Drumuri minime in graf

Raileanu Ana Arina

Universitatea Politehnica Bucuresti

Abstract. Acest document reprezinta o comparatie intre algoritmii de gasire a drumurilor minime intre oricare doua noduri dintr-un graf.

Cuprins

1 Introducere

1.1 Descrierea Problemei

O problema mult studiata, avand in vedere numarul de publicatii pe aceasta tema, este cea a aflarii drumurilor minime intre oricare doua noduri dintr-un graf. Practic, pornind de la un graf G cu V noduri si E muchii ne dorim sa aflam distanta minima intre orice noduri ale grafului. Unul dintre motivele pentru care se cauta metode cat mai eficiente de rezolvare a acestei probleme este aplicabilitatea foarte mare la diverse probleme din viata reala.

1.2 Exemple de aplicatii practice

O utilizare a drumurilor minime ar fi in sistemele GPS sau alte programe de optimizare a drumurilor, de exemplu pentru o companie care are de facut transporturi in mai multe orase. Simplist, in aceste situatii putem considera costul unei muchii ca fiind distanta dintre doua zone. O alta aplicatie in sistemele de servicii urbane este pentru a determina cat de aglomerat este un drum la o anumita ora sau care ar fi locul optim de amplasare a unei facilitati. Inclusiv in jocurile video ne intalnim cu problema drumurilor minime, cand avem un traseu de urmat sau cand inamicii isi calculeaza o strategie cat mai buna. De asemenea, acesti algoritmi sunt folosit si pentru a analiza retelele sociale, mai exact pentru a calcula proprietatea de "betweeness" intre actori [1]. Algoritmii APSP (All-Pairs Shortest Path) sunt utilizati si in analiza altor retele, cum ar fi a retelelor de calculatoare.

1.3 Specificarea Solutiilor Alese

Vom alege trei algoritmi bine-cunoscuti pentru rezolvarea problemei drumurilor minime intre oricare doua noduri, ceea ce ne va permite o comparare a eficientei si corectitudinii lor in diverse cazuri.

Algoritmul Floyd-Warshall Este un algoritm clasic, de complexitate $O(V^3)$ care poate fi folosit atat pe grafuri numai cu costuri pozitive, cat si pe grafuri cu costuri negative. In general, merita utilizat cand numarul de muchii este de ordinul patratului numarului de noduri, adica avem un graf dens.

Algoritmul Bellman-Ford Desi de obicei se foloseste pentru rezolvarea problemelor SSSP (Single Source Shortest Path), poate fi aplicat si pentru gasirea drumurilor minime intre oricare doua noduri, prin executarea lui pentru fiecare nod din graf. In implementare vom folosi o coada, adaugand in ea doar nodurile pentru care s-a imbunatatit distanta minima la un anumit pas.

Algoritmul Johnson Acest algoritm foloseste atat Bellman-Ford cat si Dijkstra, iar ideea care sta in spatele lui este de a reface costurile de pe muchiile grafului, pentru a nu mai avea muchii de cost negativ. Dupa refacere putem aplica Dijkstra. In cazul grafurilor rare, se comporta mult mai bine decat Floyd-Warshall.

1.4 Criterii de evaluare

Pentru fiecare algoritm vom testa atat corectitudinea lui, cat si eficienta.

Testarea corectitudinii In principiu ne dorim sa vedem ca algoritmul returneaza rezultatul bun pentru diverse categorii de grafuri. De aceea, vom crea teste pentru grafuri rare, dense, orientate, neorientate, cu muchii de cost negativ/pozitiv, ciclice (cu cicluri negative/pozitive), aciclice etc. Dupa aceea, vom compara matricele de distanta obtinute de cei 3 algoritmi. Eventual se poate face o comparatie si cu rezultatele obtinute de algoritmul lui Dijkstra, pentru grafuri cu muchii pozitive.
Mai mult, corectitudinea poate fi demonstrata prin inductie.

Testarea eficientei Folosind setul de teste pentru testarea corectitudinii, teste generate aleator si teste de tip worst case/best case, vom masura timpul de executie al programului si vom calcula memoria necesara.
Pentru timpul de executie fie folosim biblioteca ctime, fie chrono pentru o precizie mai mare. In final, vom reprezenta duratele in functie de numarul de noduri/muchii intr-un grafic pentru testele generate aleator pentru a compara eficienta celor 3 algoritmi pe cazuri intamplatoare. De asemenea, avand in vedere formulele de complexitate si modul de functionare al algoritmilor ne vor interesa situatiile in care un algoritm este mai avantajos decat un altul (de exemplu in cazul grafurilor rare/dense) si vom face cate un grafic pentru fiecare situatie de acest gen, cu teste ce respecta o anumita proprietate si au numar variabil de noduri/muchii.

