M. Caramihai, © 2020

STRUCTURI DE DATE & ALGORITMI

CURS 7

Grafuri (2)

Inregistrarea drumului minim

- BFS ne spune doar daca exista un drum de la sursa **s** la alte noduri **v**.
  - Nu inregistreaza drumul!
  - Algoritmul trebuie modificat in acest sens.

- Este posibil?
  - Nota: nu se cunosc nodurile ce fac parte din drum inainte ca nodul **v** sa fie atins!
  - O solutie posibila (si eficienta):
    - Se utilizeaza un vector suplimentar *pred[0..n-1]*
    - *Pred[w] = v* semnifica faptul ca nodul *w a fost vizitat din v*

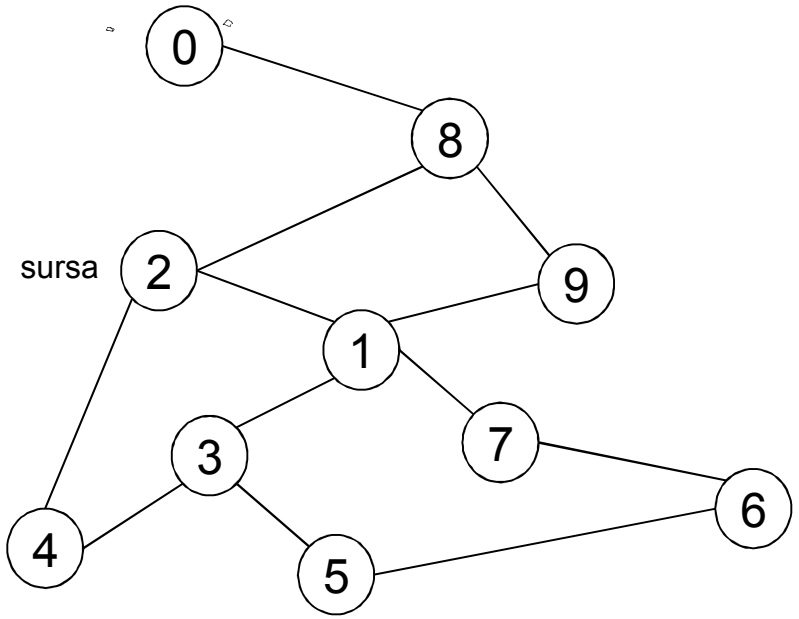
BFS + gasirea drumului

```
Algorithm BFS(s)
1.    for each vertex v
2.        do flag(v) := false;
3.           pred[v] := −1;
4.    Q = empty queue;
5.    flag[s] := true;
6.    enqueue(Q, s);
7.    while Q is not empty
8.       do v := dequeue(Q);
9.          for each w adjacent to v
10.            do if flag[w] = false
11.                  then flag[w] := true;
12.                       pred[w] := v;
13.                       enqueue(Q, w)
```

Initializare pentru toti pred[v] la -1

Inregistrare de unde se vine

Exemplu (1)



Lista adiacenta

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Pred

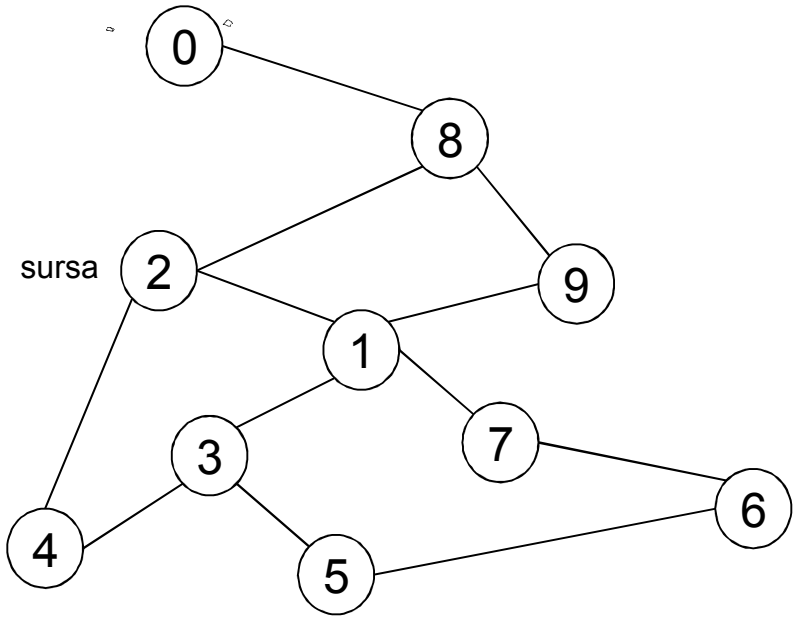
Q = { }

Initializare **Q** (gol)

Initializare tabel vizitat (*False*)

Initializare Pred cu -1

Exemplu (2)



Lista adiacenta

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	F	-
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

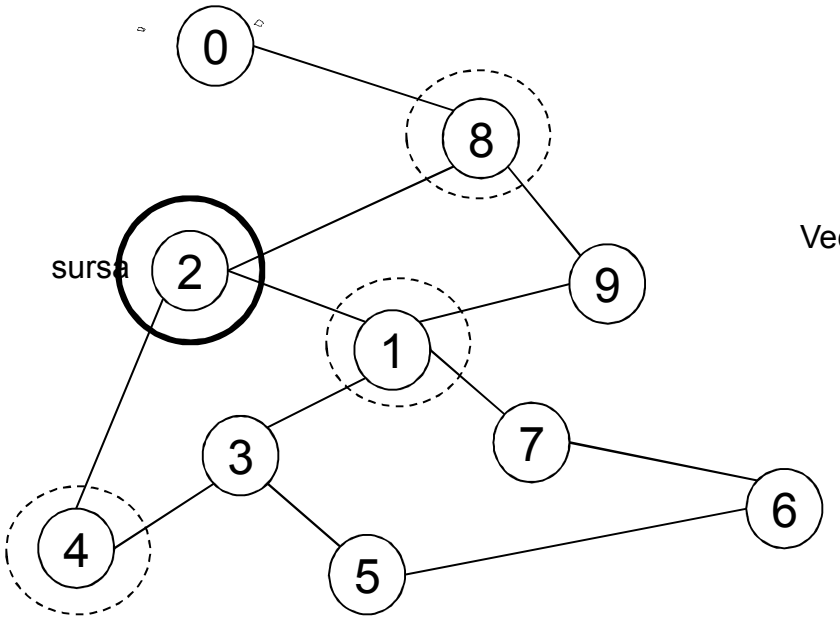
Pred

Q = { 2 }

Plasare sursa 2 in coada.

Flag faptul ca 2 a fost vizitat.

Exemplu (3)



Vecini →

Lista adiacenta

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

-
2
-
-
2
-
-
-
2
-

Pred

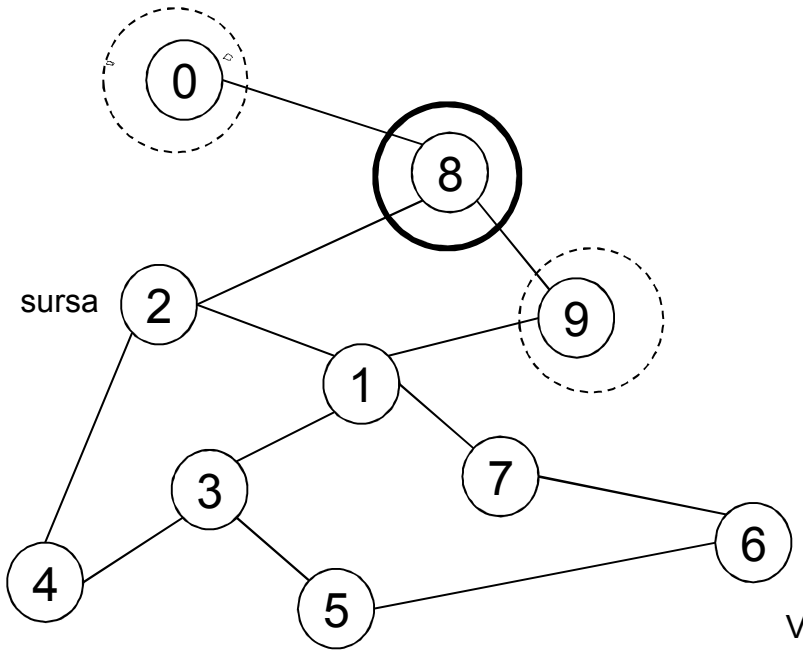
Q = {2} → { 8, 1, 4 }

Dequeue 2.
Toti vecinii nevizitati ai lui 2 se pun in coada

Marcare vecini vizitati.

Inregistrare in Pred ca se vine din 2.

Exemplu (4)



Lista adiacenta

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Vecini →

Tabel vizitat (T/F)

0	T	8
1	T	2
2	T	-
3	F	-
4	T	2
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

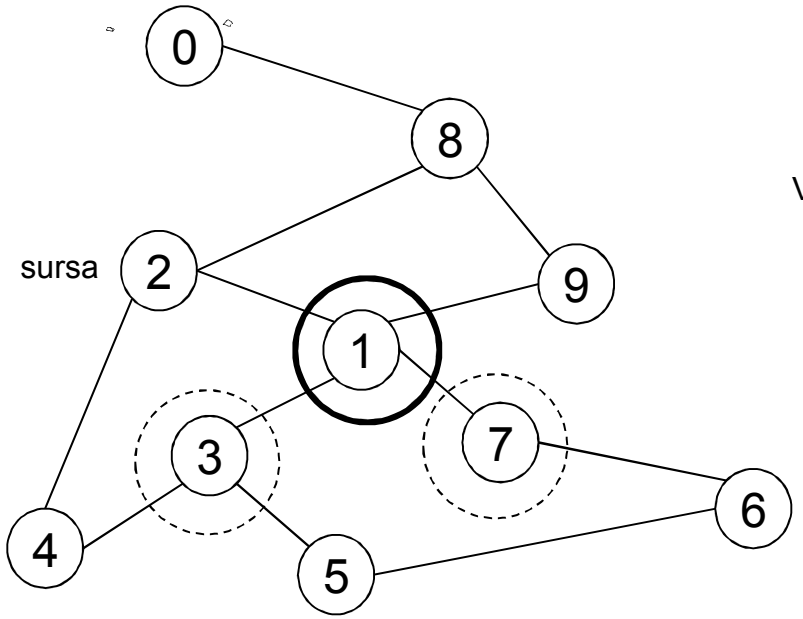
Pred

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Marcare vecini vizitati.
Inregistrare in Pred ca se vine din 8

Dequeue 8.
-- Plasare vecini nevizitati ai lui 8 in coada.
-- 2 nu a fost plasat in coada – a fost vizitat !

Exemplu (5)



Lista adiacenta

Vecini →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

8
2
-
1
2
-
-
1
2
8

Pred

Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.

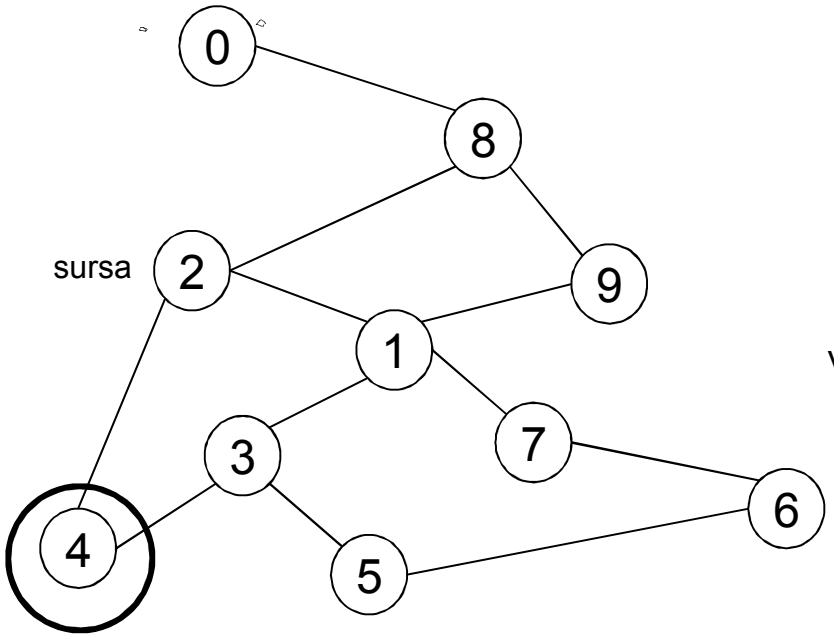
-- Plasare vecini nevizitati ai lui 1 in coada.

-- Numai nodurile 3 si 7 nu au fost vizitate acum.

Marcare vecini vizitati.

