AVANSAT · OLIMPIADĂ · LECȚIA 10

Grafuri pentru concurs: Dijkstra cu heap, sortare topologică, DP pe DAG

Durată: 90 min Nivel: olimpiadă / informatică intensiv Înainte: lecțiile xi-12, xi-19, co-01

CE VEI ȘTI DUPĂ LECȚIE

- Implementezi Dijkstra cu priority_queue, în O((n + m) log n), pe liste de adiacență.
- Reconstruiești drumul minim, nu doar lungimea lui.
- Faci sortarea topologică (algoritmul lui Kahn) și detectezi ciclurile.
- Rezolvi probleme de programare dinamică pe un graf orientat fără circuite (DAG).

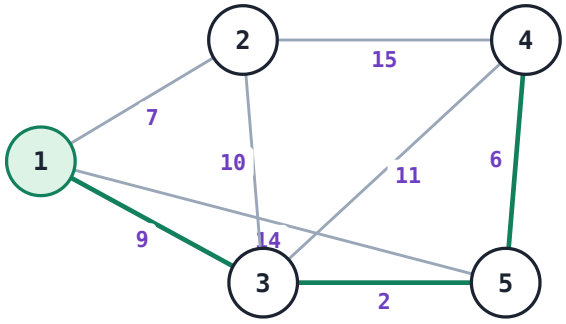
1. Ce știi deja și ce lipsește

La clasa a XI-a (lecția XI.19) ai învățat Dijkstra cu un vector de distanțe, în $O(n^2)$. Pentru $n = 10^5$ noduri, $n^2 = 10^{10}$ este prea mult. Aici îl accelerăm cu o **coadă cu priorități** (heap) și liste de adiacență, apoi învățăm o altă unealtă pentru grafuri orientate: **sortarea topologică** și **programarea dinamică pe DAG**.

2. Dijkstra cu priority_queue

Idee (recapitulare). Păstrăm `dist[v]` = cea mai bună distanță găsită până acum de la sursă. Repetăm: alegem nodul **nevizitat cu distanța cea mai mică**, îl declarăm „finalizat” și **relaxăm** muchiile lui: dacă `dist[u] + cost < dist[v]`, actualizăm `dist[v]`.

Partea lentă din $O(n^2)$ este căutarea minimului. Cu un **heap** care ține perechi `(distanță, nod)`, minimul se extrage în O(log n).



verde = drumul minim 1 → 3 → 5 → 4 (9 + 2 + 6 = 17); drumul direct 1 → 5 costă 14, dar 1 → 3 → 5 costă doar 11

Dijkstra

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

const long long INF = 1e18;
vector<pair<int, int>> g[100005];          // g[u] = lista (vecin, cost)
long long dist[100005];
int tata[100005];

int main()
{
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int a, b, c;
        cin >> a >> b >> c;
        g[a].push_back({b, c});
        g[b].push_back({a, c});            // graf neorientat: ambele sensuri
    }
    int tinta;
    cin >> tinta;

    for (int i = 1; i <= n; i++) dist[i] = INF;
    priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<pair<long
long, int>>> pq;
    dist[1] = 0;
    pq.push({0, 1});
    while (!pq.empty())
    {
        auto [d, u] = pq.top();            // nodul cu distanța cea mai mică
        pq.pop();
        if (d > dist[u]) continue;         // intrare învechită: am găsit între timp ceva
mai bun
        for (auto [v, c] : g[u])
            if (dist[u] + c < dist[v])     // relaxare
            {
                dist[v] = dist[u] + c;
                tata[v] = u;               // reținem de unde am venit
                pq.push({dist[v], v});
            }
    }

