Costul minim de la un nod la toate celelalte

Nicolau Mihnea-Andrei, 321CD

Facultatea de Automatica si Calculatoare,
Universitatea Politehnica din Bucuresti

Abstract. În acest document voi prezenta importanța problemei celui mai scurt drum și voi compara soluțiile alese.

1 Descrierea problemei rezolvate

Găsirea celui mai scurt drum între un nod si toate celelalte reprezintă o problemă importantă din teoria grafurilor, ce are aplicații practice în domeniile:

- comunicaii (ex: IP routing, telefonie);
- transport (ex: harti geografice, gps);
- computer science (ex: AI, game developement) etc.

Pentru rezolvarea acestei probleme se pot gasi mai multe solutii, unele fiind mai eficiente decat altele pentru anumite tipuri de grafuri.

Soluțiile alese. Am ales să compar algoritmii Dijkstra, Bellman-Ford și Topologic Sort (aplicat pentru shortest path).

Criteriile de evaluare. Voi compara soluțiile alese pe mai multe teste astfel încăt să observ *corectitudinea* și *eficiența* aplicării lor pe diferite inputuri.

2 Prezentarea soluțiilor

Principiul relaxării și inițializarea distanțelor.
Toate soluțiile ce vor fi prezentate în continuare se folosesc de relaxarea muchiilor unui graf și de vectorul de distanțe.
Vecotrul de distanțe. Este inițializat cu valori supraestimate ale distanțelor de la sursă la celelalte noduri.

```
int distTo[N];
for (i = 0; i < N; i++)
    distTo[N] = inf;
distTo[source] = 0;
```

Principiul relaxării. Se referă la reducerea costului drumurilor.
Fie un graf G(V, E), cu V mulțimea de noduri și E mulțimea de muchii a grafului și un nod s, ce reprezintă nodul sursă pentru care vom calcula distanțele către toate celelalte noduri.
Relaxarea unei muchii edge(u, v) (cu u = nodul sursă și v = nodul destinație și w = costul trecerii prin muchie), constă în testarea posibilității de reducere a costului drumului de la s la v, trecând prin muchia u-v (dacă s-a găsit un drum mai scurt de la s la v).

```
int u = edge.source(), v = edge.destination();
    if (distTo[v] > distTo[u] + edge.weight()) {
        distTo[v] = distTo[u] + edge.weight();
        parentOf[v] = u;  //dacă vrem să reținem calea
    } [1]
```

2.1 Algoritmul Dijkstra

Algoritmul se bazează pe „vizitarea” fiecărui nod din graf (relaxarea muchiilor care pornesc din acel nod) o singură dată. Dacă un nod a fost vizitat, distanța lui față de sursă se consideră găsită (distanța finală, nu o supraestimare).
Se utilizează un vector visited[N] pentru a se reține ce noduri au fost vizitate. Toate celelalte noduri se află într-o listă de noduri care trebuie vizitate (inițial, în listă se află doar nodul sursă).
Se selectează un nod u (din lista de noduri de vizitat), care are, la acel moment, costul estimat (al drumului față de nodul sursă) minim. Nodul este vizitat și i se adaugă toți vecinii nevizitați în lista de noduri de vizitat. Se consideră că am găsit distanța minimă pentru nodul u (îl marcăm ca vizitat).
Procesul se repetă până când nu mai rămân noduri de vizitat (lista este goală).

```
while (toVisit is not empty) {
    Node u = minimumDistance(toVisit);
    remove u from toVisit;
    for (Node v : neighbors of u) {
        if (v is not yet visited) {
            relax edge(u->v);
            add v to toVisit;
        }
        visited[u] = true;
}
```

Corectitudinea algoritmului lui Dijkstra:
Invariant: înainte de o iterație i a algoritmului, pentru toate cele i-1 noduri vizitate până în acel moment a fost găsită distanța minimă (de la sursă la ele), iar pentru toate nodurile rămase nevizitate s-a găsit drumul cu cost minim care trece doar prin noduri vizitate.

