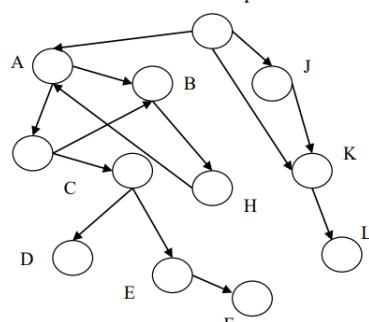


Cheatsheet PA

GRAFURI

Tipuri - orientate + neorientate

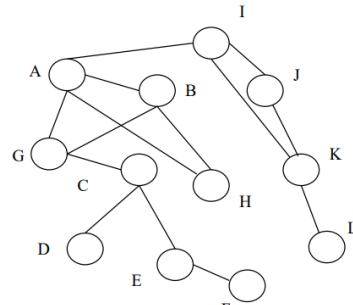
Orientate



Noduri: A, B, C ... L

Arce: AB, AG, CD ...

Neorientate



Noduri: A, B, C ... L

Muchii: AB, AI, BG ...

Notări

$G = (V, E)$;

V – mulțimea de noduri;

E – mulțimea de muchii / arce;

(u, v) – arcul / muchia u, v ;

$u \dots v$ – drum de la u la v ; dacă există mai multe variante notăm $u \dots x \dots v$, $u \dots y \dots v$;

$R(u)$ - reachable(u) = mulțimea nodurilor ce pot fi atinse pe căi ce pleacă din u ;

- $\text{succs}(u)$ – mulțimea succesorilor lui u (graf orientat) sau mulțimea nodurilor adiacente lui u (graf neorientat);

- $c(u)$ – culoarea nodului – specifică starea nodului la un anumit moment al parcurgerii:

- Alb – nedescoperit;
- Gri – descoperit, în curs de prelucrare;
- Negru – descoperit și terminat (cu semnificații diferite pentru BFS și DFS).

- $p(u)$ ($\pi(u)$) – “părintele lui u ” – identificator al nodului din care s-a ajuns în nodul u prima oară.

Parcure BFS -> COADĂ

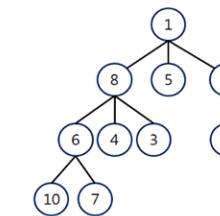
Determină numărul minim de muchii / arce între un nod s (nod de start / sursă) și orice nod din V . Acesta reprezintă cel mai scurt drum dacă graful nu are asociat o funcție de costuri.

Pseudocod

BFS(s, G);

- Pentru fiecare nod u ($u \in V$)
 - $p(u) = \text{null}$; $\text{dist}(s, u) = \text{inf}$; $c(u) = \text{alb}$; // inițializări
- $Q = ()$; // se folosește o coadă în care reținem nodurile de prelucrat
- $\text{dist}(s, s) = 0$; // actualizări: distanța de la sursă până la sursă este 0
- $Q \leftarrow Q + s$; // adăugăm sursa în coadă → începem prelucrarea lui s
- $c(s) = \text{gri}$; // și atunci culoarea lui devine gri
- Cât timp ($\text{empty}(Q)$) // cât timp mai am noduri de prelucrat
 - $u = \text{top}(Q)$; // se determină nodul din vârful cozii
 - Pentru fiecare nod v ($v \in \text{succs}(u)$) // pentru toți vecinii
 - Dacă $c(v)$ este alb // nodul nu a mai fost găsit, nu e în coadă
 - Atunci { $\text{dist}(s, v) = \text{dist}(s, u) + 1$; $p(v) = u$; $c(v) = \text{gri}$; $Q \leftarrow Q + v$; } // actualizăm structura date
 - $c(v) = \text{negru}$; // am terminat de prelucrat nodul curent
 - $Q \leftarrow Q - u$; // nodul este eliminat din coadă

Practic ia / extrage câte un nod de la începutul cozii și adaugă la finalul acesteia toți vecinii lui nevizitați. Se repetă procedeul până când coada este vidă.



BFS: 1 8 5 2 6 4 3 9 10 7

COMPLEXITATE : O(n+m)

n = număr noduri

m = număr muchii

Cheatsheet PA

Parcure DFS ->

STIVĂ / RECURSIVITATE (simulează stiva)

$d(u)$ = momentul descoperirii nodului (se trece prima oară prin u și e totodată și momentul începerii explorării zonei din graf ce poate fi atinsă din u).

$f(u)$ = timpul de finalizare al nodului (momentul în care prelucrarea nodului u a luat sfârșit)

- Tot subarborele de adâncime dominat de u a fost explorat.
- Alternativ: tot subgraful accesibil din u a fost descoperit și finalizat deja.

Pseudocod

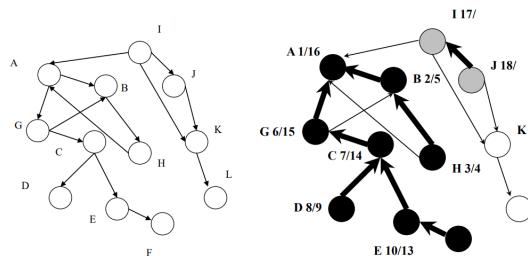
DFS(G)

- $V = \text{noduri}(G)$
- Pentru fiecare nod u ($u \in V$)
 - $c(u) = \text{alb}$; $p(u) = \text{null}$; // inițializare structură date
- $\text{timp} = 0$; // reține distanța de la rădăcina arborelui DFS până la nodul curent
- Pentru fiecare nod u ($u \in V$)
 - Dacă $c(u)$ este alb
 - Atunci $\text{explorare}(u)$; // explorez nodul

$\text{explorare}(u)$

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

$A 1/16 \Rightarrow d(A) = 1$ (timpul descoperirii)
 $f(A) = 16$ (timpul finalizării)



$I(u) = \text{intervalul de prelucrare al nodului } (d(u), f(u))$.

Teorema 5.2. $G = (V, E)$; DFS(G) sparge graful G într-o pădure de arbori $\text{Arb}(G) = \{\text{Arb}(u); p(u) = \text{null}\}$ unde $\text{Arb}(u) = (V(u), E(u))$;

- $V(u) = \{v \mid d(u) < d(v) < f(u)\} + \{u\}$;
- $E(u) = \{(v, z) \mid v, z \in V(u) \text{ și } p(z) = v\}$.

Teorema 5.3. Dacă DFS(G) generează 1 singur arbore $\Rightarrow G$ este conex. (Reciproca este adevărată?)

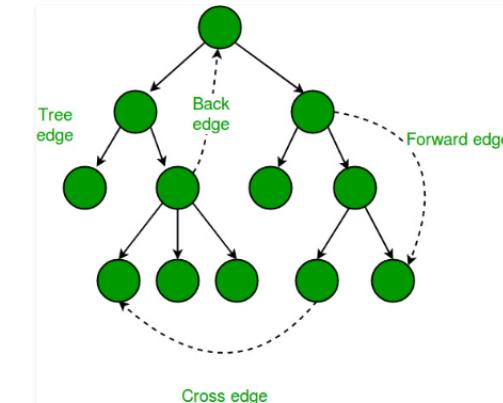
- Graf orientat conex înseamnă că prin transformarea arcelor în muchii se obține un graf neorientat conex.

