Proiectarea Algoritmilor

Curs 7 – Puncte de articulație, Punți, Drumuri minime

Bibliografie

[1] Giumale – Introducere in Analiza Algoritmilor cap. 5.3, 5.4, 5.4.1

[2] Cormen – Introducere în Algoritmi cap. 20, 21, 25.1 si 25.2

[3] R. Sedgewick, K. Wayne - Algorithms and Data Structures Fall 2007 – Curs Princeton - http://www.cs.princeton.edu/~rs/AlgsDS07/

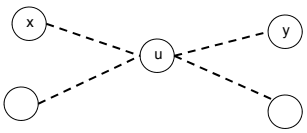
[4] Heap Fibonacci: http://www.cse.yorku.ca/~aaw/Jason/FibonacciHeapAnimation.html

Obiective

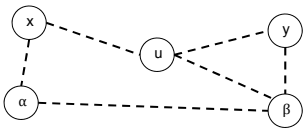
- "descoperirea" algoritmilor de:
  - Identificare a punctelor de articulație;
  - Identificare a punților;
  - Identificare a drumurilor de cost minim.
- Identificarea structurilor de date necesare pentru reducerea complexității acestor algoritmi.

Puncte de articulație. Def. Exemple

- Definiție: G = (V,E) graf neorientat, u∈V. U este punct de articulație daca ∃ x,y∈V, x ≠ y, x ≠ u, y ≠ u, a.i. ∀ x..y in G trece prin u.



Orice drum x..y trece prin u → u este punct de articulație.



Exista x..α..y care nu trece prin u → u nu mai este punct de articulație!

Algoritm naiv de detectare a punctelor de articulatie

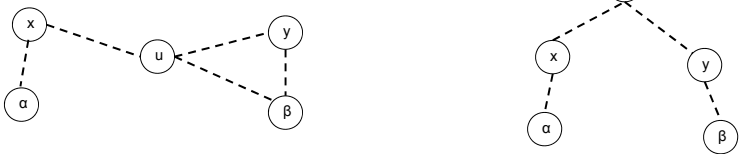
- Elimina fiecare nod si verifica conectivitatea grafului rezultat
  - Graf conex → nodul nu e punct de articulație
  - Altfel → punct de articulație
- Complexitate?
  - $O(V^2+VE)$

Puncte de articulație. Teorema

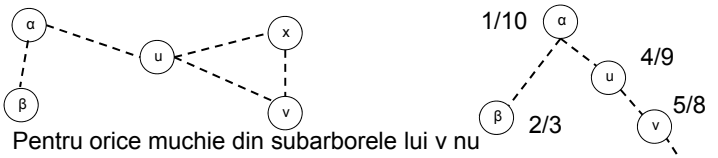
- Teorema 5.15: G = (V,E), graf neorientat si $u \in V$. u este punct de articulație in G $\Leftrightarrow$ in urma DFS in G una din proprietățile de mai jos este satisfăcută:
  - p(u) = null si u domina cel puțin 2 subarbori;
  - p(u) ≠ null si $\exists$v descendent al lui u in Arb(u) a.i. $\forall x \in$Arb(v) si $\forall$(x,z) parcurs de DFS(G) d(z) ≥ d(u).

Situații posibile

- 1) p(u) = null si u domina cel puțin 2 subarbori:



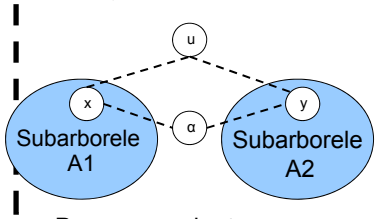
- 2) p(u) ≠ null si ∃v descendent al lui u in Arb(u) a.i. ∀x∈Arb(v) si ∀(x,z) parcurs de DFS(G) d(z) ≥ d(u):



Pentru orice muchie din subarborele lui v nu există nici o muchie înapoi spre un nod descoperit înaintea lui u.

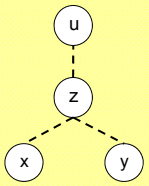
Puncte de articulație. Demonstrație teorema (Ia)

- p(u) = null si u domina cel puțin 2 subarbori ⟹ u este punct de articulație.
- Dem (Reducere la absurd): Pp ∃ x..α..y si u ∉ x..α..y.
- z = primul nod descoperit la DFS din care se poate ajunge la x si la y. Cf. T drumurilor albe x,y∈Arb(z).
- Dar x,y∈Arb(u) ➔ 2 cazuri:



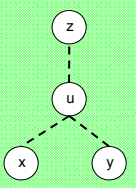
Pp ∃ x..α..y si u∉ x..α..y.

Caz 1: d(u)<d(z):



Contradictie (1) x,y nu sunt in subarbori diferiti ai lui Arb(u).

Caz 2: d(z)<d(u):

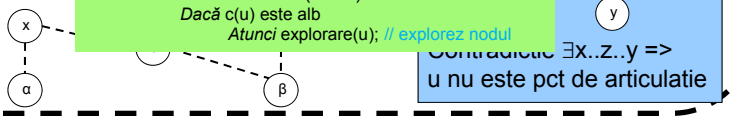


Contradictie (1), p(u)≠null.

Puncte de articulație. Demonstrație teorema (Ib)

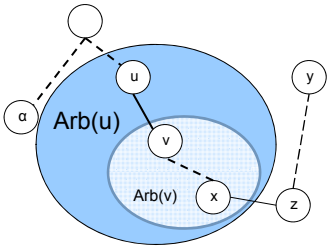
- u este punct de articulație si este descoperit in ciclul principal al DFS ⟹ p(u) = null si u domina cel puțin 2 subarbori.
- Dem (Reducere la absurd): Fie nodurile x si y a.i. u ∈ ∀ x..y. u = primul nod descoperit din cale (altfel u nu mai e descoperit in ciclul principal al DFS) => p(u) = null si x,y ∈ Arb(u).
- pp ca x, y su
  radacina ce
  → u nu e pu
  contrazice ip

```
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 până la nodul curent
    Pentru fiecare nod u (u ∈ V)
        Dacă c(u) este alb
            Atunci explorare(u); // explorez nodul
```



Puncte de articulație. Demonstrație teorema (IIa)

- p(u) ≠ null si ∃ v descendent al lui u in Arb(u) a.i. ∀ x∈Arb(v) si ∀ (x,z) parcurs de DFS(G) având d(z) ≥ d(u) ⟹ u este punct de articulație.



Dem (Reducere la absurd): Pp. u nu e punct de articulație → ∃ w∈Arb(v), y∉Arb(u) a.i. y..w. Fie z primul nod din y..w a.i. z ∉Arb(u) si x ultimul nod din y..w a.i. x∈Arb(u) →(x,z) taie frontiera Arb(u).

