati.
Se parcurge coada si se pun toate nodurile reachable din nodul curent, care nu sunt vizitate.

II. DFS

d(u) = momentul la care a fost descoperit nodul u.
f(u) = momentul la care nodul u a fost explorat complet.

Proprietati:

- Pentru fiecare v descoperit din u, este construibila o cale v, p(v), p(p(v)) … u – din parinte in parinte
- Sparge graful G intr-o padure de arbori. Toti arbori incep de la un nod ce nu are parinte.
- Daca DFS genereaza un singur arbore, G este conex.
- Teorema parantezelor: fie I(u) intervalul de prelucrare al nodului u ~ I(u) = (d(u), f (u)) ~. Pentru orice noduri diferite intre ele, u si v, intersectia I(u) si I(v) e multimea vida sau I(u) inclus in I (v) sau I(v) inclus in I(u). <=> Daca nodul v a fost descoperit dupa nodul u, explorarea lui v se va termina mereu inaintea explorarii lui u. <=> orice copil e finalizat inainte de parinti.
- Teorema drumurilor albe: v este descendent al lui u daca la un moment de timp d(u) exista o cale numai cu noduri albe u..v.
- Se pot descoperii doar muchii directe si inverse intr-un graf neorientat
- Se vor gasi arce inverse intr-un graf orientat ciclic.

Complexitate: O (V + E)

- DFS(G)
  - V = noduri(G)
  - **Pentru fiecare** nod u (u ∈ V)
    - c(u) = alb; p(u) = null; // inițializare structură date
  - timp = 0; // reține distanța de la rădăcina arborelui DFS pană la nodul curent
  - **Pentru fiecare** nod u (u ∈ V)
    - **Dacă** c(u) este alb
      - **Atunci** explorare(u); // explorez nodul

- explorare(u)
  - d(u) = ++ timp; // timpul de descoperire al nodului u
  - c(u) = gri; // nod în curs de explorare
  - **Pentru fiecare** nod v (v ∈ succs(u)) // încerc sa prelucrez vecinii
    - **Dacă** c(v) este alb
      - **Atunci** {p(v) = u; explorare(v);} // dacă nu au fost prelucrați deja
  - c(u) = negru; // am terminat de explorat nodul u
  - f(u) = ++ timp; // timpul de finalizare al nodului u

Step by step:

Foloseste un stack (FIFO) si o multime de vizitati.

Se initializeaza timpul de descoperire al sursei cu 0.
Se merge in primul descendent disponibil.
    Se marcheaza ca vizitat.
    Se incrementeaza timpul de descoperire.
    Se merge in urmatorul descendent, daca nu e vizitat.
Daca nu mai are descendenti face backtrack.
La backtrack se incrementeaza timpul de descoperire si se considera timp de finalizare pentru nod.
Merge cat timp exista noduri nevizitate.

IV. Sortarea topologica

G trebuie sa fie orientat si aciclic.

Doar pentru sortarea multimilor partial ordonate.

Sortatea topologica poate fi folosita pentru gasirea celui mai scurt drum intr-un graf ponderat, orientat, aciclic.

Daca exista undrum Hamiltonian in graf (un drum care sa treaca prin toate nodurile fix o singura data), atunci sortarea topologica este unica.

IV.I. Algoritm bazat pe DFS.

Step by step:

Se face DFS pe graf si apoi se ordoneaza nodurile in ordinea inversa timpilor de finalizare (sau, cand nodul se marcheaza negru, se introduce intr-un stack).

IV.II. Kahn.

```
TopSort(G) {
    V = noduri(G)
    L = vida;// lista care va contine elementele sortate
    // initializare S cu nodurile care nu au in-muchii
    foreach (u ∈ V) {
        if (u nu are in-muchii)
            S = S + u;
    }
    while (!empty(S)) { // cat timp mai am noduri de prelucrat
        u = random(S); // se scoate un nod din multimea S
        L = L + u;  // adaug U la lista finala
        foreach v ∈ succs(u) { // pentru toti vecinii
            sterge u-v; // sterge muchia u-v
            if (v nu are in-muchii)
                S = S + v;  // adauga v la multimea S
        } // close foreach
    } //close while
    if (G are muchii)
        print(eroare);  // graf ciclic
    else
        print(L); // ordinea topologica
}
```

Step by step:

Primul nod este unul in care nu intra nicio muchie (oricare din ele).

Se marcheaza explorate si se adauga intr-o multime next.