Inregistrare in Pred ca se vine din 1

Exemplu (6)



Lista adiacenta

Vecini

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

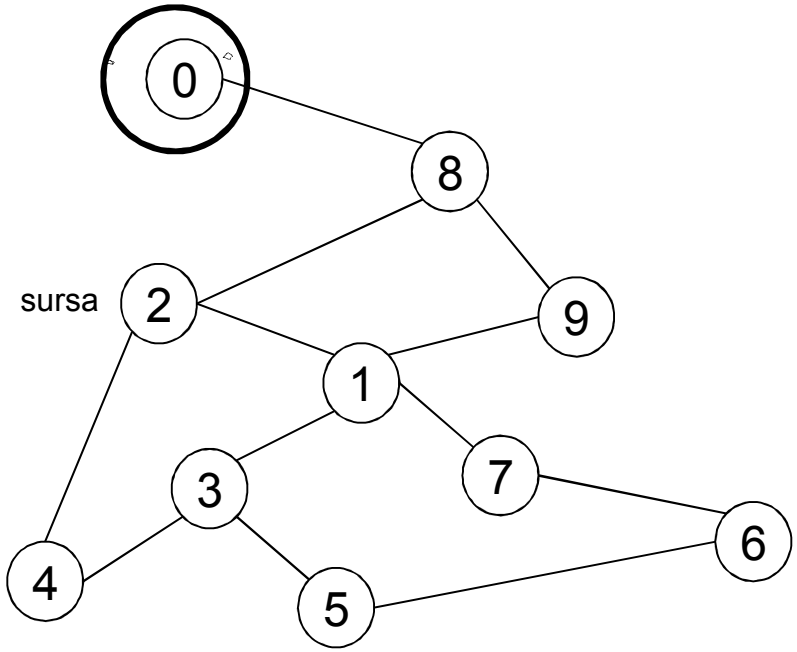
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Pred
8
2
-
1
2
-
-
1
2
8

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.
-- 4 nu are vecini nevizitati

Exemplu (7)



Lista adiacenta

Vecini →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

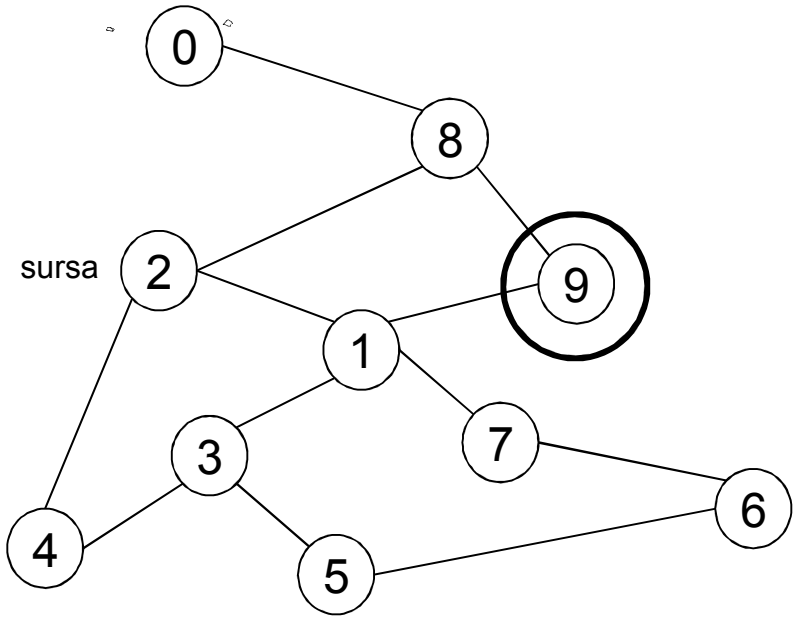
Pred

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

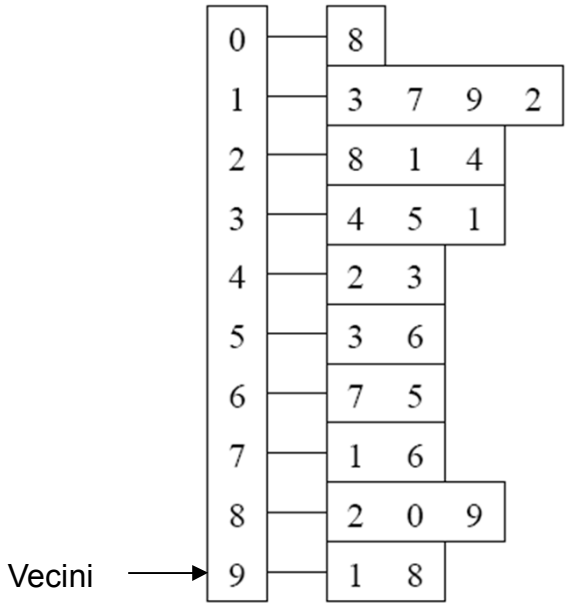
Dequeue 0.

-- 0 nu are vecini nevizitati!

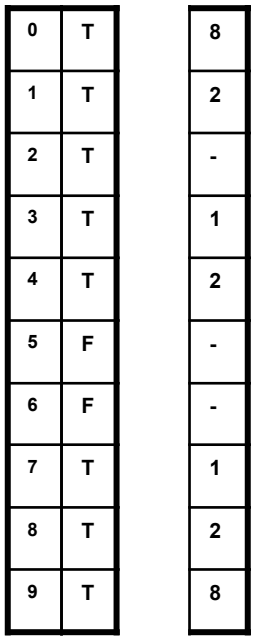
Exemplu (8)



Lista adiacenta



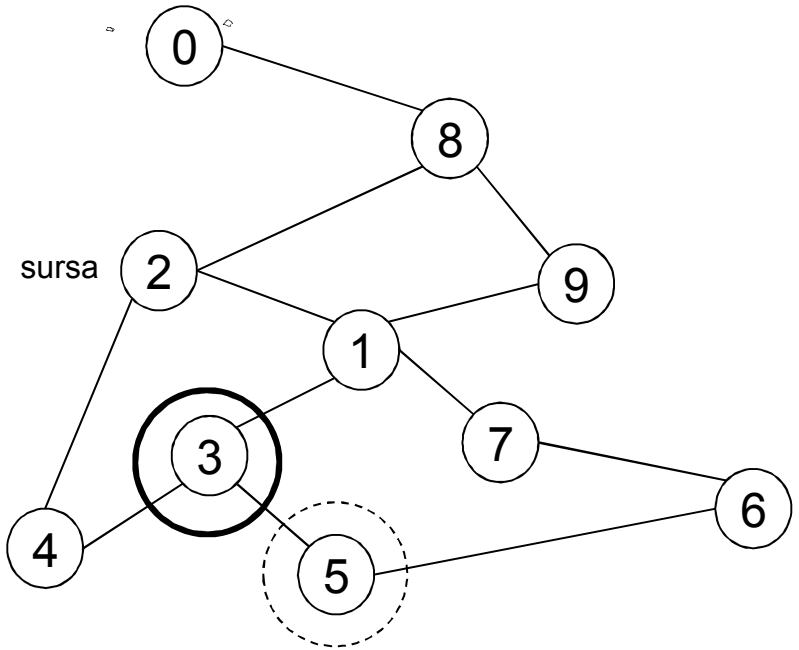
Tabel vizitat (T/F)



Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
-- 9 nu are vecini nevizitati!

Exemplu (9)



Lista adiacenta

Vecini →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

8
2
-
1
2
3
-
1
2
8

Pred

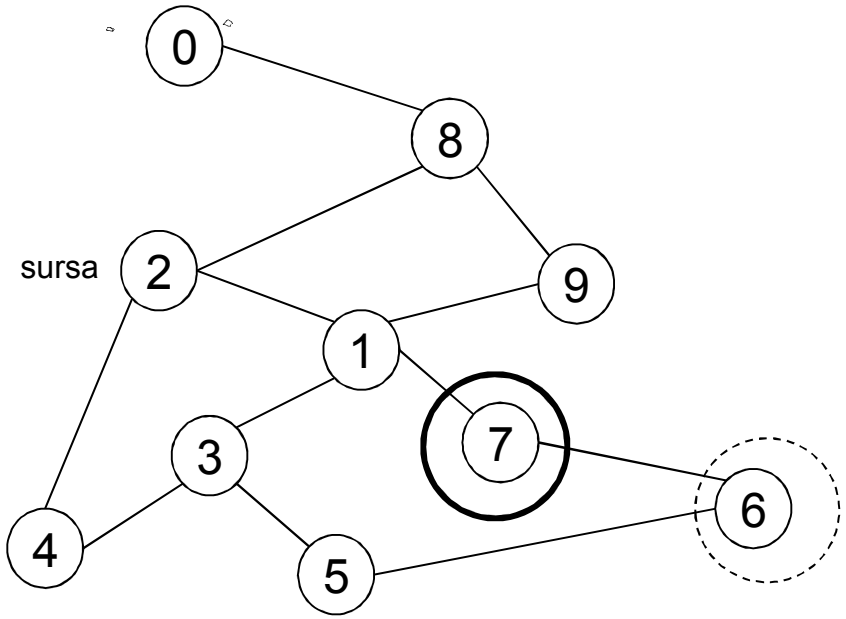
Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- vecinul 5 este plasat in coada.

Este marcat nodul 5

Se inregistreaza in Pred ca se vine din 3.

Exemplu (10)



Lista adiacenta

Vecini →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
2
-
1
2
3
7
1
2
8

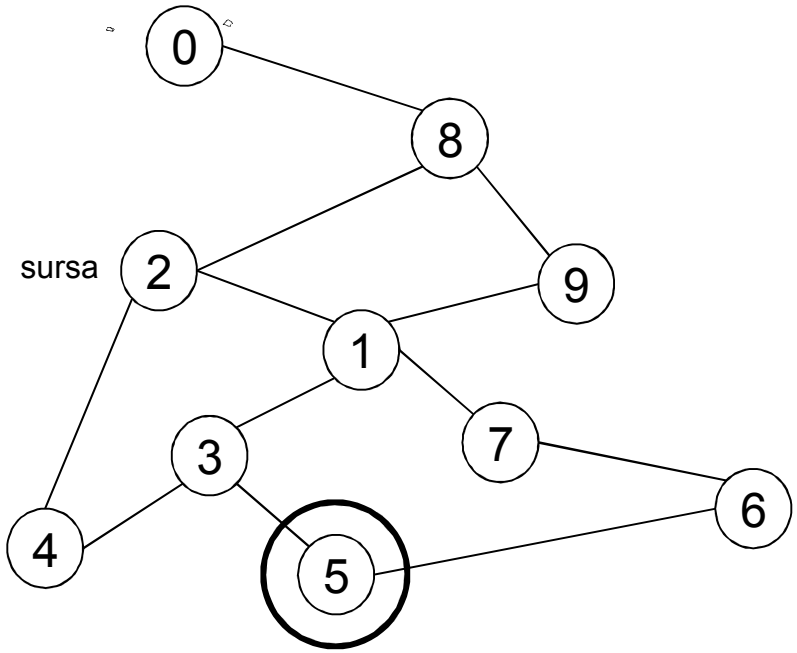
Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- vecinul 6 este plasat in coada.

Este marcat nodul 6

Se inregistreaza in Pred ca se vine din 7.

Exemplu (11)



Lista adiacenta

Vecini →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

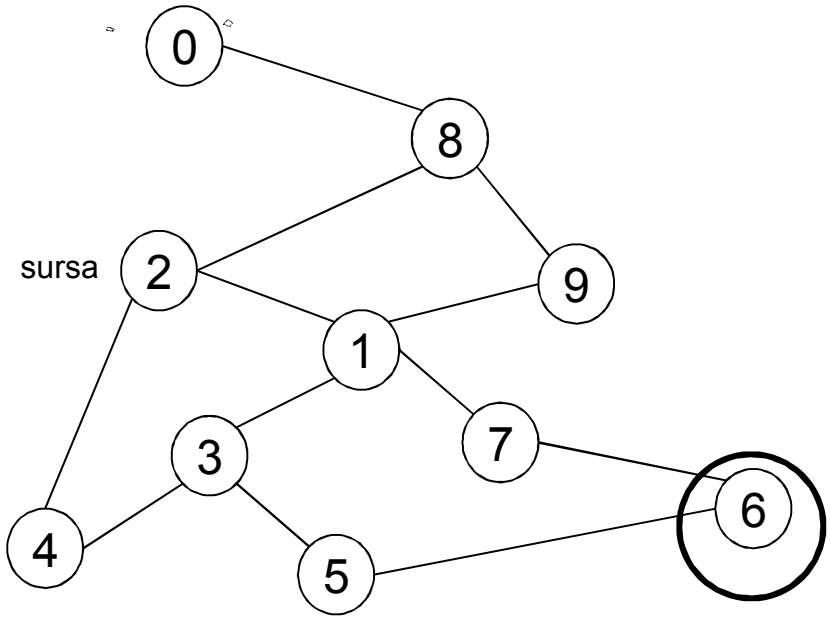
Pred

Q = { 5, 6} → { 6 }

Dequeue 5.

-- nu sunt vecini nevizitati pentru 5.

Exemplu (12)



Lista adiacenta

Vecini →

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

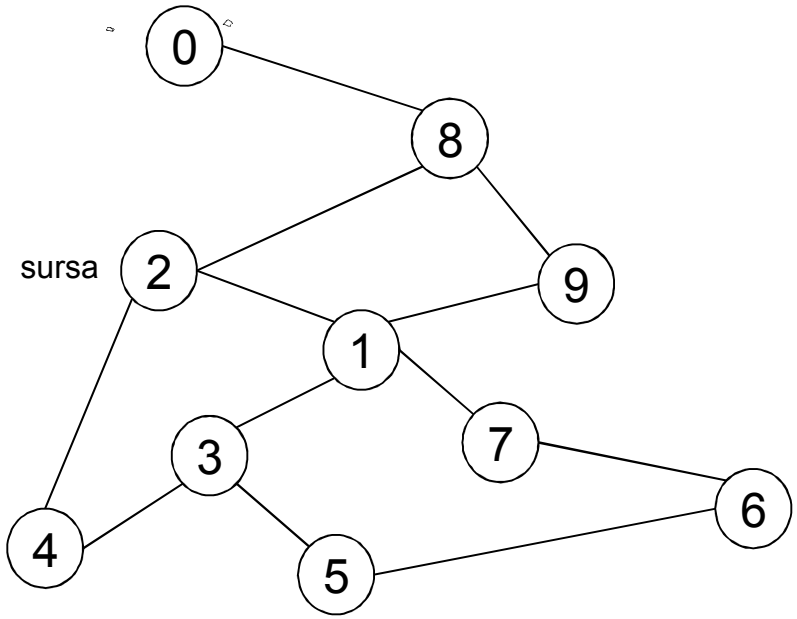
Pred

Q = { 6 } → { }

Dequeue 6.