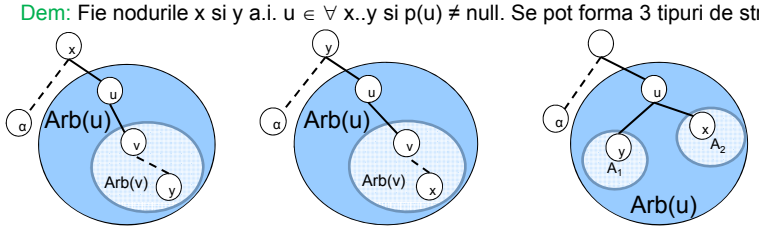
Daca d(z)>d(u) → u..x,z alb la d(u) → z∈Arb(u) → contradicție (z∉Arb(u))

Daca d(z)<d(u) → contradicție (ipoteza)

→∄ y..w → u punct de articulație

Puncte de articulație. Demonstrație teorema (IIb)

- u este punct de articulație si nu este descoperit in ciclul principal al DFS $\Rightarrow$ p(u) $\neq$ null si $\exists$ v descendent al lui u in Arb(u) a.i. $\forall$ x$\in$Arb(v) si $\forall$ (x,z) parcurs de DFS(G) având d(z) $\geq$ d(u).
- Dem: Fie nodurile x si y a.i. u $\in \forall$ x..y si p(u) $\neq$ null. Se pot forma 3 tipuri de structuri:



- Pentru primele 2 structuri, nu trebuie sa existe muchie care sa formeze ciclu de la nici un nod din Arb(v) către vreun predecesor al lui u. Altfel $\exists$ x..y a.i. u $\notin$ x..y.
- Pentru a 3-a structura, trebuie sa $\nexists$ muchie care sa formeze ciclu către un predecesor al lui u de la niciun nod din cel puțin un subarbore A_1 sau A_2.

Puncte de articulație. Structuri de date

- Structura de date de la DFS + pentru fiecare nod u$\in$V se reține:
  - Low(u) = min{d(v) | v descoperit pornind din u in cursul DFS si c(v) $\neq$ alb}
  - Subarb(u) = numărul subarborilor dominați de u (dacă e $\geq$ 2, atunci avem un punct de articulație).

Idee algoritm

- Se aplică DFS si se salvează pentru fiecare nod pana unde merge înapoi (low): low[u] = min {d(u), d(v) pt. toate muchiile înapoi (u,v), low(w) pentru toți fii w ai lui u}.
- Pentru eficienta, trebuie ca copiii sa se parcurgă înaintea părinților → ordinea inversă a d(u).

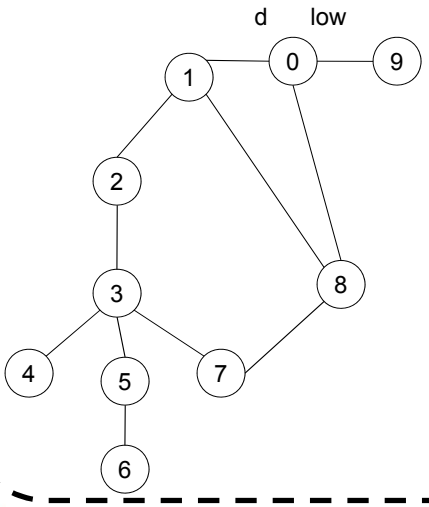
Algoritm Tarjan (I)

- Articulații (G)
  - V=noduri(G) // inițializări
  - Timp=0;
  - ***Pentru fiecare*** (u∈V)
    - culoare[u]=alb;
    - d[u]=0;
    - low[u]=0;
    - p[u]=null;
    - subarb[u]=0; // retine numărul de subarbori dominați de u
    - art[u]=0; // retine punctele de articulație
  - ***Pentru fiecare*** (u∈V)
    - ***Dacă*** (culoare(u) e alb)
      - Exploreaza(u);
      - ***Dacă*** (subarb[u] > 1) // cazul in care u este rădăcina in arborele
        - art[u] = 1 // DFS si are mai mulți subarbori → cazul 1 al // teoremei

Algoritm Tarjan (II)

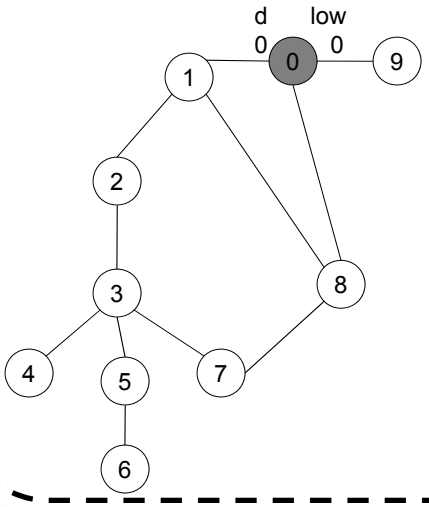
- Explorează(u)
  - d[u] = low[u] = timp++; // inițializări
  - culoare[u] = gri;
  - ***Pentru fiecare*** (v succesor al lui u)
    - ***Dacă*** (culoare[v] e alb)
      - p[v] = u; subarb[u]++; // actualizare nr subarbori // dominați de u
      - Explorează(v);
      - low[u] = min{low[u], low[v]} // actualizare low
      - ***Dacă*** (p[u] != null && low[v] ≥ d[u]) art[u] = 1; // cazul 2 al teoremei
    - ***Altfel*** low[u]=min{low[u], d[v]} // actualizare low

Exemplu rulare (1)



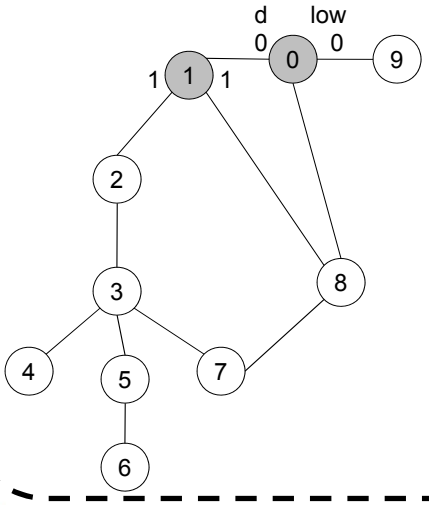
Timp =0
Cul[i] =alb
d[i]=0
Low[i] = 0
P[i]=null
Subarb[i] =0
Art[i] = 0
Exploreaza (0)

Exemplu rulare (2)