La fiecare pas, se alege un nod in care nu intra nicio muchie care porneste dintr-un nod neexplorat (din multimea next).

Adaug nodul, sterg muchia (<=> scad indegree-ul nodului).

Nodurile pot fi vizitate aleator, cu un queue sau cu un stack pentru next.

V. Tare conexitate

Un graf orientat este tare conex daca exista o cale, in ambele directii (si de la u la v si de la v la u), catre orice nod u si v. <=> Cand pot ajunge in orice nod, pornind din orice nod.

O componenta tare conexa este un subgraf tare conex maximal (cu proprietatea ca nu se mai poate adauga nicio muchie fara a strica proprietatea de tare conexitate)

Intr-un graf orientat, Gm G' e componenta tate conexa a lui G daca toate nodurile lui G' sunt in acelasi subarbore generat de DFS (pe G).

Pentru verificare: se porneste DFS din primul nod al fiecarei componente tare conexe. Daca nodurile compenentei sunt in acelasi subarborre, atubnci e ok.

Fie Φ(u) un stramos descoperit prin DFS al lui u.

- f(u) <= f (Φ(u)) ----timpul de finalizare al lui u e mai mic decat al stramosului.
- pentru orice nod v reachable din u, f (Φ(v)) <= f (Φ(u))
- Φ(Φ(u)) = Φ(u)

Primul nod descporetin intr-o CTC va avea drept succesori toate celelalte noduri ale CTC.

Inlocuind componentele tare conexe cu noduri, se obtine un graf aciclic.

Daca doua noduri apartin aceleiasi CTC, au acelasi stamos.

V.I. Kosaraju

E nevoie de doua parcurgeri DFS. Prima parcurgere trebuie sa realizeze o sortare topologica. Cea de a doua parcurgere se va face pe graful transpus.

Complexitate: O (V + E)

```
ctc(G = (V, E))
        S <- stiva vida
        culoare[1..n] = alb
        cat timp exista un nod v din V care nu e pe stiva
                dfs(G, v)
        culoare[1..n] = alb
        cat timp S != stiva vida
                v = pop(S)
                dfsT(GT, v) /* toate nodurile ce pot fi vizitate din v fac

dfs(G, v)
        culoare[v] = gri
        pentru fiecare (v, u) din E
                daca culoare[u] == alb
                        dfs(u)
        push(S, v) // nodul este terminat de expandat, este pus pe stiva
        culoare[v] = negru

dfsT(G, v) – similar cu dfs(G, v): fara stiva, dar cu retinerea solutiei
```

Step by step:

Foloseste o multime de vizitati pentru primul DFS, un stack pentru a retine nodurile in ordinea timpilor de finalizare, dupa primul DFS si inca o multime de vizitati pentru al doilea DFS.

Analog dfs pentru construirea stack-ului.
Fac graful transpus si lucrez pe el.
Se da pop.

Se adauga nodul in multimea de vizitati si in componenta curenta.
Se pune in multimea de vizitati nodul in care se poate ajunge din nodul curent.
Se updateaza nodul curent.
Se repeta pana nu mai am unde sa merg.

Se da pop din stiva pana se gaseste un nod nevizitat.
Se repeta pana cand stiva e goala.

VI. Tarjan.

Lowlink value: cel mai mic nod ID reachable din nodul curent, inclusiv el insusi.

VI.I. Tarjan pentru CTC.

```
index = 0  // nivelul pe care este nodul în arborele DFS
S = empty // se folosește o stivă care se inițializează cu Ø
Pentru fiecare v din V
  • Dacă (v.index e nedefinit) atunci // se pornește DFS din fiecare nod pe care
      • Tarjan(v)                      // nu l-am vizitat încă

Tarjan(v)
  • v.index = index // se setează nivelul nodului v
  • v.lowlink = index // reține strămoșul nodului v
  • index = index + 1  // incrementez nivelul
  • S.push(v) // introduc v în stivă
  • Pentru fiecare (v, v') din E // se prelucrează succesorii lui v
      • Dacă (v'.index e nedefinit sau v' e în S) atunci // CTC deja identificate sunt ignorate
          • Dacă (v'.index e nedefinit) atunci Tarjan(v') // dacă nu a fost vizitat v' intru în  recursivitate
          • v.lowlink = min(v.lowlink, v'.lowlink) //actualizez strămoșul
  • Dacă (v.lowlink == v.index) atunci // printez CTC începând de la coadă spre rădăcină
      • print "CTC:"
      • Repetă
          • v' = S.pop // extrag nodul din stiva și îl printez
          • print v'
      • Până când (v' == v) // până când extrag rădăcina
```

Tarjan are un inavariant pentru a impiedica DFS-ul sa aleaga noduri in ordine aleatoare: structura de date care retine noduri valide din care sa se updateze valorile de lowlink. Nodurile sunt adaugate cand sunt explorate pentru prima daata si sunt eliminate cand se descopera CTC din care fac parte.