Teorema 5.5. $\forall u, v \in V$, atunci $v \in V(u) \Leftrightarrow I(v) \subset I(u)$.

Clasificări arce de grafuri

- **Arc direct (de arbore) = Tree edge** (u, v) - între nod gri și nod alb;
- **Arc invers (de ciclu) (u, v) = Back edge** - între nod gri și nod gri;
- **Arc înainte (u, v) = Forward edge** - nod gri și nod negru și $d(u) < d(v)$;

- **Arc transversal (u, v) = Cross edge** - nod gri și nod negru și $d(u) > d(v)$.



Exemplu:

Arc direct (de arbore):

AB, BH, AG, GC, CD, CE, EF, IJ, JK, KL

Arc invers (de ciclu):

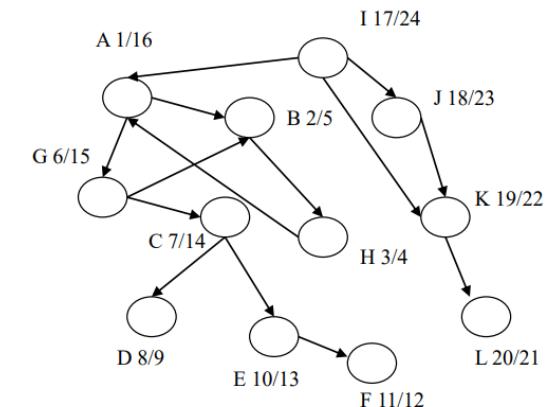
HA

Arc înainte:

IK

Arc transversal:

GB, IA



Cheatsheet PA

Teorema 5.6. Teorema drumurilor albe:

- $G = (V, E)$; $\text{Arb}(u)$; v este descendant al lui u în $\text{Arb}(u) \Leftrightarrow$ la momentul $d(u)$ există o cale numai cu noduri albe $u..v$.

Teorema 5.7. Într-un graf neorientat, DFS poate descoperi doar muchii directe și inverse.

- Dem prin considerarea cazurilor posibile!

- Fie muchia $(u, v) \in E$ și pp. $d(u) < d(v)$. Muchia poate fi străbătută din u sau din v :
 - Caz (u, v) : $c(u) = \text{gri}$, $c(v) = \text{alb} \rightarrow$ muchie directă
 - Caz (v, u) : la $d(u) \exists$ o cale cu noduri albe $u..v \rightarrow$ (Teorema drumurilor albe) v este descendant al lui u în $\text{Arb}(u) \rightarrow$ (Teorema 5.5) $d(u) < d(v) < f(v) < f(u) \rightarrow$ în intervalul $(d(v), f(v))$ când se investighează (v, u) $c(u) = c(v) = \text{gri} \rightarrow$ muchie inversă

Teorema 5.8. G = graf orientat; G ciclic \Leftrightarrow în timpul execuției DFS găsim arce inverse.

COMPLEXITATE: $O(n+m)$

n = număr de noduri

m = număr de muchii

Sortare topologică

- $G = (V, E)$ orientat, aciclic.
- V_S – secvența de noduri a.î. $\forall (u, v) \in E$, avem $\text{index}(u) < \text{index}(v)$.
- **Scop:** Sortare_topologică(G) $\Rightarrow V_S$.

Idee bazată pe DFS:

- $G = (V, E)$ orientat, aciclic; la sfârșitul DFS avem $\forall (u, v) \in E$, $f(v) < f(u)$
- \Rightarrow colectăm în V_S vârfurile în ordinea descrescătoare a timpilor f

Pseudocod

Sortare_topologică (G)

- Pentru fiecare nod u ($u \in V$) $\{c(u) = \text{alb}\}$ // inițializări
- $V_S = \emptyset$;
- Pentru fiecare nod u ($u \in V$) // pentru fiecare componentă conexă
 - Dacă $c(u)$ este alb
 - $V_S = \text{Explorează } (u, V_S)$ // prelucrez componentă conexă
- Întoarce V_S

Explorează (u, V_S)

- $c(u) = \text{gri}$ // prelucrez nodul, deci îi actualizez culoarea
- Pentru fiecare nod v ($v \in \text{succs}(u)$)
 - Dacă $c(v)$ este alb atunci $V_S = \text{Explorează } (v, V_S)$ // recursivitate
 - Dacă $c(v)$ este gri atunci întoarce eroare: graf ciclic
- $c(u) = \text{negrui}$ // am terminat prelucrarea nodului
- Întoarce $\text{cons}(u, V_S)$ // inserez nodul u la începutul lui V_S

COMPLEXITATE: $O(n+m)$

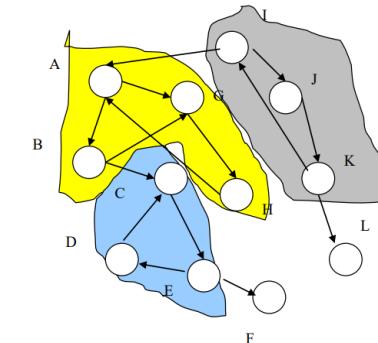
n = număr de noduri

m = număr de muchii

COMPONENTE TARE CONEXE (CTC)

Definiție: Fie $G = (V, E)$ un graf orientat. G este **tare-conex** $\Leftrightarrow \forall u, v \in V \exists$ o cale $u..v$ și o cale $v..u$ ($u \in R(v)$ și $v \in R(u)$).

Definiție: $G = (V, E)$ graf orientat. $G' = (V', E')$, $V' \subseteq V$, $E' \subseteq E$. G' este o **CTC** a lui $G \Leftrightarrow G'$ e **tare-conex** ($\forall u, v \in V'$, $u \in R(v)$ și $v \in R(u)$) și G' este **maximal** (ca număr de noduri).



Strămoș

Definiție: $G = (V, E)$ orientat, $u \in V$. $\Phi(u) =$ strămoș DFS al lui u determinat în cursul DFS(G) dacă:

- $\Phi(u) \in R(u)$
- $f(\Phi(u)) = \max\{f(v) \mid v \in R(u)\}$

Practic este primul nod din CTC descoperit în DFS.

Teorema 5.13. $G = (V, E)$ orientat, $\forall u, v \in V$; u și v aparțin aceleiași CTC $\Leftrightarrow \Phi(u) = \Phi(v)$.

Primul nod dintr-o CTC descoperit prin DFS va avea drept succesiuni în arborele generat de DFS toate elementele componentei conexe!

Algoritmul lui Kosaraju

Cheatsheet PA

- CTC(G)
- DFS(G)
 - $G^T = \text{transpun}(G)$
 - DFS(G^T) (în bucla principală se tratează nodurile în ordinea descrescătoare a timpilor de finalizare de la primul DFS)

Componentele conexe sunt reprezentate de pădurea de arbori generați de DFS(G^T).

