Algoritmica Grafurilor - Reprezentari, Parcurgeri, Drumuri

Un graf este format din noduri (varfuri) si muchii/arce.

1. Reprezentarea grafurilor :

1.1. Lista de adiacenta :

- Fiecare nod are o lista cu vecinii sai.
- Eficienta pentru grafuri "rare" (putine muchii).
- Usor de implementat cu vectori de liste.

Structura :
Adj[x] = lista nodurilor adiacente a lui x
Exemplu :
Adj[1]: 2 5
Adj[2]: 1 3 4 5
Adj[3]: 2 4
...

1.2. Matricea de adiacenta :

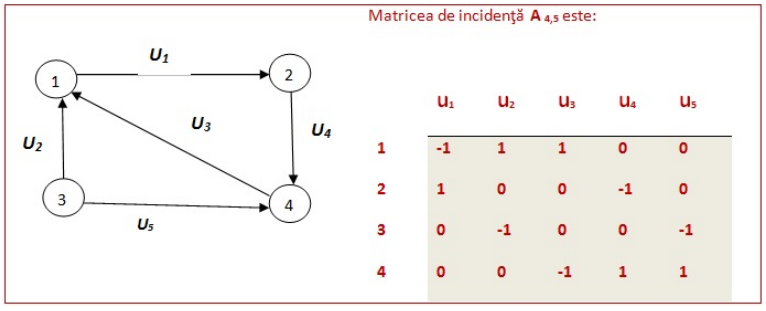
- Matrice patratica : matrice[i][j] = 1 daca exista muchie (i,j), altfel 0.
- Usor de folosit dar ocupa mult spatiu (O(n^2))

Exemplu pentru 5 noduri :

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

1.3. Matrice de incidenta :

- Linii = noduri, coloane = muchii
- 1 : nodul este destinatia
- -1 : nodul este sursa muchiei
- 0 : nu e implicat



2. Parcurgere in latime (BFS)

Ce face :

- Parcurge graful pe niveluri : intai viziteaza toti vecinii nodului curent, apoi vecinii vecinilor etc.
- Foloseste o coada (queue) pentru a tine minte ce urmeaza de vizitat.

Rezolva probleme de tip :

- Gasirea celui mai scurt drum intr-un graf neponderat.
- Determinarea daca un graf este conex.
- Numarare de componente conexe.

Cum functioneaza :

- Pleci de la un nod.
- Il vizitezzi si adaugi vecinii sai in coada.
- Apoi iei pe rand fiecare nod din coada si repeti procesul.

```cpp
void BFS(int nod, queue<int> q)
{
    q.push(nod);
    distanta[nod] = 0;
    while (!q.empty())
    {
        int nod_curent = q.front();
        q.pop();
        for (int i = 1; i <= noduri; i++)
        {
            if (matrice_adiacenta[nod_curent][i] == 1 and distanta[i] == -1)
            {
                q.push(i);
                distanta[i] = distanta[nod_curent] + 1;
            }
        }
    }
}
```

3. DFS (Depth-First Search) - Parcurgere in adancime

Ce face :

- Parcurge graful cat de adanc se poate pe o ramura.

Rezolva probleme de tip :

- Detectarea ciclurilor intr-un graf.
- Impartirea grafului in componente conexe.

Cum functioneaza :

- Pleci de la un nod.
- Vizitezi un vecin nevizitat.
- De acolo vizitezi un alt vecin nevizitat si tot asa ...

```cpp
void DFS(int nod)
{
    vizitat[nod] = true;
    for (int i = 1; i <= noduri; i++)
    {
        if (matrice_adiacenta[nod][i] == 1 and !vizitat[i])
        {
            DFS(i);
        }
    }
}
```

4. Algoritmul lui Dijkstra (Cel mai scurt drum cu costuri)

Ce face :

- Gaseste cel mai scurt drum (cost minim) de la un nod sursa la toate celelalte noduri in grafuri ponderate cu costuri pozitive.

Rezolva probleme de tip :

- Transport, retele, drumuri optime.
- Calcule de distante minime in grafuri.

Cum functioneaza :

- Pleci de la sursa si setezi distanta lui la 0, restul ∞.
- Alegi mereu nodul neprocesat cu cea mai mica distanta cunoscuta.
- Actualizezi distantele vecinilor lui.
- Repeti pana cand toate nodurile au fost procesate.