-- nu exista vecini nevizitati pentru 6.

Exemplu (13)



Lista adiacenta

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

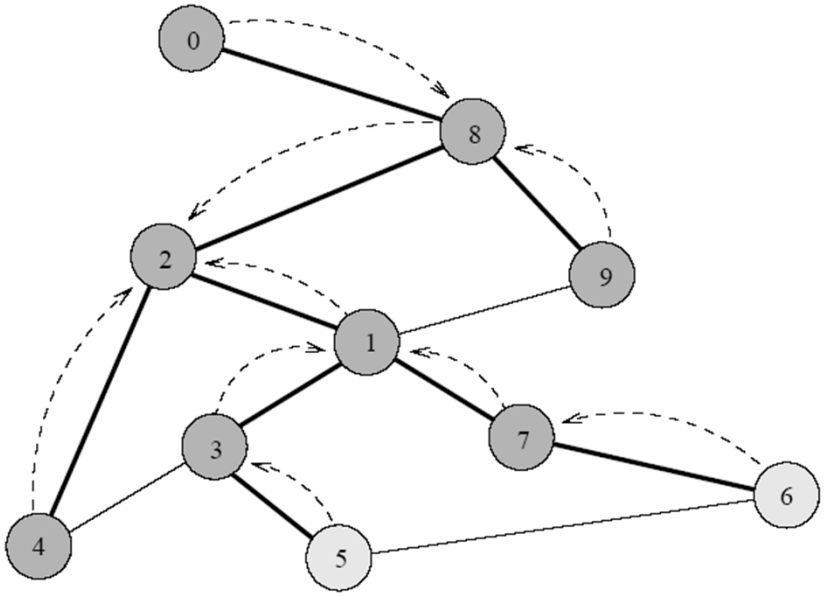
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Pred

Q = { } STOP!!! Q este gol!!!

Pred poate fi trasat ‘inapoi’ pentru a identifica calea.

Raportarea caii



noduri vizitate din

0	8
1	2
2	-
3	1
4	2
5	3
6	7
7	1
8	2
9	8

Algorithm *Path*(w)

1. **if** $pred[w] \neq -1$
2. **then**
3. *Path*$(pred[w])$;
4. output w

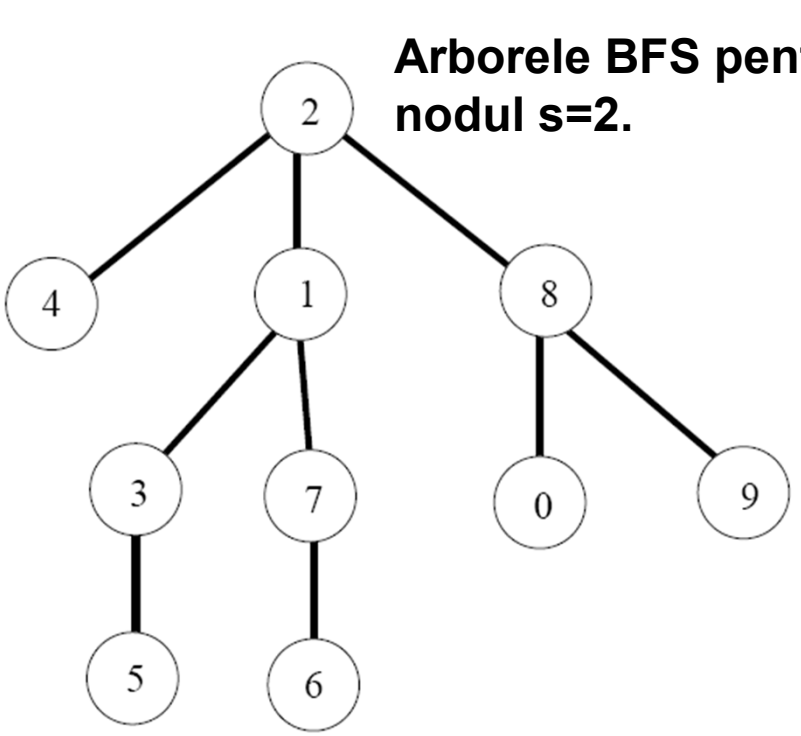
Exemple: cale de la **s** la **v**:
Path(0) →
Path(6) →
Path(1) →

Calea identificata este cea mai scurta de la s la v (numar minim de arce).

Arborele BFS

- Calea gasita de BFS apare de multe ori ca un arbore cu radacina (i.e. arbore BFS) cu nodul de start = radacina arborelui

- Terminologie:
  - Muchie de arbore: muchie ce duce un vf nedescoperit
  - Muchie de revenire: o muchie ce conduce la un vf deja vizitat (i.e. indica prezenta unui ciclu)



Arborele BFS pentru nodul s=2.

Modul de inregistrare a distantei celei mai scurte

Algorithm *BFS*(s)

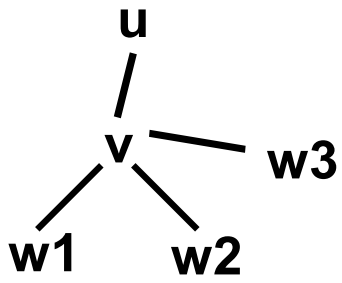
```
1.   for each vertex v
2.       do flag(v) := false;   d(v) = ∞;
3.          pred[v] := −1;
4.   Q = empty queue;
5.   flag[s] := true;          d(s) = 0;
6.   enqueue(Q, s);
7.   while Q is not empty
8.      do v := dequeue(Q);
9.         for each w adjacent to v
10.           do if flag[w] = false
11.                 then flag[w] := true;
12.                      pred[w] := v;
13.                d(w)=d(v)+1; enqueue(Q, w)
```

Traversarea in adancime a unui graf (DFS)

- **DFS** → alta varianta populara de cautare intr'un graf
- Semnificatie: "intai se viziteaza copiii"
- In cazul grafurilor neorientate, traversarea in adancime este o metoda eficienta pentru:
  - gasirea unei cai intre doua varfuri
  - determinarea faptului ca un graf este conex
  - determinarea arborelui de acoperire a unui graf conex
- DFS poate oferi informatii pe care BFS nu este capabil sa le ofere: poate preciza daca a fost (sau nu) intalnit un ciclu
- Rezolva probleme de tipul:
  - Gasirea drumului intre 2 vf (daca exista)
  - Gasirea unui ciclu

Algoritmul DFS (1)

- DFS va vizita vecinii intr'un *pattern* recursiv.
  - Oricand se viziteaza nodul **v** pornind din **u**, vor fi vizitati (recursiv) toti vecinii nevizitati ai lui **v**, dupa care se revine (*backtrack*) la **u**.
  - Nota: este posibil ca w2 sa nu fie vizitat atunci cand se viziteaza (recursiv) w1, dar va fi vizitat in momentul in care se revine din apelul recursiv.
  - In traversare, functia DFS este apelata o singura data pentru fiecare varf, a.i. fiecare muchie este examinata de 2 ori (cate o data pentru fiecare extremitate).
  - Complexitatea algoritmului DFS este **O(n+m)** .



Algoritmul DFS (2)

- **Algoritmul DFS** poate fi implementat prin intermediul unei structuri de tip stiva
- **Algoritmul DFS** merge “oricat de departe” din punctul de start si se intoarce doar daca a gasit un punct terminus
- **Algoritmul DFS** este utilizat in simularea de jocuri
- **Observatie**: DFS a fost studiat in sec XIX de matematicianul Francez **Ch Tremaux**

Algoritmul DFS (3)

Algorithm *DFS*(s)
1. **for** each vertex v
2. **do** $flag[v]$:= false;
3. *RDFS*(s);

Flag pe toate nodurile nevizitate

Algorithm *RDFS*(v)
1. $flag[v]$:= true;
2. **for** each neighbor w of v
3. **do if** $flag[w]$ = false
4. **then** *RDFS*(w);

Flag pe sine insusi ca vizitat

Pentru vecini nevizitati,
call RDFS(w) recurs

Se poate inregistra calea utilizand pred[ ].

Algoritmul DFS (4)

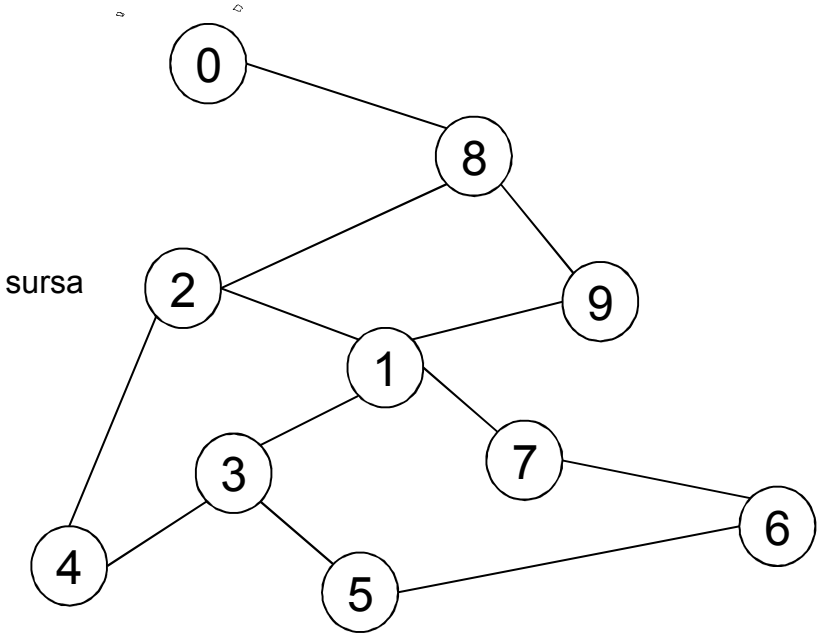
- **Algoritmul celor 3 reguli:**

Start dintr'un varf

- **R1:** a) du-te in orice varf adiacent care **nu** a fost deja vizitat

  b) pune'l in *stack* si marcheaza'l
- **R2:** daca R1 nu poate fi aplicata, atunci pop un varf din stiva
- **R3:** daca R1 si R2 nu pot fi aplicate, atunci **STOP**

Exemplu (1)



Lista de adiacente

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

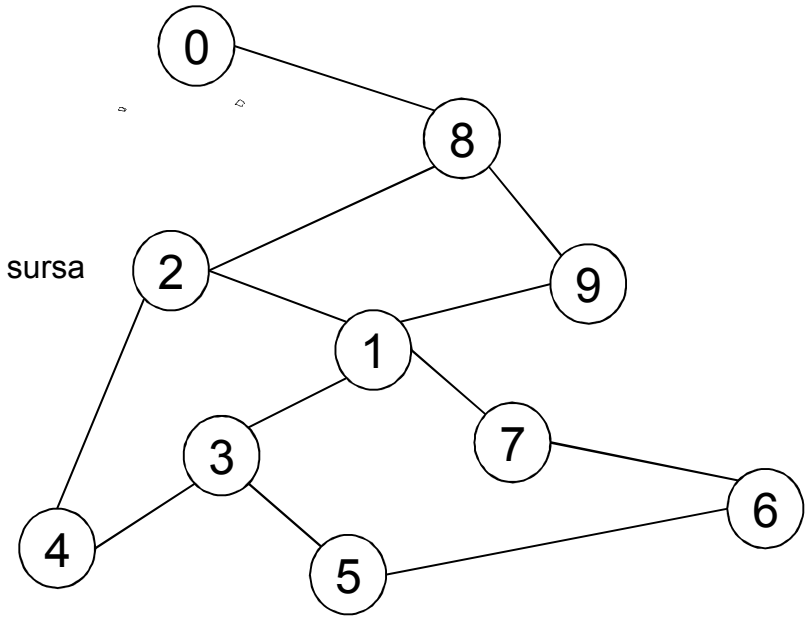
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Pred

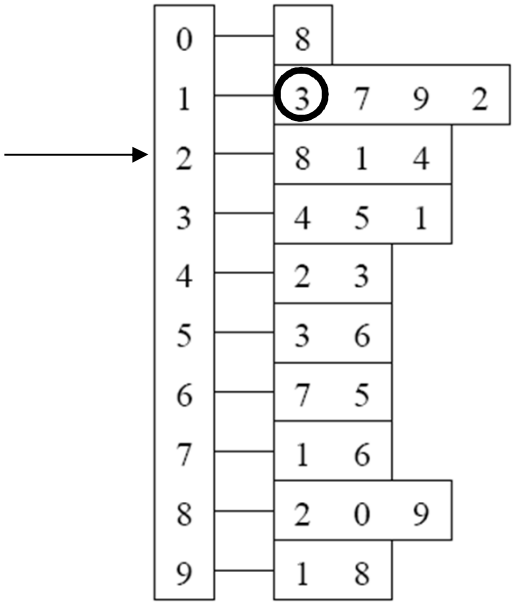
Initializare tabel vizitat (tot cu False)

Initializare Pred cu -1

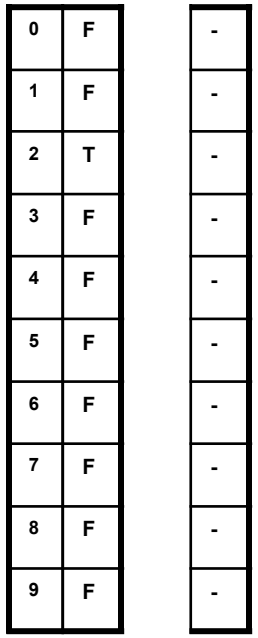
Exemplu (2)