Algoritm:

<https://www.geeksforgeeks.org/strongly-connected-components/>

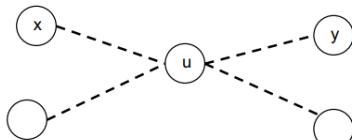
COMPLEXITATE: $O(n+m)$

n = număr de noduri

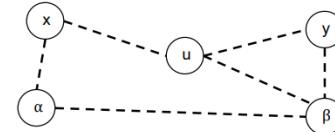
m = număr de muchii

PUNCT DE ARTICULATIE

Definiție: $G = (V, E)$ graf neorientat, $u \in V$. U este punct de articulație dacă $\exists x, y \in V$, $x \neq y$, $x \neq u$, $y \neq u$, a.î. $\forall x..y \in G$ trece prin u .



Orice drum $x..y$ trece prin $u \rightarrow u$ este punct de articulație.

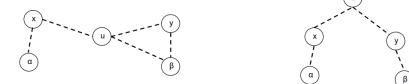


Există $x..\alpha..y$ care nu trece prin $u \rightarrow u$ nu mai este punct de articulație!

Teorema 5.15: $G = (V, E)$, graf neorientat și $u \in V$. U este punct de articulație în $G \Leftrightarrow$ în urma DFS în G una din proprietățile de mai jos este satisfăcută:

- $p(u) = \text{null}$ și u domină cel puțin 2 subarbori;
- $p(u) \neq \text{null}$ și $\exists v$ descendenter al lui u în $\text{Arb}(u)$ a.î. $\forall x \in \text{Arb}(v)$ și $\forall (x, z)$ parcursă de DFS(G) avem $d(z) \geq d(u)$.

1) $p(u) = \text{null}$ și u domină cel puțin 2 subarbori:



2) $p(u) \neq \text{null}$ și $\exists v$ descendenter al lui u în $\text{Arb}(u)$ a.î. $\forall x \in \text{Arb}(v)$ și $\forall (x, z)$ parcursă de DFS(G) $d(z) \geq d(u)$:



Structuri de date

$\text{Low}(u) = \min\{d(v) \mid v \text{ descoperit pornind din } u \text{ în cursul DFS și } c(v) \neq \text{alb}\}$

$\text{Subarb}(u) = \text{numărul subarborilor dominați de } u$ (dacă este ≥ 2 , atunci avem un punct de articulație).

Algoritmul lui Tarjan

index = 0 // nivelul pe care este nodul în arborele DFS
 $S = \text{empty}$ // se folosește o stivă care se inițializează cu \emptyset

Pentru fiecare v din V

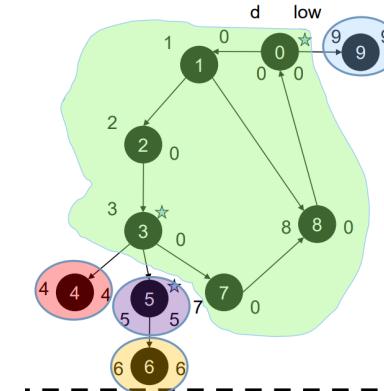
- Dacă $v.\text{index}$ e nedefinit atunci // se pornește DFS din fiecare nod pe care
- Tarjan(v) // nu l-am vizitat încă

Tarjan(v)

- $v.\text{index} = \text{index}$ // se setează nivelul nodului v
- $v.\text{lowlink} = \text{index}$ // reține strâmosul nodului v
- index = index + 1 // incrementez nivelul
- $S.\text{push}(v)$ // introduc v în stivă
- Pentru fiecare (v, v') din E // se prelucrează succesorii lui v
 - Dacă $v'.\text{index}$ e nedefinit sau $v' \in S$ atunci // CTC deja identificate sunt ignorate
 - Dacă $v'.\text{index}$ e nedefinit atunci Tarjan(v') // dacă nu a fost vizitat într-o recursitate
 - $v'.\text{lowlink} = \min(v.\text{lowlink}, v'.\text{lowlink})$ //actualizez strâmosul
- Dacă $(v.\text{lowlink} == v.\text{index})$ atunci // printez CTC începând de la coadă spre rădăcina
 - print "CTC:"
 - Repetă
 - $v = S.\text{pop}$ // extrag nodul din stivă și il printez
 - print v
 - Până când $(v == v)$ // până când extrag rădăcina

<https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/>

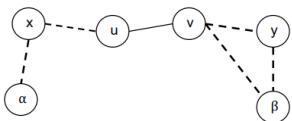
În urma aplicării algoritmului lui Tarjan, se obține:



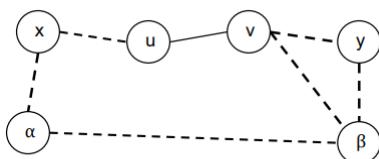
Cheatsheet PA

PUNȚI

Definiție: $G = (V, E)$, graf neorientat și $(u, v) \in E$. (u, v) este puncte în $G \Leftrightarrow \exists x, y \in V, x \neq y, \text{a.î. } \forall x..y \text{ conține muchia } (u, v)$.



Orice drum $x..y$ trece prin (u, v)
=> (u, v) este puncte



(u, v) nu este puncte

Algoritm

Punți(G)

- $V = \text{noduri}(G)$ // inițializări
- Timp = 0;
- **Pentru fiecare** nod u ($u \in V$)
 - $c(u) = \text{alb}$;
 - $d(u) = 0$;
 - $p(u) = \text{null}$;
 - $\text{low}(u) = 0$;
 - $\text{punte}(u) = 0$; // înlocuiește: $\text{subarb}(u) = 0$; $\text{art}(u) = 0$;
- **Pentru fiecare** nod u ($u \in V$)
 - Dacă $c(u)$ e alb
 - Explorează(u)

Explorează(u)

- $d(u) = \text{low}(u) = \text{timp}++$; // inițializări
- $c(u) = \text{gri}$;
- **Pentru fiecare** nod v ($v \in \text{succs}(u)$)
 - Dacă $c(v)$ e alb
 - $p(v) = u$; // se elimină: $\text{subarb}(u)++$;
 - Explorează(v);
 - $\text{low}(u) = \min\{\text{low}(u), \text{low}(v)\}$ // actualizare low
 - Dacă $(\text{low}(v) > d(u))$ $\text{punte}(v) = 1$;
 - // în loc de: Dacă($p(u) \neq \text{null} \& \& \text{low}(v) \geq d(u)$)
 - Altfel
 - Dacă ($p(u) \neq v$) $\text{low}(u) = \min\{\text{low}(u), d(v)\}$ // actualizare low

DRUMURI DE COST MINIM

TIPURI

Drumuri punct – multipunct: pentru un nod dat $s \in V$, să se găsească un drum de cost minim de la s la $\forall u \in V$; **Dijkstra, Bellman-Ford**
 Drumuri multipunct – punct: pentru un nod dat $e \in V$, să se găsească un drum de cost minim de la $\forall u \in V$ la e ; **G^T și apoi 1**
 Drumuri punct – punct: pentru două noduri date u și $v \in V$, să se găsească un drum $u..v$ de cost minim; **Folosind 1**
 Drumuri multipunct – multipunct: $\forall u, v \in V$, să se găsească un drum $u..v$ de cost minim. **Floyd-Warshall**
 Drumuri de cost maxim!