    for (int i = 1; i <= n; i++) cout << dist[i] << (i < n ? ' ' : '\n');
    vector<int> drum;                      // reconstruim drumul din tinta înapoi la sursa
    for (int v = tinta; v != 0; v = tata[v]) drum.push_back(v);
    reverse(drum.begin(), drum.end());
    for (int v : drum) cout << v << ' ';
    return 0;
}
```

Pas	Extras din heap	Distanțe după relaxare (1, 2, 3, 4, 5)
început	–	0, ∞, ∞, ∞, ∞

Pas	Extras din heap	Distanțe după relaxare (1, 2, 3, 4, 5)
1	(0, nod 1)	0, 7, 9, ∞, 14
2	(7, nod 2)	0, 7, 9, 22, 14 (2 → 4 costă 7 + 15)
3	(9, nod 3)	0, 7, 9, 20, 11 (3 → 4: 9 + 11; 3 → 5: 9 + 2)
4	(11, nod 5)	0, 7, 9, 17, 11 (5 → 4: 11 + 6)
5	(14, nod 5): intrare veche	sare (14 > 11)
6	(17, nod 4)	gata: 0, 7, 9, 17, 11

Drumul spre 4: `tata[4] = 5`, `tata[5] = 3`, `tata[3] = 1`, `tata[1] = 0` (oprire): drumul `1 3 5 4`.

DE CE AȘA?

De ce `if (d > dist[u]) continue`? Heap-ul nu poate **modifica** o intrare existentă, deci la fiecare îmbunătățire adăugăm o intrare nouă, iar cele vechi rămân. Când scoatem una veche (cu `d` mai mare decât distanța curentă), o ignorăm. **De ce `greater`?** `priority_queue` implicit scoate cel **mai mare**; cu `greater` scoate cel mai mic. **De ce `long long` și `INF = 10¹⁸`?** O distanță pe un drum lung poate depăși 2^{31}; iar `INF` trebuie să rămână sub limita tipului chiar și după adunări (de aceea nu `LLONG_MAX`). **Complexitate:** fiecare muchie provoacă cel mult o inserare: O((n + m) log m).

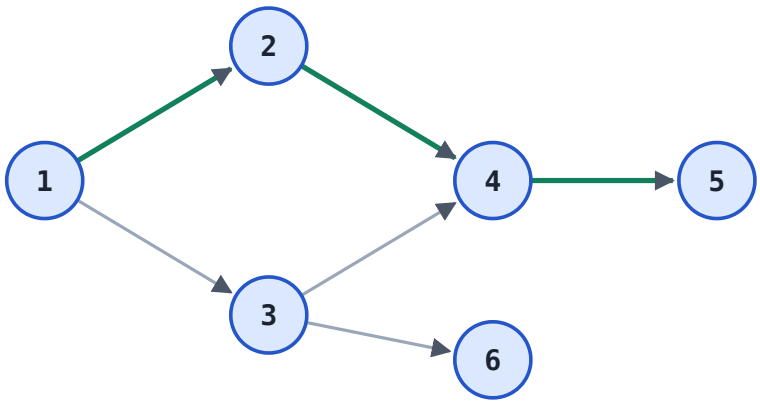
ATENȚIE — GREȘELI FRECVENTE

Dijkstra **nu merge** cu costuri **negative**: un nod finalizat nu își mai poate îmbunătăți distanța. Pentru costuri negative folosești Bellman-Ford (O(n·m)). Pentru costuri doar 0 și 1 există **BFS 0-1** cu un deque, în O(n + m).

3. Sortare topologică (Kahn)

Un graf orientat **fără circuite** se numește **DAG** (*directed acyclic graph*). Exemple: dependențele dintre cursuri („A înainte de B"), ordinea de compilare a fișierelor, ordinea de calcul a formulelor.

O **sortare topologică** așază nodurile într-un șir în care **orice arc `u → v` are `u` înaintea lui `v`**.



verde = un drum cu cele mai multe noduri (1 → 2 → 4 → 5, 4 noduri)

DAG

Algoritmul lui Kahn. Calculăm **gradul interior** al fiecărui nod (câte arce intră). Nodurile cu grad interior 0 nu depind de nimic: le punem într-o coadă. Scoatem un nod, îl adăugăm în ordine, „ștergem" arcele lui (scădem gradul interior al vecinilor) și punem în coadă vecinii care ajung la grad 0.

Dacă la sfârșit am scos mai puține noduri decât n, graful are un **ciclu** (nodurile din ciclu nu ajung niciodată la grad 0).

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

vector<int> g[100005];
int grad[100005], dp[100005];

int main()
{
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;                    // arcul a -> b
        g[a].push_back(b);
        grad[b]++;                        // gradul interior al lui b
    }
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (grad[i] == 0) q.push(i);      // nodurile fără dependențe
    vector<int> ordine;
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        ordine.push_back(u);
        for (int v : g[u])
            if (--grad[v] == 0) q.push(v); // v nu mai așteaptă nimic
    }
    if ((int)ordine.size() < n)
    {
        cout << "exista ciclu";
        return 0;
    }
    for (int u : ordine) cout << u << ' ';
    cout << '\n';

    // DP pe DAG: dp[u] = cel mai mare număr de noduri pe un drum care se termină în u
    int rezultat = 0;
    for (int i = 1; i <= n; i++) dp[i] = 1;
    for (int u : ordine)                  // în ordine topologică: dp[u] este deja final
    {
        rezultat = max(rezultat, dp[u]);
        for (int v : g[u]) dp[v] = max(dp[v], dp[u] + 1);
    }
    cout << rezultat;
    return 0;
}
```

