Structuri de Date și Algoritmi
Grafuri neorientate

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

Anul Universitar 2022–2023

Conținutul cursului

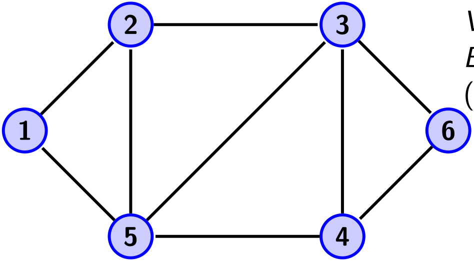
Definiție

- Un graf este o modalitate de a reprezenta **relații între obiecte**.

Un graf G este definit printr-o mulțime V de noduri sau vârfuri și o mulțime E de legături între aceste noduri numite arce sau muchii.
Din punct de vedere matematic, avem următoarea definiție:

$$G = (V, E), E \subseteq V \times V, E = \{(u, v) | u, v \in V\}$$

unde V este mulțimea de noduri și E este o relație binară peste V.



$V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (3, 5), (3, 6), (4, 5), (4, 6)\}$

Noțiuni elementare

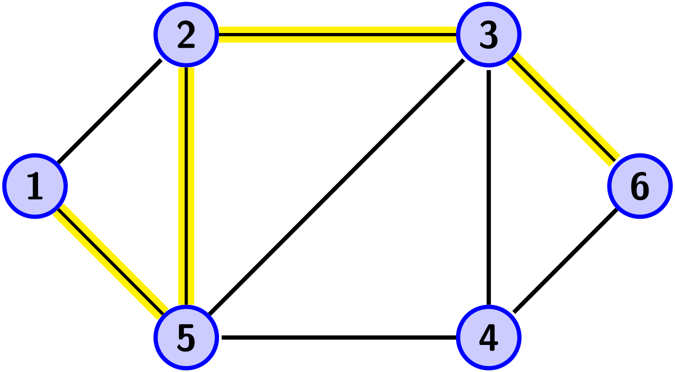
- În cazul grafurilor neorientate, perechile de vârfuri din mulțimea E sunt neordonate și sunt denumite **muchii**.
- Perechea neordonată formată din vârfurile u și v se notează (u, v), vârfurile u și v numindu-se **extremitățile** muchiei (u, v).
- Dacă există un **arc** sau o **muchie** cu extremitățile u și v, atunci vârfurile u și v sunt **adiacente**, fiecare extremitate a muchiei fiind considerată **incidentă** cu muchia respectivă.
- Gradul unui nod v, notat $grad(v)$, este numărul de muchii incidente cu acesta.

$$grad(v) = |\{e \in E | e = (v, x) \text{ } sau \text{ } e = (x, v), x \in V\}|$$

- Elementul $v \in V$ se numește **vârf izolat** dacă, pentru orice $e \in E$, v nu este incident cu e. Altfel spus, $grad(v) = 0$.
- Ordinul unui graf este numărul de noduri din graf $n = |V|$.
- Dimensiunea unui graf este egală cu numărul de muchii $m = |E|$.

Noțiuni elementare

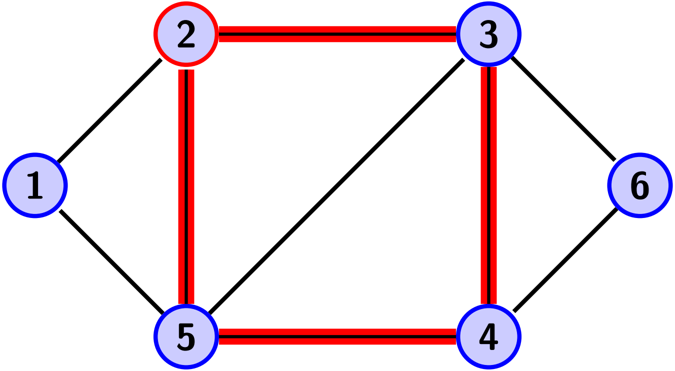
- O **cale** (*drum*) de lungime k între două noduri a și b este o succesiune de noduri $v_0, v_1, \ldots, v_k$ cu $v_0 = a$ și $v_k = b$ și $\{v_{i-1}, v_i\} \in E$ pentru $i = \overline{1, k}$.



Cale de lungime 4 între 1 și 6: $1 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$

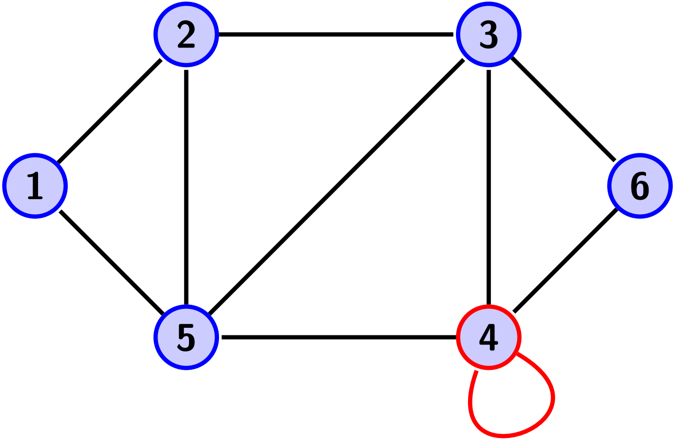
Noțiuni elementare

- Dacă există în G o cale de la a la b, atunci spunem că b este accesibil din a.
- O **cale simplă** are toate nodurile distincte.
- Un **ciclu** este o cale $v_0, v_1, \ldots, v_k$ în care $v_0 = v_k$.



Noțiuni elementare

- O **buclă** (a, a) este un ciclu de lungime 1.



Noțiuni elementare

- Un **ciclu simplu** are toate nodurile distincte (exceptând nodul de plecare) și nu conține bucle.
- Un **graf aciclic** nu conține cicluri.

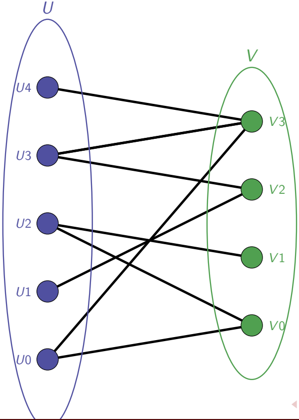
Un graf neorientat este considerat **graf conex** dacă pentru oricare două vârfuri există o cale care le unește.

- Componentele conexe ale unui graf neorientat sunt clasele de echivalență ale vârfurilor prin relația *este accesibil din*.
- Un graf neorientat este considerat conex dacă are o singură componentă conexă.
- Un graf conex aciclic este arbore liber.
- Un graf neorientat aciclic care nu este conex (are mai mult de o componentă conexă) se numește pădure.

Noțiuni elementare

Graf bipartit

Un **graf bipartit** este un graf în care mulțimea nodurilor V se partiționează $V = V_1 \cup V_2$, unde $V_1 \cap V_2 = \emptyset$ astfel încât pentru $\forall(u, v) \in E$ avem $u \in V_1$ și $v \in V_2$ sau $u \in V_2$ și $v \in V_1$.



Proprietăți

- Fie un graf neorientat $G = (V, E)$ cu m muchii ($|E| = m$) atunci este îndeplinită relația:

$$\sum_{v \in V} grad(v) = 2m$$

- Fie un graf neorientat G cu n noduri și m arce. Atunci este îndeplinită relația:

$$m \leq \frac{n \cdot (n-1)}{2}$$

Fie un graf G cu n noduri și m arce.