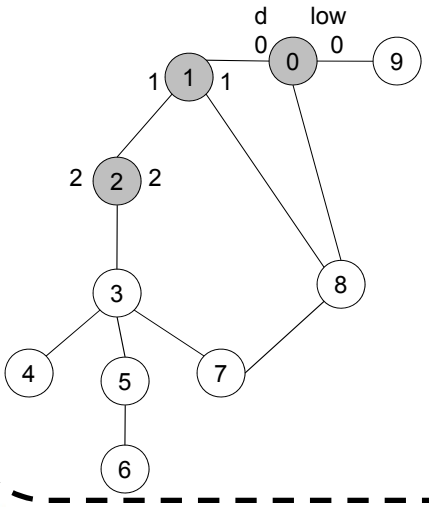
Low[0] = d[0]=0
Timp =1
Cul[0] =gri
P[1]=0
Subarb[0] =1
Exploreaza (1)

Exemplu rulare (3)



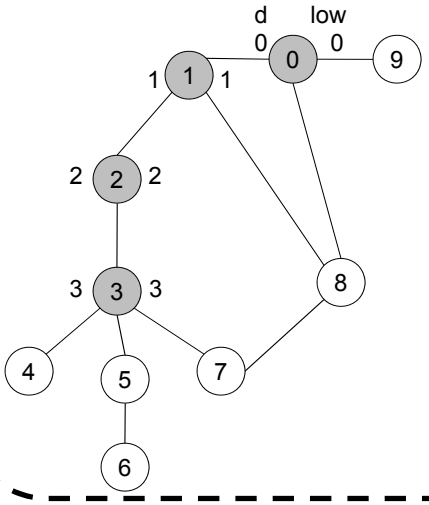
Low[1] = d[1]=1
Timp =2
Cul[1] =gri
P[2]=1
Subarb[1] =1
Exploreaza (2)

Exemplu rulare (4)



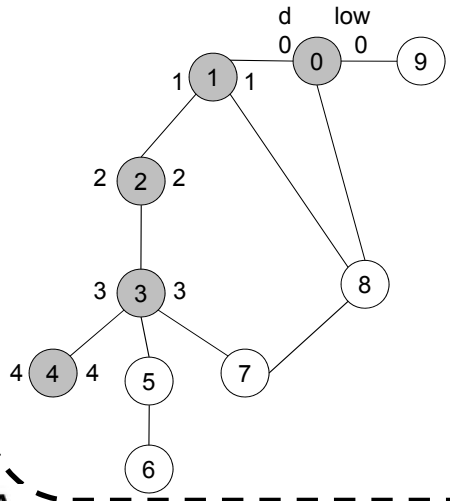
Low[2] = d[2]=2
Timp =3
Cul[2] =gri
P[3]=2
Subarb[2] =1
Exploreaza (3)

Exemplu rulare (5)



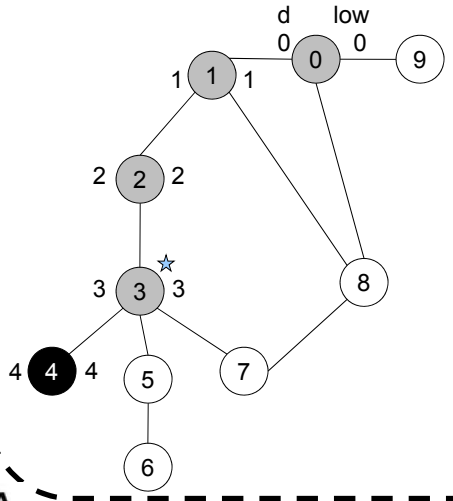
Low[3] = d[3]=3
Timp =4
Cul[3] =gri
P[4]=3
Subarb[3] =1
Exploreaza (4)

Exemplu rulare (6)



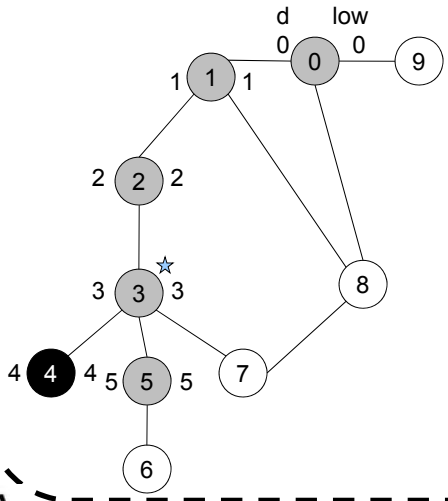
Low[4] = d[4]=4
Timp =5
Cul[4] =gri
revenire

Exemplu rulare (7)



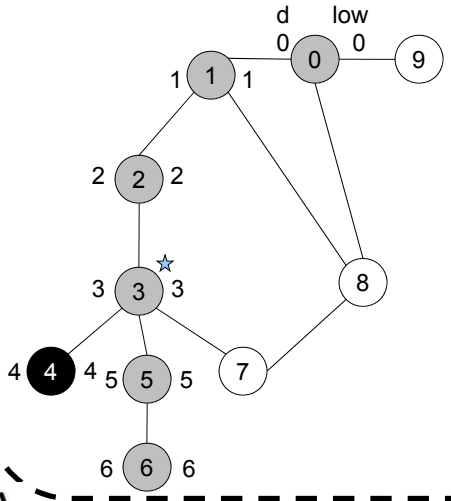
Low[4] = d[4]=4
Timp =5
Cul[4] =gri
revenire
Low [3] = min {low[3], low[4]}= 3
Low[4] > d[3] → art[3] = 1
P[5]=3
Subarb[3] =2
Exploreaza (5)

Exemplu rulare (8)



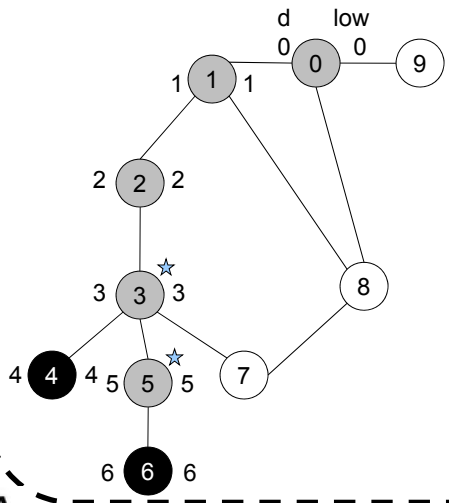
Low[5] = d[5]=5
Timp =6
Cul[5] =gri
P[6]=5
Subarb[5] =1
Exploreaza (6)

Exemplu rulare (9)



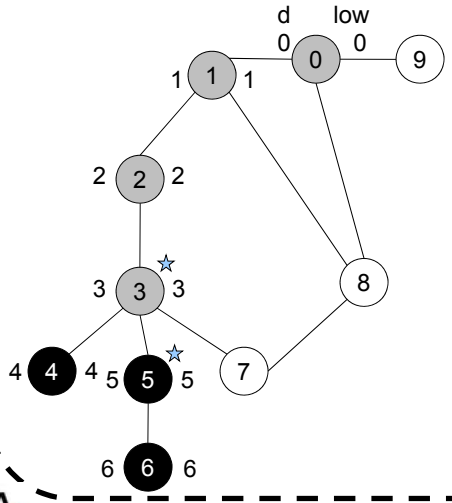
Low[6] = d[6]=6
Timp =7
Cul[6] =gri
revenire