Inițializare: inițial, niciun nod nu a fost vizitat, deci toate cele 0 noduri vizitate au distanța minimă față de sursă, și toate nodurile nevizitate au distanța minimă trecând prin 0 noduri.
Menținere: presupun că înainte de iterația i se respectă ipoteza invariantului. În cadrul iterației i, se va extrage din lista de noduri de vizitat nodul u cu distanța cea mai mică față de sursă, deci distTo[u] < distTo[v], unde v este oricare alt nod din listă.
distTo[u] va fi considerată cea mai mică distanță de la sursă la u, deoarece, dacă ar fi existat un alt drum mai scurt, care ar începe de la un nod w nevizitat, rezultă din ipoteză faptul că distTo[w] > distTo[u], lucru care duce la contradicție.
Distanțele către nodurile nevizitate vor rămâne la fel sau vor deveni mai mici prin relaxarea muchiei (u->nodul respectiv), deci vor rămâne cele mai scurte distanțe față de sursă și vor trece doar prin noduri vizitate.

Terminare: Înainte de iterația N+1, unde N reprezintă numărul de noduri, toate nodurile la care se poate ajunge din sursă vor avea găsită (în distTo[node]) distanța minimă față de sursă (lista de noduri nevizitate va fi goală).

Complexitatea algoritmului:
În cadrul algoritmului, fiecare nod u este adăugat în lista de vizitat o singură dată (este scos din listă când se vizitează și considerat vizitat la sfârșitul iterației). Astfel, într-un graf orientat, algoritmul lui Dijkstra va relaxa fiecare muchie (u->v) o singură data.
Pentru a se alege următorul nod de vizitat u, trebuie găsit nodul cu distanța cea mai mică față de sursă, reprezantând o operație de căutare în listă.
Deci, complexitatea temporară a algoritmul va fi: $O(|E|*dk_Q + |V|*em_Q)$ [2], unde dk_Q reprezintă timpul necesar adaugării (și ordonării) nodului v în lista de vizitat și em_Q reprezintă timpul necesar extragerii nodului cu distanța minimă u din lista de vizitat.

Am ales să studiez două implementări ale algoritmului: cu listă și cu priority queue.

Dijkstra implementat cu listă neordonată de noduri de vizitat:
Lista fiind neordonată. adăugarea se va face în timp liniar, deci $dm_Q = 1$, iar extragerea din listă, în cel mai rău caz, va necesita parcurgerea unei întregi liste de |V| elemente, deci $em_Q = |V|$.
De aici rezultă complexitatea temporară a acestei implementări ca fiind $O(|E| + |V|*|V|)$, deci $O(|V|^2)$.

Dijkstra implementat cu priority queue (min heap) pentru nodurile de vizitat:
În acest caz, lista este reprezentată de o coadă de prioritate. Adăugarea unui nod în coadă (astfel încât coada să rămână ordonată) se face în O(logn), deci dm_Q = log |V|, iar extragerea nodului potrivit din coadă se va face tot în O(logn), deci dmq = log |V|. Rezultă complexitatea temporară a acestei implementări ca fiind $O((|E| + |V|) \log |V|)$.

Ambele implementări folosesc $O(|V|^3)$ spațiu adițional (un vector de distanțe distTo[|V|], o coadă ce poate ține |V|-1 noduri în cel mai rău caz și un vector visited[|V|]).

Avantaje: Algoritmul lui Dijkstra este o modalitate rapidă (poate ajunge până la $O(|E|+|V|\log|V|)$ la implementarea cu Fibonacci Heap) de rezolvare a problemei celui mai scurt drum.

Dezavantaje: Algoritmul nu garantează obținerea unui rezultat corect atunci când este aplicat pe un graf ce conține muchii de cost negativ (relaxarea muchiilor se face doar o singură dată, iar în caz de cost negativ ele ar trebui relaxate de mai multe ori).

