Laborator 09: Drumuri minime

Responsabili:

- Darius-Florentin Neațu (2017-2021) [mailto:neatudarius@gmail.com]
- Radu Nichita (2021) [mailto:radunichita99@gmail.com]

Obiective laborator

- Înțelegerea conceptelor de cost, relaxare a unei muchii, drum minim
- Prezentarea si asimilarea algoritmilor pentru calculul drumurilor minime

Importanţă – aplicaţii practice

Algoritmii pentru determinarea drumurilor minime au multiple aplicații practice si reprezintă clasa de algoritmi pe grafuri cel mai des utilizata:

- Rutare in cadrul unei rețele (telefonice, de calculatoare etc.)
- Găsirea drumului minim dintre doua locații (Google Maps, GPS etc.)
- Stabilirea unei agende de zbor in vederea asigurării unor conexiuni optime
- Asignarea unui peer / server de fișiere in funcție de metricile definite pe fiecare linie de comunicație

Concepte

Costul unei muchii si al unui drum

> Fiind dat un graf orientat G = (V, E), se considera funcția w: E → W, numita funcție de cost, care asociază fiecărei muchii o valoare numerica.

Domeniul funcției poate fi extins, pentru a include si perechile de noduri intre care nu exista muchie directa, caz in care valoarea este $+\infty$.

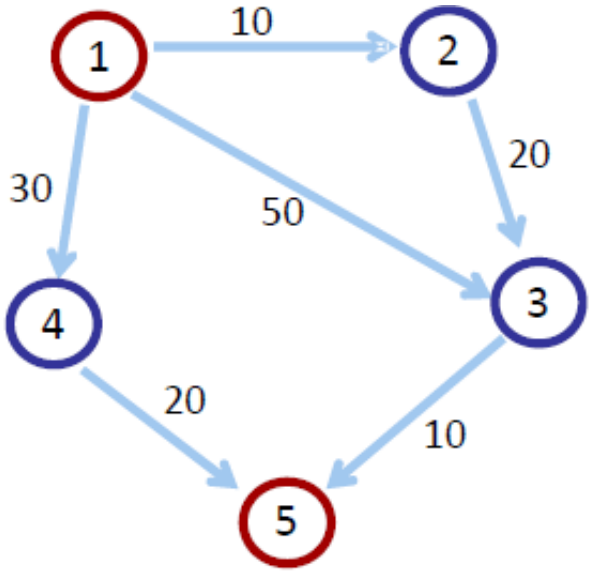
> Costul unui drum format din muchiile p12 p23 ... p(n-1)n, având costurile w12, w23, ..., w(n-1)n, este suma w = w12 + w23 + ... + w(n-1)n.

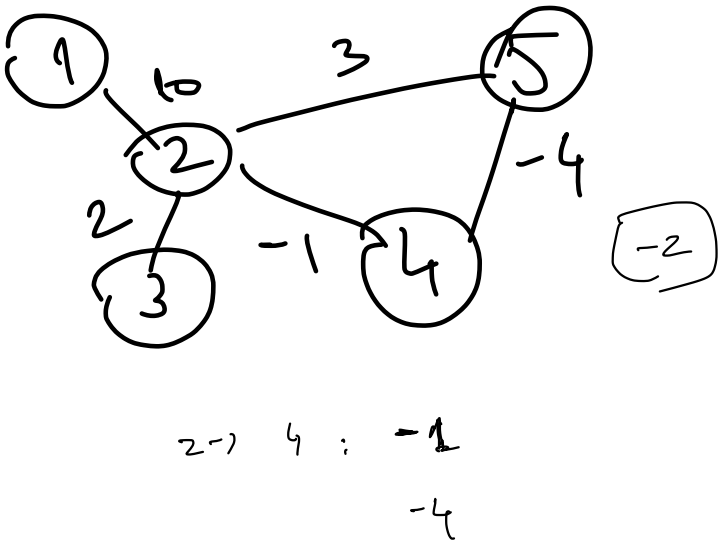
In exemplul alăturat, costul drumului de la nodul 1 la 5 este:

drumul 1: w14 + w45 = 30 + 20 = 50

drumul 2: w12 + w23 + w35 = 10 + 20 + 10 = 40

drumul 3: w13 + w35 = 50 + 10 = 60



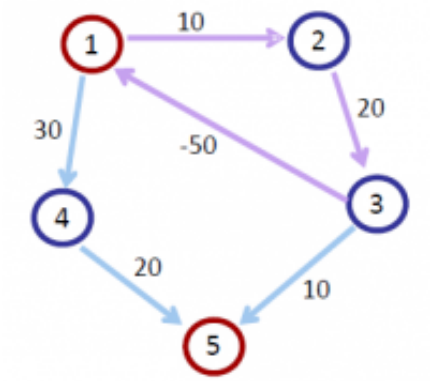


Drumul de cost minim

> Costul minim al drumului dintre doua noduri este minimul dintre costurile drumurilor existente intre cele doua noduri.