Complexitate: O (V + E)

<u>Step by step:</u>

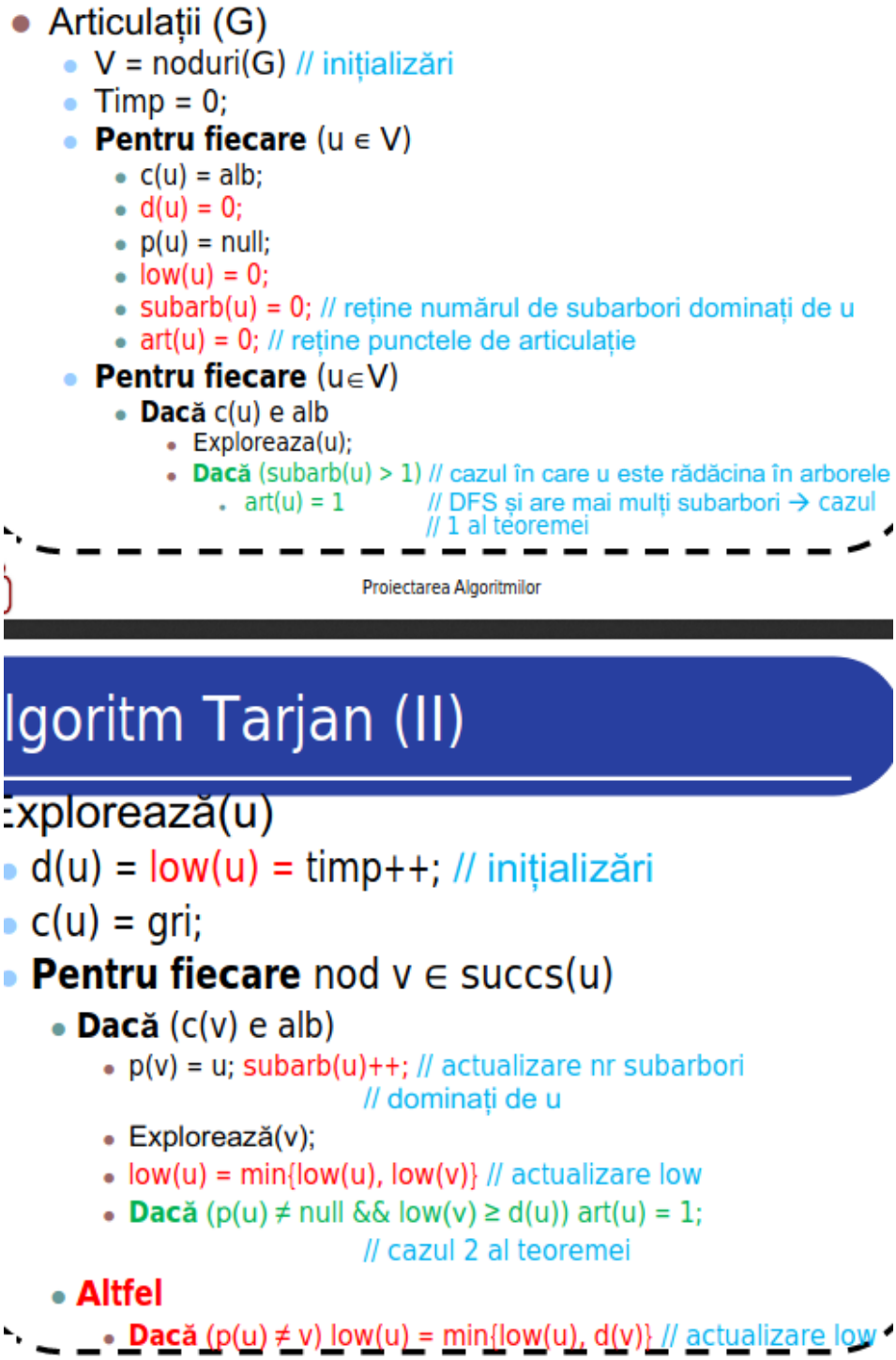
Foloseste un Stack si un array pentru a retine low-ul nodurilor.
Se incepe exlorarea cu dfs dintr-un nod.
Se pune pe stack si i se atribuie un low (egal cu id-ul lui).
Merg in urmatorul vecin.
Cand vecinul e deja pe stack, low-ul nodului curent devine minimul dintre low-ul sau actual si cel al vecinului.
Cand ne intoarcem intr-un nod (backtrack de la DFS), daca nu mai are niciun vecin si daca are low = propriul id => s-a gasit o componenta conexa.
Toate nodurile din acea componenta (cu acelasi low) vor fi eliminate de pe stack.