2.2 Algoritmul Bellman-Ford

Algoritmul se bazează, ca și Dijkstra, pe principiul relaxării. Spre deosebire de Dijkstra, fiecare muchie este relaxată de |V|-1 ori. Astfel, este asigurat rezultatul corect și în cazul muchiilor de cost negativ.

```
for i from 1 to |V|-1:

 for each edge (u, v) with weight w in edges:

  if distance[u] + w < distance[v]:

    distance[v] := distance[u] + w
```

[2]

După finalizarea celor |V|-1 iterații, se mai poate parcurge setul de muchii încă o dată. Dacă unul din drumuri se schimbă, s-a detectat un ciclu de cost negativ.

Corectitudinea algoritmului:
Invariant: înainte de o iterație i a algoritmului, distTo[v] (unde v este un nod oarecare al grafului, diferit de sursă) este ori infinit (dacă încă nu s-a găsit un drum de la sursă la u), ori cel mai mic drum de la sursă la v care trece prin maxim i-1 muchii.

Inițializare: înainte de iterația i = 1, toate distanțele către noduri diferite de sursă sunt infinit.

Menținere: presupun că înainte de iterația i se respectă ipoteza invariantului (toate drumurile către noduri u trec prin maxim i-1 muchii). În cadrul iterației i, distTo[v] ori a fost infinit și va rămâne infinit (nu s-a găsit încă un drum), ori a fost oricât și se va schimba, primind valoarea distTo[u] + w, unde distTo[u] reprezintă un drum care trece prin maxim i-1 muchii și w reprezintă o muchie (deci i muchii în total).
La final iterației i (înainte de iterația i+1), toate drumurile către un nod n vor trece prin maxim i muchii.

Terminare: După finalizarea iterației N-1, unde N reprezintă numărul de noduri, toate nodurile vor avea distanța fie infinit (nu se poate ajunge din sursă în ele), fie costul unui drum care trece prin maxim N-1 muchii.

Complexitatea algoritmului Bellman-Ford:
Întregul set de muchii (E) este vizitat de |V|-1 ori, deci complexitatea temporară a algoritmului va fi ***O(|V|*|E|)***.
Algoritmul folosește ***O(|V|)*** spațiu adițional (pentru vectorul de distanțe distTo[|V|]).

Avantaje: Se poate aplica pe mai multe cazuri față de Dijkstra (ia în considerare posibilitatea costurilor negative și detectează ciclurile de cost negativ) și ocupă mai puțin spațiu adițional.

Dezavantaje: Algoritmul are o complexitate temporară mai mare față de Dijkstra și rulează mult mai lent pentru grafuri dense.

Algoritmul Bellman-Ford bazat pe o coadă de noduri candidat:
Singurele muchii care ar putea duce la o schimbare a distanței nodului u sunt cele care părăsesc un nod a cărei distanțe s-a schimbat la iterația trecută (noduri candidat). Această implementare a Bellman-Ford folosește o coadă pentru a reține nodurile candidat (se iterează pe muchiile ce părăsesc un nod din coadă, spre deosebire de tot setul de muchii). Coada nu va conține noduri candidat duplicate (se va adăuga un nod în coadă doar dacă nu există deja).
Algoritmul nu funcționează la fel de bine ca implementarea normală a Bellman-Ford pentru detectarea ciclurilor negative, deoarece nu are un număr fix de parcurgeri ale setului de muchii.

Complexitatea temporară a acestei implementări este tot ***O(|V|*|E|)*** în cel mai rău caz (la fel ca în prima implementare), dar, în medie, algoritmul va rula în ***O(|E|)*** (în general nu toate nodurile au muchii către toate celelalte noduri).

Spațiul adițional folosit de algoritm este $O(|V|^2)$.