Pas	Coada	Ordine	Gradele interioare (1..6) după pas
început	1	–	0, 1, 1, 2, 1, 1

Pas	Coada	Ordine	Gradele interioare (1..6) după pas
scoate 1	2, 3	1	0, 0, 0, 2, 1, 1
scoate 2	3	1 2	0, 0, 0, 1, 1, 1
scoate 3	4, 6	1 2 3	0, 0, 0, 0, 1, 0
scoate 4	6, 5	1 2 3 4	0, 0, 0, 0, 0, 0
scoate 6, apoi 5	–	1 2 3 4 6 5	–

DE CE AȘA?

De ce funcționează? Un nod cu grad interior 0 nu are niciun nod care trebuie să fie înaintea lui care să nu fi fost deja pus în ordine: deci poate fi pus acum. Când un nod este scos, arcele lui nu mai blochează nimic. **De ce arată ciclul?** Nodurile unui ciclu se așteaptă unele pe altele: gradul interior al fiecăruia rămâne ≥ 1, deci nu intră niciodată în coadă și `ordine.size() < n`. **Ordinea nu este unică:** orice nod cu grad 0 poate fi ales; de aceea, problemele cer de obicei „orice ordine validă" sau „cea mai mică lexicografic" (cu `priority_queue` în loc de `queue`).

4. DP pe DAG

Într-un DAG, dacă parcurgi nodurile în ordine topologică, când ajungi la `u` toate nodurile care ajung în `u` au fost deja prelucrate: `dp[u]` este final. Poți calcula astfel:

Problema	Recurența
cel mai lung drum (număr de noduri) până în u	`dp[v] = max(dp[v], dp[u] + 1)` pentru fiecare arc `u → v`
cel mai lung drum cu costuri	`dp[v] = max(dp[v], dp[u] + cost)`
numărul de drumuri de la sursă la u	`dp[v] += dp[u]` (modulo, dacă e cazul)
drumul minim (chiar cu costuri negative!)	`dp[v] = min(dp[v], dp[u] + cost)`

SFAT

Într-un DAG, drumul **minim** cu costuri negative se rezolvă în O(n + m) prin DP în ordine topologică, mai rapid decât Dijkstra și chiar cu costuri negative.

5. Dijkstra sau topologic sau BFS?

Graful	Costuri	Algoritm
fără costuri (sau toate 1)	–	BFS
doar 0 și 1	–	BFS 0-1 (deque)
orice sens, costuri ≥ 0	pozitive	Dijkstra cu heap
orice sens, costuri cu minus	negative	Bellman-Ford

Graful	Costuri	Algoritm
orientat, fără circuite	oricare	DP în ordine topologică
între toate perechile, n ≤ 400	oricare	Roy-Floyd

6. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
uiți `if (d > dist[u]) continue`	corect, dar mai lent (nodul se prelucrează de mai multe ori)	sari peste intrările vechi
`priority_queue` fără `greater`	scoate maximul: rezultate greșite	`greater<...>` sau costuri negate
`INF = INT_MAX` și `dist[u] + c`	depășire	`long long` și `INF = 1e18`
un singur sens pe graful neorientat	drumuri lipsă	adaugi ambele arce
Dijkstra cu costuri negative	rezultat greșit	Bellman-Ford
cod pentru `tata` fără oprire	buclă infinită la reconstruire	`tata[sursa] = 0`, oprire la 0
nu verifici ciclul la sortare topologică	ordine incompletă ca și cum ar fi corectă	compară `ordine.size()` cu n
DP pe DAG fără ordinea topologică	valori nefinale	prelucrezi nodurile în ordine topologică

7. La olimpiadă

Dijkstra cu heap este o unealtă de bază: „drum minim cu restricții" se rezolvă adesea pe un graf **cu stări** (nod, ceva în plus), de exemplu (nod, câte cupoane au rămas). Sortarea topologică apare la „dependențe", „ordinea de execuție" și în DP pe DAG („numărul de drumuri", „drumul de cost maxim"). Când un graf cere „cel mai lung drum" și are cicluri, problema devine foarte grea: verifică întâi dacă e DAG.

8. Exersează pe hârtie

1. Rulează Dijkstra din nodul 1 pe: 1–2 (4), 1–3 (1), 3–2 (2), 2–4 (5), 3–4 (8). Care sunt distanțele?

2. Care este drumul minim de la 1 la 4 în exercițiul 1?

3. Care este gradul interior al fiecărui nod în graful orientat 1→2, 1→3, 2→4, 3→4, 4→5?

4. Scrie o sortare topologică pentru graful de la exercițiul 3 (algoritmul lui Kahn).

5. Are graful 1→2, 2→3, 3→4, 4→2 o sortare topologică? Ce rămâne după Kahn?

6. Pentru DAG-ul 1→2, 1→3, 2→4, 3→4: câte drumuri există de la 1 la 4?

7. Pentru același DAG, câte noduri are cel mai lung drum?

8. De ce nu poate Dijkstra să folosească `queue` în loc de `priority_queue`?

9. Ce complexitate are Dijkstra cu heap pentru n = 10^5, m = $2 \cdot 10^5$? Aproximează numărul de operații.

10. De ce un DAG cu costuri negative se rezolvă bine, dar un graf oarecare cu costuri negative nu cu Dijkstra?

9. Mini-grile

1. Dijkstra cu heap are complexitatea: a) $O(n^2)$ b) $O((n + m) \log n)$ c) $O(n \cdot m)$ d) $O(m)$

2. Un graf orientat are sortare topologică dacă și numai dacă: a) este conex b) nu are circuite c) are $n - 1$ arce d) toate costurile sunt pozitive

3. În algoritmul lui Kahn, nodurile intră în coadă când: a) au gradul exterior 0 b) au gradul interior 0 c) sunt vizitate d) au cel mai mare grad

4. Pentru costuri negative, într-un graf oarecare, folosești: a) BFS b) Dijkstra c) Bellman-Ford d) Kruskal

10. Mai departe (pentru cine vrea mai mult)

- **Bellman-Ford:** costuri negative și detectarea ciclurilor negative.
- **BFS 0-1:** `deque`, nodurile cu cost 0 la început și cele cu cost 1 la sfârșit.
- **Dijkstra pe grafuri cu stări:** (nod, stare) ca „nod" al unui graf mai mare.
- **A*:** Dijkstra ghidat de o estimare.
- **Sortare topologică prin DFS** (ordinea inversă a terminării).
- **Componente tare conexe** (Kosaraju, Tarjan) și contracția lor într-un DAG.

Rezumat

- Dijkstra cu `priority_queue<pair<long long, int>, …, greater<…>>`: $O((n + m) \log n)$; sari peste intrările vechi.
- Pentru drum, reții `tata[v]` la fiecare relaxare și reconstruiești de la țintă la sursă.
- Costurile negative strică Dijkstra; folosește Bellman-Ford (sau DP pe DAG dacă nu are circuite).
- Kahn: grad interior 0 → coadă; dacă ies mai puține noduri decât n, există ciclu.
- DP pe DAG în ordine topologică: cel mai lung drum, numărul de drumuri, drum minim.