In exemplul de mai sus, drumul de cost minim de la nodul 1 la 5 este prin nodurile 2 si 3.

Deși, in cele mai multe cazuri, costul este o funcție cu valori nenegative, exista situații in care un graf cu muchii de cost negativ are relevanta practica. O parte din algoritmi pot determina drumul corect de cost minim inclusiv pe astfel de grafuri. Totuși, nu are sens căutarea drumului minim in cazurile in care graful conține cicluri de cost negativ – un drum minim ar avea lungimea infinita, intrucat costul sau s-ar reduce la fiecare reparcurgere a ciclului:



In exemplul alăturat, ciclul 1 → 2 → 3 → 1 are costul -20.

drumul 1: w12 + w23 + w35 = 10 + 20 + 10 = 40

drumul 2: (w12 + w23 + w31) + w12 + w23 + w35 = -20 + 10 + 20 + 10 = 20

drumul 3: (w12 + w23 + w31) + (w12 + w23 + w31) + w12 + w23 + w35 = -20 + (-20) + 10 + 20 + 10 = 0

Relaxarea unei muchii

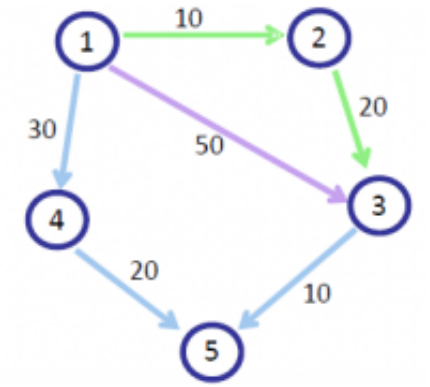
> Relaxarea unei muchii **(u, v)** constă în a testa dacă se poate **reduce distanța** / **costul drumului** de la **sursă s** până la nodul **v**, trecând prin nodul intermediar **u** și apoi circuland pe muchia **(u, v)**.

Fie **w[u][v]** costul inițial al muchiei **(u, v)**, **d[s][u]** costul **drumului** de la sursa **s** la **u** și **d[s][v]** costul **drumului** de la sursa **s** la **v**.

Daca **d[s][v]** > **d[s][u]** + **w[u][v]**, muchia **(u, v)** este **relaxată** și drumul anterior **s** − ... − **v** (care **nu** trece prin **u**) este înlocuit cu drumul **s** − ... − **u** − **v** (care trece prin u și care are cost mai mic).

In exemplul alăturat, sursa este **s** = **1**. Muchia de la 1 la 3 are cost w[1][3] = 50, deci inițial **d[1][3]** = **50**. Analog, **d[1][2]** = **10** și **d[2][3]** = **20**.

Prin relaxarea muchiei **(u, v)** = **(2, 3)**, distanța de la **s** = **1** la **v** = **3** se poate actualiza la **30** (**d[1][3]** > **d[1][2]** + **w[2][3]**).



Toți algoritmii prezentați in continuare se bazează pe relaxare pentru a determina drumul minim.

Drumuri minime de sursa unica

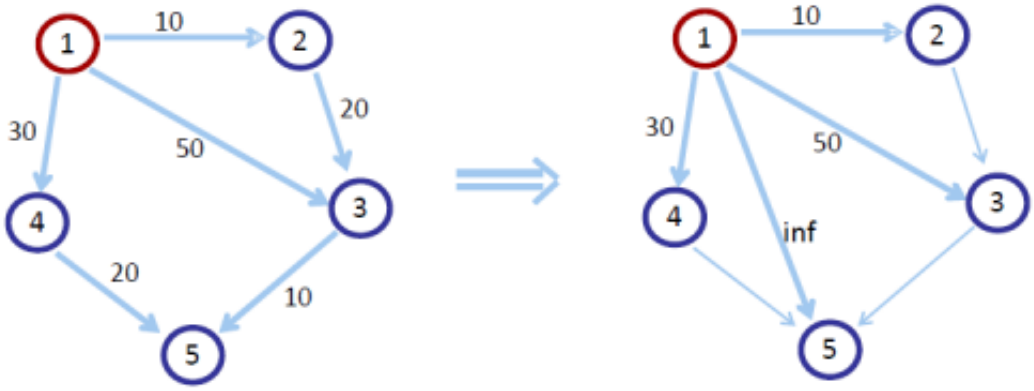
Algoritmii din aceasta secțiune determina drumul de cost minim de la un nod sursa, la restul nodurilor din graf, pe baza de relaxări repetate.