Exemplu rulare (10)



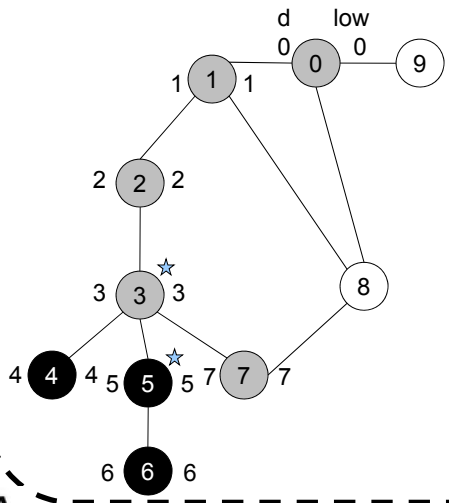
Low[6] = d[6]=6
Timp =7
Cul[6] =gri
revenire
Low [5] = min {low[5], low[6]}= 5
Low[6] > d[5] → art[5] = 1
revenire

Exemplu rulare (11)



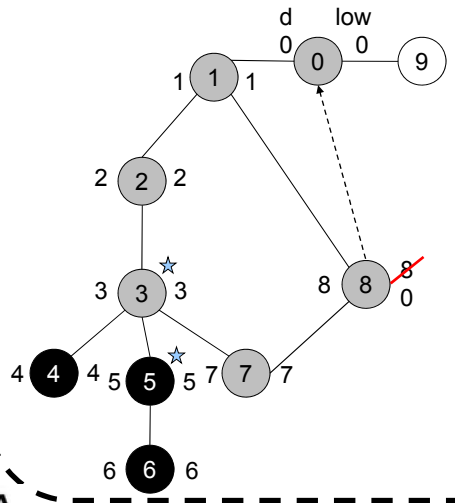
Low[5] = d[5]=5
Timp =7
Cul[5] =gri
revenire
Low [3] = min {low[3], low[5]}= 3
Low[5] > d[3] → art[3] = 1
P[7]=3
Subarb[3] =3
Exploreaza (7)

Exemplu rulare (12)



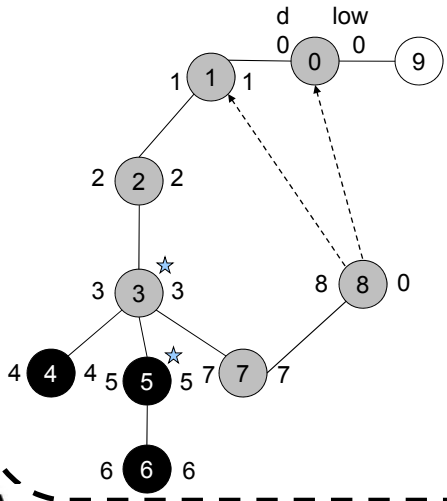
Low[7] = d[7]=7
Timp =8
Cul[7] =gri
P[8]=7
Subarb[7] =1
Exploreaza (8)

Exemplu rulare (13)



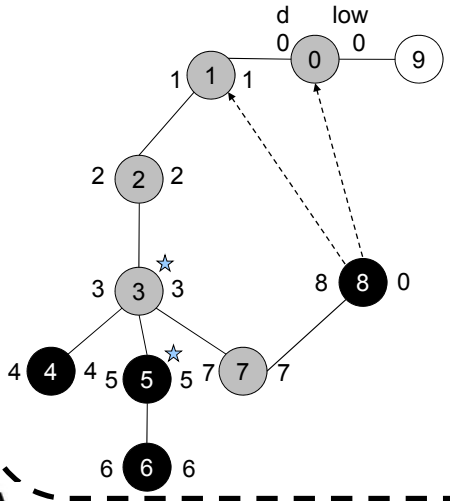
Low[8] = d[8]=8
Timp =9
Cul[8] =gri
Low[8] = min{d[0], low[8]}=0

Exemplu rulare (14)



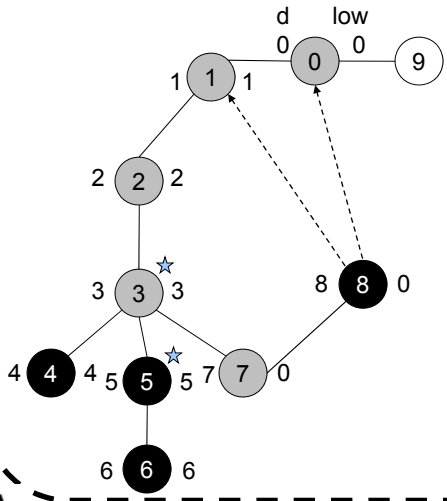
d[8]=8
Low[8] =0
Timp =9
Cul[8] =gri
Low[8] = min{d[1], low[8]}=0

Exemplu rulare (15)



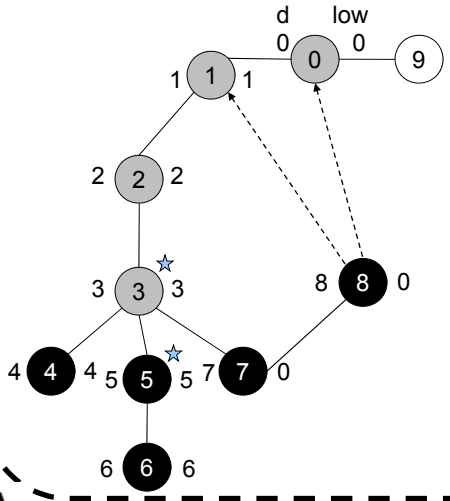
d[8]=8
Low[8] =0
Timp =9
Cul[8] =gri
revenire

Exemplu rulare (16)



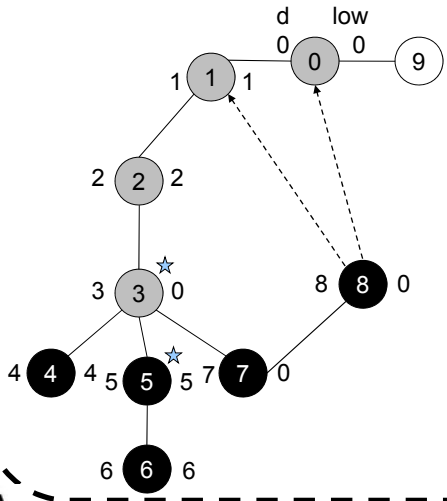
low[7] = min(low[7], low[8]) = 0
low[8] < d[7] → nu se modifica art[7]