Lista adiacenta



Tabel vizitat (T/F)



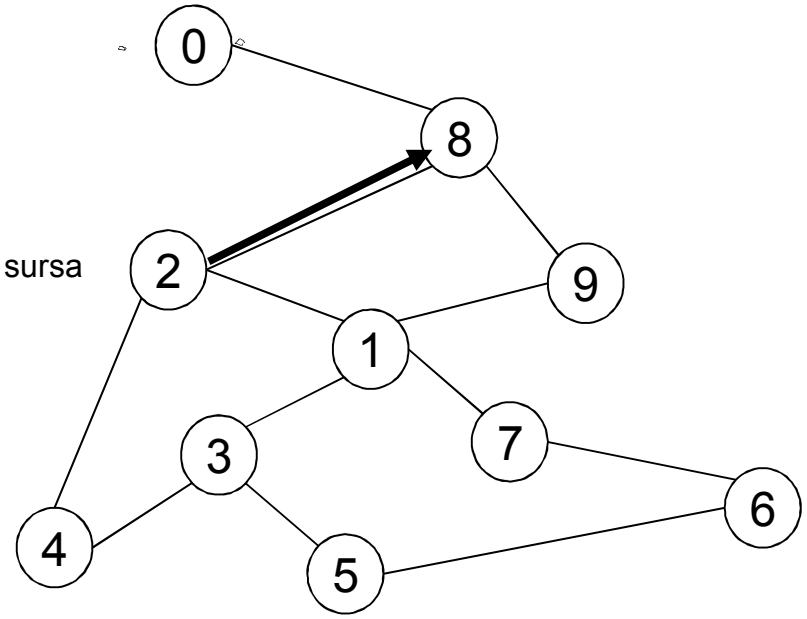
Pred

Se marcheaza 2
ca fiind vizitat

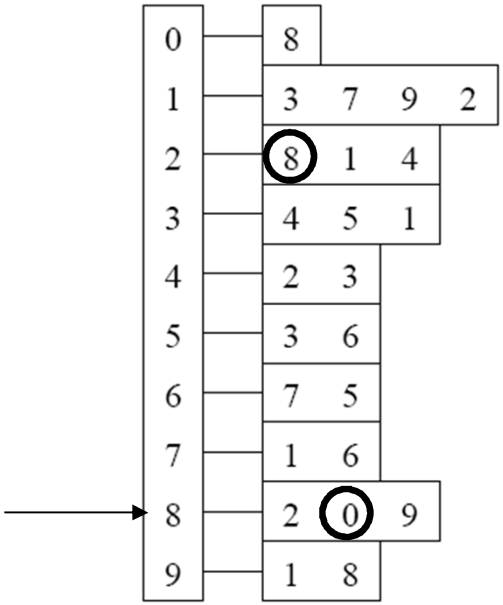
RDFS(2)

Se viziteaza RDFS(8)

Exemplu (3)



Lista adiacente



Tabel vizitat (T/F)

0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

Pred
-
-
-
-
-
-
-
-
2
-

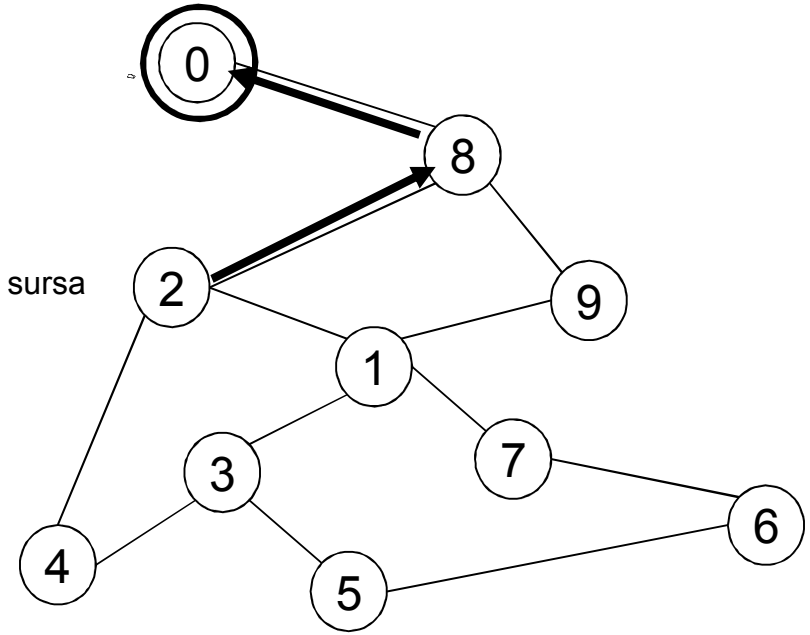
Marcare 8 - vizitat

marcare Pred[8]

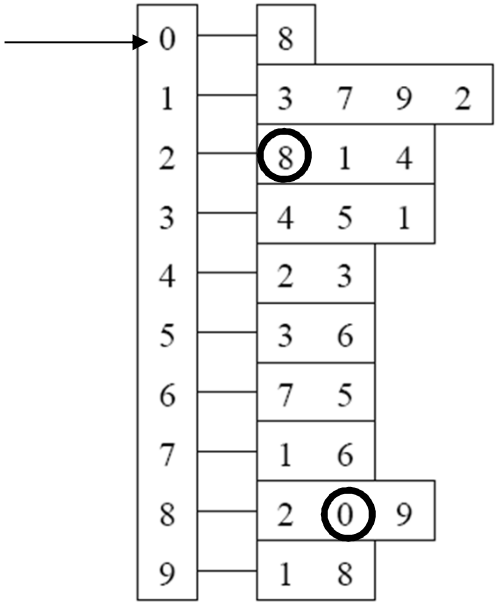
Apel recursiv

RDFS(2)
RDFS(8)
2 a fost deja vizitat,
→ se viziteaza RDFS(0)

Exemplu (4)



Lista adiacente



Tabel vizitat (T/F)

0	T
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	T
9	F

8
-
-
-
-
-
-
-
2
-

Pred

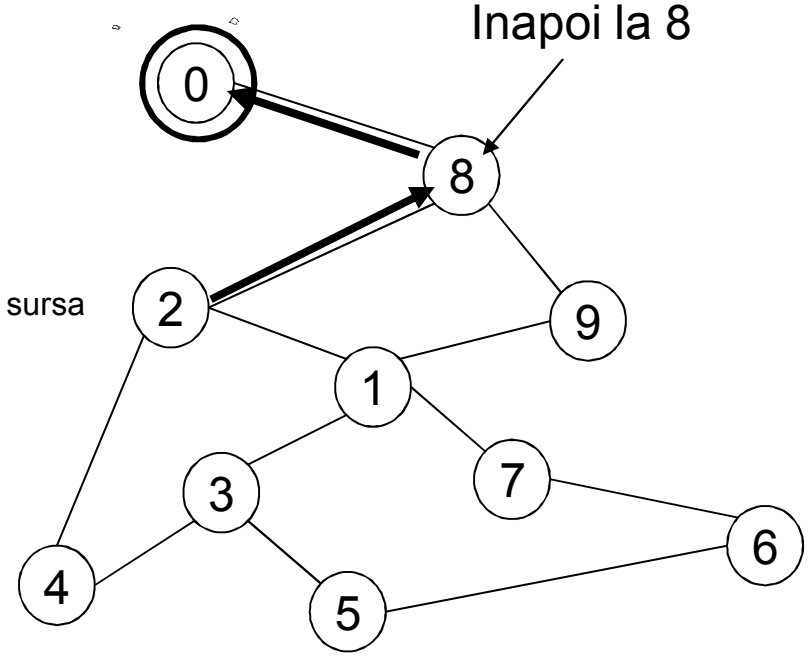
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(0) → nu sunt vecini nevizitati,
return pt apel RDFS(8)

Marcare 0 - vizitat

Marcare Pred[0]

Exemplu (5)



Lista de adiacente

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitat (T/F)

0	T	8
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

Pred

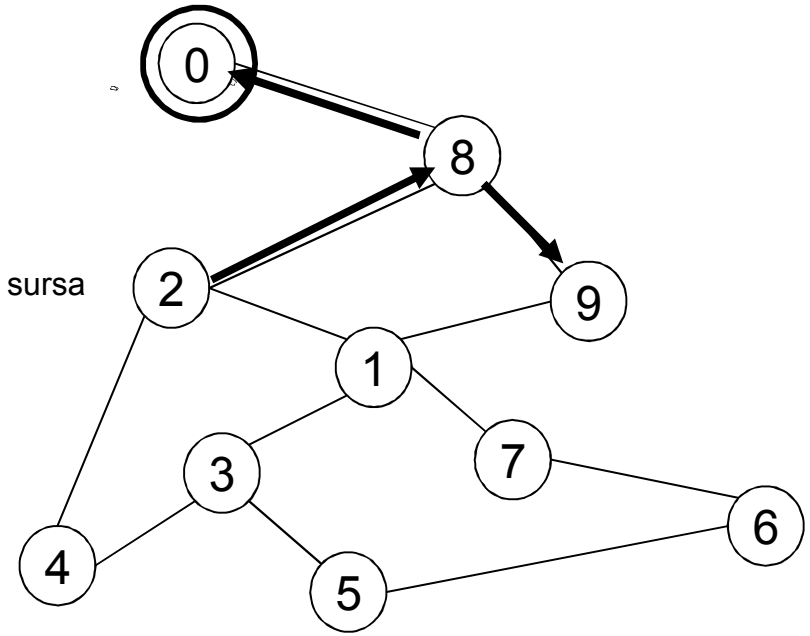
Apeluri recursive

RDFS(2)

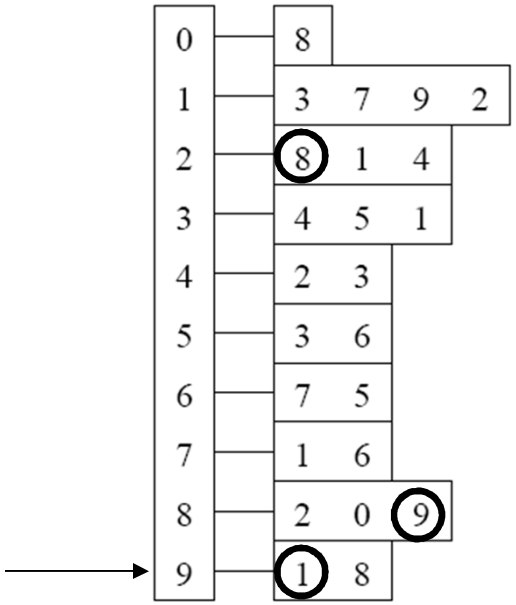
RDFS(8)

Acum viziteaza 9 → RDFS(9)

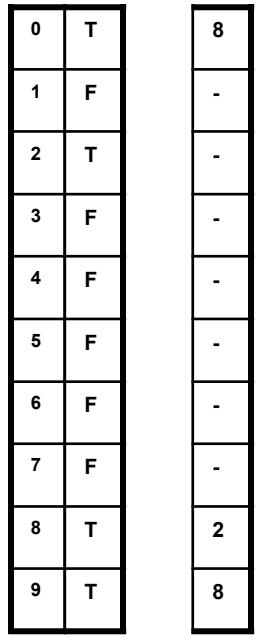
Exemplu (6)



Lista adiacente



Tabel vizitat (T/F)



Pred

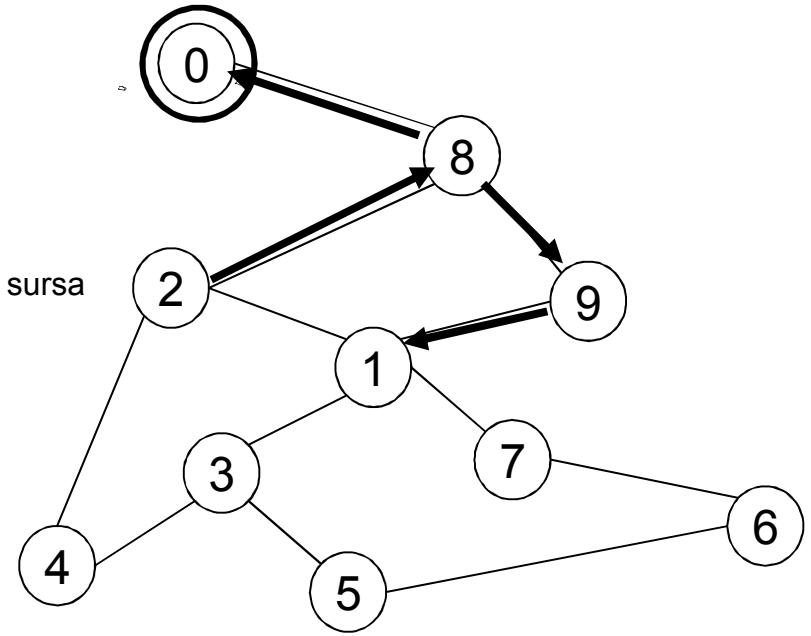
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
→ viziteaza 1, RDFS(1)