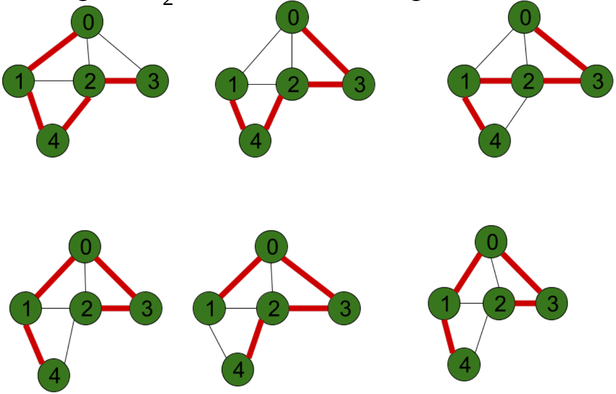
1. Dacă G este conex atunci $m \geq n - 1$.
2. Dacă G este arbore atunci $m = n - 1$.
3. Dacă G este o pădure atunci $m \leq n - 1$.

Noțiuni elementare

- O cale $v_0, v_1, \ldots, v_k$ a unui graf $G = (V, E)$ care conține fiecare muchie o dată și numai o dată se numește **cale euleriană**.
- Dacă $v_0 = v_k$ și calea este euleriană, atunci avem un **ciclu eulerian**.
- Un graf care conține un ciclu eulerian se numește **graf eulerian**.
- Un graf $G = (V, E)$, fără vârfuri izolate, este eulerian dacă și numai dacă este conex și gradele tututor vârfurilor sale sunt numere pare.
- Fie $G = (V, E)$ un graf. Se numește **ciclu hamiltonian** un ciclu care trece o singură dată prin toate nodurile grafului (cu excepția nodului de început).
- Un graf care conține un ciclu hamiltonian se numește **graf hamiltonian**.
- Fie $G = (V, E)$ un graf neorientat și o cale elementară care trece prin toate nodurile grafului: $v_1, v_2, \ldots v_n$. Dacă $grad(v_1) + grad(v_n) \geq n$, atunci graful este **hamiltonian**.

Noțiuni elementare

- Fie graful neorientat $G = (V, E)$ cu n noduri. Dacă pentru orice pereche de noduri neadiacente $v_i \neq v_j$ avem îndeplinită relația $grad(v_i) + grad(v_j) \geq n$ atunci graful este **hamiltonian**.
- Dacă $G = (V, E)$ este un graf cu n noduri și gradul oricărui nod este mai mare sau egal cu $\frac{n}{2}$, atunci G este un graf **hamiltonian**.



Definirea TAD-ului pentru un graf

Constructori

- Aceștia au ca rezultat un graf nou pentru care considerăm tipul `TGraph`.

1. Inițializarea grafului: `initGraph` : `Int` $\rightarrow$ `TGraph`
2. Adăugarea unui nod: `insertVetex` : `TGraph` $\times T \rightarrow$ `TGraph`
3. Adăugarea unei muchii: `insertEdge` : `TGraph` $\times T \times T \rightarrow$ `TGraph`
4. Ștergerea unui nod: `removeVertex` : `TGraph` $\times T \rightarrow$ `TGraph`
5. Ștergerea unei muchii: `removeEdge` : `TGraph` $\times T \times T \rightarrow$ `TGraph`

Funcții

- Operații care furnizează informații despre un graf.

1. Numărul de noduri din graf: `numV` : `TGraph` $\rightarrow$ `Int`
2. Numărul de muchii din graf: `numE` : `TGraph` $\rightarrow$ `Int`
3. Verificarea existenței unei muchii:
   `checkEdge` : `TGraph` $\times T \times T \rightarrow \{0, 1\}$
4. Gradul unui nod: `deg` : $T \times$ `TGraph` $\rightarrow$ `Int`

Modalități de reprezentare

Ce structură de date putem folosi pentru a reprezenta un graf?

Modalități de reprezentare

Ce structură de date putem folosi pentru a reprezenta un graf?

Putem folosi o matrice!

Modalități de reprezentare

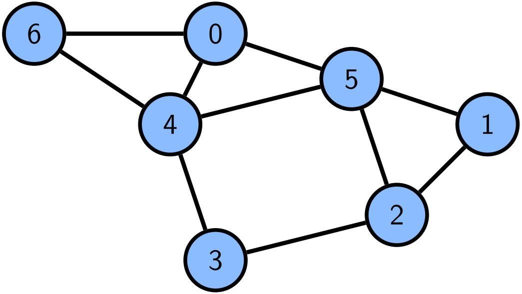
Ce structură de date putem folosi pentru a reprezenta un graf?

Putem folosi o matrice!

- Fie **G=(V,E)** un graf neorientat cu **n** vârfuri și **m** muchii. Matricea de adiacență, asociată grafului **G**, este o matrice pătratică de ordinul **n**, cu elementele definite astfel:

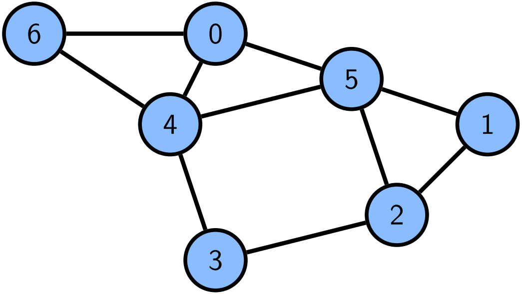
$$a_{i,j} = \begin{cases} 1 & \text{daca } (i,j) \in E \\ 0 & \text{daca } (i,j) \notin E \end{cases}$$

Matrice de adiacență



	0	1	2	3	4	5	6
0	0	0	0	0	1	1	1
1	0	0	1	0	0	1	0
2	0	1	0	1	0	1	0
3	0	0	1	0	1	0	0
4	1	0	0	1	0	1	1
5	1	1	1	0	1	0	0
6	1	0	0	0	1	0	0

Matrice de adiacență



	0	1	2	3	4	5	6
0	0	0	0	0	1	1	1
1	0	0	1	0	0	1	0
2	0	1	0	1	0	1	0
3	0	0	1	0	1	0	0
4	1	0	0	1	0	1	1
5	1	1	1	0	1	0	0
6	1	0	0	0	1	0	0

- Suma elementelor de pe linia **i** și respectiv suma elementelor de pe coloana **j** au ca rezultat gradul nodului **i**, respectiv **j**.
- Suma tuturor elementelor matricei de adiacență este suma gradelor tuturor nodurilor, adică dublul numărului de muchii.

Implementare folosind matrice de adiacență

Cum putem reprezenta structura?

Implementare folosind matrice de adiacență

Cum putem reprezenta structura?

```c
typedef struct Graph {
    int num_vertices;
    int** matrix;
} Graph;
```

Implementare folosind matrice de adiacență

Cum putem reprezenta structura?

```c
typedef struct Graph {
    int num_vertices;
    int** matrix;
} Graph;
```

Ce presupune operația de inițializare a grafului?

Implementare folosind matrice de adiacență

Trebuie să alocăm dinamic memoria și să inițializăm cu 0 toate elementele matricei.

Implementare folosind matrice de adiacență

Trebuie să alocăm dinamic memoria și să inițializăm cu 0 toate elementele matricei.

```c
void initGraph(Graph* g, int num_vertices) {
    int i;
    g->num_vertices = num_vertices;
    g->matrix = (int**) malloc(num_vertices * sizeof(int*));
    for (i = 0; i < num_vertices; i++) {
        g->matrix[i] = (int*) calloc(num_vertices, sizeof(int));
    }
}
```

Implementare folosind matrice de adiacență

Cum putem insera sau șterge o muchie?