- 1) **Punct - multipunct** -> Dijkstra, Bellman-Ford
- 2) **Multipunct - punct** -> transpunem graful și aplicăm unul din cei 2 algoritmi de la 1
- 3) **Punct - punct** -> folosim 1

4) **Multipunct - multipunct** -> Floyd-Warshall

5) **Cost maxim** ->

ALGORITMUL LUI DIJKSTRA

Se folosește **NUMAI** pentru costuri pozitive ($w(u, v) > 0, \forall u, v \in V$).

Dijkstra(G, s)

- **Pentru fiecare** ($u \in V$) // inițializări
 - $d[u] = \infty$; $p[u] = \text{null}$;
- $d[s] = 0$;
- $Q = \text{construiește_coada}(V)$ // coadă cu priorități
- **Cât timp** ($Q \neq \emptyset$)
 - $u = \text{ExtragMin}(Q)$; // extrage din V elementul cu $d[u]$ minim
 - // $Q = Q - \{u\}$ – se execută în cadrul lui ExtragMin
 - **Pentru fiecare** ($v \in Q$ și v din succesorii lui u)
 - Dacă ($d[v] > d[u] + w(u, v)$)
 - $d[v] = d[u] + w(u, v)$ // actualizez distanța
 - $p[v] = u$ // și părantele

https://infoarena.ro/job_detail/2376287?action=view-source

Complexitate?

Vectori - $O(V^2)$

HB - $O(E \log V)$

HF - $O(V \log V+E)$

Cheatsheet PA

Algoritmul lui Bellman-Ford

<https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>

BellmanFord(G,s) // G=(V,E),s=sursa

- Pentru fiecare nod $v \in V$ // initializări
 - $d[v] = \infty$;
 - $p[v] = \text{null}$;
- $d[s] = 0$; // 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");

Teorema. $G = (V, E)$, $w : E \rightarrow \mathbb{R}$ funcție de cost asociată. Algoritmul Bellman-Ford aplicat acestui graf plecând din sursa s nu returnează EROARE dacă G nu conține cicluri negative, iar la terminare $d[v] = \delta(s, v)$ pentru $\forall v \in V$. Dacă G conține cel puțin un ciclu negativ accesibil din s, atunci algoritmul întoarce EROARE.

BellmanFordOpt2(G,s)
• Pentru fiecare nod $v \in V$

- $d[v] = \infty$;
- $p[v] = \text{null}$;
- $\text{marcat}[v] = \text{false}$; // marcăm nodurile pentru care am făcut relaxare

- $Q = \emptyset$; // coadă cu priorități
- $d[s] = 0$; $\text{marcat}[s] = \text{true}$; Introdu(Q,s);
- Cât timp ($Q \neq \emptyset$)
- $u = \text{ExtragăMin}(Q)$; $\text{marcat}[u] = \text{false}$; // extrag minimul
- Pentru fiecare nod v ($v \in V$)
 - Dacă $d[v] > d[u] + w(u,v)$ atunci // relaxez arcele ce pleacă din u
 - $d[v] = d[u] + w(u,v)$;
 - $p[v] = u$;
 - Dacă $(\text{marcat}[v] == \text{false})$ { $\text{marcat}[v] = \text{true}$; Introdu(Q,v);}

Observație: nu mai detectează cicluri negative!

FLOYD-WARSHALL (ROY FLOYD)

Calculează distanțele minime între oricare 2 noduri dintr-un graf (drumuri optime multipunct-multipunct).

Idee: la pasul k se calculează cel mai bun cost între u și v folosind cel mai bun cost u..k și cel mai bun cost k..v calculat până în momentul respectiv.

Rezultatele furnizate sunt corecte dacă algoritmul se aplică pe grafuri ce nu conțin cicluri de cost negativ.

Notății

$G = (V, E)$; $V = \{1, 2, \dots, n\}$;

$w : V \times V \rightarrow \mathbb{R}$; $w(i, i) = 0$; $w(i, j) = \infty$ dacă $(i, j) \notin E$;

$d^k(i, j) =$ costul drumului i..j construit astfel încât drumul trece doar prin noduri din mulțimea $\{1, 2, \dots, k\}$;

$\delta(i, j) =$ costul drumului optim i..j; $\delta(i, j) = \infty$ dacă nu există drum i..j;

$\delta^k(i, j) =$ costul drumului optim i..j ce trece doar prin noduri din mulțimea $\{1, 2, \dots, k\}$;

$p^k(i, j) =$ predecesorul lui j pe drumul i..j ce trece doar prin noduri din mulțimea $\{1, 2, \dots, k\}$.

Recurență

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

Soluție finală: $d^n(i, j) = \delta(i, j)$.

Algoritm

Pentru i de la 1 la n

- Pentru j de la 1 la n // initializări
 - $d^0(i, j) = w(i, j)$
 - Dacă $(w(i, j) == \infty)$
 - $p^0(i, j) = \text{null}$;
 - Altfel $p^0(i, j) = i$;

Cheatsheet PA

Pentru k de la 1 la n

- Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - Dacă $d^{k-1}(i,j) > d^{k-1}(i,k) + d^{k-1}(k,j)$
 - $d^k(i,j) = d^{k-1}(i,k) + d^{k-1}(k,j)$
 - $p^k(i,j) = p^{k-1}(k,j); // și actualiză$
 - Altfel
 - $d^k(i,j) = d^{k-1}(i,j)$
 - $p^k(i,j) = p^{k-1}(i,j);$

Complexitate -> $O(V^3)$

Se poate reduce complexitatea spațială, folosind o singură matrice de dimensiune $n \times n \Rightarrow$ complexitate spațială $O(N^2)$

Problemă: în pasul k, pentru $k < i$ și $k < j$, $d(i,k)$ și $d(k,j)$ folosite la calculul $d(i,j)$ sunt $d^k(k,j)$ și $d^k(i,k)$ în loc de $d^{k-1}(k,j)$ și $d^{k-1}(i,k)$. Dacă dem. că $d^k(k,j) = d^{k-1}(k,j)$ și $d^k(i,k) = d^{k-1}(i,k)$, atunci putem folosi o singură matrice.