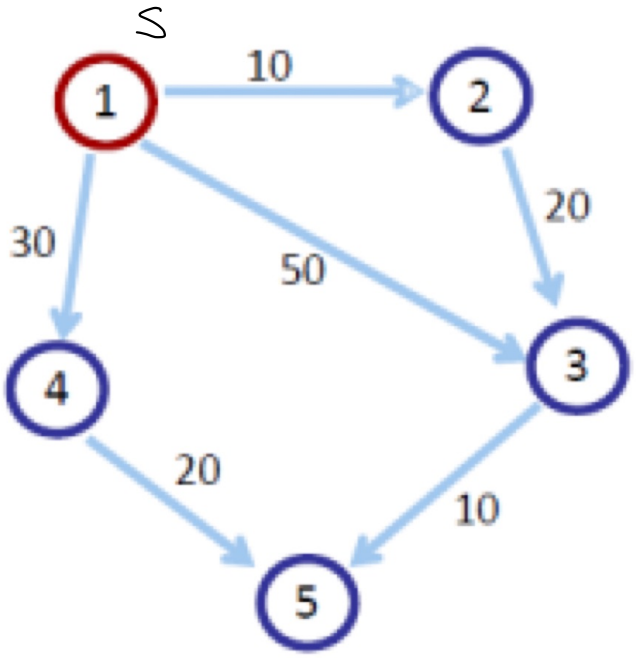
Algoritmul lui Dijkstra

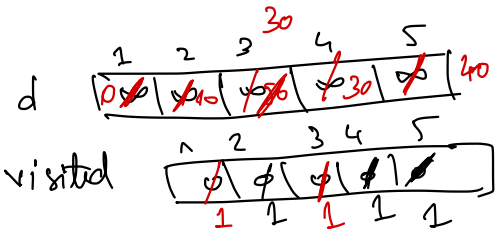
Dijkstra poate fi folosit doar in grafuri care au toate muchiile nenegative.

Algoritmul este de tip Greedy:
optimul local căutat este reprezentat de costul drumului dintre nodul sursa s si un nod v. Pentru fiecare nod se retine un cost estimat d[v], inițializat la început cu costul muchiei s → v, sau cu +∞, daca nu exista muchie.

In exemplul următor, sursa s este nodul 1. Inițializarea va fi:







$H = \{\}$

$H = \{(0,1)\}$

$x = 1$; $H = \{\}$

1: 2 3 4

2: 3

3: 5

4: 5

5: –

$y = 2$, cost = 10, $d[2] > d[1] + cost \Rightarrow d[2] = 10$

$H = \{(10,2)\}$

$y = 3$, cost = 50, $d[3] = 50$

$H = \{(50,3)\ (10,2)\}$

$y = 4$, cost = 30, $d[4] = 30$

$H = \{(30,4), (50,3), (10,2)\}$

$x = 2$, $H = \{(30,4), (50,3)\}$

$y = 3$, cost = 20, $d[3] = 50 > d[2] + cost = 10 + 20 = 30 \Rightarrow d[3] = 30$

$H = \{(30,4), (50,3), (30,3)\}$

$x = 3$, $H = \{(30,4), (50,3)\}$

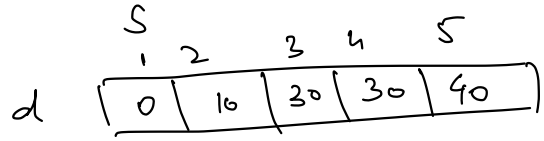
$y = 5$, cost = 10, $d[5] > d[3] + 10 = 40$