Implementare folosind matrice de adiacență

Cum putem insera sau șterge o muchie?

Pornind de la muchia (u, v) va trebui să modificăm elementele `matrix[u][v]` și `matrix[v][u]`.

```c
void insertEdge(Graph* g, int vertex1, int vertex2) {
    if (vertex1 < 0 || vertex1 >= g->num_vertices || vertex2 < 0 || vertex2 >= g->num_vertices) {
        return;
    }
    g->matrix[vertex1][vertex2] = 1;
    g->matrix[vertex2][vertex1] = 1;
}
```

Cum putem insera sau șterge o muchie?

Pornind de la muchia (u, v) va trebui să modificăm elementele `matrix[u][v]` și `matrix[v][u]`.

```c
void removeEdge(Graph* g, int vertex1, int vertex2) {
    if (vertex1 < 0 || vertex1 >= g->num_vertices || vertex2 < 0 || vertex2 >= g->num_vertices) {
        return;
    }
    g->matrix[vertex1][vertex2] = 0;
    g->matrix[vertex2][vertex1] = 0;
}
```

Implementare folosind matrice de adiacență

Ce se întâmplă când vrem să inserăm un nod sau să ștergem un nod?

Implementare folosind matrice de adiacență

Ce se întâmplă când vrem să inserăm un nod sau să ștergem un nod?

Se modifică radical structura matricei de adiacență.

Implementare folosind matrice de adiacență

Ce se întâmplă când vrem să inserăm un nod sau să ștergem un nod?

Se modifică radical structura matricei de adiacență.

Ce se întâmplă cu performanțele acestor operații?

Implementare folosind matrice de adiacență

Ce se întâmplă când vrem să inserăm un nod sau să ștergem un nod?

Se modifică radical structura matricei de adiacență.

Ce se întâmplă cu performanțele acestor operații?

Aceste operații vor fi foarte ineficiente.

Modalități de reprezentare

Ce altă modalitate de reprezentare am putea utiliza?

Modalități de reprezentare

Ce altă modalitate de reprezentare am putea utiliza?

Putem reprezenta graful prin liste de adiacență!

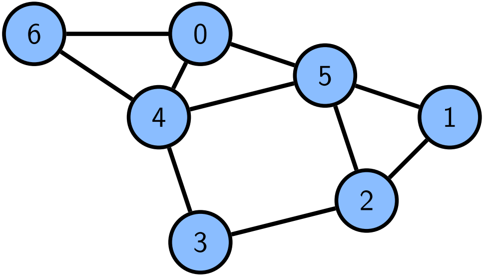
Modalități de reprezentare

Ce altă modalitate de reprezentare am putea utiliza?

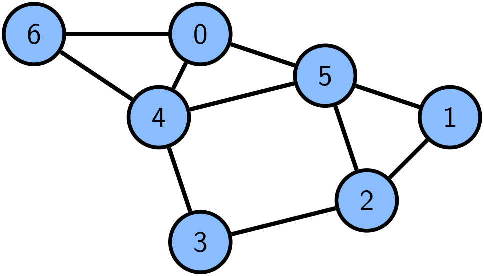
Putem reprezenta graful prin liste de adiacență!

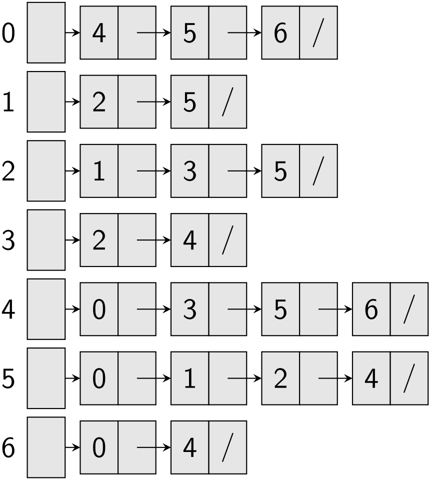
- Fie **G=(V,E)** un graf neorientat cu **n** vârfuri și **m** muchii. Reprezentarea grafului **G** prin **liste de adiacență** constă în:
  1. precizarea numărului de vârfuri și a numărului de muchii;
  2. pentru fiecare vârf i se precizează lista vecinilor săi, adică lista nodurilor adiacente cu nodul i.

Liste de adiacență



Liste de adiacență





Algoritmi de parcurgere

Parcurgerea unui graf presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informațiilor asociate vârfurilor. Exista doua metode fundamentale de parcurgere a grafurilor:

Algoritmi de parcurgere

Parcurgerea unui graf presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informațiilor asociate vârfurilor. Exista doua metode fundamentale de parcurgere a grafurilor:

- **Parcurgerea în lățime (BFS - Breadth First Search)**

Algoritmi de parcurgere

Parcurgerea unui graf presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informațiilor asociate vârfurilor. Exista doua metode fundamentale de parcurgere a grafurilor:

- **Parcurgerea în lățime (BFS - Breadth First Search)**
- **Parcurgerea în adâncime (DFS - Depth First Search)**

Algoritmi de parcurgere

Parcurgerea unui graf presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informațiilor asociate vârfurilor. Exista doua metode fundamentale de parcurgere a grafurilor:

- **Parcurgerea în lățime (BFS - Breadth First Search)**
- **Parcurgerea în adâncime (DFS - Depth First Search)**

Prin parcurgerea unui graf neorientat se înțelege examinarea nodurilor sale, plecând dintr-un vârf dat **i**, astfel încât fiecare nod accesibil din **i**, pe muchii adiacente două câte două, să fie atins o singură dată.

Algoritmi de parcurgere

Parcurgerea unui graf presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informațiilor asociate vârfurilor. Exista doua metode fundamentale de parcurgere a grafurilor:

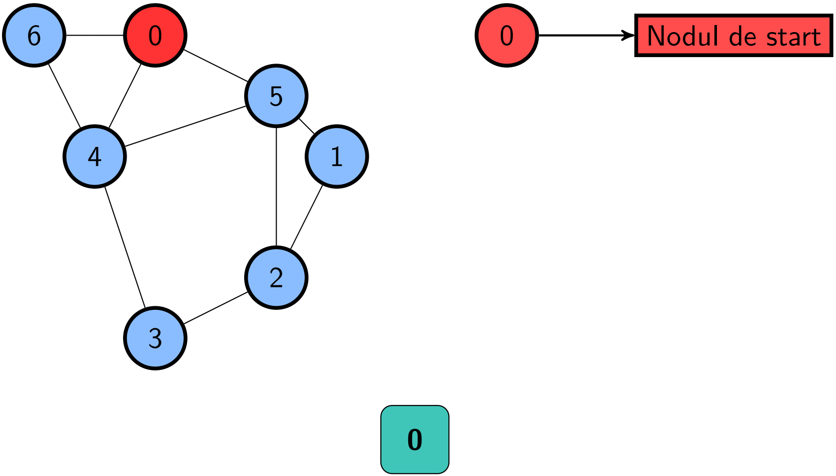
- **Parcurgerea în lățime (BFS - Breadth First Search)**
- **Parcurgerea în adâncime (DFS - Depth First Search)**

Prin parcurgerea unui graf neorientat se înțelege examinarea nodurilor sale, plecând dintr-un vârf dat **i**, astfel încât fiecare nod accesibil din **i**, pe muchii adiacente două câte două, să fie atins o singură dată.
Graful este structura neliniară de organizare a datelor, iar rolul traversării sale poate fi și determinarea unei aranjării liniare a nodurilor în vederea trecerii de la unul la altul.