Dar: $d^k(i,j) = \min\{d^{k-1}(i,j), d^{k-1}(i,k) + d^{k-1}(k,j)\}$

- $d^k(k,j) = \min\{d^{k-1}(k,j), d^{k-1}(k,k) + d^{k-1}(k,j)\} = d^{k-1}(k,j)$
- $d^k(i,k) = \min\{d^{k-1}(i,k), d^{k-1}(i,k) + d^{k-1}(k,k)\} = d^{k-1}(i,k)$

Alg. după optimizare

Pentru i de la 1 la n

- Pentru j de la 1 la n // inițializări
 - $d(i,j) = w(i,j)$
 - Dacă $w(i,j) == \infty$
 - $p(i,j) = \text{null};$
 - Altfel $p(i,j) = i;$

C
(

Pentru k de la 1 la n

- Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - Dacă $d(i,j) > d(i,k) + d(k,j)) // determină$
 - $d(i,j) = d(i,k) + d(k,j)$
 - $p(i,j) = p(k,j); // și actualizăm pă$

Închidere tranzitivă

Fie $G = (V,E)$. Închiderea tranzitivă a lui E este $G^* = (V,E^*)$, unde

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

Obținerea închiderii tranzitive a unui graf se face prin modificarea algoritmului Floyd-Warshall, astfel:

- $\min \Rightarrow$ operatorul boolean sau (\vee)
- $+$ \Rightarrow operatorul boolean și (\wedge)

Închidere_tranzitivă(G)

Pentru i de la 1 la n

- Pentru j de la 1 la n
 - $E^*(i,j) = ((i,j) \in E) \vee (i = j)) // inițializări$

Pentru k de la 1 la n

- Pentru i de la 1 la n
 - Pentru j de la 1 la n
 - $E^*(i,j) = E^*(i,j) \vee (E^*(i,k) \wedge E^*(k,j))$

Complexitate? Complexitate spațială?
 $O(V^3)$ $O(V^2)$

Algoritmul lui Johnson

Este folosit pentru grafuri rare. Mai eficient decât Floyd-Warshall pentru acest tip de grafuri.

Complexitate: $O(V^2 \log V + VE)$

Dacă graful conține doar arce de cost pozitiv - se aplică Dijkstra pentru fiecare nod

Astfel, se recalculează costurile muchiilor astfel încât să se respecte următoarele proprietăți:

$w_1(u,v) \geq 0, \forall (u,v) \in E;$
p este drum minim utilizând w \Leftrightarrow p este drum minim utilizând w_1 .

CheatSheet PA

Adăugăm un nod nou în graf și legăm muchii de la el la toate nodurile din graf cu costul 0. Aplicăm algoritmul lui Bellman-Ford și verificăm dacă graful conține cicluri negative. Dacă nu conține determinăm în urma alg. BF vectorul $h(x) =$ distanța minimă de la nodul adăugat la nodul x . Calculăm noua funcție de costuri astfel:

$$w_1(u,v) = w(u,v) + h(u) - h(v);$$

Se aplică alg Johnson pentru graf cu arce de cost pozitiv (după ce eliminăm nodul adăugat inițial).

Johnson(G)

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

Arbore minim de acoperire

Definiție: Un arbore liber se numește arbore de acoperire dacă $V' = V$.

Definiție: Un arbore de acoperire (Arb) se numește arbore minim de acoperire (notăm AMA) dacă $Arb \in ARB(G)$ a.î. $C(Arb) = \min\{C(Arb') \mid Arb' \in ARB(G)\}$.

$G = (V, E)$, $C = (V', E')$ – ciclu în G ; $e \in E'$ a.î. $w(e) = \max\{w(e') \mid e' \in E'\} \Rightarrow e \notin Arb(G)$ unde $Arb(G) = AMA$ în G .

Proprietăți (I)

$| G = (V, E)$, $C = (V', E')$ – ciclu în G ; $e \in E'$ a.î. $w(e) = \max\{w(e') \mid e' \in E'\} \Rightarrow e \notin Arb(G)$ unde $Arb(G) = AMA$ în G .

Proprietăți (II)

$| G = (V, E)$, $S = (V', E')$ un AMA parțial al lui G , $V' \subset V$; $e = (u, v)$ a.î. $e \notin E'$ și $(u \in V' \text{ și } v \notin V')$ sau $(u \notin V' \text{ și } v \in V')$ cu proprietatea că: $w(u, v) = \min\{w(u', v') \mid (u' \in V' \text{ și } v' \notin V') \text{ sau } (u' \notin V' \text{ și } v' \in V')\} \Rightarrow (u, v) \in AMA$.

Algoritmi AMA / APM

2 algoritmi greedy

Prim: se pornește cu un nod și se extinde pe rând cu muchiile cele mai ieftine care au un singur capăt în mulțimea de muchii deja formată (Proprietatea 2). Algoritmul este asemănător algoritmului Dijkstra.

Kruskal: inițial toate nodurile formează câte o mulțime și la fiecare pas se reunesc 2 mulțimi printr-o muchie. Muchiile sunt considerate în ordinea costurilor și sunt adăugate în arbore doar dacă nu creează ciclu (Proprietatea 1).

PRIM

- asemănător algoritm Dijkstra

Prim(G, w, s)

- $A = \emptyset$ // inițializare AMA
- Pentru fiecare $(u \in V)$
 - $d[u] = \infty$; $p[u] = \text{null}$ // inițializăm distanța și părintele
 - $d[s] = 0$; // nodul de start are distanța 0
 - $Q = \text{constr}Q(V, d)$; // ordonată după costul muchiei // care unește nodul de AMA deja creat
- Cât timp ($Q \neq \emptyset$) // cât timp mai sunt noduri neadăugate
 - $u = \text{ExtractMin}(Q)$; // extrag nodul aflat cel mai aproape
 - $A = A \cup \{(u, p[u])\}$; // adaug muchia în AMA
 - Pentru fiecare $(v \in \text{succs}(u))$
 - Dacă $d[v] > w(u, v)$ atunci
 - $d[v] = w(u, v)$; // actualizăm distanțele și părintii nodurilor
 - $p[v] = u$; // adiacente care nu sunt în AMA încă
- Întoarce $A - \{(s, p(s))\}$ // prima muchie adăugată

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

Cheatsheet PA

Complexitate Prim

- Depinde de implementare (vezi Dijkstra)
 - Matrice de adiacență $O(V^2)$
 - Heap binar $O(E \log V)$
 - Heap Fibonacci $O(V \log V + E)$
- Concluzii**
 - Grafuluri dese
 - Matrice de adiacență preferată
 - Grafuluri rare
 - Heap binar sau Fibonacci

KRUSKAL

- Kruskal(G, w) **Implementare în Java la [4] !**
- $A = \emptyset$; // inițializare AMA
 - Pentru fiecare** ($v \in V$)
 - 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** ($(u, v) \in E$) // muchiile se extrag în ordinea // costului
 - Dacă $\text{Arb}(u) \neq \text{Arb}(v)$ atunci // verificăm dacă se creează ciclu
 - $\text{Arb}(u) = \text{Arb}(u) \cup \text{Arb}(v)$ // se reunesc mulțimile de noduri (arborii)
 - $A = A \cup \{(u, v)\}$ // se adaugă muchia sigură în AMA
 - Întoarce A