$H = \{(30,4)\ (50,3), (40,5)\}$

$x = 4$ $H = \{(50,3), (40,5)\}$

$y = 5$, cost = 20, $d[5] > 30 + 20 = 50$

$x = 5$ $H = \{(50,3)\}$



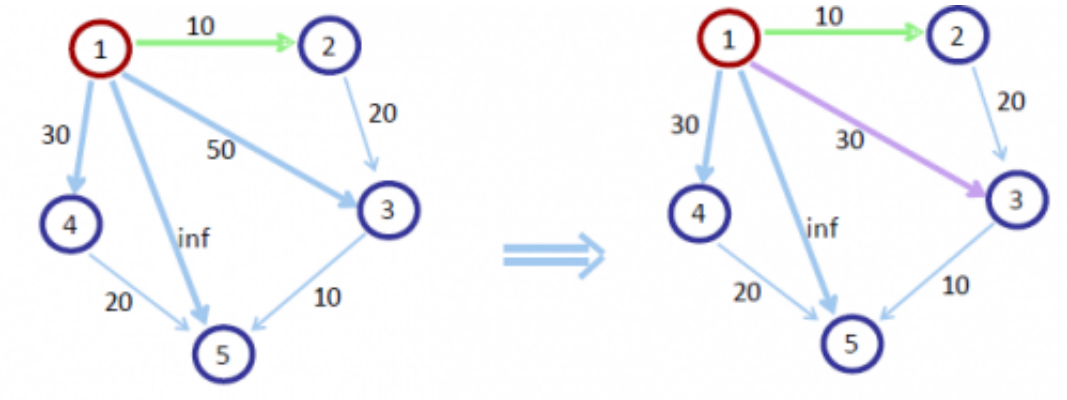
Aceste drumuri sunt îmbunătățite la fiecare pas, pe baza celorlalte costuri estimate.

Algoritmul selectează, in mod repetat, nodul u care are, la momentul respectiv, costul estimat minim (fata de nodul sursa). In continuare, se încearcă sa se relaxeze restul costurilor d[v]. Daca d[v] >= d[u] + wuv , d[v] ia valoarea d[u] + wuv.

Pentru a tine evidenta muchiilor care trebuie relaxate, se folosesc doua structuri: S (mulțimea de vârfuri deja vizitate) si Q (o coada cu priorități, in care nodurile se afla ordonate după distanta fata de sursa) din care este mereu extras nodul aflat la distanta minima. In S se afla inițial doar sursa, iar in Q doar nodurile spre care exista muchie directa de la sursa, deci care au d[nod] < +∞.

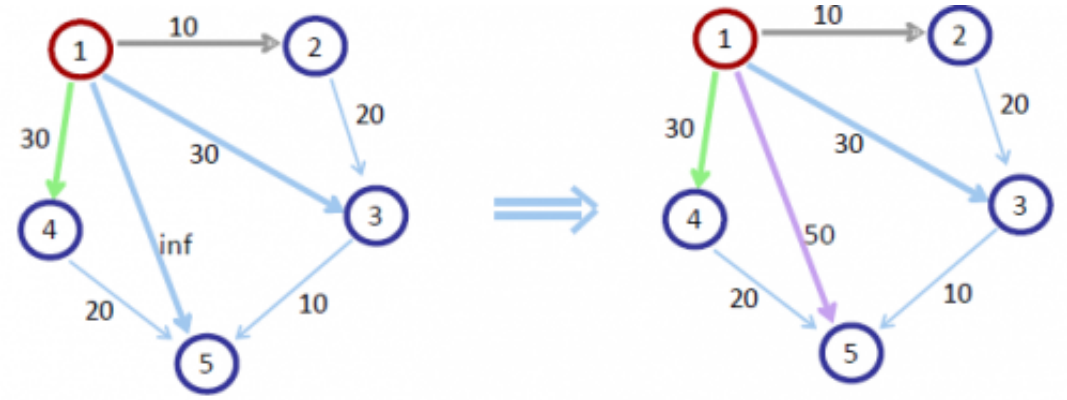
In exemplul de mai sus, vom inițializa S = {1} si Q = {2, 4, 3}.

La primul pas este selectat nodul 2, care are d[2] = 10.
Singurul nod pentru care d[nod] poate fi relaxat este 3 : d[3] = 50 > d[2] + w23 = 10 + 20 = 30



După primul pas, S = {1, 2} si Q = {4, 3}.

La următorul pas este selectat nodul 4, care are d[4] = 30.
Pe baza lui, se poate modifica d[5] : d[5] = +∞ > d[4] + w45 = 30 + 20 = 50



După al doilea pas, S = {1, 2, 4} si Q = {3, 5}.

La următorul pas este selectat nodul 3, care are d[3] = 30, si se modifica din nou d[5]: d[5] = 50 > d[3] + w35 = 30 + 10 = 40.

Algoritmul se încheie când coada Q devine vida, sau când S conține toate nodurile. Pentru a putea determina si muchiile din care este alcătuit drumul minim căutat, nu doar costul sau final, este necesar sa reținem un vector de părinți P. Pentru nodurile care au muchie directa de la sursa, P[nod] este inițializat cu sursa, pentru restul cu null.