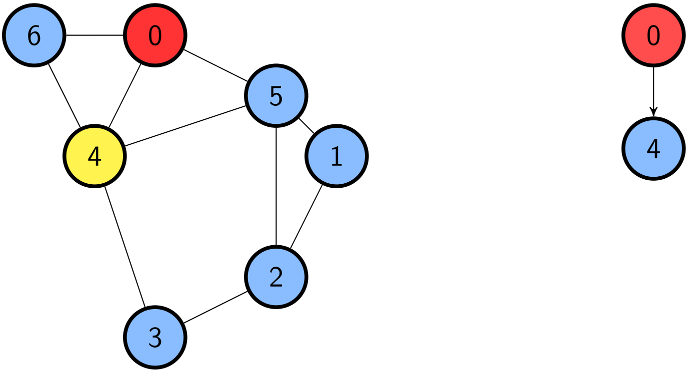
Parcurgerea în lățime

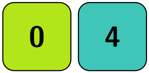
Parcurgerea în lățime este unul dintre cei mai simpli și, poate, folositori algoritmi de căutare în grafuri. Se obțin drumurile dintr-un nod sursa către orice nod din graf astfel:
Fiind date un graf **G=(V,E)** și un nod sursă **s**, această parcurgere va permite explorarea sistematica în **G** și descoperirea fiecărui nod, plecând din **s**. Totodată, se poate calcula și distanta de la **s** la fiecare nod ce poate fi vizitat. În felul acesta, se contruiește un arbore ,,pe lățime" cu rădăcina în **s** ce conține toate nodurile ce pot fi vizitate. Pentru orice nod **v**, ce poate fi vizitat plecând din **s**, drumul de la rădăcină la acest nod, drum refăcut din arborele ,,pe lățime", este cel mai scurt de la **s** la **v**, în sensul ca acest drum conține cele mai puține muchii.