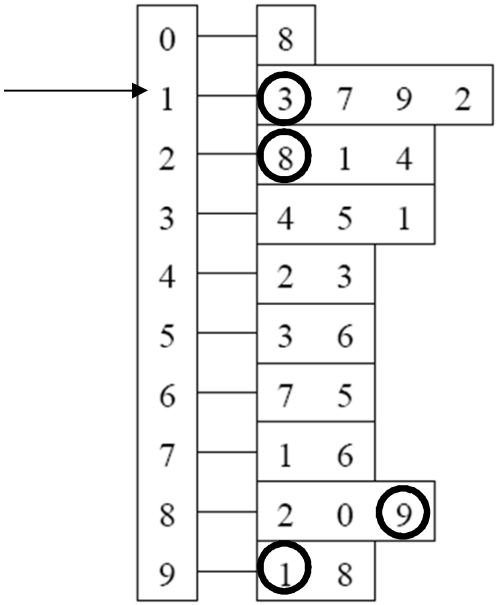
Marcare 9 - vizitat

Marcare Pred[9]

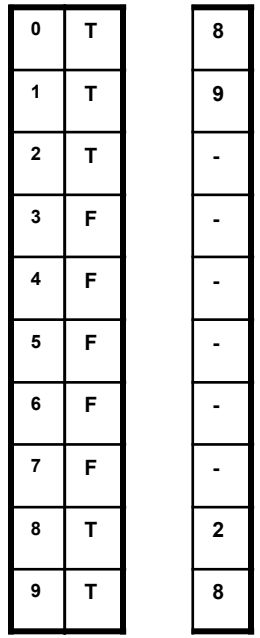
Exemplu (7)



Lista adiacente



Tabel vizitare (T/F)



Pred

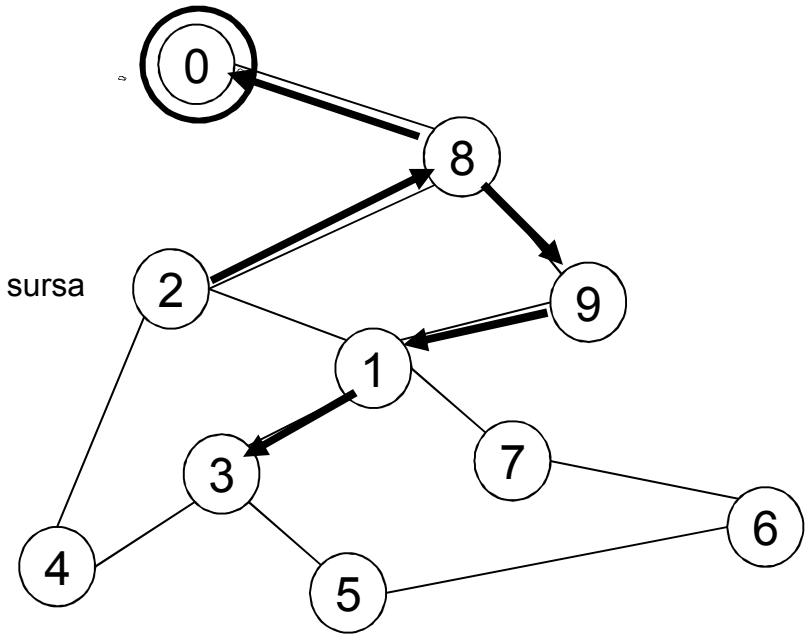
Apeluri recursiv

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
viziteaza RDFS(3)

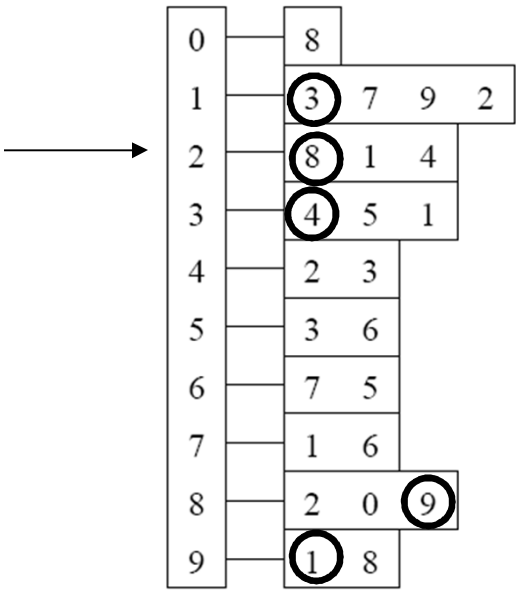
Marcare 1 - vizitat

Marcare Pred[1]

Exemplu (8)



Lista adiacente



Tabel vizitat (T/F)

0	T
1	T
2	T
3	T
4	F
5	F
6	F
7	F
8	T
9	T

Pred
8
9
-
1
-
-
-
-
2
8

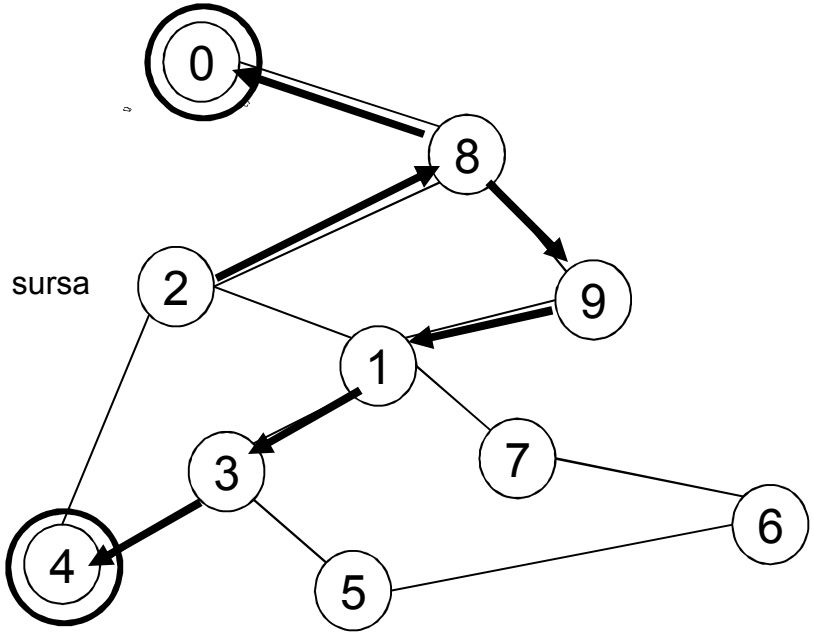
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)

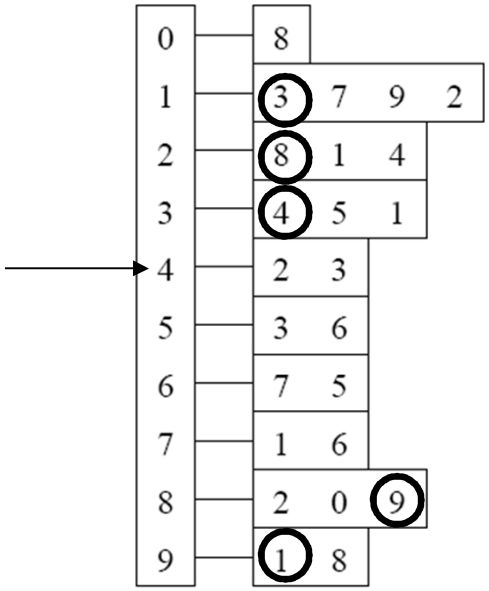
Marcare 3 - vizitat

Marcare Pred[3]

Exemplu (9)



Lista adiacente



Tabel vizitat (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	F
8	T
9	T

8
9
-
1
3
-
-
-
2
8

Pred

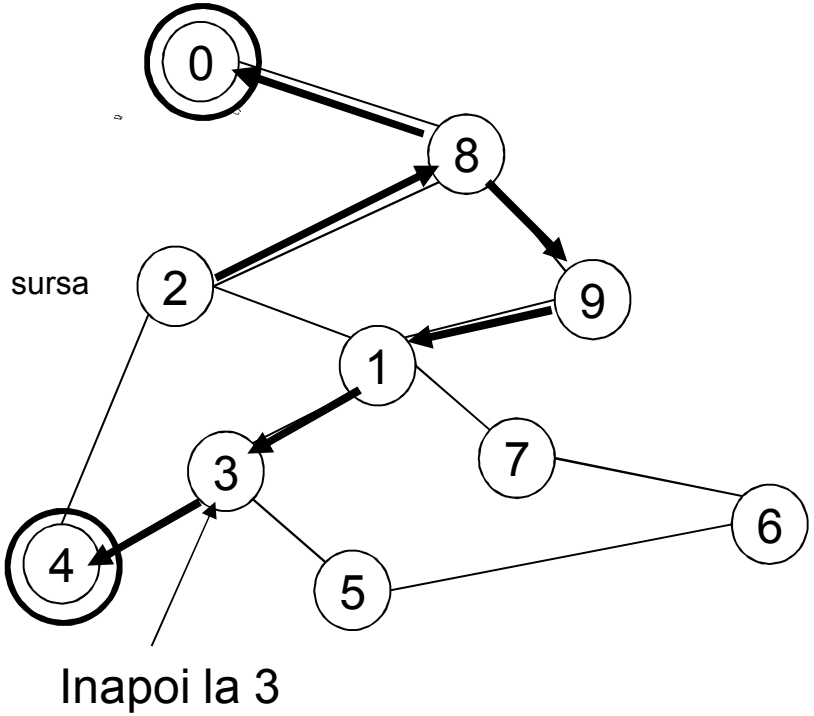
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(4) → STOP toti vecinii lui 4 au fost vizitati
revenire pentru apel RDFS(3)

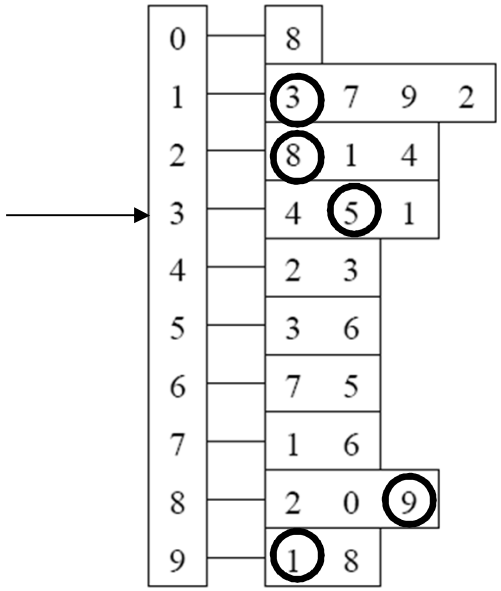
Marcare 4 - vizitat

Marcare Pred[4]

Exemplu (10)



Lista adiacenta



Tabel vizitare (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	F
8	T
9	T

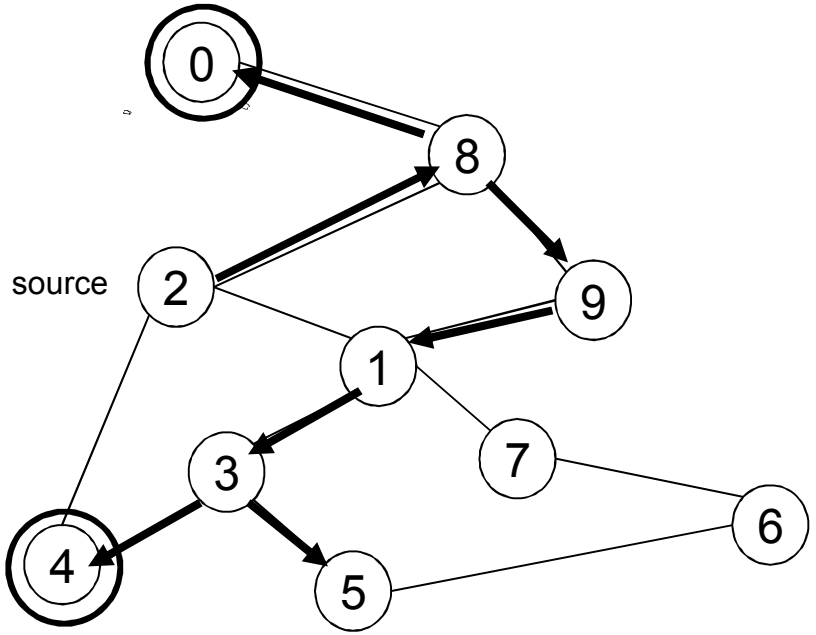
8
9
-
1
3
-
-
-
2
8

Pred

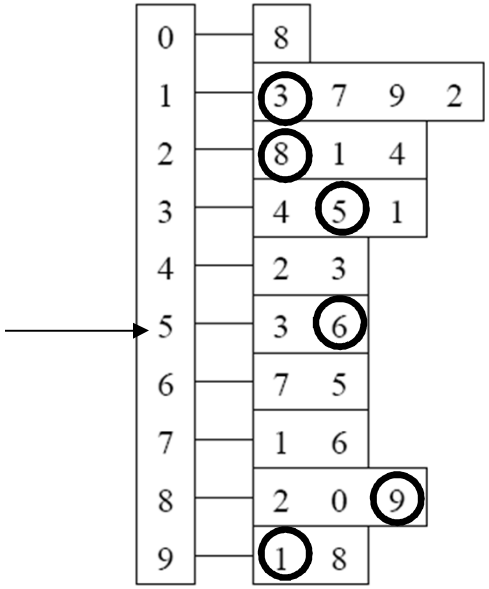
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
viziteaza 5 → RDFS(5)

Exemplu (11)