2.3 Algoritm pentru grafuri orientate aciclice bazat pe Topologic Sort

Pentru grafuri orientate aciclice (DAG), putem combina relaxarea cu sortarea topologică pentru a găsi, mai simplu și rapid, drumurile cele mai scurte.
Pașii de urmat sunt:
- găsirea unei ordini topologice a grafului (pornind de la sursă);
- calcularea drumului cel mai scurt către fiecare nod, relaxând fiecare muchie o singură dată.

Deoarece graful aciclic nu conține cicluri, se poate urma un drum de la sursă până la un ultim nod (la care se poate ajunge), fără a avea riscul de a ne întoarce înapoi la un nod pe care deja l-am vizitat.

Corectitudinea algoritmului:
Invariant: înainte de o iterație i a algoritmului, distTo[v] (unde v este un nod oarecare al grafului, diferit de sursă), dacă este diferită de supraestimarea inițială (infinit), este cel mai mic drum de la sursă la v care trece prin maxim i-1 muchii.

Inițializare: înainte de iterația i = 1, toate distanțele către noduri diferite de sursă sunt infinit.

Menținere: presupun că înainte de iterația i se respectă ipoteza invariantului (toate drumurile către noduri u trec prin maxim i-1 muchii). În cadrul iterației i se pot modifica doar distanțele veciniilor nodului evaluat la iterația i-1. Deci, dacă nodul evaluat în mod curent are un drum minim de la sursă, care trece prin i-1 noduri, atunci vecinii lui vor avea un drum care trece prin maxim i-1 muchii + muchia nod_i->vecin (în total i muchii).

Terminare: După finalizarea iterației N-1, unde N reprezintă numărul de noduri, toate nodurile vor avea distanța fie infinit (nu se poate ajunge din sursă în ele), fie costul unui drum care trece prin maxim N-1 muchii.

Complexitatea algoritmului:
Fiecare muchie este relaxată maxim o singură dată și fiecare nod considerat o singură dată, deci complexitatea temporară a algoritmului va fi ***O(|V|+|E|)***.
Algoritmul folosește ***O(|V|)*** spațiu adițional, pentru vectorul de distanțe distTo[|V|] (dacă luăm în considerare și sortarea topologică, folosește $O(|V|^3)$ spațiu adițional).

Avantaje: Este mult mai rapid decât Dijkstra sau Bellman-Ford (timp liniar) și se poate aplica și pe grafuri cu muchii de cost negativ.

Dezavantaje: Algoritmul este aplicat doar pe grafuri ce admit o ordonare topologică.

3 Evaluarea soluțiilor

3.1 Construirea setului de teste

Pentru a compara corectitudinea și viteza soluțiilor am compus un set de 7 cazuri care să cuprindă un număr mare de tipuri de grafuri (toate grafurile considerate sunt orientate):
caz 1 – lanț (graf aciclic) (N = 1000, M = 999);
caz 2 – graf cu M = 0,5*numărul maxim de muchii (N = 1000, M = 499500);
caz 3 – graf cu M = numărul maxim de muchii (N = 1000, M = 999000);
caz 4 – graf cu număr mare de noduri (N = 10000, M = 50000);
caz 5 – graf aciclic cu muchii de cost negativ (N = 1000, M = 2000);
caz 6 – graf cu ciclu de cost negativ (N = 10, M = 20);
caz 7 – graf aciclic cu număr maxim de muchii (N = 1000, M = 499500);

Specificațiile sistemului de calcul pe care au fost rulate testele:
procesor Intel i7-6500U, 2.50GHz, Usable RAM: 7.90GB.