Atentie :

- Merge doar daca toate costurile sunt pozitive!

- De ce folosim priority queue ca sa scoatem nodul cu cel mai mic cost in Dijkstra? Pentru ca construim drumurile pas cu pas si vrem sa construim intai drumurile cele mai scurte, apoi cele mai lungi. Facem asta pentru ca daca ajungi la un nod pe cel mai scurt drum posibil, atunci orice alt drum care ajunge acolo mai tarziu va fi mai lung, deci odata ce am gasit cel mai scurt drum la un nod, nu mai trebuie sa il actualizam.

```cpp
vector<int> dijkstra(int V, vector<vector<pair<int,int>>> &adj, int src){

    // Create a priority queue to store vertices that are being preprocessed.
    priority_queue<pair<int,int>, vector<pair<int,int>>,greater<pair<int,int>>> pq;

    // Create a vector for distances and initialize all distances as infinite
    vector<int> dist(n: V + 1, x: INT_MAX);

    // Insert source itself in priority queue and initialize its distance as 0.
    pq.push(v: [&]{src, 0});
    dist[src] = 0;

    // Looping till priority queue becomes empty (or all distances are not finalized)
    while (!pq.empty()){
        // The first vertex in pair is the minimum distance vertex, extract it from priority queue.
        int u = pq.top().first;
        pq.pop();
        // Get all adjacent of u.
        for (auto x :pair<int, int> : adj[u]){
            // Get vertex label and weight of current adjacent of u.
            int v = x.first;
            int weight = x.second;
            // If there is shorter path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(v: [&]{v, [&]dist[v]});
            }
        }
    }
    return dist;
}
```

Parte scrisă	Ce înseamnă natural
`priority_queue`	Este o coadă cu priorități.
`<pair<int, int>>`	Tipul datelor stocate: fiecare element este o pereche (cost, nod).
`vector<pair<int, int>>`	Containerul intern folosit pentru stocare (un vector simplu).
`greater<pair<int, int>>`	Criteriul de comparare: **prioritate mai mare pentru costuri mai mici.**

5. Algoritmul lui Kosaraju (Componente tare conexe)

Ce face :

- Identifica toate componentele tare conexe intr-un graf orientat.
- O componenta tare conexa este un grup de noduri intre care poti ajunge de la oricare la oricare.

Cum functioneaza :

- Faci DFS normal si pui nodurile intr-o stiva in functie de cand se termina procesarea lor.
- Inversezi directiile muchiilor (graful transpus).
- Faci DFS din stiva → fiecare DFS gaseste o componenta tare conexa.

```cpp
void dfs1(int u, vector<vector<int>>& adj, vector<bool>& visited, stack<int>& S) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs1(v, adj, visited, S);
        }
    }
    S.push(u); // punem pe stiva la finalizarea nodului
}

void dfs2(int u, vector<vector<int>>& adjT, vector<bool>& visited) {
    visited[u] = true;
    cout << u << " ";
    for (int v : adjT[u]) {
        if (!visited[v]) {
            dfs2(v, adjT, visited);
        }
    }
}
void Kosaraju(int V, vector<vector<int>>& adj) {
    stack<int> S;
    vector<bool> visited(V + 1, false);

    // Pas 1: DFS normal
    for (int i = 1; i <= V; ++i) {
        if (!visited[i]) {
            dfs1(i, adj, visited, S);
        }
    }

    // Pas 2: Construim transpusul
    vector<vector<int>> adjT(V + 1);
    for (int u = 1; u <= V; ++u) {
        for (int v : adj[u]) {
            adjT[v].push_back(u);
        }
    }

    // Pas 3: DFS pe transpus
    fill(visited.begin(), visited.end(), false);

    cout << "Componente tare conexe:\n";
    while (!S.empty()) {
        int u = S.top();
        S.pop();
        if (!visited[u]) {
            dfs2(u, adjT, visited);
            cout << "\n"; // fiecare DFS descoperă o componentă
        }
    }
}
int main() {
    int V = 5;
    vector<vector<int>> adj(V + 1);

    adj[1] = {2};
    adj[2] = {3};
    adj[3] = {1};
    adj[4] = {5};

    Kosaraju(V, adj);

    return 0;
}
```