Exemplu rulare (17)



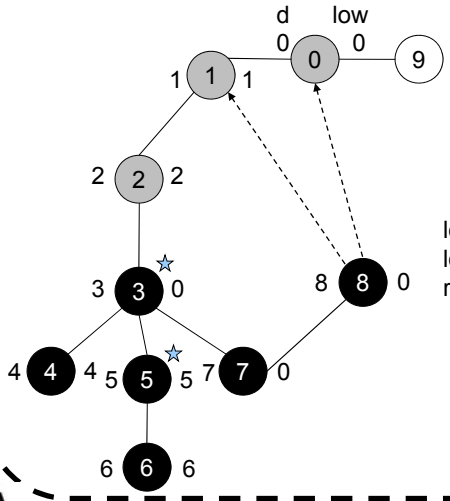
low[7] = min(low[7], low[8]) = 0
revenire

Exemplu rulare (18)



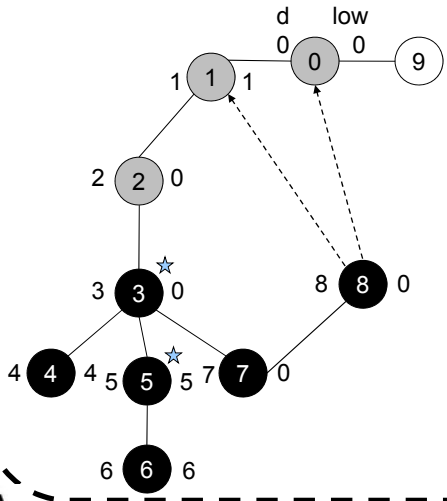
low[3] = min(low[3], low[7]) = 0
low[7] < d[3] → nu se modifica art[3]

Exemplu rulare (19)



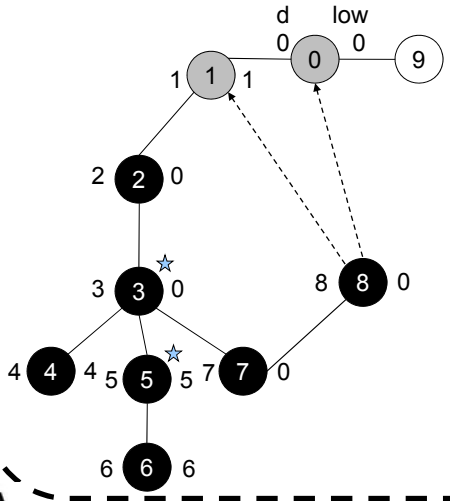
low[3] = min(low[3], low[7]) = 0
low[7] < d[3] → nu se modifica art[3]
revenire

Exemplu rulare (20)



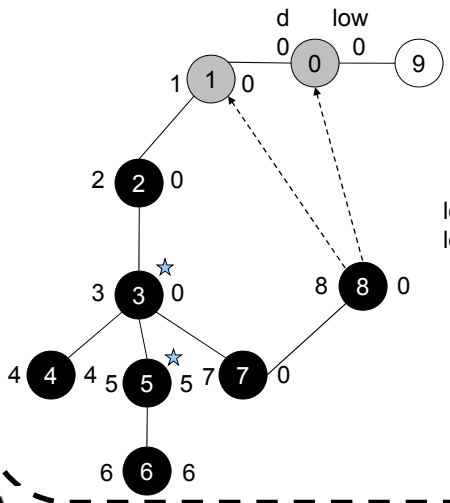
low[2] = min(low[2], low[3]) = 0
low[3] < d[2] → nu se modifica art[2]

Exemplu rulare (21)



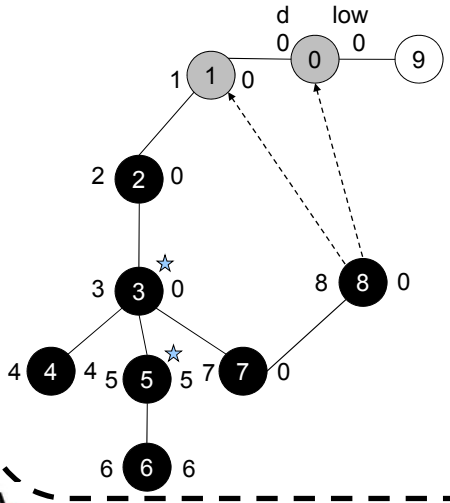
low[2] = min(low[2], low[3]) = 0
low[3] < d[2] → nu se modifica art[2]
revenire

Exemplu rulare (22)



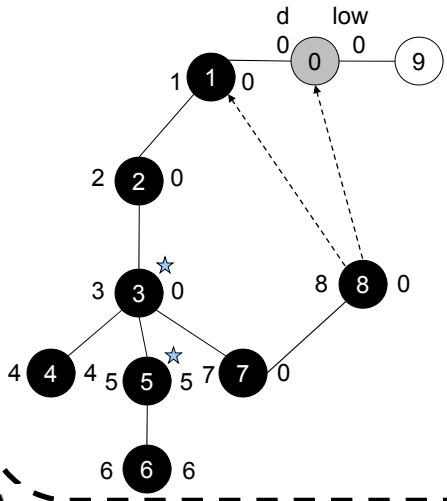
low[1] = min(low[1], low[2]) = 0
low[2] < d[1] → nu se modifica art[1]

Exemplu rulare (23)



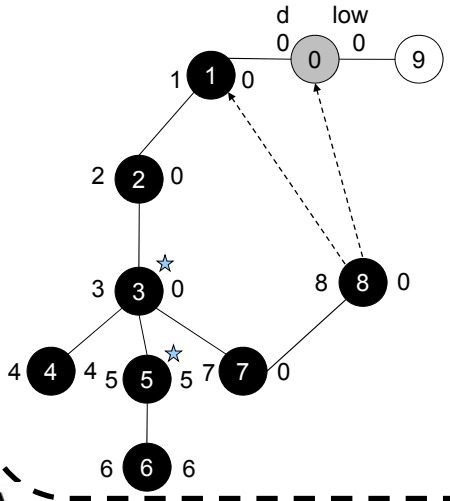
low[1] = 0
low[1] = min(low[1], d[8]) = 0

Exemplu rulare (24)