Lista adiacente



Tabel vizitare (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	T

Pred
8
9
-
1
3
3
-
-
2
8

Apeluri recursive

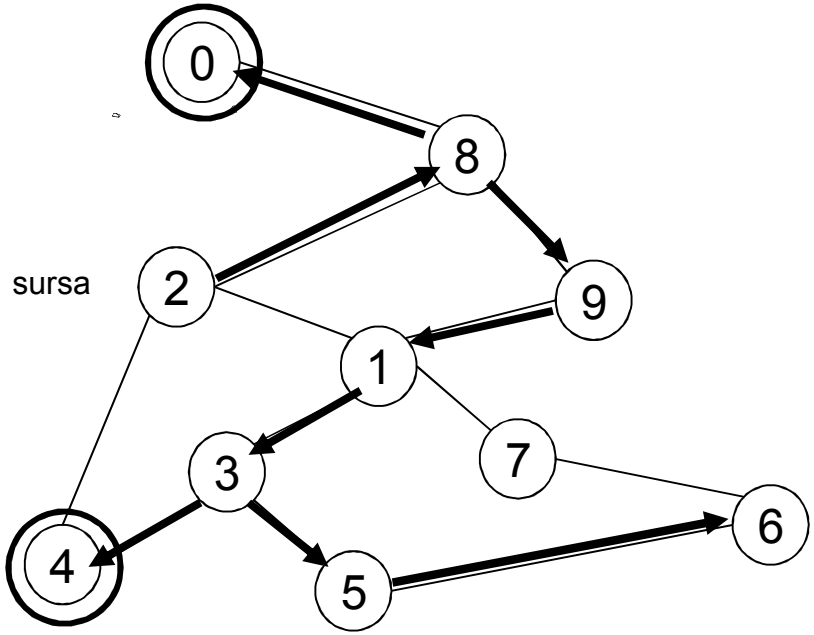
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)

Marcare 5 → vizitat

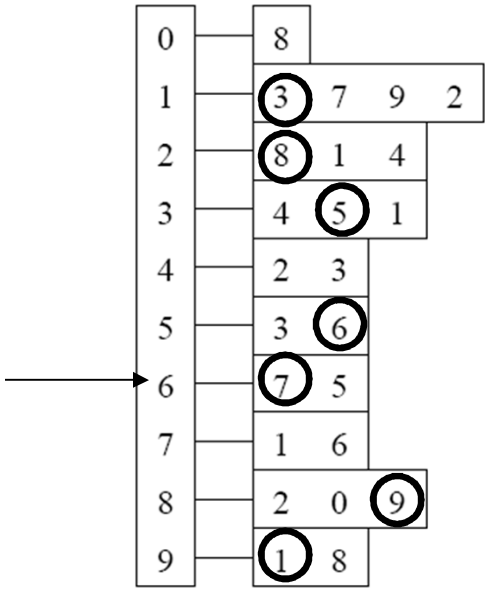
Marcare Pred[5]

3 a fost deja vizitat, se viziteaza 6 → RDFS(6)

Exemplu (12)



Lista adiacente



Tabel vizitare (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	F
8	T
9	T

Pred
8
9
-
1
3
3
5
-
2
8

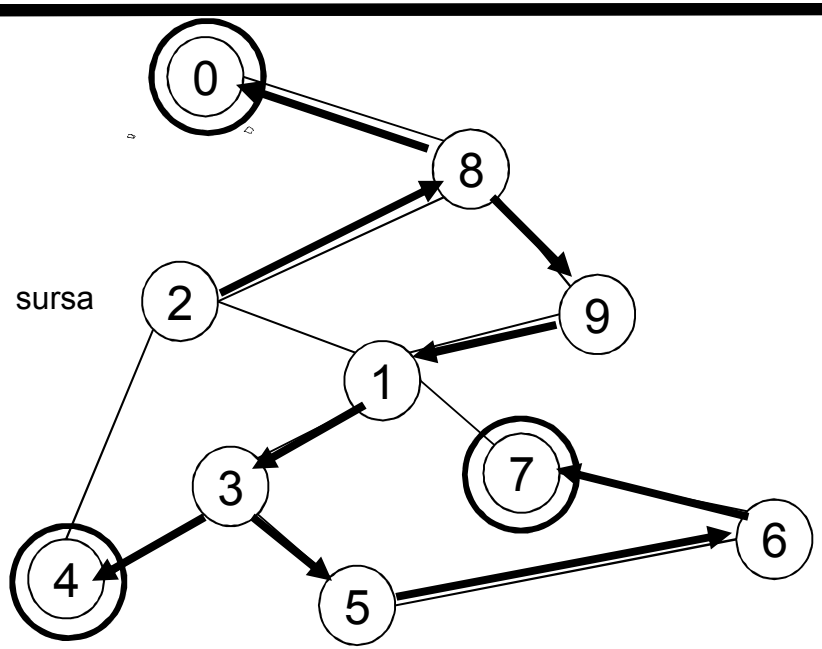
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
viziteaza 7 → RDFS(7)

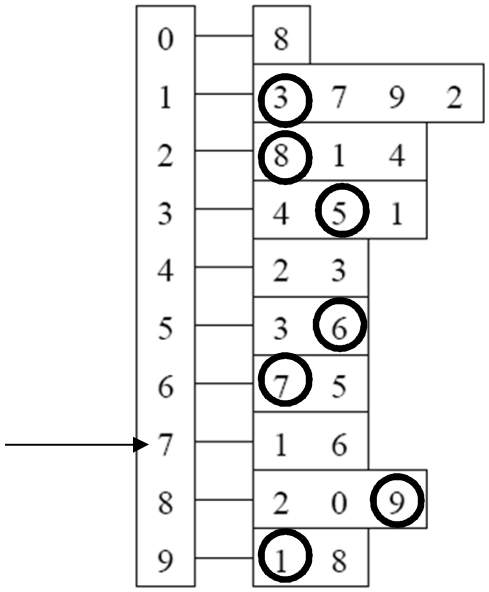
Marcare 6 - vizitat

Marcare Pred[6]

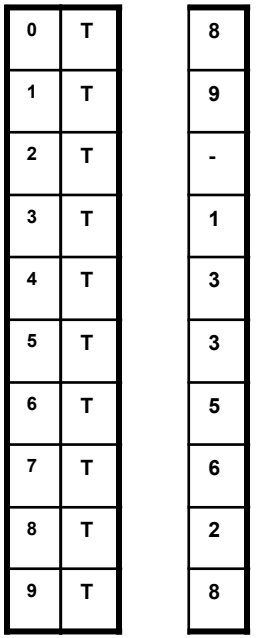
Exemplu (13)



Lista adiacente



Tabel vizitare (T/F)



Pred

RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1)

RDFS(3)

RDFS(5)

RDFS(6)

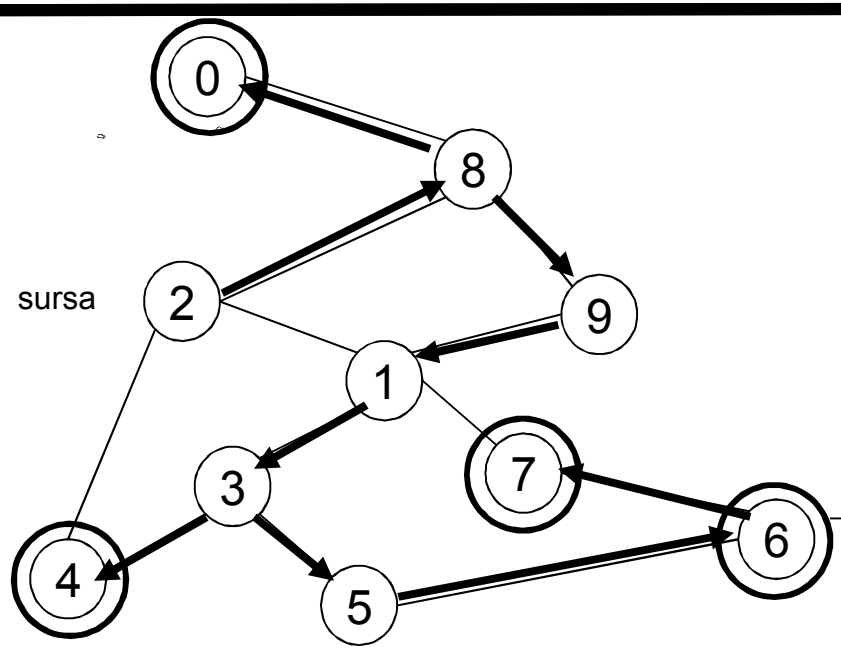
RDFS(7) -> Stop nu mai sunt vecini nevizitati

Apeluri recursive

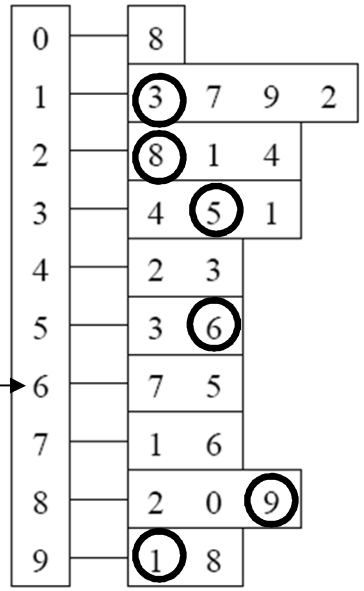
Marcare 7 - vizitat

Marcare Pred[7]

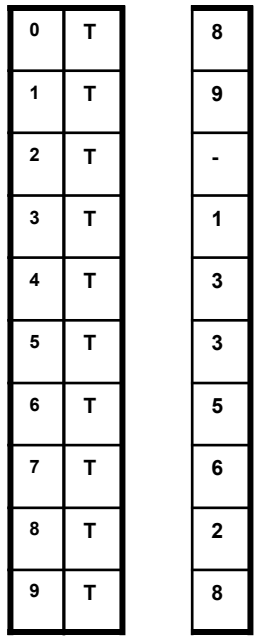
Exemplu (14)



Lista adiacente



Tabel vizitare (T/F)

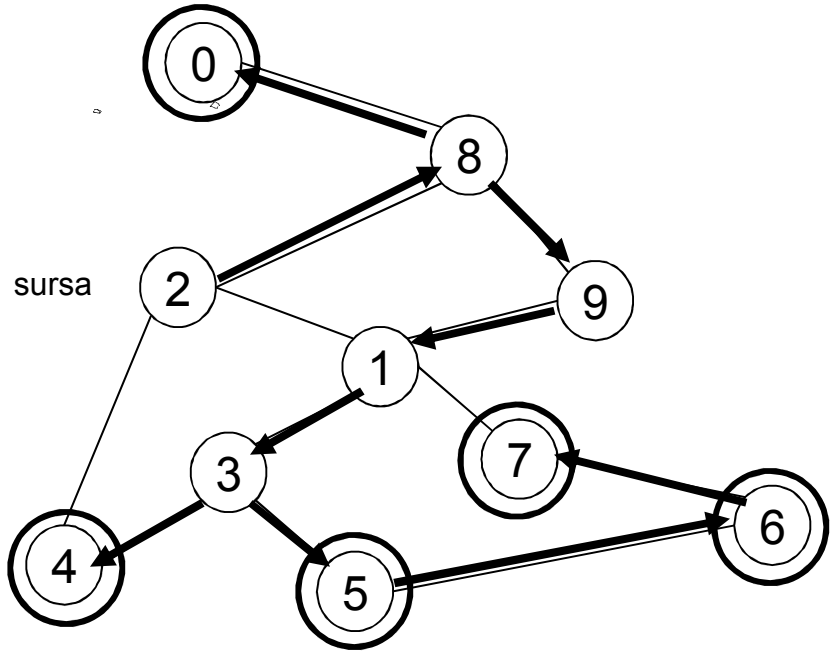


Pred

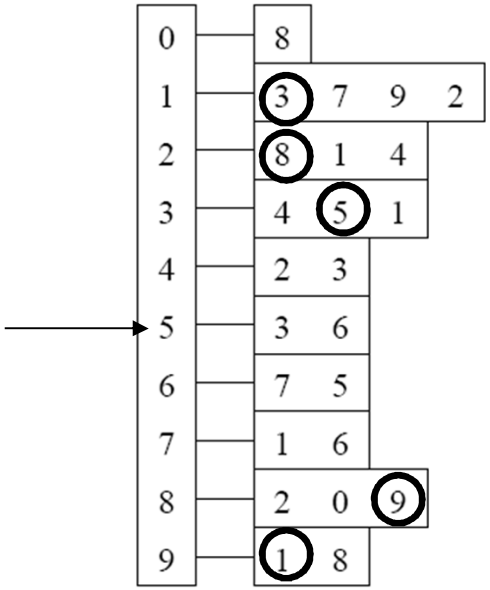
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop

Exemplu (15)



Lista adiacente



Tabel vizitare (T/F)

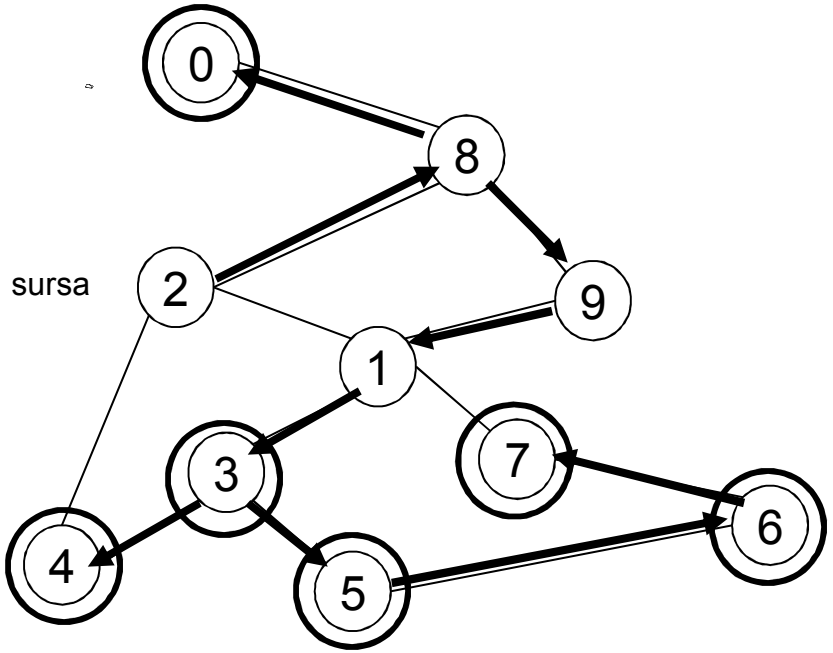
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