Parcurgerea în lățime

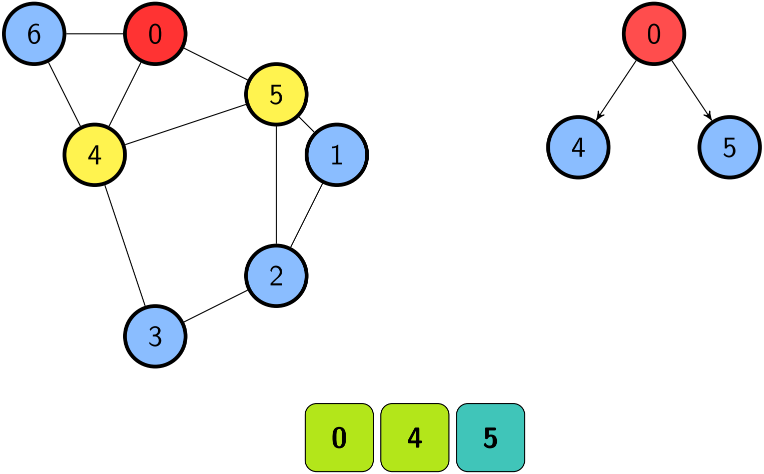


Parcurgerea în lățime

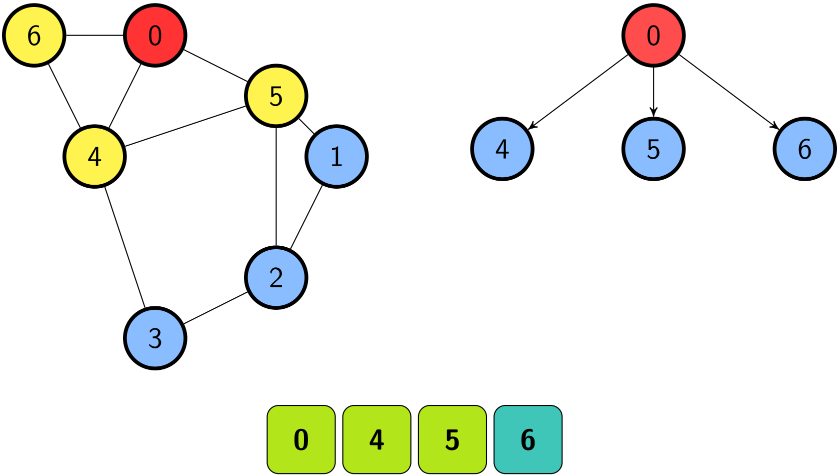




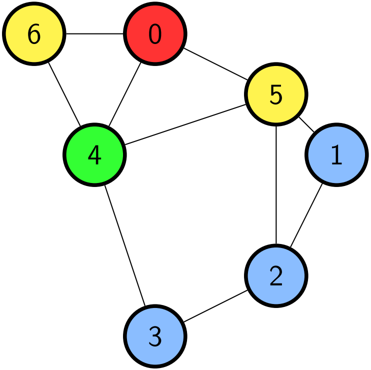
Parcurgerea în lățime

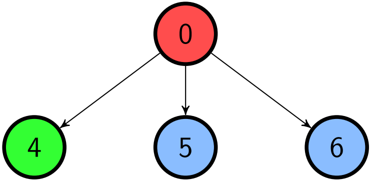


Parcurgerea în lățime



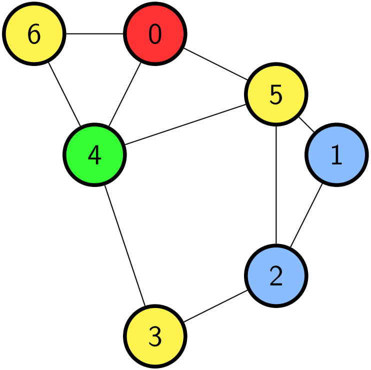
Parcurgerea în lățime

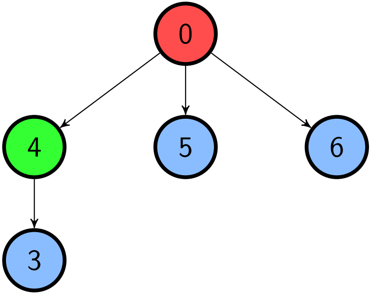




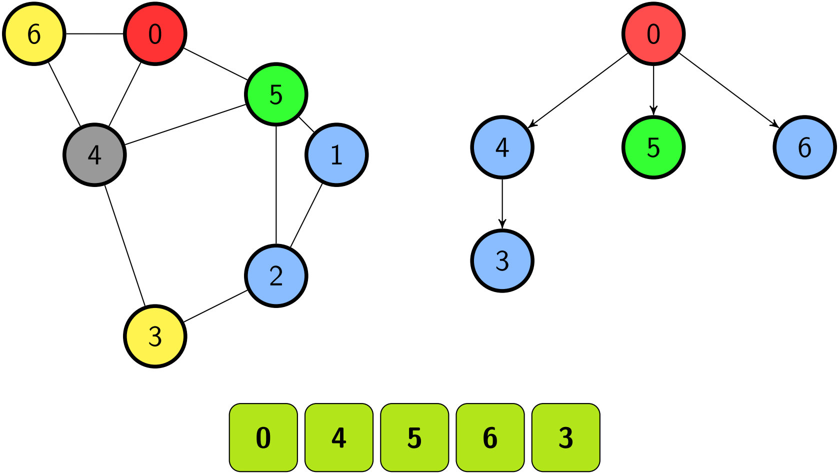
0	**4**	**5**	**6**

Parcurgerea în lățime

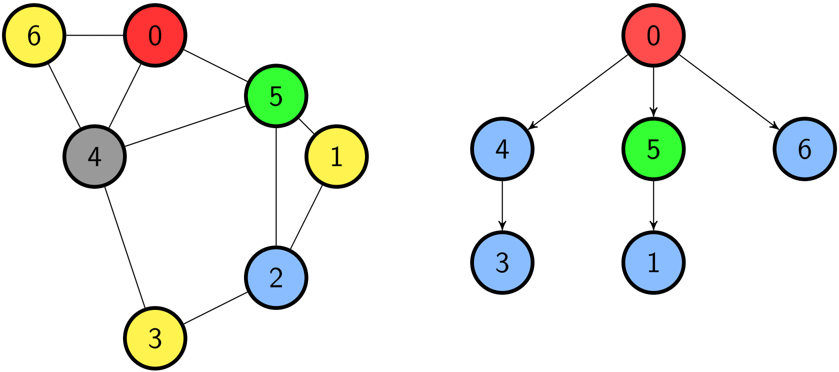




0	**4**	**5**	**6**	**3**

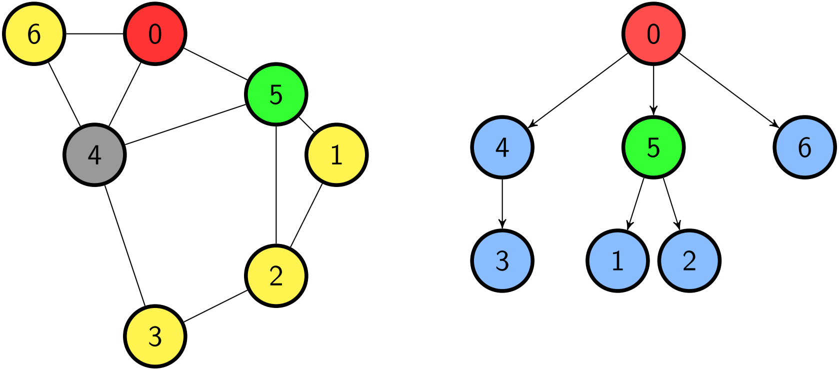

6
0
5
4
1
2
3
0
4
5
6
3
0 4 5 6 3

Parcurgerea în lățime



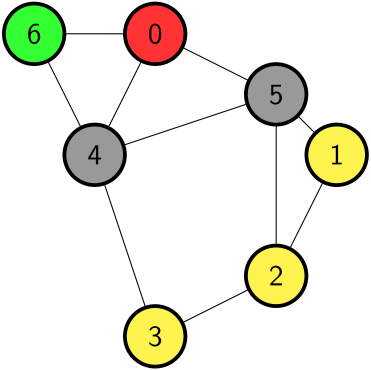
0	**4**	**5**	**6**	**3**	**1**

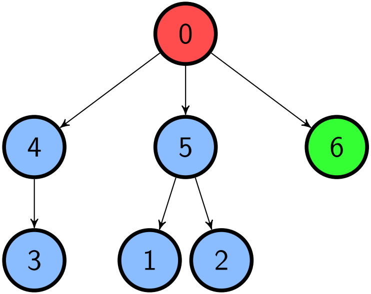
Parcurgerea în lățime



0	**4**	**5**	**6**	**3**	**1**	**2**

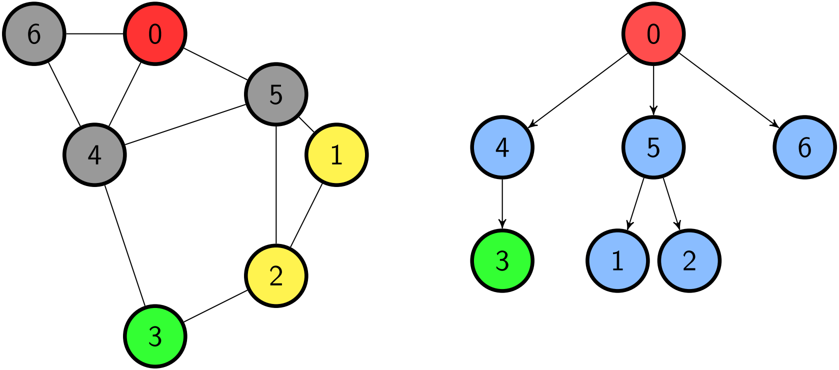
Parcurgerea în lățime





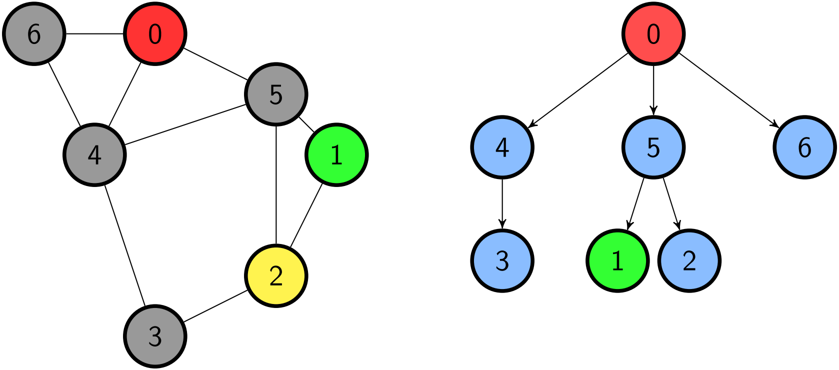
0	**4**	**5**	**6**	**3**	**1**	**2**

Parcurgerea în lățime



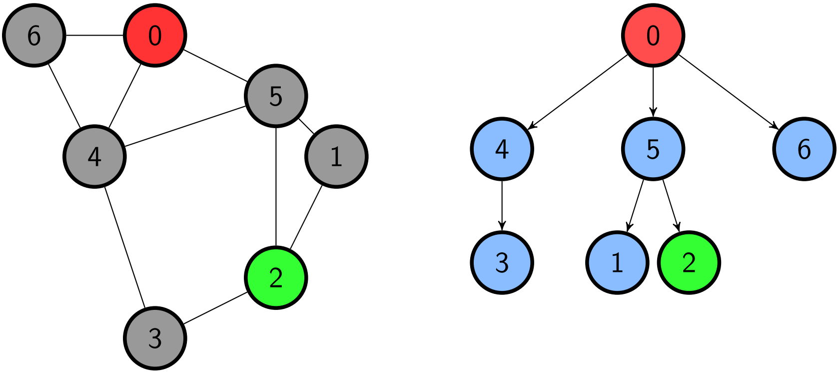
0	**4**	**5**	**6**	**3**	**1**	**2**

Parcurgerea în lățime



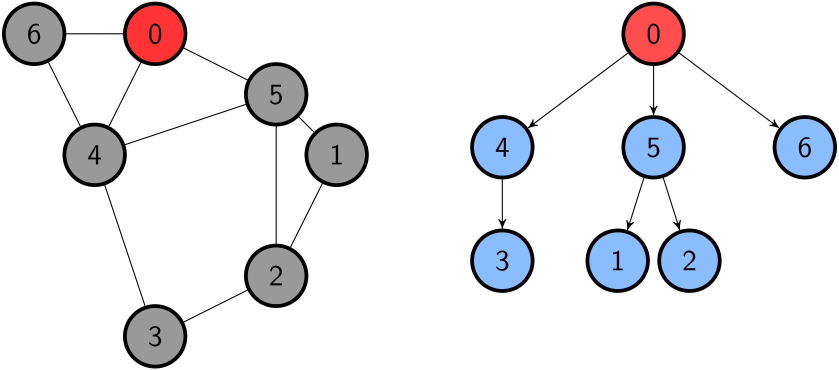
0	**4**	**5**	**6**	**3**	**1**	**2**

Parcurgerea în lățime



0	**4**	**5**	**6**	**3**	**1**	**2**

Parcurgerea în lățime



0	**4**	**5**	**6**	**3**	**1**	**2**

Parcurgerea în lățime

Vârfurile vizitate trebuie marcate:

$$viz[i] = \begin{cases} 1 & \text{dacă } \mathbf{i} \textit{ a fost vizitat} \\ 0 & \text{altfel} \end{cases}$$

Parcurgerea în lățime

Vârfurile vizitate trebuie marcate:

$$viz[i] = \begin{cases} 1 & \text{dacă } \mathbf{i} \textit{ a fost vizitat} \\ 0 & \text{altfel} \end{cases}$$

Important

Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore**, ce poate fi util pentru determinarea de lanțuri de la rădăcină la alte vârfuri.