3.2 Tabele și grafice cu rezultatele evaluării:

Table 1. Rezultatul rulării algoritmilor pe lanț

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Corect	16.925241
Dijkstra Priority Queue	Corect	8.937090
Bellman-Ford	Corect	109.484685
Bellman-Ford Queue	Corect	4.963558
Topologic Sort (DAG)	Corect	4.781039

Table 2. Rezultatul rulării algoritmilor pe graf cu nr. mediu de muchii

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Corect	56.315677
Dijkstra Priority Queue	Corect	24.047417
Bellman-Ford	Corect	121.062764
Bellman-Ford Queue	Corect	31.481691
Topologic Sort (DAG)	Greșit	9.326621

Table 3. Rezultatul rulării algoritmilor pe graf cu nr. maxim de muchii

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Corect	87.989368
Dijkstra Priority Queue	Corect	162.420608
Bellman-Ford	Corect	14481.941524
Bellman-Ford Queue	Corect	173.857834
Topologic Sort (DAG)	Greșit	66.795879

Table 4. Rezultatul rulării algoritmilor pe graf cu nr. mai mare de noduri

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Corect	747.587455
Dijkstra Priority Queue	Corect	269.303810
Bellman-Ford	Corect	8265.652253
Bellman-Ford Queue	Corect	448.964918
Topologic Sort (DAG)	Greșit	29.129888

Table 5. Rezultatul rulării algoritmilor pe graf aciclic cu muchii de cost negativ

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Greșit	54.642984
Dijkstra Priority Queue	Greșit	18.392501
Bellman-Ford	Corect	118.547009
Bellman-Ford Queue	Corect	7.552793
Topologic Sort (DAG)	Corect	2.772940

Table 6. Rezultatul rulării algoritmilor pe graf cu ciclu de cost negativ

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Greșit	5.105780
Dijkstra Priority Queue	Greșit	4.275360
Bellman-Ford	Corect	4.715854
Bellman-Ford Queue	Greșit	37134.983954
Topologic Sort (DAG)	Greșit	1.865877

Table 7. Rezultatul rulării algoritmilor pe graf aciclic cu nr. max de muchii

Algoritm	Output	Timp de rulare (ms)
Dijkstra	Corect	145.654576
Dijkstra Priority Queue	Corect	195.835336
Bellman-Ford	Corect	10859.791747
Bellman-Ford Queue	Corect	50.020366
Topologic Sort (DAG)	Corect	48.102735