VI.II. Tarjan pentru puncte critice.

Punct critic = punct prin care trece orice drum dintr-un graf.

Intr-un graf neorientat, u este punct de articulatie daca are una dintre proprietatie:

- nu are parinte si este parintele a 2 fii independenti
- are parinte si exista un descendent al lui in arborele rezultat din DFS -v, astfel incat timpul de descoperire al tuturor nodurilor adiacente lui v sa fie mai mari decat ale lui u. <=> in arbirele v sa nu exista muchii inapoi spre u sau spre un nod descoperit inainte lui u



Step by step:

Foloseste o multime pentru vizitati.
Se retine pentru fiecare nod timpul de descoperire, low-ul, parintele.
Se parcurge graful DFS.
Se pune nodul in vizitati si se updateaza timpul de descoperire, low-ul la propriul ID si parintele la nodul din care am pornit.
Daca urmeaza sa ajungem intr-un nod deja vizitat se merge inapoi (backtracking de la DFS)
Se verifica una din cele 2 conditii pt a testa daca e punct critic:
- parinte a 2 fii independenti
- discovery type al nodului <= low-ul tuturor nodurilor adiacente.
Daca da, e punct critic, daca nu
Se updateaza low-un nodulu curent cu id-ul nodului din care ne-am intors (daca e mai mic).

VI.III. Tarjan pentru punti.

Punte = muchia prin care trece orice drum, intr-un graf neorientat.

- Punți(G)
  - V = noduri(G) // inițializări
  - Timp = 0;
  - **Pentru fiecare** nod u (u ∈ V)
    - c(u) = alb;
    - d(u) = 0;
    - p(u) = null;
    - low(u) = 0;
    - punte(u) =0; // înlocuiește: subarb(u) = 0; art(u) = 0;
  - **Pentru fiecare** nod u (u ∈ V)
    - **Dacă** c(u) e alb
      - Explorează(u)

Step by step:

Analog puncte critice doar ca se schimba conditia:

- low (crt) > index (prev), la backtracking.

VII. Drumuri de cost minim.

Drumuri:

1) punct – multipunct: Dijkstra, ellman-Ford
2) multimunct – punct: se face G transpus si 1)
3) punct-punct: 1)
4) multipunct-multipunct: Roy-Floyd

Folositi pe grafuri orientate.

Proprietati:

- subdrumurile unui drum minim sunt drumuri optimale
- daca drumului s...uv este optim, atunci este egal cu costul optim al drumului s..u + ponderea muchiei u,v
- drumul optim s..v <= drumul optim s...u + ponderea muchiei u, v <=> orice drum optic are costul mai mic decat al altui drum.

VII.I. Dijkstra.

1) drumuri minime de la sursa catre celelalte noduri
2) nu merge cu muchii de cost negativ
3) complexitate: O (V + E)

Proprietati:

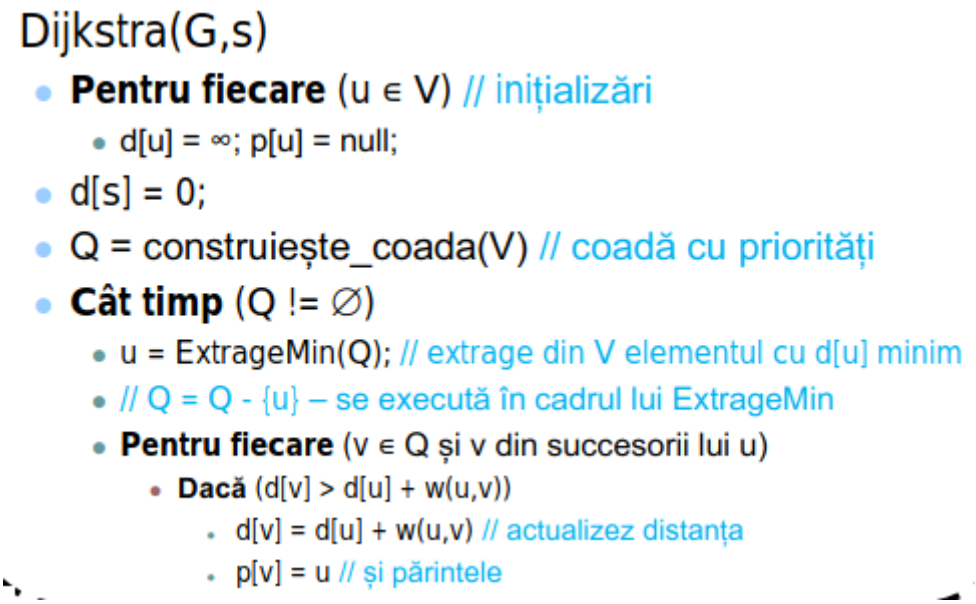
- obtine costul optim si odata atinsa aceasta valoare, nu se mai modifica
- pentru un drum optim s..uv, daca drumul s..u este optim la un moment de timp, atunci, imediat dupa relaxarea arcului u, v drumul s...u e optim.

Complexitatea (depinde de cum e graful si implementare)

- Grafuri DESE: O(v^2)
- Grafuri RARE: cu Heap Fribonacci: O (VlogV + E)
  cu Heap Binar: O (E logV)

- Vector:
  - insert – 1 * V = V;
  - delete – V * V = V2 (necesită căutarea minimului);
  - conține? – 1 * E = E;
  - micșorează_val – 1 * E = E;
  - este_vidă? – 1 * V = V;
- Heap binar:
  - structură de date de tip arbore binar + 2 constrângeri:
  - Fiecare nivel este complet; ultimul se umple de la stânga la dreapta; oricare u ∈ Heap; u ≥ răd(st(u)) și u ≥ răd(dr(u)) (u este ≥ decât ambii copii ai săi) unde ≥ este o relație de ordine pe mulțimea pe care sunt definite elementele heapului.
  - insert – logV * V = VlogV;
  - delete – logV * V = VlogV;
  - conține? – 1 * E = E;
  - micșorează_val – logV * E = ElogV;
  - este_vidă? – 1 * V = V.
  - Eficient dacă graful are arce puține comparativ cu numărul de noduri.
- Heap Fibonacci
  - Poate fi format din mai mulți arbori.
  - Cheia unui părinte ≤ cheia oricărui copil
  - Fiind dat un nod u și un heap H:
    - p(u) – părintele lui u;
    - copil(u) – legătura către unul din copiii lui u;
    - st(u), dr(u) – legătura la frații din stânga și din dreapta (cei de pe primul nivel sunt legați între ei astfel);

    - grad(u) – numărul de copii ai lui u;
    - min(H) – cel mai mic nod din H;
    - n(H) – numărul de noduri din H.
  - insert – 1 * V = V;
  - delete – logV * V = VlogV(amortizat!);
  - micșorează_val – 1 * E = E;
  - este_vidă? – 1 * V = V.
  - Cea mai rapidă structură dpdv teoretic.



Step by step:

Are nevoie de o multime de vizitati.
Fiecare nod retine de distanta pana la elinsusi.
Se alege o sursa si se updateaza distantele tuturor vecinilor.

Se alege nodul cu cea mai mica distanta.
Se aleg toate nodurile adiacentte, nevizitate.
Se updateaza distanta nodurilor vecine nodului ales, daca e mai mica.

VII.II. Bellman-Ford.

1) drumuri minime de la sursa catre celelalte noduri
2) detecteaza cicluri de cost negativ
3) complexitate O (V * E)

Daca d[v] nu se modifica la pasul i, atunci nu trebuie relaxat niciun arc care pleaca din v la momentul i + 1. => pastram o coada cu varfurile modificate.

Daca se face aceasta optimizare, nu mai pot fi detectate cicluri negative.

```
BellmanFord(G,s)   // G=(V,E),s=sursa
  Pentru fiecare nod v (v ∈ V) // inițializări
    d[v] = ∞;
    p[v] = null;
  d[s] = 0; // actualizare distanță de la s la s
  Pentru i de la 1 la |V| -1 // pentru fiecare pas pornind din s
  // spre restul nodurilor se încearcă construcția unor drumuri
  // optime de dimensiune i
    Pentru fiecare (u,v) din E
      // pentru arcele ce pleacă de la nodurile deja considerate
      Dacă d[v] > d[u] + w(u,v) atunci // se relaxează arcele corespunzătoare
        d[v] = d[u] + w(u,v);
        p[v] = u;
  Pentru fiecare (u,v) din E
    Dacă d[v] > d[u] + w(u,v) atunci
      Eroare ("ciclu negativ");
```

Step by step:

Pentru fiecare nod se retine distanta pana la el.
Se itereaza prin fiecare muchie.
Se updateaza costul din noduri daca e cazul.
Se repeta de n-1 ori.