Pseudocodul pentru determinarea drumului minim de la o sursa către celelalte noduri utilizând algoritmul

lui Dijkstra este:

```
Dijkstra(sursa, dest):
introdu sursa in Q
d[sursa] = 0
d[nod] = +∞ // pentru orice nod != sursa
P[nod] = null // pentru orice nod din V

// relaxari succesive
cat timp Q nu e vida
    u = extrage_min (Q)
    selectat(u) = true
    foreach nod in vecini[u] // (*)
        /* daca drumul de la sursa la nod prin u este mai mic decat cel curent */
        daca not selectat(nod) si d[nod] > d[u] + w[u, nod]
            // actualizeaza distanta si parinte
            d[nod] = d[u] + w[u, nod]
            P[nod] = u
            /* actualizeaza pozitia nodului in coada prioritara */
            actualizeaza (Q,nod)

// gasirea drumului efectiv
Initializeaza Drum = {}
nod = P[dest]
cat timp nod != null
    insereaza nod la inceputul lui Drum
    nod = P[nod]
```

Reprezentarea grafului ca matrice de adiacenta duce la o implementare ineficienta pentru orice graf care nu este complet, datorita parcurgerii vecinilor nodului u, din linia (*), care se va executa în |V| pași pentru fiecare extragere din Q, iar pe întreg algoritmul vor rezulta |V|^2 pași. Este preferata reprezentarea grafului cu liste de adiacenta, pentru care numărul total de operații cauzate de linia (*) va fi egal cu |E|. Complexitatea algoritmului este O(|V|^2+|E|) în cazul în care coada cu priorități este implementata ca o căutare liniara. În acest caz funcția extrage_min se executa în timp O(|V|), iar actualizează(Q) in timp O(1).

O varianta mai eficienta este implementarea cozii ca heap binar. Funcția extrage_min se va executa în timp O(lg|V|); funcția actualizează(Q) se va executa tot în timp O(lg|V|), dar trebuie cunoscuta poziția cheii nod în heap, adică heapul trebuie sa fie indexat. Complexitatea obținută este O(|E|lg|V|) pentru un graf conex.

Cea mai eficienta implementare se obține folosind un heap Fibonacci pentru coada cu priorități:

Aceasta este o structura de date complexa, dezvoltata în mod special pentru optimizarea algoritmului Dijkstra, caracterizata de un timp amortizat de O(lg|V|) pentru operația extrage_min si numai O(1) pentru actualizeaza(Q). Complexitatea obținută este O(|V|lg|V| + |E|), foarte bună pentru grafuri rare.

Algoritmul Bellman – Ford

Algoritmul Bellman Ford poate fi folosit si pentru grafuri ce conțin muchii de cost negativ, dar nu poate fi folosit pentru grafuri ce conțin cicluri de cost negativ (când căutarea unui drum minim nu are sens).
Cu ajutorul sau putem afla daca un graf conține cicluri. Algoritmul folosește același mecanism de relaxare ca si Dijkstra, dar, spre deosebire de acesta, nu optimizează o soluție folosind un criteriu de optim local, ci parcurge fiecare muchie de un număr de ori egal cu numărul de noduri si încearcă sa o relaxeze de fiecare data, pentru a îmbunătăți distanta până la nodul destinație al muchiei curente.

Motivul pentru care se face acest lucru este ca drumul minim dintre sursa si orice nod destinație poate sa treacă prin maximum |V| noduri (adică toate nodurile grafului), respectiv |V|-1 muchii; prin urmare, relaxarea tuturor muchiilor de |V|-1 ori este suficienta pentru a propaga până la toate nodurile informația despre distanta minima de la sursa.

Daca, la sfârșitul acestor |E|*(|V|-1) relaxări, mai poate fi îmbunătățită o distanță, atunci graful are un ciclu de cost negativ si problema nu are soluție.

Menținând notațiile anterioare, pseudocodul algoritmului este:

```
BellmanFord(sursa):

d[sursa] = 0
d[nod] = +∞ // pentru orice nod != sursa
p[nod] = null // pentru orice nod din V

// relaxari succesive
// cum in initializare se face o relaxare (daca exista drum direct de la sursa la nod =>
// d[nod] = w[sursa, nod]) mai sunt necesare |V-1| relaxari
for i = 1 to |V|-1
    foreach (u, v) in E  // E = multimea muchiilor
        daca d[v] > d[u] + w(u,v)
            d[v] = d[u] + w(u,v)
            p[v] = u;

// daca se mai pot relaxa muchii
foreach (u, v) in E
    daca d[v] > d[u] + w(u,v)
        fail ("exista cicluri negativ")
```

Complexitatea algoritmului este O(|E|*|V|).