Parcurgerea în lățime

Putem reține, în plus, vectorul tata, unde **tata[j]** reprezintă acel vârf **i** din care este descoperit (vizitat) vârful **j**.

Dacă dorim și determinarea de **distanțe minime** de la **s** la alte vârfuri, putem reține, în plus, vectorul de distanțe: **d[i]** = lungimea drumului determinat de algoritm de la **s** la **i**.

Parcurgerea în lățime

Putem reține, în plus, vectorul tata, unde **tata[j]** reprezintă acel vârf **i** din care este descoperit (vizitat) vârful **j**.
Dacă dorim și determinarea de **distanțe minime** de la **s** la alte vârfuri, putem reține, în plus, vectorul de distanțe: **d[i]** = lungimea drumului determinat de algoritm de la **s** la **i**.

$$d[j] = d[tatat[j]] + 1;$$

//nivelul lui j în arborele asociat parcurgerii
//d[i] = distanța de la s la i (s = nodul de start)

Parcurgerea în lățime

Pseudocod

Algorithm 1 BreadthFirstSearch

```
procedure BFS(G = {V, E}, start)
    for each vertex u ∈ V\{start} do
        color[u] ← WHITE
        dist[u] ← ∞
        parent[u] ← NIL
    color[start] ← GRAY
    dist[start] ← 0
    Q ← ∅
    Q ← enqueue(Q, start)
    while Q ≠ ∅ do
        u ← dequeue(Q)
        for each vertex v ∈ V\{u} do
            if (u, v) ∈ E then
                if color[v] = WHITE then
                    color[v] ← GRAY
                    dist[v] ← dist[u] + 1
                    parent[v] ← u
                    Q ← enqueue(Q, v)
        color[u] ← BLACK
```

Parcurgerea în lățime

```
void bfs(int mat[][20], int n, int s) {
   int viz[20], tata[20], d[20], p, u, c[20], i, j;
   for(i = 0; i < n; i++) {
      viz[i] = tata[i] = 0;
      d[i] = 32000; //distanta infinita
   }
   p = u = 1; c[1] = s; viz[s] = 1; d[s] = 0;
   while(p <= u) {
      i = c[p++];
      printf("%d\n", i);
      for(j = 0; j < n; j++) {
         if(mat[i][j] == 1 && !viz[j]) {
            c[u++] = j; viz[j] = 1;
            tata[j] = i; d[j] = d[i] + 1;
         }
      }
   }
}
```

Principalele aplicații ale acestui algoritm sunt următoarele:

- test graf conex (testam dacă toate vârfurile au fost vizitate atunci când aplicăm algoritmul de parcurgere în lățime dintr-un nod de start);

Principalele aplicații ale acestui algoritm sunt următoarele:

- test graf conex (testam dacă toate vârfurile au fost vizitate atunci când aplicăm algoritmul de parcurgere în lățime dintr-un nod de start);
- determinarea numărului de componente conexe;

Principalele aplicații ale acestui algoritm sunt următoarele:

- test graf conex (testam dacă toate vârfurile au fost vizitate atunci când aplicăm algoritmul de parcurgere în lățime dintr-un nod de start);
- determinarea numărului de componente conexe;
- determinarea unui arbore parțial al unui graf conex;

Principalele aplicații ale acestui algoritm sunt următoarele:

- test graf conex (testam dacă toate vârfurile au fost vizitate atunci când aplicăm algoritmul de parcurgere în lățime dintr-un nod de start);
- determinarea numărului de componente conexe;
- determinarea unui arbore parțial al unui graf conex;
- determinarea unui lanț / drum minim între două vârfuri u și v (se apelează bf pentru *u* și apoi se afișează drumul de la *u* la *v*, folosind vectorul tata) **dacă există**.

```
int componente(int mat[][20], int n, int s) {
   ...
   int nrComp = 0;
   //determinarea numarului de componente conexe
   for(i = 0; i < n; i++) {
      if(viz[i] == 0) {
         nrComp++;
         bfs(mat, n, i);
      }
   }
   return nrComp;
}
```

Parcurgerea în lățime

Observație

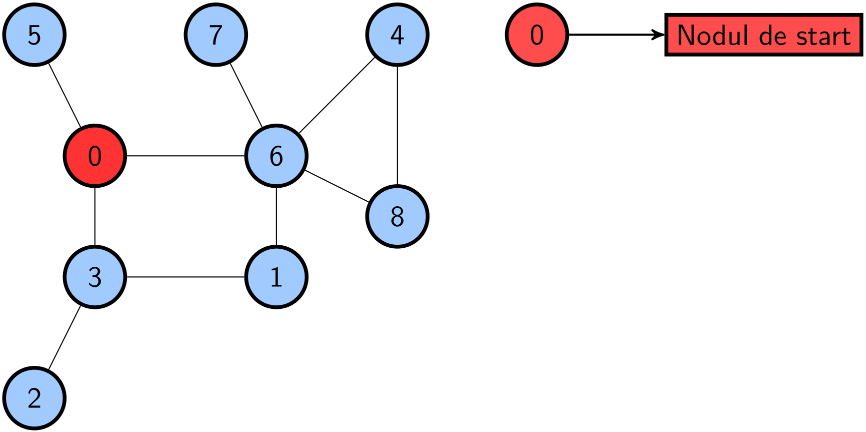
Parcurgerea în lățime este cunoscuta si ca algoritmul lui Lee in lumea algoritmicii romanesti. Implementarea algoritmului de mai sus are la baza o metoda iterativa si foloseste, ca structura auxiliara de date, o coada. Fiecare vecin va fi introdus in coada, iar la extragerea unuia din structura vom introduce in coada toti vecinii nevizitati ai nodului curent, avand grija sa eliminam nodul curent. Algoritmul se repeta pana cand coada devine vida.

Parcurgerea în adâncime

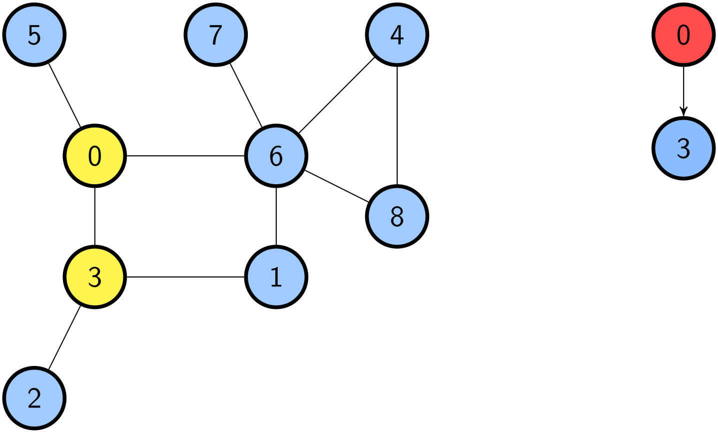
Se vizitează:

- **Inițial**: vârful de start s - devine vârf curent;
- **La un pas**:
  - se trece la primul vecin nevizitat al vârfului curent, **dacă există**;
  - **altfel**:
    - se merge **înapoi** pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați;
    - se trece la **primul** dintre aceștia și se reia procesul.