VII.III. Floyd - Warshall (Roy – Floyd).

1) drumuri minime intre oricare 2 noduri.
2) nu detecteaza cicluri de cost negativ.
3) complexitate O(V^3).

Algoritmul se poate implementa cu o matrice 2D, pentru o complexitate spatiala O(V^2)

Algoritm bazat pe DP.
Recurentele:

- $d^0(i,j) = w(i,j);$
- $d^k(i,j) = \min\{d^{k-1}(i,j), d^{k-1}(i,k) + d^{k-1}(k,j)\}$, pentru $0 < k \leq n;$

- **Pentru** i **de la** 1 **la** n
  - **Pentru** j **de la** 1 **la** n // inițializ
    - d(i,j) = w(i,j)
    - **Dacă** (w(i,j) == ∞)
      - p(i,j) = null;
    - **Altfel** p(i,j) = i;
- **Pentru** k **de la** 1 **la** n
  - **Pentru** i **de la** 1 **la** n
    - **Pentru** j **de la** 1 **la** n
      - **Dacă** (d(i,j) > d(i,k) + d(k,j)) // dete
        - d(i,j) = d(i,k) + d(k,j)
        - p(i,j) = p(k,j); // și actualizăm

Step by step:

E nevoie de o matrice cu E linii si E coloane.

Se pune 0 pe diagonala principala.

Se completeaza distantele pentru nodurile ce pot fi accesate direct din nodul sursa.

La urmatorul pas, se mai pune la dispozitie un nod v. Se updateaza distantele de la sursa catre nodurile la care se poate ajunge trecand prin noul nod: de ex: le la u la v pot ajunge prin y: u..y → y..v. Daca acest cost este mai mic decat costul din tabel pentru u,v acesta se updateaza.

VII.IV. Floyd - Warshall pentru inchiderea tranzitiva.

Intr-un graf G(V, E), inchiderea tranzitiva lui E este un graf G* = (V, E*), unde

$$E^*(i,j)=\begin{cases}1, \text{ dacă } \exists\ i..j\\ 0, \text{ dacă } \nexists\ i..j\end{cases}$$

Se modifica Floyd-Warshall astfel:
min se inlocuieste cu sau
\+ se inlocuieste cu si

- **Pentru** i **de la** 1 **la** n
  - **Pentru** j **de la** 1 **la** n
    - E* (i,j) = (((i,j) ∈ E) ∨ (i = j)) // inițializări
- **Pentru** k **de la** 1 **la** n
  - **Pentru** i **de la** 1 **la** n
    - **Pentru** j **de la** 1 **la** n
      - E* (i,j) = E* (i,j) ∨ (E* (i,k) ∧ E* (k,j))

VII.V. Jhonson

Se foloseste pentru grafuri rare.

Este mai bun decat Roy-Floyd pe grafuri rare.

Foloseste liste de adiacenta.

Bazat pe Dijkstra si Bellman-Ford.

Complexitate: O(V^2 log V + VE).

Daca graful are numai arce pozitive se aplica Dijkstra pentru fiecare nod.
Altfel: se calculeaza costuri pozitive pentru fiecare arc mentinand proprietatile:

- ponderea noua a arcului >= 0, pentru orice arc.
- Daca p este un drum minim ce trece prin arcul original, el va ramane minim trecand si prin arcul cu noua pondere.

Cum se construieste noua pondere?

$W1(u,v) = W(u, v) + h(u) – h(v)$, unde h e o functie de cost.

Se adauga un nod S.
Se uneste s cu toate nodurile grafului prin arce de cost 0.
Se aplica BFS pe acest graf.