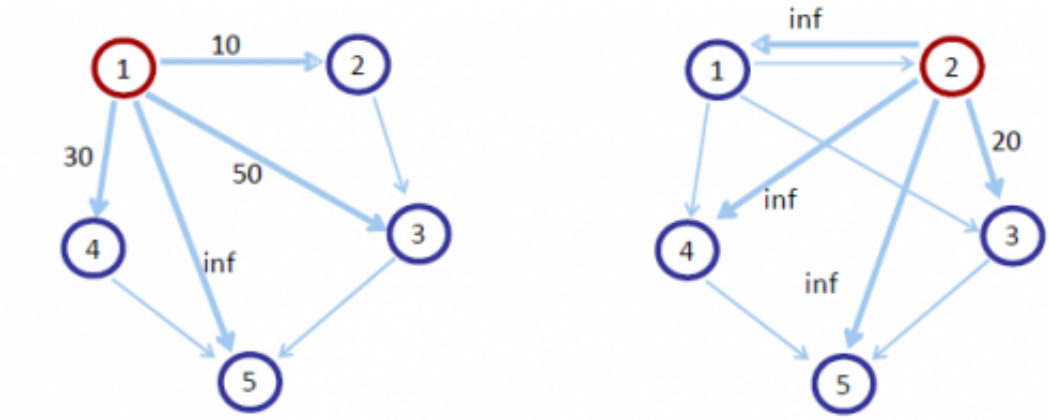
Drumuri minime intre oricare doua noduri

Floyd-Warshall

Algoritmii din aceasta secțiune determina drumul de cost minim dintre oricare doua noduri dintr-un graf. Pentru a rezolva aceasta problema s-ar putea aplica unul din algoritmii de mai sus, considerând ca sursa fiecare nod, pe rând, dar o astfel de abordare ar fi ineficienta.

Algoritmul Floyd-Warshall(intalnit si sub numele de Roy-Floyd) compara toate drumurile posibile din graf dintre fiecare 2 noduri, si poate fi utilizat si in grafuri cu muchii de cost negativ.

Estimarea drumului optim poate fi reținut intr-o structura tridimensionala d[v1, v2, k], cu semnificația - costul minim al drumului de la v1 la v2, folosind ca noduri intermediare doar noduri pana la nodul k. Daca nodurile sunt numerotate de la 1, atunci d[v1, v2, 0] reprezintă costul muchiei directe de la v1 la v2, considerând +∞ daca aceasta nu exista. Exemplu, pentru v1 = 1, respectiv 2:



Pornind cu valori ale lui k de la 1 la |V|, ne interesează să găsim cea mai scurta cale de la fiecare v1 la fiecare v2 folosind doar noduri intermedire din mulțimea {1, ..., k}. De fiecare data, comparam costul deja estimat al drumului de la v1 la v2, deci d[v1, v2, k-1] obținut la pasul anterior, cu costul drumurilor de la v1 la k si de la k la v2, adică d[v1, k, k-1] + d[k, v2, k-1], obținutae la pasul anterior. Atunci, d[v1, v2, |V|]

va conține costul drumului minim de la v1 la v2.

Pseudocodul acestui algoritm este:

```
FloydWarshall(G):
n = |V|
int d[n, n, n]
foreach (i, j) in (1..n,1..n)
    d[i, j, 0] = w[i,j] // costul muchiei, sau infinit
for k = 1 to n
    foreach (i,j) in (1..n,1..n)
        d[i, j, k] = min(d[i, j, k-1], d[i, k, k-1] + d[k, j, k-1])
```

Complexitatea temporala este O(|V|^3), iar cea spațială este tot O(|V|^3). O complexitate spațială cu un ordin mai mic se obține observând ca la un pas nu este nevoie decât de matricea de la pasul precedent d[i, j, k-1] si cea de la pasul curent d[i, j, k]. O observație și mai bună este că, de la un pas k-1 la k, estimările lungimilor nu pot decât sa scadă, deci putem sa lucram pe o singura matrice. Deci, spațiul de memorie necesar este de dimensiune |V|^2.