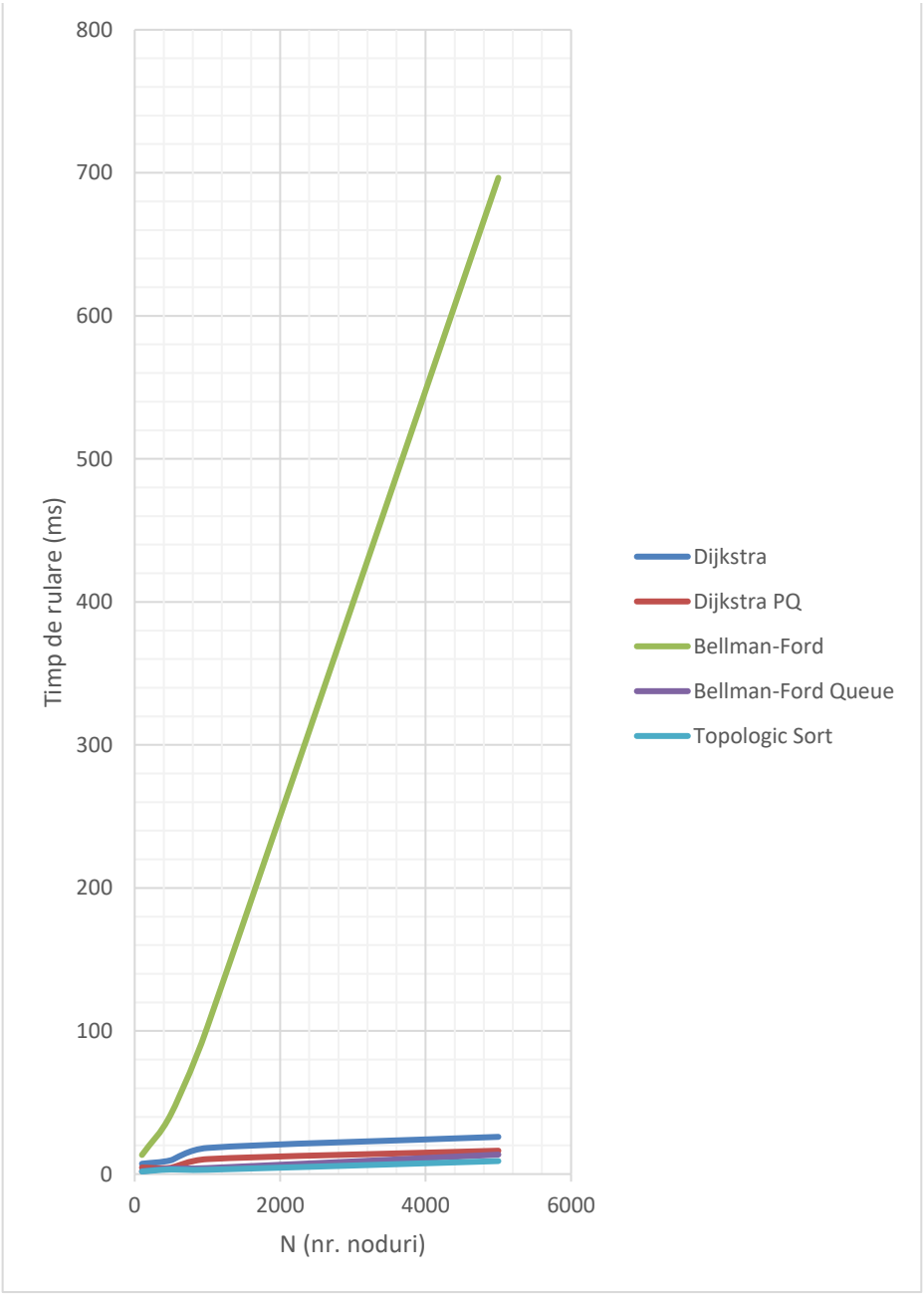

Fig. 1. Compararea timpiilor de rulare a algoritmilor pe graf tip lanț (spars)