2 Prezentarea solutiilor

In continuare vom prezenta cei 3 algoritmi specificati pentru rezolvarea problemei.

2.1 Descrierea algoritmilor

Algoritmul Floyd-Warshall Este un algoritm de programare dinamica. Pentru a intelege mai bine acest algoritm este util sa privim formula de recurenta.

$$bestPath_{ij}^{(k)} = \begin{cases} distance_{ij}, k = 0 \\ min(bestPath_{ij}^{(k-1)}, bestPath_{ik}^{(k-1)} + bestPath_{kj}^{(k-1)}, k >= 1 \end{cases}$$

Algoritmul este format din 3 bucle for, una pentru k, una pentru i si una pentru j. Ideea care sta in spatele algoritmului este ca la o iteratie prin bucla exterioara construim cele mai scurte drumuri intre nodurile din buclele interioare, trecand doar prin nodurile definite de bucla exterioara. Mai exact, in primul for alegem o submultime de noduri, $K = \{1, 2, ..., k\}$, al carei cardinal va creste de fiecare data cu 1, dupa care luam pe rand toate perechile (i, j) de noduri apartinand multimii de noduri V. Daca exista un drum de la nodul i la nodul j trecand prin noul nod adaugat la submultimea K, si acest drum este mai scurt decat drumul gasit anterior, dintr-o submultime mai mica, atunci actualizam distanta pentru perechea noastra. Un pseudocod care descrie acest algoritm este prezentat mai jos[2].

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  D = W
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              d_ij = min(d_ij, d_ik + d_kj)
7  return D
```

Ordinea celor 3 bucle for este relevanta, deoarece schimbarea acestora va duce la un calcul diferit. De exemplu, daca in loc de k, i, j ar fi i, j, k progamul ar fi echivalent cu calculul distantei minime dintre i si j cu doar un nod intermediar. Algoritmul nu verifica ciclurile de cost negativ.

Algoritmul Bellman-Ford Este folosit in general pentru rezolvarea problemei SSSP, adica distanta minima de la un nou la toate celelalte. Algoritmul pentru aceasta problema este prezentat mai jos[2].

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Se alege un nod sursa, s, si se calculeaza distantele minime plecand din el si ajungand in celelalte noduri. Pseudocodul este executat pentru un graf cu costuri, reprezentate prin $w(u, v)$. Sensul celor doua bucle for este ca dupa k iteratii prin bucla exterioara, distanta pana la nodul v va fi distanta minima intre sursa si nod, cu cel mult k muchii.
Pentru a extinde acest algoritm la cazul APSP mai cream o bucla in care apelam de $|V|$ ori functia, considerand pe rand fiecare nod ca fiind sursa. Calculam vectorul de distante corespunzatoare unui nod pe care il adaugam dupa la matricea noastra de distante.
Daca dorim sa facem acest algoritm si mai eficient din punct de vedere al timpului de executie folosim o coada si inlocuim bucla for exterioara cu un $while$. Astfel, cat timp avem noduri in coada extragem un nod si incercam sa imbunatatim distantele catre vecinii lui. Cand reusim sa gasim o distanta mai buna catre unul dintre vecini e o posibilitate sa putem actualiza si alte distante, deci inseram vecinul in coada. In concluzie, vom face mai putine verificari si incercam sa actualizam distantele doar cand are sens.
Verificarea faptului ca nu exista cicluri de cost negativ se face intre liniile 5-8. O alta metoda de verificare este numararea aparitiilor unui nod in bucla. Daca apare de un numar mai mare decat numarul de noduri, este clar ca am gasit un ciclu negativ.

Algoritmul Johnson Este un algoritm care are in componenta sa alti doi algoritmi: Dijkstra si Bellman-Ford. Acesta poate fi folosit pe grafuri cu muchii de cost negativ deoarece foloseste tehnica de "reweight", adica modificare a costului muchiilor astfel incat sa aiba numai muchii pozitive, urmand ca la sfarsitul executiei sa se faca operatia inversa. Mai jos este prezentat pseudocodul lui[2].