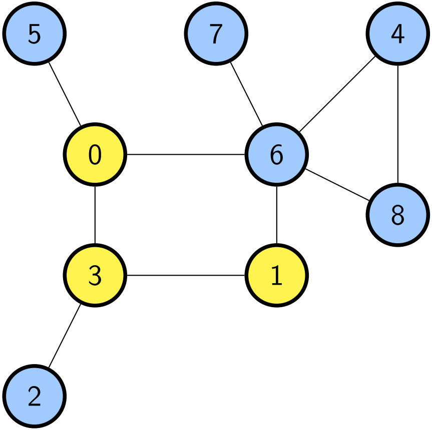
Parcurgerea în adâncime

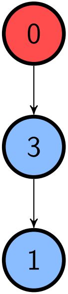


Parcurgerea în adâncime

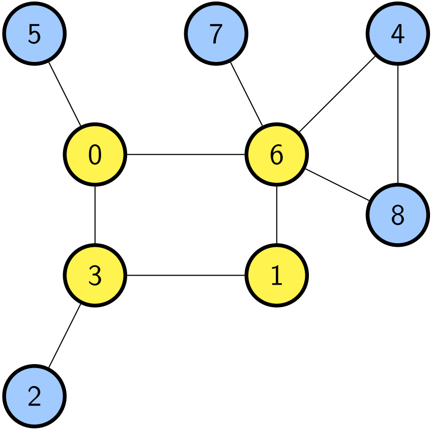


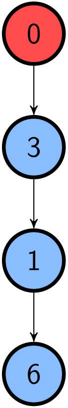
Parcurgerea în adâncime



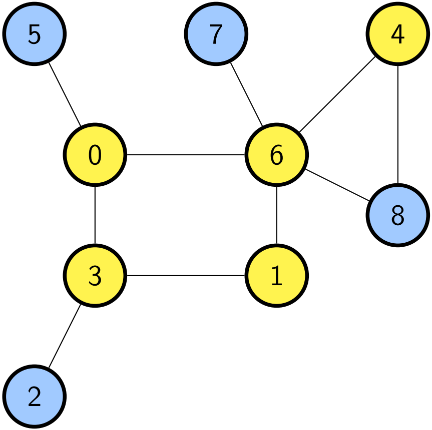


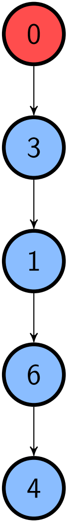
Parcurgerea în adâncime



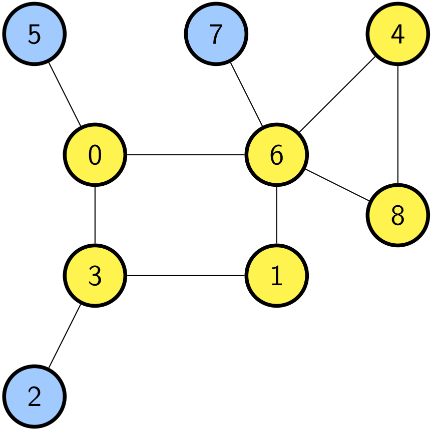


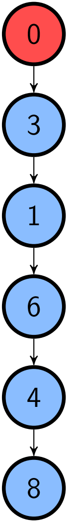
Parcurgerea în adâncime



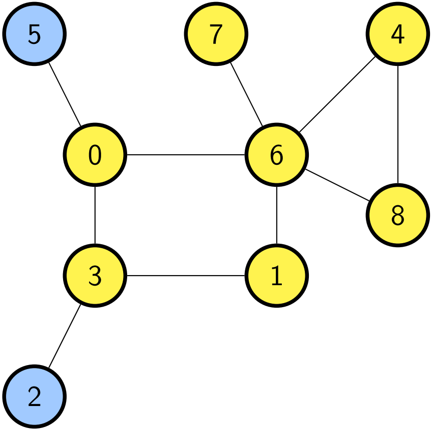


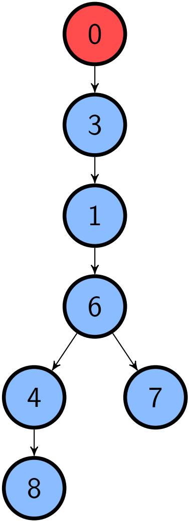
Parcurgerea în adâncime



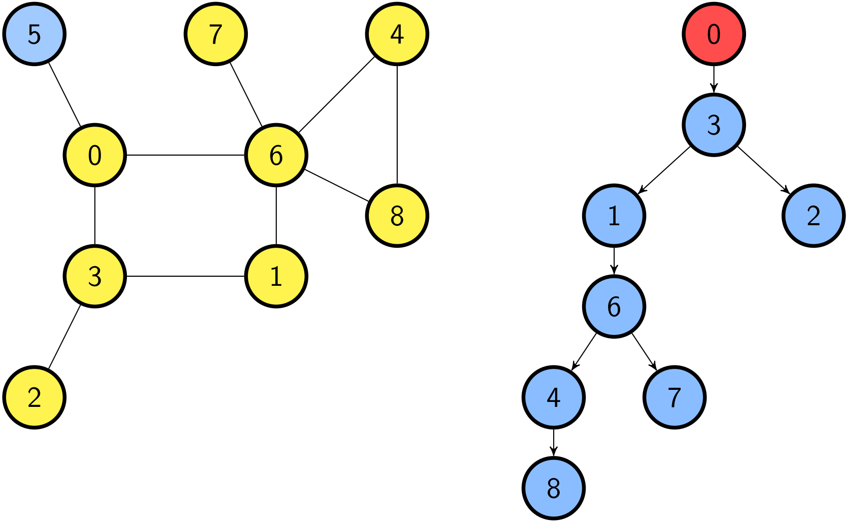


Parcurgerea în adâncime

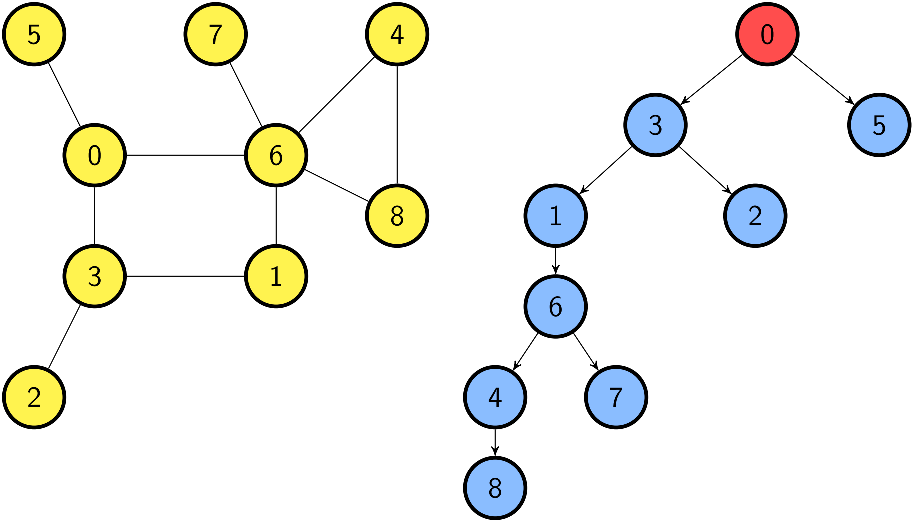




Parcurgerea în adâncime



Parcurgerea în adâncime



Parcurgerea în adâncime

Pseudocod

Algorithm 2 Depth First Search

procedure DFS(G = {V, E}, start)
    $S \leftarrow \emptyset$
    $S \leftarrow \boldsymbol{push}(S, v)$
    while $\boldsymbol{!isEmpty}(S)$ **do**
        $u \leftarrow \boldsymbol{pop}(S)$
        if $viz[u] = \boldsymbol{false}$ **then**
            $viz[u] \leftarrow \boldsymbol{true}$
            for each $vertex\ x \in V \setminus \{u\}$ **do**
                if $(u, x) \in E$ **then**
                    if $viz[x] = \boldsymbol{false}$ **then**
                        $S \leftarrow \boldsymbol{push}(S, x)$

Parcurgerea în adâncime

- Putem realiza o varianta recursivă care folosește implicit stiva.

```c
void dfs(int mat[][20], int n, int viz[], int tata[], int s){
   int i;
   viz[s] = 1;
   printf("%d\n", s);
   for(i = 0; i < n; i++) {
      if(mat[s][i] == 1 && !viz[j]) {
         tata[i] = s;
         dfs(mat, n, viz, tata, i);
      }
   }
}
```

Parcurgerea în adâncime

Aplicații

Principalele aplicații ale acestui algoritm sunt următoarele:

- verificarea existenței ciclurilor într-un graf (un ciclu se închide în parcurgere când vârful curent are un vecin deja vizitat, care nu este tatăl lui);

Principalele aplicații ale acestui algoritm sunt următoarele:

- verificarea existenței ciclurilor într-un graf (un ciclu se închide în parcurgere când vârful curent are un vecin deja vizitat, care nu este tatăl lui);
- să se verifice dacă un graf neorientat este bipartit;

Teorema Konig

Fie **G = (V, E)** un graf simplu cu **n** vârfuri ($n > 1$). Avem următoarea propozitie:
G este bipartit dacă și numai dacă toate ciclurile elementare din G sunt pare.

Principalele aplicații ale acestui algoritm sunt următoarele:

- verificarea existenței ciclurilor într-un graf (un ciclu se închide în parcurgere când vârful curent are un vecin deja vizitat, care nu este tatăl lui);
- să se verifice dacă un graf neorientat este bipartit;

Teorema Konig

Fie **G = (V, E)** un graf simplu cu **n** vârfuri ($n > 1$). Avem următoarea propozitie:
G este bipartit dacă și numai dacă toate ciclurile elementare din G sunt pare.

- în cazul grafurilor orientate, se poate folosi pentru a determina o sortare topologică a grafului;

Principalele aplicații ale acestui algoritm sunt următoarele:

- verificarea existenței ciclurilor într-un graf (un ciclu se închide în parcurgere când vârful curent are un vecin deja vizitat, care nu este tatăl lui);
- să se verifice dacă un graf neorientat este bipartit;

Teorema Konig

Fie **G = (V, E)** un graf simplu cu **n** vârfuri ($n > 1$). Avem următoarea propozitie:
G este bipartit dacă și numai dacă toate ciclurile elementare din G sunt pare.

- în cazul grafurilor orientate, se poate folosi pentru a determina o sortare topologică a grafului;
- de asemenea, poate fi folosit pentru a determina numărul de componente conexe.