Rescris, pseudocodul algoritmului arata astfel:

```
FloydWarshall(G):
n = |V|
int d[n, n]
foreach (i, j) in (1..n,1..n)
    d[i, j] = w[i,j] // costul muchiei, sau infinit
for k = 1 to n
    foreach (i,j) in (1..n,1..n)
        d[i, j] = min(d[i, j], d[i, k] + d[k, j])
```

Pentru a determina drumul efectiv, nu doar costul acestuia, avem doua variante:

1. Se retine o structura de părinți, similara cu cea de la Dijkstra, dar, bineînțeles, bidimensionala.
2. Se folosește divide et impera astfel:

- se caută un pivot k astfel încât cost[i][j] = cost[i][k] + cost[j][k]
- se apelează funcția recursiv pentru ambele drumuri → (i,k),(k,j)
- dacă pivotul nu poate fi găsit, afișăm i
- după terminarea funcției recursie afișăm extremitatea dreapta a drumului

TL;DR

1. Daca avem un graf **neorientat**, **fara cicluri** (un arbore), exista un singur drum intre oricare doua noduri, care poate fi aflat printr-o simpla parcurgere DFS. Folosind diferite preprocesari [8],[9], putem calcula distanta intre oricare doua noduri in timp constant, O(1).

2. Daca avem un graf **orientat**, **fara cicluri** (un DAG [10]), putem sa sa relaxam muchiile nodurilor, parcurgandu-le pe acestea in ordinea data de sortarea topologica. O(|V|+|E|)

3. Daca avem un graf unde toate muchiile au **cost egal**, putem afla distanta minima de la un nod sursa la orice alt nod printr-o parcurgere BFS. (de asemenea, tinand cont de faptul ca pot exista mai multe drumuri pana la un anumit nod). O(|V|+|E|)

4. Pentru grafuri orientate, **rare** (relativ putine muchii), putem folosi algoritmul lui Johnson([11]) pentru calcularea distantei minime de la un nod, la oricare alt nod. O(|V|^2log|V| + |V||E|)

Concluzii

- **Dijkstra***

- calculează drumurile minime de la o sursa către celelalte noduri
- nu poate fi folosit daca exista muchii de cost negativ
- complexitate minima O(|V|lg|V| + |E|) utilizând heapuri Fibonacci;

- **Bellman - Ford**

- calculează drumurile minime de la o sursă către celelalte noduri
- detectează existența ciclurilor de cost negativ
- complexitate O(|V| * |E|)

- **Floyd - Warshall**

- calculează drumurile minime intre oricare doua noduri din graf
- poate fi folosit in grafuri cu cicluri de cost negativ, dar nu le detectează
- complexitate O(|V|^3)

Exercitii

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab09 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab09].

Inainte de a rezolva exercitiile, asigurati-va ca ati citit si inteles toate precizarile din sectiunea Precizari laboratoare 07-12 [https://ocw.cs.pub.ro/courses/pa/skel_graph].

Prin citirea acestor precizari va asigurati ca:

- cunoasteti **conventiile** folosite
- evitati **buguri**
- evitati **depunctari** la lab/teme/test

Dijkstra

Se da un graf **orientat** cu **n** noduri si **m** arce. Graful are pe arce **costuri pozitive**.

Folositi **Dijkstra** pentru a gasi **costul minim** (**lungimea minima**) a unui drum de la o sursa data (**source**) la toate celelalte n - 1 noduri din graf.

Costul / lungimea unui drum este suma costurilor/lungimilor arcelor care compun drumul.

Restrictii si precizari:

- $n <= 50.000$
- $m <= 2.5 * 10^5$
- $0 <= c <= 20.000$, unde c este costul/lungimea unui arc
- timp de executie
    - C++: 1s
    - Java: 2s

Rezultatul se va returna sub forma unui vector **d** cu **n** + **1** elemente.

Conventie:

- **d[node]** = costul minim / lungimea minima a unui drum de la **source** la nodul **node**
- **d[source]** = **0**
- **d[node]** = **-1** , daca nu se poate ajunge de la **source** la **node**

d[0] nu este folosit, deci ca fi initializat cu 0! (am pastrat indexarea nodurilor de la 1)

Bellman-Ford

Se da un graf **orientat conex** cu **n** noduri si **m** arce. Graful are pe arce **costuri pozitive sau negative**.

Folositi **Bellman-Ford** pentru a gasi **costul minim** (**lungimea minima**) a unui drum de la o sursa data (**source**) la toate celelalte n - 1 noduri din graf. In caz ca se va detecta un ciclu de cost negativ, se va semnala acest lucru.

Costul / lungimea unui drum este suma costurilor/lungimilor arcelor care compun drumul.