Elementele algoritmului:

- Sortarea muchiilor:** $O(E \log E) \approx O(E \log V)$
- Arb(u) = Arb(v)** – compararea a 2 mulțimi disjuncte $\{1, 2, 3\} \{4, 5, 6\}$ – mai precis trebuie identificat dacă 2 elemente sunt în aceeași mulțime
- Arb(u) \cup Arb(v)** – reunirea a 2 mulțimi disjuncte într-o singură

Alternative pentru operațiile cu mulțimi

$\text{id}[i]$ = părintele lui i

pentru rădăcină $\text{id}[i] = i$

Arb(i) // identificarea rădăcinii unei componente

- Cât timp** ($i \neq \text{id}[i]$) $i = \text{id}[i];$
- Întoarce** i

Comparare (u, v)

- Întoarce** $\text{Arb}(u) \neq \text{Arb}(v)$

Reuniune (u, v) // implică identificarea rădăcinii

- $v = \text{Arb}(v)$
- $\text{id}[v] = u;$

Worst case comparare + reunire - $O(V)$

Optimizare

Se menține numărul de noduri din fiecare subarbore. Se adaugă arborele mic la cel mare pentru a face mai puține căutări \rightarrow înălțimea arborelui e mai mică și numărul de căutări scade de la V la $\lg V$. Complexitatea pentru comparare și reunire devine $O(\lg V)$.

Identificarea rădăcinii:

- Arb(i)
 - Cât timp** ($i \neq \text{id}[i]$)
 - $\text{id}[i] = \text{id}[\text{id}[i]];$
 - $i = \text{id}[i];$
 - Întoarce** i

Complexitate Kruskal - $O(E \log V)$ - complexitatea este dată de sortarea muchiilor

Aplicație Kruskal

K-clustering

- Împărțirea unui set de obiecte în k grupuri astfel încât obiectele din cadrul unui grup să fie "apropiate" considerând o "distanță" dată.

Se formează V clustere (un cluster per obiect). Găsește cele mai apropiate 2 obiecte din clustere diferite și unește cele 2 clustere. Se oprește când au mai rămas k clustere. \rightarrow chiar algoritmul Kruskal

FLUX

Cheatsheet PA

$G(V, E)$ graf orientat;

$c(u,v) \geq 0 \quad \forall (u,v) - c =$ capacitatea arcelor;

Dacă $(u,v) \notin E \rightarrow c(u,v) = 0$;

S – sursa traficului;

T – destinația traficului (drena);

Presupunem că $\forall u \in V \setminus \{s, t\} \exists s..u..t.$

$f: V \times V \rightarrow \mathbb{R}$ - fluxul prin rețeaua G ;

Proprietăți:

- $\forall u, v \in V, f(u,v) \leq c(u,v)$ (fluxul printr-un arc este mai mic sau egal cu capacitatea arcului) – respectarea capacitatii arcelor;
- $\forall u, v \in V, f(u,v) = -f(v,u)$ – simetria fluxului;
- $\sum f(u,v) = 0$ pentru $\forall u \in V \setminus \{s,t\}$ – conservarea fluxului.

$f(u,v)$ – fluxul din u spre v ;

$f_i(u) = \sum f(v,u)$ – fluxul total care intra în nodul u ;

$f_o(u) = \sum f(u,v)$ – fluxul total careiese din nodul u ;

Valoarea totală a fluxului:

- $|f| = \sum f(s,v) = f_o(s)$;
- $|f| =$ fluxul ce părăsește sursa;
- Cf. proprietăților P1-P3: $|f| = \sum f(s,v) = \sum f(v,t) = f_i(t)$.

Pentru surse multiple / destinații multiple se adaugă o sursă / destinație unică.

Surse multiple $\{s_1, s_2, \dots, s_n\}$;

Destinații multiple $\{t_1, t_2, \dots, t_m\}$;

Se adaugă o **sursă unică** cu arce de capacitate infinită spre sursele $s_1..s_n$ și **flux egal** cu fluxul generat de sursele respective;

Se adaugă o **destinație unică** t și arce de capacitate infinită între $t_1..t_m$ și t și **flux egal** cu fluxul ce intră în destinațiile respective.

Operații cu fluxuri

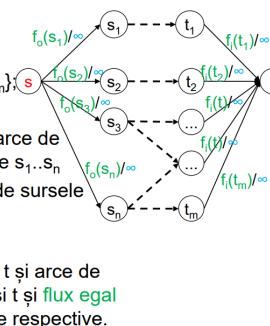
X, Y – mulțimi de noduri;

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y) = \text{fluxul între } X \text{ și } Y;$$

Operații:

- $\forall X \in V: f(X, X) = 0$;
- $\forall X, Y \in V: f(X, Y) = -f(Y, X)$;
- $\forall X, Y, Z \in V$ și $Y \subseteq X$:
 - $f(X \setminus Y, Z) = f(X, Z) - f(Y, Z)$;
 - $f(Z, X \setminus Y) = f(Z, X) - f(Z, Y)$;
- $\forall X, Y, Z \in V$ și $X \cap Y = \emptyset$:
 - $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$;
 - $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$
- $f(s, V) = f(V, t)$

Arc rezidual \rightarrow fluxul pe acest arc se poate mări



Definiție: Un arc (u,v) pentru care $f(u,v) < c(u,v)$ se numește **arc rezidual**.

Definiție: Cantitatea cu care se poate mări fluxul pe arcul (u,v) se numește **capacitatea reziduală a arcului (u,v)** ($c_f(u,v)$):

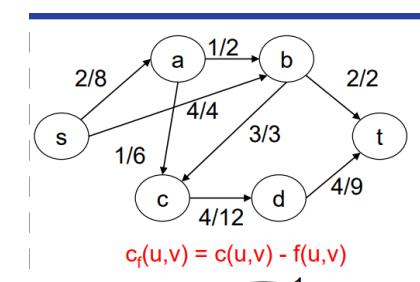
$$c_f(u,v) = c(u,v) - f(u,v)$$

Definiție: Rețea reziduală ($G_f = (V, E_f)$) este o rețea de flux formată din arcele ce admit creșterea fluxului:
 $E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$.

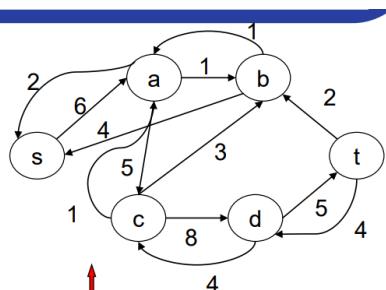
Definiție: Cale reziduală e un drum $s..t \subseteq G_f$, unde $c_f(u,v)$ este capacitatea reziduală a arcului (u,v) .

Definiție: Capacitatea reziduală a căii = capacitatea reziduală minimă de pe calea $s..t$ descoperită.