- Johnson(G)
  - // Construim G' = (V',E');
  - V' = V ∪ {s}; // adăugăm nodul s
  - E' = E ∪ (s,u), ∀u∈V; w(s,u) = 0; // și îl legăm de toate nodurile
  - **Dacă** BF(G',s) e fals // aplic BF pe G'
    - **Eroare** "ciclu negativ"
  - **Altfel**
    - **Pentru fiecare** v ∈ V
      - h(v) = δ(s,v); // calculat prin BF
    - **Pentru fiecare** (u,v) ∈ E
      - $w_1(u,v) = w(u,v) + h(u) - h(v)$ // calculez noile costuri pozitive
    - **Pentru fiecare** (u ∈ V)
      - Dijkstra(G,w_1,u) // aplic Dijkstra pentru fiecare nod
      - **Pentru fiecare** (v ∈ V)
        - $d(u,v) = \delta_1(u,v) + h(v) - h(u)$ // calculez costurile pe graful inițial

VIII. Arbori Minimi de Acoperire.

Multimi disjuncte.

Pentru un graf neorientat si conex, cu functii de cost asociate muchiilor.

Un arbore liber al lui G este un graf neorientat conex si aciclic. Costul arborerul este suma ponderilor arcelor.

Un arbore liber se numeste arbore de acoperire daca V' = V.

Un arbore de acoperire se numeste arbore minim de acoperire daca are costul egal cu minimul dintre toti arborii de acoperire posibili.

Fie A o multime de muchii inclusa in E. si (S, V-S) o partitionare a lui V. Partitionarea respecta multimea A daca nu exista un nod din A care sa taie frontiera dintre S si V-S.

O muchie oarecare dintr-un graf este sigura in raport cu o multimea A daca A reunit cu muchia face parte de arborele minim de acoperire al grafului.

- Teorema 5.23: Fie A o mulțime de muchii ale unui AMA parțial al grafului G = (V,E). Fie (S,V-S) o partiționare care respectă A, iar (u,v) ∈ E o muchie care taie frontiera dintre S și V-S a.î.

w(u,v) = min{w(x,y) | (x,y) ∈ E și (x ∈ S, y ∈ V-S) sau (x ∈ V-S, y ∈ S)}

Muchia (u,v) este sigură in raport cu A.

VIII.I. Kruskall.

Complexitatea depinde de impementarea multimilor disjuncte.

Sortarea muchiilor: O (E log V) + compararea a doua multimi disjuncte + reuniunea a doua multimi disjuncte.

Multimi disjuncte: implementare:

- ca vector: comparare: O(V^2) ; reuniune: O(V); E apeluri => O (E * V^2)
- regasire rapida: comparare: O(1); reuniune: O (V); E apeluri => O(E * V) -nu se foloseste pentru grafuri mari!!!
- reuniune rapida: O(logV); reuniune: O (logV); in mare: O (E)

Complexitatea algoritmului e data de sortare: O (E log V)

- A = ∅; // inițializare AMA
- **Pentru fiecare** (v ∈ V)
  - Constr_Arb(v) // creează o mulțime formată din nodul respectiv
    // (un arbore cu un singur nod)
- Sortează_asc(E,w) // se sortează muchiile în funcție de
  // costul lor
- **Pentru fiecare** ((u,v) ∈ E) // muchiile se extrag în ordinea
  // costului
  - **Dacă** Arb(u) != Arb(v) **atunci** // verificăm dacă se creează ciclu
    - Arb(u) = Arb(u) ∪ Arb(v) // se reunesc mulțimile de noduri (arborii)
    - A = A ∪ {(u,v)} // se adaugă muchia sigură în AMA
- **Întoarce** A

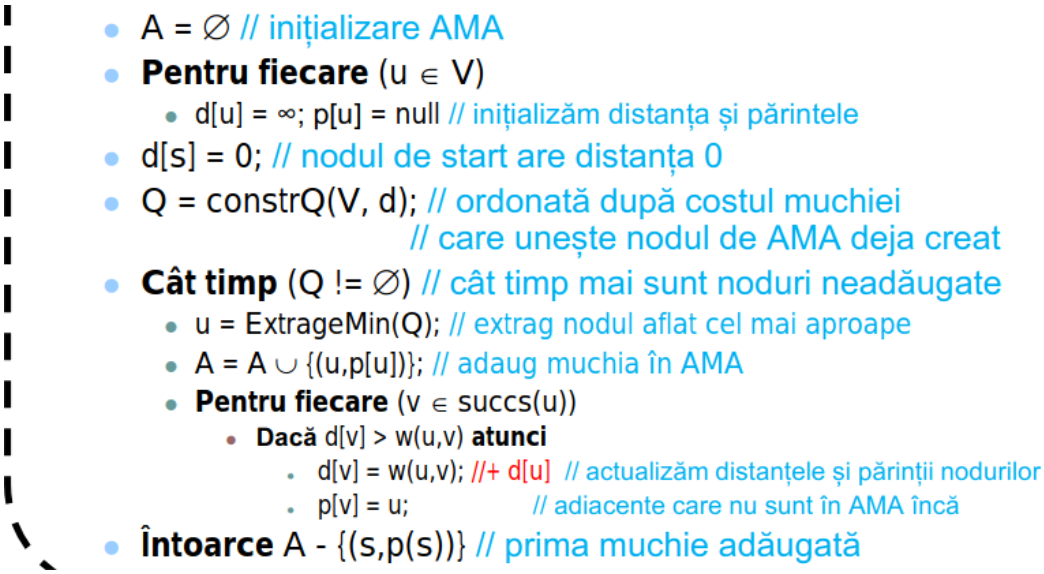
Step by step Kruskall cu union find optimizat.