Restrictii si precizari:

- $n <= 50.000$
- $m <= 2.5 * 10^5$
- $-1.000 <= c <= +1.000$, unde c este costul/lungimea unui arc
- timp de executie
    - C++: 1s
    - Java: 2s
- Pentru punctaj maxim, implementarea din laborator trebuie sa treaca toate testele, cu exceptia testelor 8 si 9, pe care va lua TLE. Pentru cei curiosi, exista si o implementare mai eficienta a algoritmului, oarecum similara cu cea de la Dijkstra (pentru mai multe detalii: https://infoarena.ro/problema/bellmanford [https://infoarena.ro/problema/bellmanford])

Rezultatul se va returna sub forma unui vector **d** cu **n** + **1** elemente.

Conventie:

- **d[node]** = costul minim / lungimea minima a unui drum de la **source** la nodul **node**
- **d[source]** = **0**
- **d[node]** = **-1** , daca nu se poate ajunge de la **source** la **node**

d[0] nu este folosit, deci ca fi initializat cu 0! (am pastrat indexarea nodurilor de la 1)

ATENTIE!!! Este posibil ca un astfel de graf sa aiba ciclu de cost negativ. In cazul detectarii unui ciclu de cost negativ, functia voastra va returna un vector gol! (**std::vector<int>()** / **{}** sau **new ArrayList<Integer>()**).

RoyFloyd

Se da un graf **orientat** cu **n** noduri. Graful are **costuri pozitive** pe arce.

Se da **matricea ponderilor** , se cere **matricea drumurilor minime**.

Restrictii si precizari:

- $n <= 100$
- $0 <= c <= 1.000$, unde c este costul unui arc

- daca **nu exista muchie** intre o pereche de noduri x si y, distanta de la nodul x la nodul y din **matricea ponderilor** va fi 0
- daca dupa aplicarea algoritmului **nu se gaseste drum** pentru o pereche de noduri x si y, se va considera **distanta** dintre ele egala cu 0 (se stocheaza in **matricea distantelor** valoarea 0)
- drumul de la nodul i la nodul i are lungime 0 (prin conventie)
- timp de executie
    - C++: 1s
    - Java: 2s

Rezultatul se va **stoca** in matricea **d** declarata in schelet! Algoritmul vostru trebuie doar sa o populeze corect, tinand cont ca nodurile sunt indexate de la 1.

BONUS

Pentru exercitiul cu Dijkstra, **reconstituiti** drumul de lungime minima **source** la celelalte noduri din graf.

Extra

Rezolvati problema rfinv [https://infoarena.ro/problema/rfinv] pe infoarena.
Rezolvati problema coach [https://infoarena.ro/problema/coach] pe infoarena.
Rezolvati problema rf [https://infoarena.ro/problema/rf] pe infoarena.
Rezolvati problema TODO [https://infoarena.ro/problema/TODO] pe infoarena.

Referințe:

[1] http://en.wikipedia.org/wiki/Dijkstra's_algorithm [http://en.wikipedia.org/wiki/Dijkstra's_algorithm]

[2] http://en.wikipedia.org/wiki/Bellman-Ford_algorithm [http://en.wikipedia.org/wiki/Bellman-Ford_algorithm]

[3] http://www.algorithmist.com/index.php/Floyd-Warshall's_Algorithm [http://www.algorithmist.com/index.php/Floyd-Warshall's_Algorithm]

[4] http://en.wikipedia.org/wiki/Binary_heap [http://en.wikipedia.org/wiki/Binary_heap]

[5] http://en.wikipedia.org/wiki/Fibonacci_heap [http://en.wikipedia.org/wiki/Fibonacci_heap]

[6] T. Cormen, C. Leiserson, R. Rivest, C. Stein – Introducere în Algoritmi

[7] C. Giumale – Introducere în analiza algoritmilor

[8] http://en.wikipedia.org/wiki/Range_Minimum_Query [http://en.wikipedia.org/wiki/Range_Minimum_Query]

[9] http://en.wikipedia.org/wiki/Lowest_common_ancestor [http://en.wikipedia.org/wiki/Lowest_common_ancestor]

[10] http://en.wikipedia.org/wiki/Directed_acyclic_graph [http://en.wikipedia.org/wiki/Directed_acyclic_graph]

[11] http://en.wikipedia.org/wiki/Johnson%27s_algorithm
[http://en.wikipedia.org/wiki/Johnson%27s_algorithm]

pa/laboratoare/laborator-09.txt · Last modified: 2021/05/19 11:21 by darius.neatu