Probleme de lucru (concurs)

Probleme de pe infoarena, CSES, Kilonova sau pbinfo, în ordinea dificultății. Trimite soluțiile pe platformă și bifează ce ai terminat.

- [ ] `infoarena` **dijkstra** — ușoară
  problema clasică: drumuri minime dintr-un nod, graf orientat
  https://www.infoarena.ro/problema/dijkstra

- [ ] CSES **Shortest Routes I** — ușoară
  Dijkstra din nodul 1, exact programul din lecție
  https://cses.fi/problemset/task/1671

- [ ] CSES **Course Schedule** — medie
  sortare topologică și detectarea ciclurilor
  https://cses.fi/problemset/task/1679

- [ ] CSES **Longest Flight Route** — medie
  DP pe DAG cu reconstruirea drumului
  https://cses.fi/problemset/task/1680

Temă (concurs)

Mai grele: încearcă-le după ce le-ai terminat pe cele de mai sus.

- [ ] CSES **Flight Discount** — dificilă
  Dijkstra cu două stări pe nod: cu și fără cupon folosit
  https://cses.fi/problemset/task/1195

- [ ] CSES **Game Routes** — medie
  numărarea drumurilor într-un DAG
  https://cses.fi/problemset/task/1681

Citește și exersează în plus

- CP-Algorithms — Dijkstra — Dijkstra pentru grafuri rare, cu set sau coadă cu priorități (engleză)
  https://cp-algorithms.com/graph/dijkstra_sparse.html
- CP-Algorithms — Topological Sorting — sortare topologică cu DFS (engleză) https://cp-algorithms.com/graph/topological-sort.html
- USACO Guide — Shortest Paths — Dijkstra și probleme graduate (engleză)
  https://usaco.guide/gold/shortest-paths
- VisuAlgo — Single-Source Shortest Paths — animație pentru Dijkstra și Bellman-Ford (engleză)
  https://visualgo.net/en/sssp

Răspunsuri

Exersează pe hârtie

1. `dist[1] = 0`; `dist[3] = 1`; `dist[2] = min(4, 1 + 2) = 3`; `dist[4] = min(3 + 5, 1 + 8) = 8`. Distanțele: **0, 3, 1, 8**.

2. Drumul 1 → 3 → 2 → 4: 1 + 2 + 5 = 8 (față de 1 → 3 → 4 cu 9).

3. Grade interioare: 1: 0; 2: 1; 3: 1; 4: 2; 5: 1.

4. Coada începe cu 1; după 1 intră 2 și 3; după 2 scade gradul lui 4 la 1; după 3, 4 ajunge la 0 și intră; după 4 intră 5. Ordine: **1 2 3 4 5**.

5. Nu: 2 → 3 → 4 → 2 este un circuit. Kahn scoate doar nodul 1; nodurile 2, 3, 4 rămân cu grad interior ≥ 1 (ordine de lungime 1 < 4).

6. Două drumuri: 1 → 2 → 4 și 1 → 3 → 4.

7. Trei noduri (de exemplu 1 → 2 → 4).

8. `queue` extrage în ordinea inserării, nu a distanței: ar procesa nodul cu distanța mai mare înaintea celui cu distanța mai mică, iar distanțele ar fi finalizate greșit.

9. $O((n + m) \log n) \approx 3 \cdot 10^5 \cdot 17 \approx 5 \cdot 10^6$ operații.

10. În DAG, ordinea topologică garantează că dp[u] este final când îl folosim, indiferent de semnul costurilor. În Dijkstra, ordinea de finalizare depinde de distanțe crescătoare, ceea ce costurile negative strică.

Mini-grile

1. b) O((n + m) log n). **2.** b) nu are circuite. **3.** b) au gradul interior 0. **4.** c) Bellman-Ford.