```
JOHNSON(G, w)
 1  compute G', where G'.V = G.V ∪ {s},
        G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
 2  if BELLMAN-FORD(G', w, s) == FALSE
 3      print "the input graph contains a negative-weight cycle"
 4  else for each vertex v ∈ G'.V
 5          set h(v) to the value of δ(s, v)
                computed by the Bellman-Ford algorithm
 6      for each edge (u, v) ∈ G'.E
 7          ŵ(u, v) = w(u, v) + h(u) − h(v)
 8      let D = (d_uv) be a new n × n matrix
 9      for each vertex u ∈ G.V
10          run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11          for each vertex v ∈ G.V
12              d_uv = δ̂(u, v) + h(v) − h(u)
13      return D
```

Pentru acest algoritm este necesar sa modificam graful initial astfel: adaugam un nou nod sursa, s, care va avea muchii catre toate celelalte noduri din graful initial. Aceste muchii noi adaugate vor avea costul 0. Folosim algoritmul Bellman-Ford in varianta sa originala, SSSP, pornind din noul nod. In cazul in care avem un ciclu negativ, intrerupem executia programului. In continuare, luam fiecare nod din graful nou si ii schimbam costul, astfel incat sa nu mai avem nicio muchie de cost negativ. Schimbarea de cost se face cu ajutorul distantelor calculate de algoritmul Bellman-Ford cu sursa creata (linia 5). Pe noul graf se poate aplica Dijkstra, explicat in sectiunea urmatoare, pornind din fiecare nod. La final, deoarece Dijkstra functioneaza numai pe grafuri cu muchii pozitive, trebuie sa refacem matricea reala a distantelor, deci anulam modificarea facuta anterior (linia 12).

Algoritmul Dijkstra (bonus) Este un algoritm de tip SSSP ce poate fi modificat pentru a rezolva probleme de tip APSP. Pseudocodul pentru acest algoritm este prezentat mai jos[2]. In aceasta varianta avem o multime S care retine toate

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

nodurile pentru care am calculat deja cu siguranta distanta minima posibila de la sursa la ele (in aceasta multime stim sa nu mai cautam). In coada Q (min-heap/priority-queue) inseram initial toate nodurile si extragem pe rand nodul cel mai apropiat de sursa. Astfel, $Q = V - S$. Dupa ce luam un nod din coada ii verificam toti vecinii si incercam sa gasim o distanta mai buna pentru ei.
In implementarea mea pornim cu o coada de prioritati de perechi nod-distanta goala, cu regula de sortare dupa distanta minima catre nodul sursa. Initial introducem sursa in coada si de fiecare data cand reusim sa imbunatatim distanta pentru un nod il introducem si pe el in coada. La extragerea unui nod continuam cu actualizarea vecinilor numai daca distanta cu care a fost inserat in coada nu este mai mare decat distanta lui minima in acel moment.
La fel ca si in cazul Bellman-Ford, pentru a aplica algoritmul in rezolvarea problemei APSP pornim cu fiecare nod al grafului drept radacina.

2.2 Analiza Complexitatii Solutiilor

Algoritmul Floyd-Warshall Luand in considerare faptul ca lucram pe liste de adiacenta, avem o etapa de initializare in care cream matricea de distante astfel: luam fiecare nod din graf, iar pentru un nod luam setul de muchii adiacente lui si punem costul fiecarei muchii in matricea de distante. Aceasta operatie va consuma un timp $O(E)$, deoarece practic luam fiecare muchie din graf si atribuirea se face in timp constant. Initial matricea este declarata sub forma unui vector din stl cu valoare inf pe fiecare pozitie, iar o matrice o aiba dimensiunea $|V| \cdot |V|$, intrucat ne intereseaza distantele minime intre oricare doua noduri. Pentru calcularea distantelor sunt 3 for-uri imbricate fiecare cu mergand de la 1 la $n = |V|$. In cadrul ultimului for se face operatia de calculare a minimului, in timp constant, precum si o atribuire. Astfel, complexitatea temporala a algoritmului este $O(E + V^3)$ si se poate reduce la $O(V^3)$.
Complexitatea spatiala a algoritmului poate fi considerata drept spatiul ocupat de matricea de rezultate, adica $O(V^2)$. Avand in vedere faptul ca poate fi rezolvat iterativ, este nevoie de un singur stack call.