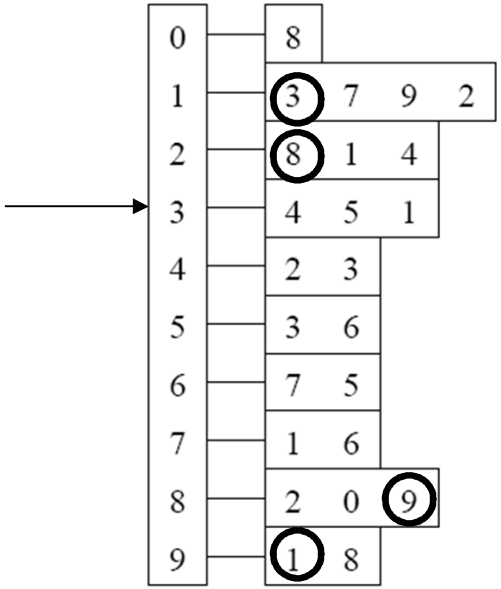
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) -> **Stop**

Exemplu (16)



Lista adiacente



Tabel vizitare (T/F)

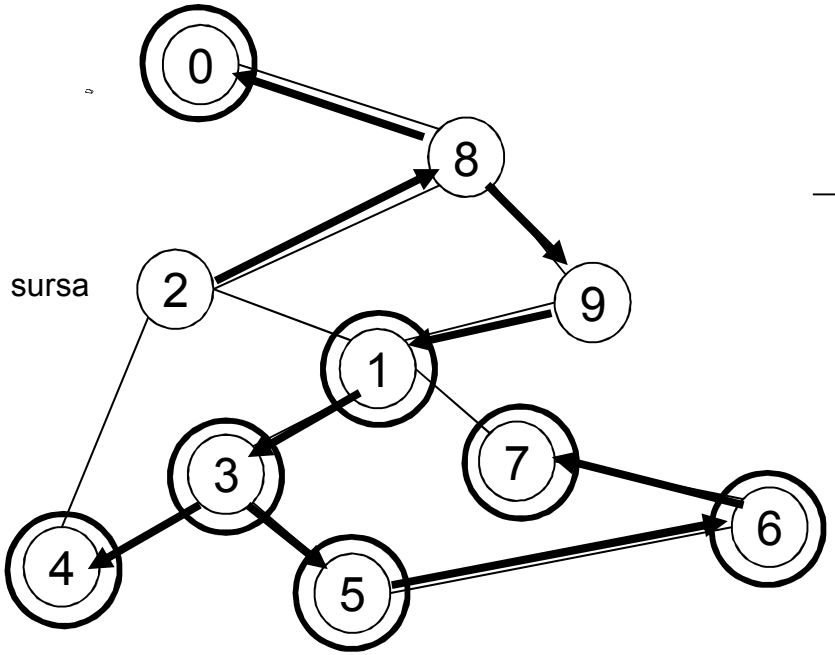
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

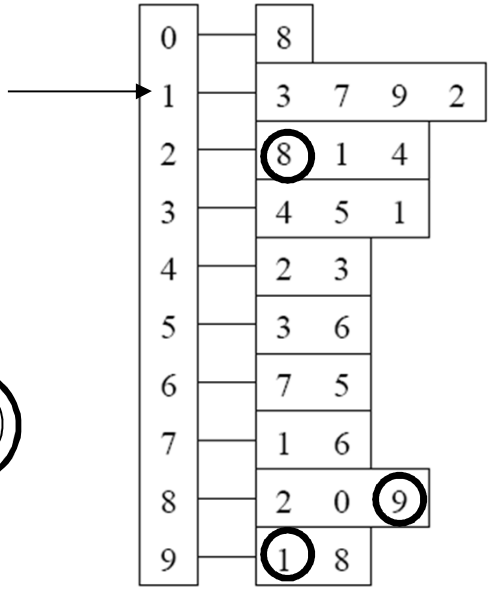
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) -> **Stop**

Exemplu (17)



Lista adiacente



Tabel vizitare (T/F)

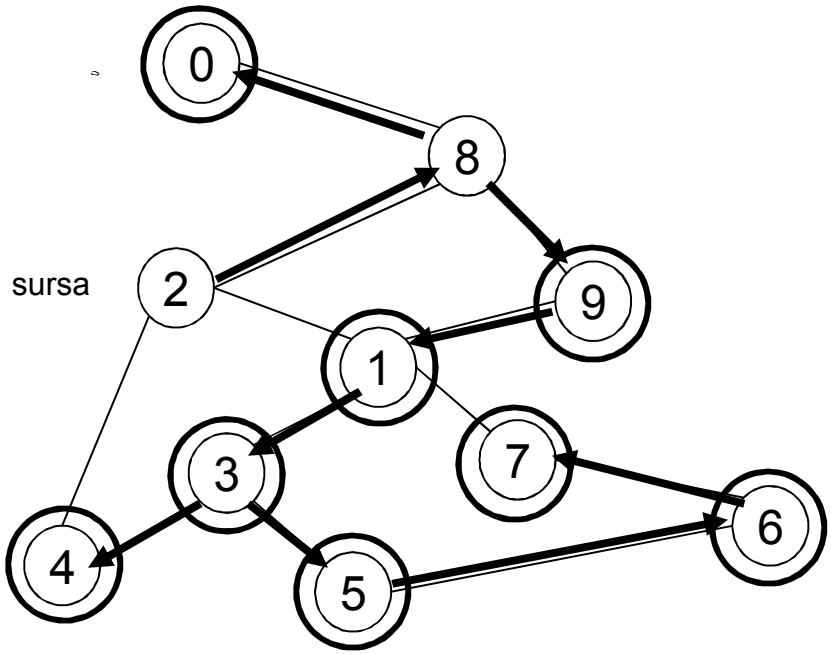
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

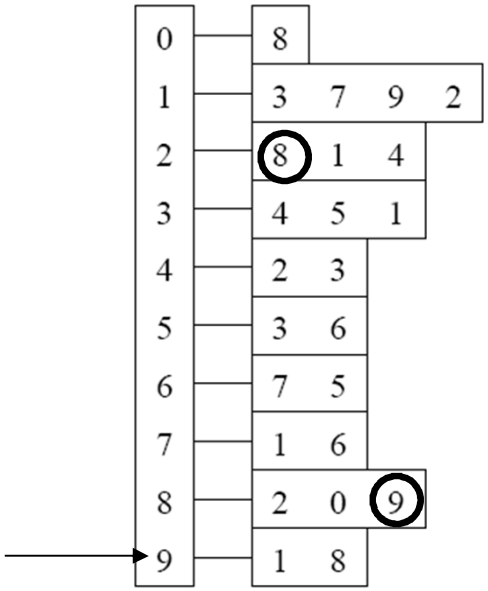
Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1) -> **Stop**

Exemplu (18)



Lista adiacente



Tabel vizitare (T/F)

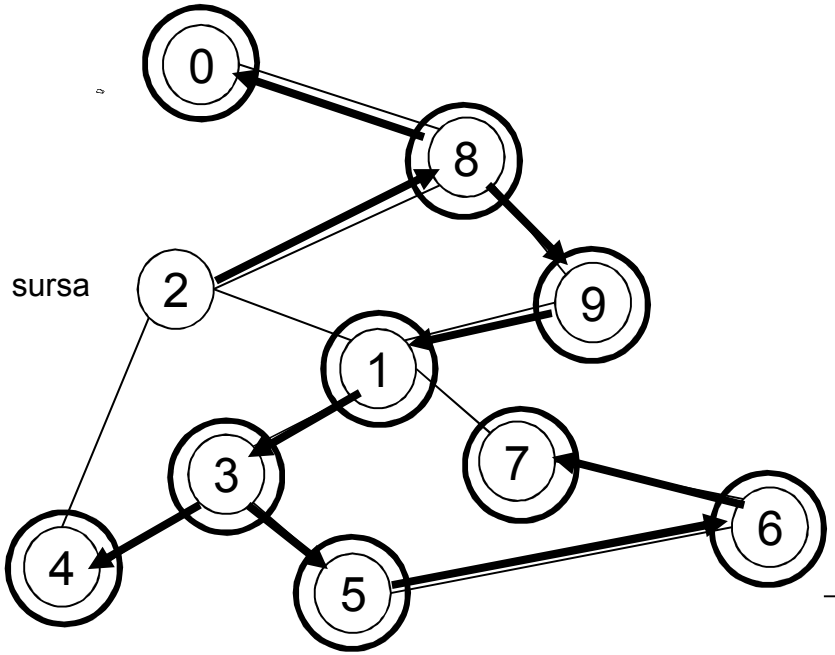
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Apeluri recursive

RDFS(2)
RDFS(8)
RDFS(9) -> **Stop**

Exemplu (19)



Lista adiacente

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitare (T/F)

0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

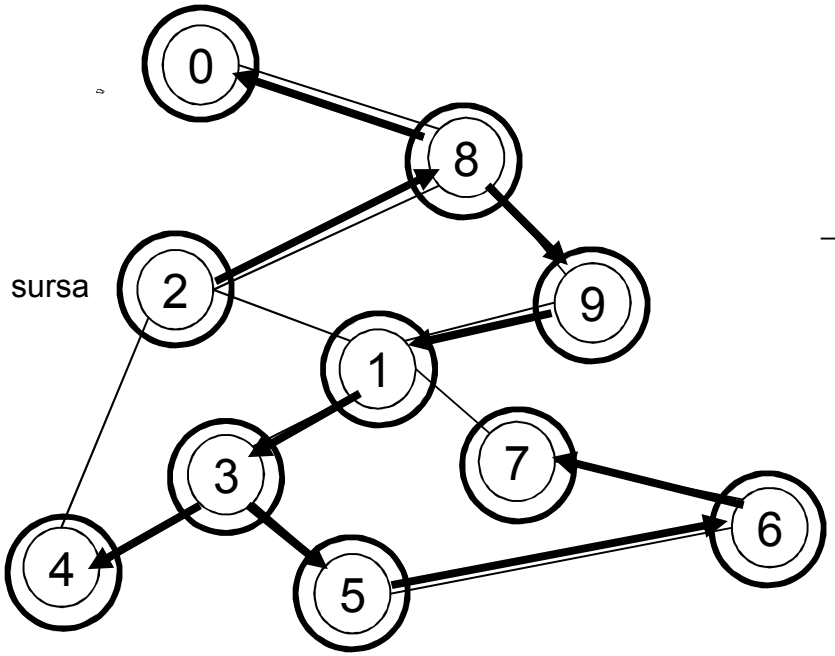
Pred

Apeluri recursive

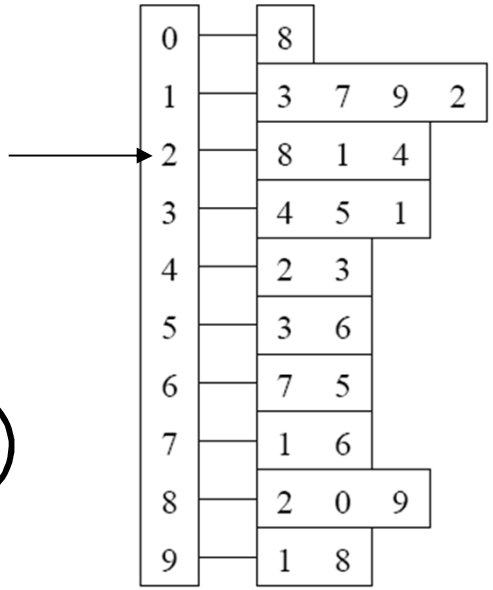
RDFS(2)

RDFS(8) -> **Stop**

Exemplu (20)



Lista adiacente



Tabel vizitare (T/F)

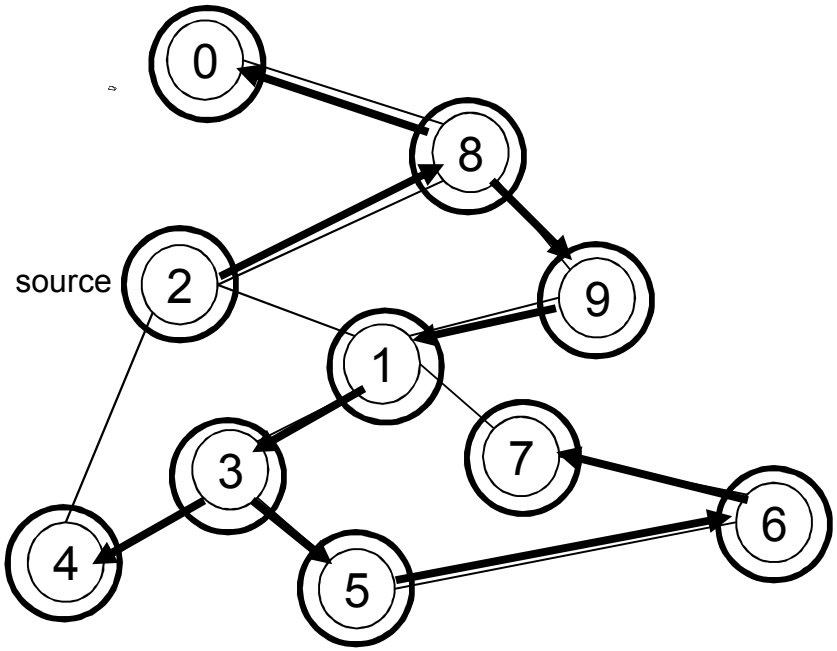
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Apeluri recursive