Parcurgerea în adâncime
Aplicații

Principalele aplicații ale acestui algoritm sunt următoarele:

- verificarea existenței ciclurilor într-un graf (un ciclu se închide în parcurgere când vârful curent are un vecin deja vizitat, care nu este tatăl lui);
- să se verifice dacă un graf neorientat este bipartit;

Teorema Konig

Fie **G = (V, E)** un graf simplu cu **n** vârfuri ($n > 1$). Avem următoarea propozitie:
G este bipartit dacă și numai dacă toate ciclurile elementare din G sunt pare.

- în cazul grafurilor orientate, se poate folosi pentru a determina o sortare topologică a grafului;
- de asemenea, poate fi folosit pentru a determina numărul de componente conexe.

Implementare graf utilizând liste de adiacență

```c
typedef struct graph {
    int V; // nr de noduri din graf
    List *adjLists; // vectorul cu listele de adiacență
    int *visited; // vector pentru marcarea vizitate
}*Graph;
Graph initGraph(int V) {
    Graph g;
    int i;
    g = (Graph) malloc(sizeof(struct graph));
    g->V = V;
    g->adjLists = (List*) malloc(V * sizeof(List));
    for (i = 0; i < V; i++)
        g->adjLists[i] = NULL;
    g->visited = calloc(V, sizeof(int));
    return g;
}
```

Implementare graf utilizând liste de adiacență

```c
Graph insertEdge(Graph g, int u, int v) {
    g->adjLists[u] = addLast(g->adjLists[u], v);
    g->adjLists[v] = addLast(g->adjLists[v], u);
    return g;
}
int checkEdge(Graph g, int u, int v) {
    List tmp = g->adjLists[u];
    while (tmp != NULL) {
        if (tmp->data == v)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

Implementare graf utilizând liste de adiacență

```c
void dfs(Graph g, int start) {
    Stack stack = NULL;
    int i;
    stack = push(stack, start);
    for (i = 0; i < g->V; i++) {
        g->visited[i] = 0;
    }
    g->visited[start] = 1;
    int current;
    while (!isEmptyStack(stack)) {
        current = top(stack);
        stack = pop(stack);
        printf("%d ", current);
        List tmp = g->adjLists[current];
```

Implementare graf utilizând liste de adiacență

```
        while (tmp != NULL) {
            if (!g->visited[tmp->data]) {
                stack = push(stack, tmp->data);
                g->visited[tmp->data] = 1;
            }
            tmp = tmp->next;
        }
    }
}
```

Implementare graf utilizând liste de adiacență

```c
int bfs(Graph g, int start, int end) {
    Queue q = NULL;
    int i;
    q = enqueue(q, start);
    int current, *dist, *parent, result;
    dist = calloc(g->V, sizeof(int));
    parent = calloc(g->V, sizeof(int));
    for (i = 0; i < g->V; i++) {
        g->visited[i] = 0;
        parent[i] = -1;
        dist[i] = -1;
    }
    g->visited[start] = 1;
    dist[start] = 0;
```

Implementare graf utilizând liste de adiacență

```
    while (!isEmptyQueue(q)) {
        current = first(q);
        q = dequeue(q);
        printf("%d ", current);
        List tmp = g->adjLists[current];
        while (tmp != NULL) {
            if (!g->visited[tmp->data]) {
                q = enqueue(q, tmp->data);
                g->visited[tmp->data] = 1;
                dist[tmp->data] = dist[current] + 1;
                parent[tmp->data] = current;
            }
            tmp = tmp->next;
        }
    }
```

```
    result = dist[end];
    current = end;
    printf("\n");
    while (current != start) {
        printf("%d <- ", current);
        current = parent[current];
    }
    printf("%d\n", start);
    free(parent);
    free(dist);
    return result;
}
```

Vă mulțumesc pentru atenție!