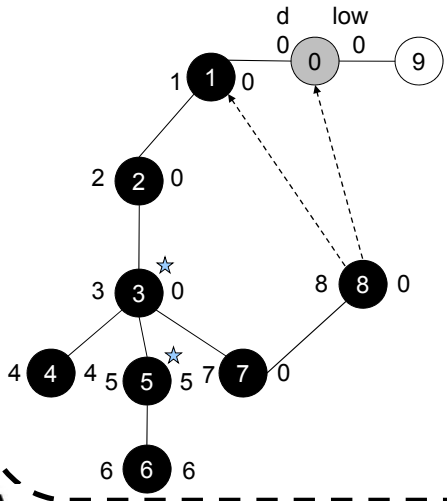
low[1] = 0
revenire

Exemplu rulare (25)



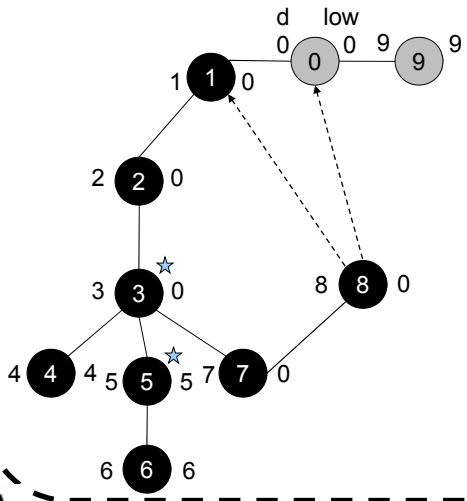
low[0] = min{low[1], low[0]} =0
P[0] = null → continuă cu următorul copil

Exemplu rulare (26)



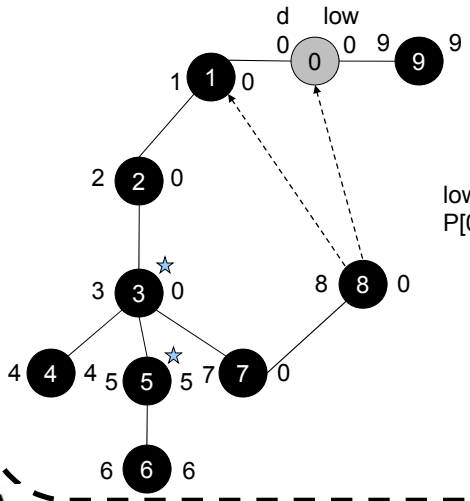
low[0] = min(low[0],d [8])=0
P[9]=0
Subarb[0] = 2
Exploreaza (9)

Exemplu rulare (27)



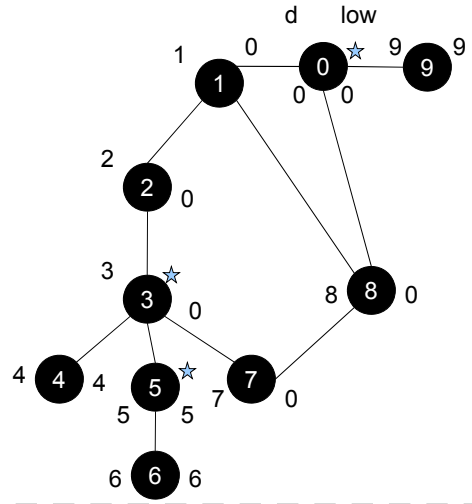
Low[9] = d[9]=9
Timp = 10
Cul[9] =gri
revenire

Exemplu rulare (28)



low[0] = min(low[0], low[9]) = 0
P[0] = null → revenire

Exemplu rulare (29)

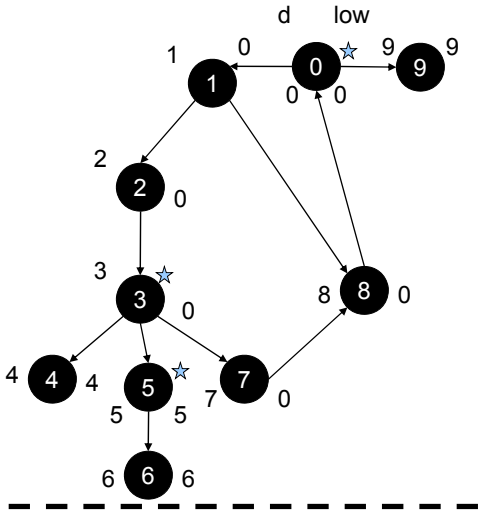


Algoritmul lui Tarjan adaptat pentru determinarea CTC

- index = 0 // nivelul pe care este nodul in arborele DFS
- S = empty // se folosește o stiva care se inițializează cu Ø
- ***Pentru fiecare*** v in V do
  - ***Dacă*** (v.index is undefined) ***atunci*** // se pornește DFS din fiecare nod pe care
    - tarjan(v) // nu l-am vizitat încă

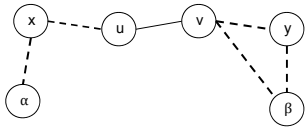
- procedure tarjan(v)
  - v.index = index // se setează nivelul nodului v
  - v.lowlink = index // retine strămoșul nodului v
  - index = index + 1 // incrementez nivelul
  - S.push(v) // introduc v in stiva
  - ***Pentru fiecare*** (v, v') in E do // se prelucrează succesorii lui v
    - ***Dacă*** (v'.index is undefined or v' is in S) ***atunci*** // CTC deja identificate sunt ignorate
      - ***Dacă*** (v'.index is undefined) ***atunci***
        - tarjan(v') // daca nu a fost vizitat v' intru in recursivitate
        - v.lowlink = min(v.lowlink, v'.lowlink) //actualizez strămoșul
      - ***Altfel Dacă*** (v' is in S) ***atunci***
        - v.lowlink = min(v.lowlink, v'.index) //muchie inapoi catre un nod din aceeasi CTC
  - ***Dacă*** (v.lowlink == v.index) ***atunci*** // printez CTC începând de la coadă spre rădăcină
    - print "Nodurile unei CTC:"
    - ***Repetă***
      - v' = S.pop // extrag nodul din stiva si il printez
      - print v'
    - ***Până când*** (v' == v) // până când extrag rădăcina

Exemplu rulare (CTC)

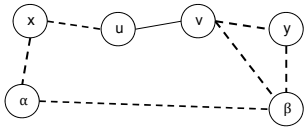


Punți

- Definiție: G = (V,E), graf neorientat si (u,v)∈E. (u,v) este punte in G ⇔∃ x,y∈V, x ≠ y, a.i. ∀ x..y conține muchia (u,v).