RDFS(2) -> **Stop**

Exemplu (21)



Lista adiacente

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Tabel vizitare (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Poate DFS sa gaseasca cai valide ? **DA**

Algorithm *Path*(w)

1. **if** $pred[w] \neq -1$
2. **then**
3. *Path*$(pred[w])$;
4. output w

Exemple….
Path(0) →
Path(6) →
Path(7) →

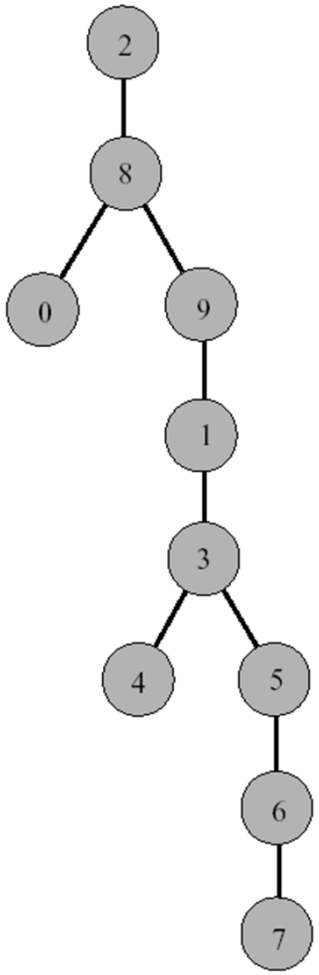
DFS – aspecte temporale (1)

- Un nod se viziteaza o singura data
- Trebuiesc examinate toate arcele (ce leaga nodurile)
  - Se stie ca $\Sigma_{\text{vertex } v}$ **degree(v) = 2m** unde m este numarul de arce
- Rezulta ca timpul de rulare pentru DFS este proportional cu numarul de arce si de noduri (similar cu BFS)
  - **O(n + m)**
- In final se poate scrie::
  - **O(|v|+|e|)** **|v| = numar noduri (n)**
    |e| = numar arce (m)

DFS – aspecte temporale (2)

- Setarea unui label pentru un vf / arc: **O(1)**
- Fiecare vf este marcat de doua ori : ne-explorat si vizitat
- Fiecare muchie este marcata de doua ori: ne-explorat si descoperit (sau *back*)
- Implementarea DFS cu matrice de adiacenta: **O(n^2)**

Arbore DFS



Arborele DFS.
Este mul mai “precis” decat un arbore BFS.

Pune in evidenta structura apelurilor recursive

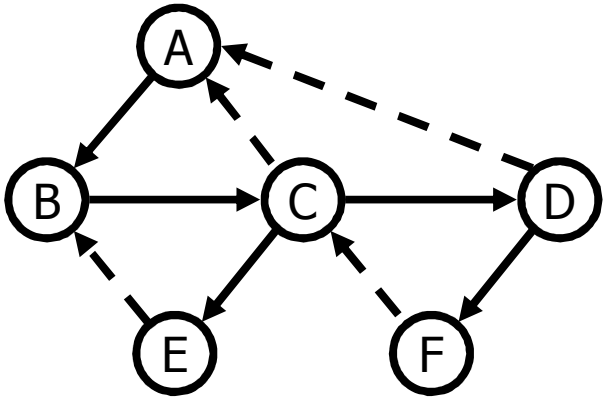
- Cand se viziteaza vecinul **w** al lui **v**, **w** va deveni “copilul” lui **v**
- de cate ori DFS revine de la un nod **v**, se “urca” in arbore de la **v** la “parintele” sau

Aplicatii DFS

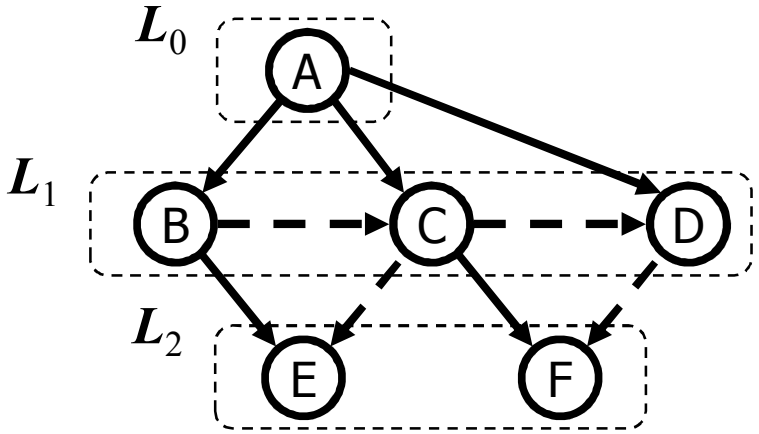
- Gasirea unui drum:
  - Se precizeaza p de start si cel de sfarsit
  - Se defineste o stiva pentru pastrarea drumului intre sursa si nodul curent
  - La atingerea varfului destinatie, drumul se va afla stocat in stiva
- Identificarea unui ciclu
  - Se precizeaza p de start si cel de sfarsit
  - Se defineste o stiva pentru pastrarea drumului intre sursa si nodul curent
  - Cand se intalneste o “muchie inapoi” (de ex (v,w)) se defineste ciclul ca fiind portiunea de stiva de la **top** la varful **w**.
- Determinarea faptului ca un graf este bi-conex)
- Determinarea faptului ca un graf este tare conex

DFS *vs.* BFS

Aplicatii	DFS	BFS
Conectare componente, cale/cai, cicli	√	√
Calea cea mai scurta		√
Componente bi-conexate	√	



DFS



BFS

Algoritmul Prim (1)

(sursa: Calin Jebelean, Algoritmul lui Prim)

- **AP** returneaza un arbore de acoperire (care poate sa nu fie unic) pentru un graf ponderat (graf in care fiecare arc are asociat un cost)
- Un arbore de acoperire pentru un graf este un subgraf alcatuit din toate nodurile grafului initial dar nu din toate arcele, ci doar din atatea arce cat sa nu apara cicluri
- Pentru un graf conex pot fi gasiti mai multi arbori de acoperire, in functie de arcele care sunt alese pentru a forma arborele
- Costul total al arborelui de acoperire este dat de suma costurilor arcelor alese, deci vom avea arbori “mai scumpi” si arbori “mai ieftini”
- Algoritmul lui Prim gaseste arborele de acoperire cel mai ieftin pentru un graf conex ponderat, pe care-l vom numi “arborele de acoperire minim” (acesta poate sa nu fie unic)
- Daca graful nu este conex, el este alcatuit din subgrafuri (componente) conexe
- In cazul unui astfel de graf, algoritmul lui Prim gaseste cate un arbore de acoperire minim pentru fiecare componenta conexa a grafului (neconex) dat, adica o “padure de arbori de acoperire minimi”

Algoritmul Prim (2)

(sursa: Calin Jebelean, Algoritmul lui Prim)

- Initial, toate nodurile grafului se considera nevizitate
- Se porneste de la ***un nod oarecare*** al grafului care se marcheaza ca vizitat
- In permanenta vor fi mentinute doua multimi:
  - Multimea U a nodurilor vizitate (initial, U va contine doar nodul de start)
  - Multimea N\U a nodurilor nevizitate (N este multimea tuturor nodurilor)
- La fiecare pas se alege acel nod din multimea N\U care este legat ***prin arc de cost minim*** de ***oricare*** din nodurile din multimea U
- Nodul ales va fi scos din multimea N\U si trecut in multimea U
- Algoritmul continua pana cand U = N

Algoritmul Dijkstra (1)

(sursa: Calin Jebelean, Algoritmul lui Dijkstra)

- Este un algoritm care calculeaza drumurile minime de la un nod al unui graf la toate celelalte noduri din graf
- Grafurile pe care poate lucra algoritmul lui Dijkstra sunt, in general, ponderate si orientate – arcele sunt orientate de la un nod la alt nod (nu se poate merge si invers) si au un anumit cost de care se va tine seama in aflarea drumului minim
- Daca graful este neponderat (arcele nu au costuri asociate) atunci drum minim intre doua noduri se considera drumul alcatuit din numar minim de arce
- Pentru a gasi drumul minim de la un nod X la un nod Y se poate aplica o cautare prin cuprindere pornind de la nodul X – prima aparitie a lui Y in coada algoritmului de cautare prin cuprindere presupune existenta unui drum cu numar minim de arce de la X la Y, care poate fi reconstituit
- Pe un astfel de graf se poate aplica si algoritmul lui Dijkstra, daca transformam in prealabil graful intr-unul ponderat, asociind fiecarui arc acelasi cost (de exemplu, costul 1)
- Drumul de cost minim intre doua noduri obtinut in urma aplicarii algoritmului lui Dijkstra va avea si numar minim de arce din moment ce toate arcele au acelasi cost

Algoritmul Dijkstra (2)

(sursa: Calin Jebelean, Algoritmul lui Dijkstra)

- Algoritmul lui Dijkstra functioneaza atat pe grafuri conexe cat si pe grafuri neconexe
- Un graf este conex daca din orice nod al sau se poate ajunge in orice alt nod
- In cazul grafurilor orientate, pentru ca intre doua noduri sa existe un drum in graf, nu este suficient sa existe o succesiune de arce intre cele doua noduri, ci arcele trebuie sa fie si orientate in sensul corespunzator
- Un drum intr-un graf orientat trebuie sa parcurga numai arce orientate identic, de la nodul sursa pana la nodul destinatie
- Daca nu exista nici un drum de la nodul de start la un alt nod al grafului atunci algoritmul lui Dijkstra va raporta existenta unui drum de lungime infinita intre ele – acest rezultat indica, de fapt, lipsa oricarui drum intre cele doua noduri

Algoritmul Dijkstra (3)

(sursa: Calin Jebelean, Algoritmul lui Dijkstra)

- Intrare:
  - Algoritmul porneste de la un graf orientat si ponderat cu N noduri
  - De asemenea, e nevoie de un nod de start apartinand grafului – acesta este nodul de la care se doreste aflarea drumurilor minime pana la celelalte noduri din graf
- Iesire:
  - Rezultatul algoritmului se prezinta sub forma unui tablou D cu N intrari, continand distantele minime de la nodul de start la toate celelalte noduri din graf
  - De asemenea, tot ca rezultat se poate obtine si arborele drumurilor minime (in cazul in care ne intereseaza nu numai lungimile minime ale drumurilor, ci si drumurile propriu-zise) – acesta este un arbore generalizat care se va obtine sub forma unui tablou T cu N intrari (implementarea cu indicatori spre parinte)

Algoritmul Dijkstra (4)

(sursa: Calin Jebelean, Algoritmul lui Dijkstra)

- Fie X nodul de start – acesta se marcheaza ca vizitat
- Ideea gasirii drumului minim de la X la un un alt nod este cautarea treptata: se presupune ca drumul minim este alcatuit dintr-un singur arc (arcul direct intre X si nodul tinta, care poate sa nu existe, costul sau fiind infinit in acest caz), apoi se cauta drumuri mai scurte alcatuite din 2 arce, apoi din 3, etc. – ***un drum minim nu poate avea mai mult de N-1 arce***, deci algoritmul are N-1 pasi (al N-lea este banal)
- Dupa pasul k ($1 \leq k \leq N-1$), tabloul D va contine lungimile drumurilor minime de la nodul X la celelalte noduri, toate aceste drumuri fiind alcatuite din ***maxim k arce***
- Astfel, D[Y] = L dupa pasul k inseamna ca de la X la Y ***exista un drum minim de lungime L alcatuit din maxim k arce***
- Deci, dupa pasul k, au fost gasite numai drumurile minime alcatuite din maxim k arce – abia la finalul algoritmului (dupa pasul N-1) drumurile minime obtinute sunt definitive, ele fiind drumuri minime alcatuite din maxim N-1 arce

Algoritmul Dijkstra (5)

(sursa: Calin Jebelean, Algoritmul lui Dijkstra)

- La inceputul fiecarui pas k avem un set de k-1 noduri marcate (in cadrul pasilor precedenti) – nodurile marcate sunt cele pentru care se cunoaste drumul minim (initial, doar nodul de start este marcat deoarece doar pentru el se cunoaste drumul minim)
- In cadrul pasului k trebuie executate 3 operatiuni:
  - Se gaseste acel nod Y ***nemarcat*** care are ***D[Y] minim*** (acesta este singurul dintre nodurile nemarcate pentru care se poate spune sigur ca drumul minim are lungimea D[Y]) – pentru celelalte noduri nemarcate valoarea corespunzatoare din tabloul D s-ar putea sa nu reprezinte lungimea drumului minim ci un drum minim intermediar (alcatuit din maxim k-1 arce) care poate fi imbunatatit in cadrul pasilor viitori ai algoritmului
  - Nodul Y se marcheaza ca vizitat
  - Pentru fiecare nod Z ***ramas nemarcat*** se executa urmatorul algoritm:

```
if D[Z] > D[Y] + Arc(Y , Z) then begin
        D[Z] := D[Y] + Arc(Y , Z);
        T[Z] := Y
end
```