Exemplu rețea reziduală



Cheatsheet PA



Rețeaua reziduală $G_f = (V, E_f)$ unde
 $E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$

Calea reziduală: $s \rightarrow a \rightarrow c \rightarrow d \rightarrow t$

Capacitatea reziduală a căii:

$$c_f(p) = \min\{6, 5, 8, 5\} = 5$$

Lemă 5.17: G – rețea de flux, f flux în G , $p = s..t$ – cale reziduală în G_f , $f_p: V \times V \rightarrow \mathbb{R}$ se definește ca fiind:

$$f_p(u,v) = \begin{cases} c_f(p), & \text{dacă } (u,v) \in p \\ -c_f(p), & \text{dacă } (v,u) \in p \\ 0, & \text{dacă } (u,v) \text{ și } (v,u) \notin p \end{cases}$$

$$f_p = \text{flux în } G_f, |f_p| = c_f(p)$$

Corolar 5.4: $f' = f + f_p$ = flux în G , astfel încât
 $|f'| = |f| + |f_p| > |f|$

Această Lemă ne spune cum se definește fluxul
printr-o rețea reziduală

Calcularea fluxului maxim

Metoda Ford-Fulkerson

- $f(u,v) = 0 \forall (u,v)$ // inițializarea fluxului
- Repetă // creștere iterativă a fluxului
 - găsește un drum $s..p..t$ pe care se poate mări fluxul (cale reziduală)
 - $f = f + \text{flux}(s..p..t)$
- Până când nu se mai poate găsi nici un drum $s..p..t$
- Întoarce f

FORD-FULKERSON

Ford – Fulkerson(G, s, t)

- Pentru fiecare (u,v) din E
 - $f(u,v) = f(v,u) = 0$ // inițializare
- Cât timp
 - Există o cale reziduală p între $s..t$ în G_f
 - $c_f(p) = \min\{c_f(u,v) \mid (u,v) \in p\}$ // capacitatea reziduală
 - Pentru fiecare (u,v) din p
 - $f(u,v) = f(u,v) + c_f(p)$
 - $f(v,u) = -f(u,v)$
 - Întoarce $|f|$

Complexitate?

Complexitate $O(E * f_{\max})$

f_{\max} = fluxul maxim

Îmbunătățiri:

- Se aleg căile reziduale cu capacitate maximă – complexitatea va depinde în continuare de f_{\max} și de valoarea capacitaților;
- Se aleg căile reziduale cele mai scurte → în acest caz complexitatea nu mai depinde de f_{\max} ci numai de numărul de arce (ex. Edmonds-Karp: identificarea căilor reziduale minime prin aplicarea unui BFS)

Edmonds – Karp(G, s, t)

- Pentru fiecare (u,v) din E
 - $f(u,v) = f(v,u) = 0$ // inițializare
- Cât timp
 - Există căi reziduale între $s..t$ în G_f
 - Determină calea reziduală de lungime minimă p aplicând BFS
 - $c_f(p) = \min\{c_f(u,v) \mid (u,v) \in p\}$ // capacitatea reziduală
 - Pentru fiecare (u,v) din p
 - $f(u,v) = f(u,v) + c_f(p)$
 - $f(v,u) = -f(u,v)$
 - Întoarce $|f|$

Complexitate?

https://infoarena.ro/job_detail/2391089?action=view-source

Preflux

Definiție: Preflux = $f: V \times V \rightarrow \mathbb{R}$ astfel încât să fie satisfăcute restricțiile:

- $f(u,v) \leq c(u,v), \forall (u,v) \in E$ – respectarea capacitații arcelor;
- $f(u,v) = -f(v,u), \forall u,v \in V$ – simetria fluxului;
- $\sum_{v \in V} f(v,u) \geq 0, \forall u \in V \setminus \{s\}$ – conservarea fluxului.

Cheatsheet PA

Definiție: O funcție $h: V \rightarrow N$ este o **funcție de înălțime** dacă îndeplinește restricțiile:

- $h(s) = |V| - \text{fixă}$;
- $h(t) = 0 - \text{fixă}$;
- $h(u) \leq h(v) + 1$ pentru orice arc rezidual $(u,v) \in G_f$ – variabilă.

Lema 5.19: G – rețea de flux, $h: V \rightarrow N$ este o funcție de înălțime. Dacă $\forall u, v \in V, h(u) > h(v) + 1$ atunci arcul (u,v) nu este arc rezidual.

Pompare(u,v) // pompează fluxul în exces ($e(u) > 0$)
// are loc doar dacă diferența de înălțime dintre u și v este 1
// $(h(u) = h(v) + 1)$, altfel nu e arc rezidual și nu ne interesează

- $d = \min(e(u), c_f(u,v))$; // cantitatea de flux pompată
- $f(u,v) = f(u,v) + d$; // actualizare flux pe arcul (u,v)
- $f(v,u) = -f(u,v)$; // respectarea simetriei
- $e(u) = e(u) - d$; // actualizare supraîncărcare la sursă
- $e(v) = e(v) + d$; // actualizare supraîncărcare la destinație

Înălțare(u) // mărește $h(u)$ dacă u are flux în exces
// $(e(u) > 0)$ și $u \notin \{s, t\} \forall (u,v) \in G_f$ avem $h(u) \leq h(v)$
• $h(u) = 1 + \min\{h(v) \mid (u,v) \in G_f\}$

Init_preflux(G, s, t)

- **Pentru fiecare** $(u \in V)$
 - $e(u) = 0$ // inițializare exces flux în nodul u
 - $h(u) = 0$ // inițializare înălțime nod u
- **Pentru fiecare** $((u,v) \in E)$ // inițializare fluxuri
 - $f(u,v) = 0$
 - $f(v,u) = 0$
- $h(s) = |V|$ // inițializare înălțime sursă
- **Pentru fiecare** $(u \in \text{succs}(s) \setminus \{s\})$
 - $f(s,u) = c(s,u)$; // actualizare flux
 - $f(u,s) = -c(s,u)$;
 - $e(u) = c(s,u)$ // actualizare exces

Complexitate

Pompare_preflux(G, s, t)

- **Init_preflux(G, s, t)** // inițializarea prefluxului
- **Cât timp (1) // cât timp pot face pompări sau înălțări**
 - **Dacă** $(\exists u \in V \setminus \{s, t\}, v \in V \mid e(u) > 0 \text{ și } c_f(u,v) > 0 \text{ și } h(u) = h(v) + 1)$ // încerc să pompeze
 - Pompare(u,v); continuă;
 - **Dacă** $(\exists u \in V \setminus \{s, t\}, v \in V \mid e(u) > 0 \text{ și } \forall (u,v) \in E_f, h(u) \leq h(v))$ // înălțare(u); continuă; // încerc să înălțe
 - **Întrerupe;** // nu mai pot face nimic → am ajuns la flux max
- **Întoarce** $e(t) // e(t) = |f|$ = fluxul total în rețea

Init_preflux: O(E)

Pompare(u,v): O(1)

Înălțare(u): O(V) – implică găsirea minimului dintre nodurile succesoare

Cât timp: [vezi Cormen]

- Câte înălțări?
 - Care e înălțimea maximă? $2(|V| - 1)$ – drum rezidual de lungime maximă
 - Care este numărul maxim total de înălțări? $(2|V| - 1)(|V| - 2) \sim 2|V|^2$
- Câte pompări?
 - Pompări saturate: $2|V||E|$ - de căte ori un arc poate fi saturat? (în funcție de suma $h(u) + h(v)$)
 - Pompări nesaturate: $4|V|^2(|V| + |E|)$ – sumă înălțimi noduri excedentare

Complexitate totală: $O(V^2 * E)$ [vezi Cormen]

ALGORITMI EURISTICI

Stare a problemei = abstractizare a unei configurații valide a universului problemei, configurație ce determină univoc comportarea locală a fenomenului descris de problemă.