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



(u,v) nu este punte

Algoritm (I)

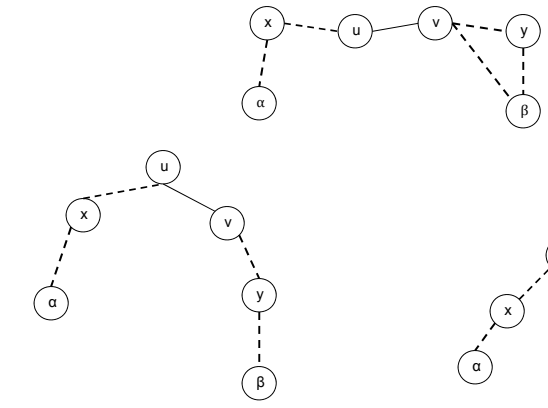
- Punți(G)
  - V = noduri(G) // inițializări
  - Timp = 0;
  - ***Pentru fiecare*** (u∈V)
    - culoare[u]=alb;
    - d[u]=0;
    - low[u]=0;
    - p[u]=null;
    - punte(u) =0; // subarb[u]=0; art[u]=0;
  - ***Pentru fiecare***(u∈V)
    - ***Dacă*** (culoare(u) e alb)
      - Explorează(u)

Algoritm (II)

- Explorează(u)
  - d[u] = low[u] = timp++; // inițializări
  - culoare[u] = gri;
  - ***Pentru fiecare***(v succesor al lui u)
    - ***Dacă*** (culoare[v] e alb)
      - P[v]=u; // subarb[u]++;
      - Explorează(v);
      - low[u]=min{low[u],low[v]} // actualizare low
      - ***Dacă*** (low[v]>d[u]) punte[v]=1;

      // ***Dacă***(p[u]!=null && low[v]>=d[u])
    - ***altfel***
      - ***Dacă*** (p[u] ≠ v) low[u]=min{low[u], d[v]} // actualizare low

Exemplu



DFS din u; puntea este detectata in v

DFS din v; puntea este detectata in u

Drumuri de cost minim

- G = (V,E) un graf, iar w:E -> $\Re$ o funcție de cost asociată arcelor grafului (w(u,v) = costul arcului (u,v)).
- Cost(u..v) = costul drumului u..v (este aditiv – costul drumului = suma costurilor arcelor).
- Variante:
  1. Drumuri punct – multipunct: pentru un nod dat s∈V, să se găsească un drum de cost minim de la s la ∀u∈V; Dijkstra, Bellman-Ford
  2. Drumuri multipunct – punct: pentru un nod dat e∈V, să se găsească un drum de cost minim de la ∀u∈V la e; G^T si apoi 1
  3. Drumuri punct – punct: pentru două noduri date u si v∈V, să se găsească un drum u..v de cost minim; Folosind 1
  4. Drumuri multipunct – multipunct: ∀u, v∈V, să se găsească un drum u..v de cost minim. Floyd-Warshall
  5. Drumuri de cost maxim!

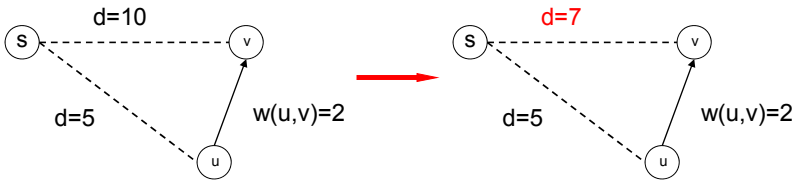
Temă de gândire pentru acasă – posibil subiect de examen!

Drumuri minime de sursa unica

- Sunt concepuți pentru grafuri orientate.
- Bazați pe algoritmi greedy.
- Se pornește de la nodul de start si pe baza unui optim local, drumurile sunt extinse si optimizate pană la soluția finală.
- Notații:
  - d(v) = costul drumului descoperit s..v;
  - δ(u,v) = costul drumului optim u..v; δ(u,v)=∞ daca v∉R(u);
  - p(v) = predecesorul lui v pe drumul s..v.

Drumuri minime de sursa unica

- Relaxarea muchiei → dacă d[v] > d[u] + w(u,v), atunci actualizează d[v].



- Exemple: Dijkstra si Bellman–Ford.

Algoritmul lui Dijkstra (I)

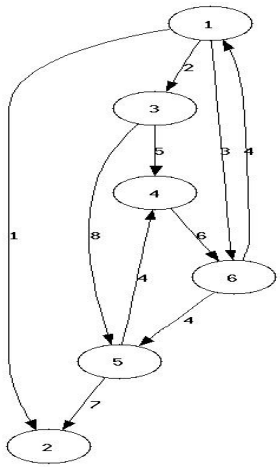
- Folosește o coada de priorități in care se adaugă nodurile in funcție de distanța cunoscută in momentul respectiv de la s până la nod.
- Se folosește **NUMAI** pentru costuri pozitive (w(u,v) > 0, ∀u,v∈V).
- Dijkstra_generic (G,s)
  - V = nodurile lui G
  - ***Cat timp*** (V != ∅)
    - u = nod din V cu d[u] min
    - V = V - {u}
    - ***Pentru fiecare*** (v∈succesorii lui u) relaxare_arc(u,v)

    // optimizare drum s..v pentru v∈succesorilor lui u

Algoritmul lui Dijkstra (II)

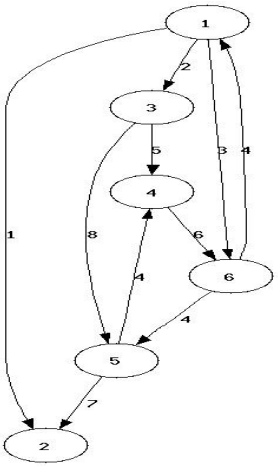
- Dijkstra(G,s)
  - ***Pentru fiecare*** (u ∈ V)
    - d[u] = ∞; p[u] = null;
  - d[s] = 0;
  - Q = construiește_coada(V) // coadă cu priorități
  - ***Cat timp*** (Q != ∅)
    - u = ExtrageMin(Q); // extrage din V elementul cu d[u] minim
    - // Q = Q - {u} – se execută in cadrul lui ExtrageMin
    - ***Pentru fiecare*** (v ∈ Q si v din succesorii lui u)
      - ***Daca*** (d[v] > d[u] + w(u,v))
        - d[v] = d[u] + w(u,v) // actualizez distanța
        - p[v] = u // si părintele

Exemplu (I)



Exemplu (II)

- d[1] = 0;
- (1): d[2] = 1; d[3] = 2; d[6] = 3;
- (2): d[4] = 7; d[5] = 10;
- (3): d[5] = 7;



Complexitate Dijkstra

- Depinde de ExtrageMin – coadă cu priorități.
- Operații ce trebuie realizate pe coadă + frecvența lor:
  - insert – V;
  - delete – V;
  - conține? – V;
  - micșorează_val – E;
  - este_vidă? – V.