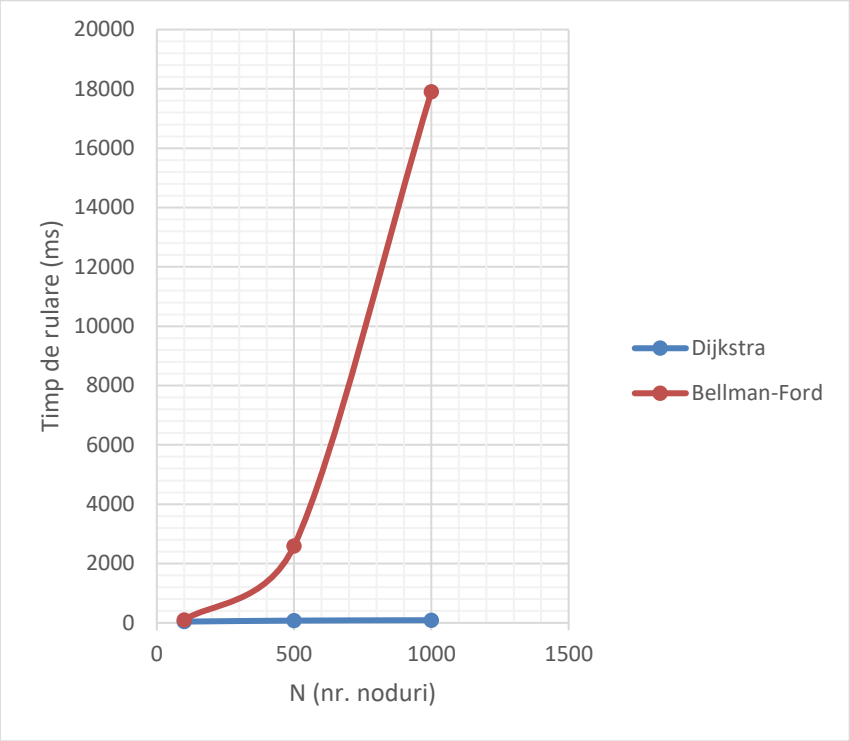

Fig. 2. Timpul de rulare pentru Dijkstra vs Bellman-Ford pentru graf dens

3.3 Interpretarea rezultatelor

Pentru lanțuri (grafuri aciclice) se observă că 4 din cei 5 algoritmi au performanță similară pentru testele considerate. Așa cum ne-am aștepta, algoritmul bazat pe Sortare Topologică este cel mai rapid, urmat de Bellman-Ford cu coadă și Dijkstra (fig 1).
Pentru grafurile ciclice, se observă că algoritmul cu Sortare Topologică nu mai oferă rezultatul corect (nu există o sortare topologică pentru acest tip de graf). Se mai observă faptul că Bellman-Ford devine din ce în ce mai ineficient (comparat cu Dijkstra) pentru grafuri din ce în ce mai dense (fig 2).
În cazul grafurilor cu muchii de cost negativ, Dijkstra nu mai oferă rezultatul corect, iar pentru grafurile cu cicluri de muchii de cost negativ doar Bellman-Ford detectează existența unui astfel de ciclu (deși nu întoarce nici el un rezultat corect, rezultatul fiind inexistent).
În ultimul caz, cel al grafurilor aciclice cu număr mai mare de muchii, se observă, din nou, că algoritmul bazat pe Sortare Topologică este cel mai eficient pentru graf aciclic.

Timpii de rulare obținuți. Aceștia nu sunt exacți, deoarece sunt de ordin mic și există și alte procese care rulează în background în același timp cu algoritmii observați. Ei sunt folosiți doar estimativ.

4 Concluzii

În practică, fiecare din cei 3 algoritmi sunt folosiți pentru a calcula cel mai mic drum în graf.
Dacă este cunoscut tipul de graf și intervalul de valori pentru costul muchiilor, atunci se poate opta pentru algoritmul cel mai eficient în acel caz (atâta timp cât oferă și un rezultat corect).
Bellman-Ford este cel mai versatil dintre cei trei, deoarece întoarce un rezultat corect pe toate tipurile de graf testate. Îl alegem când nu avem nevoie de viteză mare și nu știm dacă graful are sau nu cicluri sau muchii de cost negativ.
Dijkstra este o alegere bună dacă știm că nu vom avea muchii de cost negativ (există o versiune care permită relaxarea repetată a muchiilor și întoarce rezultatul corect și pentru muchii negative) sau dacă avem nevoie de un algoritm rapid.
Algoritmul bazat pe sortare topologică este cel pe care l-aș alege dacă aș știi că nu există cicluri în graf. Dacă nu suntem siguri de acest lucru, atunci nu are rost să-l încercăm.

5 Referințe

1. https://algs4.cs.princeton.edu/44sp/ ultima accesare: 13.12.2018
2. http://www.cse.unt.edu/~tarau/teaching/AnAlgo/Dijkstra's%20algorithm.pdf ultima accesaere: 14.12.2018

6 Bibliografie

- Dijkstra, E. W. (1959). „A note on two problems in connexion with graphs” (http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf, ultima accesare 13.12.2018)
- Drumuri minime, Laboratorul 8 PA (http://elf.cs.pub.ro/pa/wiki/laboratoare/laborator-08, ultima accesare: 13.12.2018)
- Shortest Path Faster Algorithm (https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm, ultima accesare: 12.12.2018)
- Pathing (http://web.cs.unlv.edu/larmore/Courses/CSC269/pathing, ultima accesare: 13.12.2018)
- Topological sorting (https://en.wikipedia.org/wiki/Topological_sorting, ultima accesare: 13.12.2018)