Spațiul stărilor = graf în care nodurile corespund stărilor problemei, iar arcele desemnează tranzițiile valide între stări.

Descriere

- Nodul de start (starea inițială);
- Funcție de expandare a nodurilor (produce lista nodurilor asociate stărilor valide în care se poate ajunge din starea curentă);
- Predicat de testare dacă un nod corespunde unei stări soluție.

Definiție: Dacă explorarea se bazează pe informația acumulată în cursul explorării, informație prelucrată **euristică** (costuri) → **algoritm informat**.

Definiție: Dacă explorarea este 'la întâmplare' → **algoritm neinformat**.

Definiție: Dacă algoritmul de explorare are posibilitatea să abandoneze calea curentă de rezolvare și să revină la o cale anterioară → **algoritmi tentativi**.

Definiție: Altfel (algoritmul avansează pe o singură direcție) → **algoritmi irevocabili**.

OPEN = mulțimea (lista) nodurilor **explorate** (frontiera dintre zona cunoscută și cea necunoscută).

CLOSED = mulțimea (lista) nodurilor **expandate** (regiunea cunoscută în totalitate).

Explorarea zonelor necunoscute se face prin **alegerea** și **expandarea** unui nod din **OPEN**. După expandare, nodul respectiv e trecut în **CLOSED**.

Majoritatea algoritmilor tentativi folosesc lista **OPEN**, dar doar o parte folosesc lista **CLOSED**.

Cheatsheet PA

Definiție: Algoritm complet = algoritm de explorare care garantează descoperirea unei soluții, dacă problema acceptă soluție.

- Algoritmii irevocabili sunt mai rapizi și consumă mai puține resurse decât cei tentativi, dar nu sunt compleți pentru că pierd informație.

Definiție: Algoritm optimal = algoritm de explorare care descoperă soluția optimă a problemei.

Explorare(StInit, test_sol)

- OPEN = {constr_nod(StInit)}; // starea inițială
- **Cât timp** (OPEN ≠ Ø)
 - // mai am noduri de prelucrat
 - nod = selecție_nod(OPEN); // aleg un nod
 - Dacă (test_sol(nod)) Întoarce nod;
 - // am găsit o soluție
 - OPEN = OPEN \ {nod} U expandare{nod}; // extind căutarea
 - Întoarce insucces; // nu s-a găsit nicio soluție

Explorarea tentativă completă BF* (BEST FIRST)

BF*(StInit, f, test_sol)

- nod = constr_nod(StInit); // starea inițială
- π(nod) = null;
- OPEN = {nod}; // noduri explorate dar neexpandate
- CLOSED = Ø; // noduri expandate

Inițializări

• **Cât timp** (OPEN ≠ Ø)

- nod = selecție_nod(OPEN); // $f(nod) = \min\{f(n) | n \in OPEN\}$

• Dacă (test_sol(nod)) Întoarce nod;

Soluția

- OPEN = OPEN \ {nod};
- CLOSED = CLOSED U {nod};
- succs = expand(nod);

Continuarea căutării

• **Pentru fiecare** (succ ∈ succs)

- Dacă (succ ∉ CLOSED U OPEN) atunci

• OPEN = OPEN U {succ}; π(succ) = nod;

• Altfel

- succ' = apariția lui succ în CLOSED U OPEN

• Dacă ($f(succ) < f(succ')$) // am găsit o cale mai bună către succ și

// redescidem nodul

$\pi(succ') = nod$; // actualizez părintele

$f(succ') = f(succ)$; // și costul nodului

Dacă (succ' ∈ CLOSED) // dacă era considerat expandat, îl redescidem

CLOSED = CLOSED \ {succ'}; OPEN = OPEN U {succ'}

Nod nou

Actualizări

Reprelucrare

Întoarce insucces;

Optimalitate?

Completitudine?

Complexitate?

ex: Best-first cu diverse euristici

Algoritmul este complet dar nu este optim
→ optimalitatea depinde de euristică f!

Complexitate: $O(b^{d+1})$

A*

A*(StInit, h, test_sol)

- $n_0 = \text{constr_nod}(StInit)$; // starea inițială

Initializări

- $f(n_0) = h(n_0)$; $g(n_0) = 0$; $\pi(n_0) = \text{null}$; // euristici

- OPEN = { n_0 }; CLOSED = Ø; // și cozi

• **Cât timp** (OPEN ≠ Ø) // mai am noduri de prelucrat

- nod = selecție_nod(OPEN); // $f(nod) = \min\{f(n) | n \in OPEN\}$

• Dacă (test_sol(nod)) Întoarce nod;

Soluția

- OPEN = OPEN \ {nod}; // updateaza OPEN

- CLOSED = CLOSED U {nod}; // și CLOSE

- succs = expand(nod); // determin nodurile succesoare

Continuarea căutării

• **Pentru fiecare** (succ ∈ succs) { // prelucrare succs

- $g_{succ} = g(\text{nod}) + c(\text{nod}, \text{succ})$; // calculez g

Prelucrare

- $f_{succ} = g_{succ} + h(\text{succ})$; // calculez f = g + h

succesor

- Dacă (succ ∉ CLOSED U OPEN) atunci // nod nou descooperit →

- OPEN = OPEN U {succ}; // îl bag în OPEN

- $g(\text{succ}) = g_{succ}$; $f(\text{succ}) = f_{succ}$; $\pi(\text{succ}) = \text{nod}$;

Nod nou

- altfel // a mai fost prelucrat

- Dacă ($g_{succ} < g(\text{succ})$) // verific dacă noui g este mai mic decât

- // anteriorul

- $g(\text{succ}) = g_{succ}$; $f(\text{succ}) = f_{succ}$; $\pi(\text{succ}) = \text{nod}$ // cale mai bună

- Reprelucrare Dacă (succ ∈ CLOSED) // dacă era considerat expandat, îl redescidem

- CLOSED = CLOSED \ {succ}; OPEN = OPEN U {succ};

- Întoarce Insucces; Insucces

ex: A* cu diverse euristici