- Dijkstra(G,s)
  - ***Pentru fiecare*** (u ∈ V)
    - d[u] = ∞; p[u] = null;
  - d[s] = 0;
  - Q = construiește_coada(V) // coadă cu priorități
  - ***Cat timp*** (Q != ∅)
    - u = ExtrageMin(Q); // extrage din V elementul cu d[u] minim
    - // Q = Q - {u} – se execută in cadrul lui ExtrageMin
    - ***Pentru fiecare*** (v ∈ Q si v din succesorii lui u)
      - ***Daca*** (d[v] > d[u] + w(u,v))
        - d[v] = d[u] + w(u,v) // actualizez distanța
        - p[v] = u // si părintele

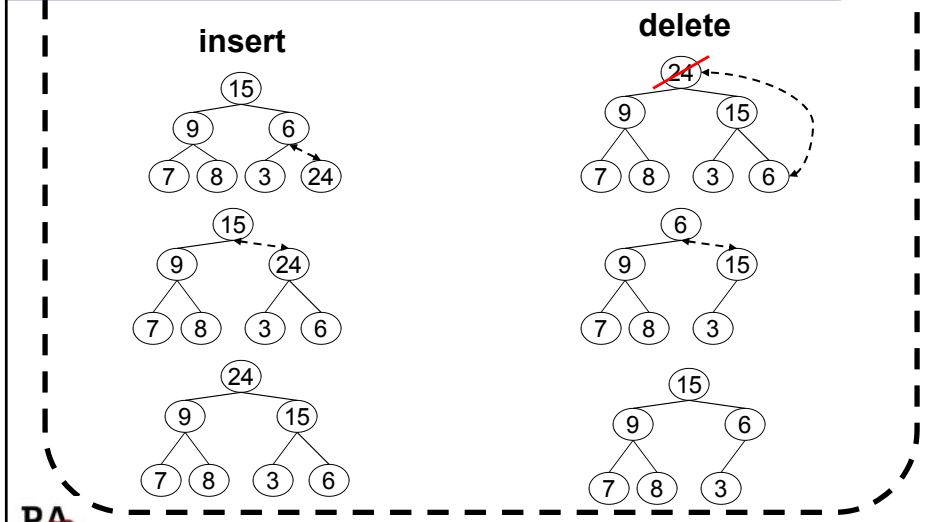
Implementare cu vectori

- Costuri:
  - insert – 1 * V = V;
  - delete – V * V = V^2 (necesită căutarea minimului);
  - conține? – 1 * V = V;
  - micșorează_val – 1 * E = E;
  - este_vidă? – 1 * V = V;
- Cea mai bună metodă pentru grafuri "dese" ($E \approx V^2$)!

Implementare cu heap binar

- Heap binar – structură de date de tip arbore binar + 2 constrângeri:
  - Fiecare nivel este complet; ultimul se umple de la stânga la dreapta;
  - $\forall u \in$ Heap; u $\geq$ răd(st(u)) && u $\geq$ răd(dr(u)) unde $\geq$ este o relație de ordine pe mulțimea pe care sunt definite elementele heapului.

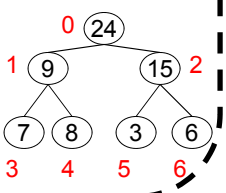
Operatii pe Heap Binar



Implementare Heap Binar

- Implementare folosind vectori.
- Poziție[i] = unde se găsește in indexul de valori elementul de pe poziția i din heap.
- Reverse[i] = unde se găsește in heap elementul de pe poziția i din valoare.
- Implementare disponibila la [3].

Index	0	1	2	3	4	5	6
Valoare	7	6	15	8	24	9	3
Poziție	4	5	2	0	3	6	1
Reverse	3	6	2	4	0	1	5



Heap Binar

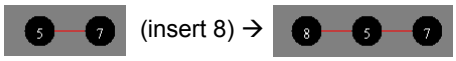
- Costuri:
  - insert – logV * V = VlogV;
  - delete – logV * V = VlogV;
  - conține? – 1 * V = V;
  - micșorează_val – logV * E = ElogV;
  - este_vidă? – 1 * V = V.
- Eficient dacă graful are muchii puține comparativ cu numărul de noduri.

Heap Fibonacci

- Poate fi format din mai mulți arbori.
- Cheia unui părinte ≤ cheia oricărui copil.
- Fiind dat un nod u si 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 si din dreapta (cei de pe primul nivel sunt legați intre 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.

Operatii Heap Fibonacci

- Inserare nod – O(1)
  - construiește un nou arbore cu un singur nod

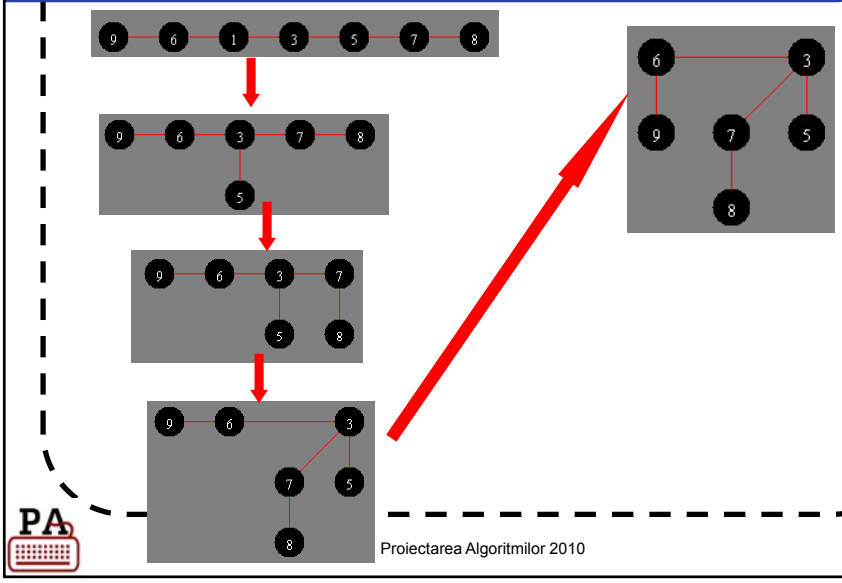


- Min – accesibil direct - min(H) – O(1)
- ExtrageMin O(logn) – cost amortizat!
  - Mută copiii minimului pe prima coloană;
  - **Consolidează** heap-ul.

Operatii Heap Fibonacci

- Consolidare Heap
  - Cat timp există 2 arbori cu grade diferite Arb(x) si Arb(y), x < y:
    - Arb(y) adăugat ca si copil al lui x;
    - grad[x] ++;
- Applet si implementare disponibile la [4].

Consolidare Heap



Costuri Heap Fibonacci

- Costuri:
  - 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.

Concluzii Dijkstra

- Implementarea trebuie realizată in funcție de tipul grafului pe care lucrăm:
  - vectori pentru grafuri “dese”;
  - heap pentru grafuri “rare”.
- Heapul Fibonacci este mai eficient decât heapul binar dar mai dificil de implementat.

Întrebări?