E nevoie de toate muchiile ordonate crescator dupa cost. Se itereaza prin ele si se da break cand se gasesc E – 1 muchii bune.
Este nevoie de un vector de parinti pentru fiecare nod.
Este nevoie sde un vector de size pentur a retine dimensiunea arborelui care incepe din nodul respectiv.
Se itereaza prin muchii si se verifica ca muchia sa nu introduca cicluri:
- se determina radacina arborelui din care e u si din care e v. Acestea trebuie sa fie diferite.
Se actualizeaza parintele la root, daca e cazul.
Se updateaza size-ul, daca e cazul.

VIII.II. Prim.

Complexitatea depinde de graf.

- Grafuri DESE: cu matrice de adiacenta: O(v^2)
- Grafuri RARE: cu Heap Fribonacci: O (VlogV + E)
  cu Heap Binar: O (E logV)



Step by step:

Se foloseste un vector de parinti si un vector de distante si o multime de vizitati.
Se aleg elementele cu distanta minima.
Pentru toti vecinii noduui ales, care nu sunt deja in arbore, se verifica daca pot fi adauga.

IX. Flux.

Retea de transport: graf orientat, prin care trebuie trimis cat mai mult flux. Fiecare muchie are un flux si o capacitate: x (cat flux se trimite efectiv) / y (capacitatea <=> cat se poate trimite maxim).

In muchie nu poate intra mai mult decat iese.
Daca fluxul de la u la v e x, fluxul de la v la u e -x.
Fluxul se conserva in retea.
Capacitate reziduala = ce ramane posibil sa transportam = capacitate – flux efectiv

Retea reziduala = retea de flux formata din retea ce permite marirea fluxului.

Putem mari fluxul printr-o retea de flux: |f + f'| = |f |+ |f'|

c(u, v) = capacitatea u, v
f (u, v) = fluxul u, v

ex: u --(10/15)---> v
2 capacitati reziduale: u -> v = 5; v -> u = -10 (c (u, v) – f (u, v))

Taieturi minime: o taietura este o partitionare a 2 noduri in multimi disjuncte a.i:

- destinatia sa fie separata de sursa
- muchiile care trec de la partitia cu sursa la cea cu destinatia sa fie saturate.
- muchiile care trec de la partitia cu destinatia la cea cu sursa sa fie goale.

Daca exista o taietura minima, fluxul in graf este maxim.

IX.I. Ford-Fulkerson.

Nu este un algoritm, este o metoda de calcul.

Pentru orice cale de la s la t in graul cu muchii reziduale
    daca are capacitatea > 0
        adaug cat mai incape sa trimit catre fluxul maxim

caile din reteaua reziduala = drum de ameliorare.

Complexitate: O (E * flux maxim)

Probleme: se folosesc cai cu capacitate mica.
Se pun fluxuri pe mai multe arce decat e nevoie.
Imbunatatiri: se aleg cai reziduale cu capacitate maxima
Se aleg cele mai scurte cai reziduale.

IX.II. Edmonds-Karp.

Este o implementare pentru metoda Ford-Fulkerson.

E bazata pe BFS.

Complexitate: O (E^2 * V)

Nu cicleaza ptc fluxul de ameliorare creste la fiecare pas.

Step by step:
    Se aplica BFS pe graf pana se ajunge la destinatie.
    Pun pe muchii cat pot sa trimit maxim catre nodul descoperit.
    Se repeta cat timp se gaseste o cale.
Obs: Ma pot razgandi: daca am muchie de la u la v, cu capacitate 5/7, ma pot razgandi si sa trimit mai putin de la v la u (-5)