Algoritmul Bellman-Ford Pornim de la algoritmul clasic, SSSP. Avem doua for-uri imbricate, cel exterior mergand de la 1 la $n = |V| - 1$ si cel interior iterand prin fiecare muchie a grafului. Relaxarea unei muchii se face in timp constant, deoarece consta intr-o verificare si o atribuire a unei valori. Astfel, este clar ca timpul este dat de iteratia prin cate E muchii de cate n ori, deci $O(E \cdot V - V) = O(E \cdot V)$. Avand in vedere faptul ca aplicam Bellman-Ford pe fiecare nod al grafului, ajungem la o complexitate totala $O(V^2 \cdot E)$. Folosind o coada putem face algoritmul mai eficient, dar complexitatea in cazul cel mai rau va fi in continuare egala cu cea calculata mai devreme.
Pe langa spatiul suplimentar folosit de matricea de distante, $O(V^2)$ folosind o coada putem avea un caz in care actualizam distantele foarte multor noduri in acelasi timp, astfel poate fi ocupata o memorie aditionala $O(V)$. In plus, avand in vedere ca verificam de cate ori inseram un nod in coada, vom folosi inca $O(V)$ pentru a retine aceste date.

Algoritmul Johnson Avand in vedere ca a fost nevoie sa copiam graful initial si sa adaugam un nou nod la el, aceasta modificare o sa ocupe un timp $O(V)$. Executarea algoritmului Bellman-Ford pentru noua sursa are complexitatea $O(V \cdot E)$. Modificarea distantelor pentru fiecare muchie, "reweighting", va dura $O(E)$. In continuare, initializarea matricei de distante rezultate se executa in $O(V^2)$. Deoarece luam fiecare nod si executam Dijkstra cu el ca sursa, o sa avem de $|V|$-ori complexitatea algoritmului Dijkstra implementat cu coada cu prioritati, $O((E + V) \cdot logV)$. Dupa aceea, trebuie sa recalculam distantele corespunzatoare, adica sa iteram din nou prin matricea de distante si sa o actualizam in $O(V^2)$.Deci, complexitatea va fi $O(V + V \cdot E + E + V^2 + V \cdot (E + V) \cdot logV = O(V \cdot E \cdot logV + V^2 \cdot logV)$. Se poate atinge o complexitate temporala si mai buna folosind heap-uri Fibonacci, $O(V \cdot E + V^2 \cdot logV)$.

Complexitatea spatiala in cazul algoritmului implementat este $O(V^2+E)$ deoarece a fost nevoie sa copiam graful initial la care am adaugat un nod si pentru a retine rezultatul folosim o matrice de distante cu $|V| \cdot |V|$ elemente. Deoarece se folosesc doar functii iterative, putem folosi un singur stack frame la un moment de timp.

Algoritmul Dijkstra Algoritmul lui Dijkstra pentru SSSP, va avea o complexitate $O(E \cdot logV)$, deoarece vom folosi o coada de prioritati in care vom insera nodurile. Numarul de elemente din coada de prioritati va fi cel mult $|V|$, deci operatiile pe coada vor avea in cel mai rau caz $O(logV)$. Avand in vedere ca vom lua in considerare toate muchiile din graf, ajungem la o complexitate $O((E+V) \cdot logV)$. In cazul in care am fi folosit un heap Fibonacci am fi putut ajunge la complexitate $O(E + V \cdot logV)$[3].
Complexitatea spatiala este aproximativ $O(V^2 + V)$ deoarece avem nevoie sa retinem matricea de distante cu $|V| \cdot |V|$ elemente si vom avea cel mult $|V|$ elemente in coada de prioritati. Algoritmul este iterativ deci vom avea doar un stack call.

2.3 Avantaje si Dezavantaje

Algoritmul Floyd-Warshall

1. Avantaje
   - foarte simplu de implementat
   - functioneaza pe grafuri cu muchii negative
   - usor de tinut minte
   - intuitiv
   - complexitatea lui temporala depinde numai de numarul de noduri, pentru grafurile foarte dense se comporta la fel de bine precum ceilalti algoritmi prezentati
   - nu necesita memorie suplimentara, este nevoie doar sa retinem matricea de rezultate
2. Dezavantaje
   - produce rezultate eronate pe grafuri cu cicluri de cost negativ, nu gaseste ciclurile negative
   - pe grafuri rare este ineficient din punct de vedere al timpului

Algoritmul Bellman-Ford

1. Avantaje
   - functioneaza pe grafuri cu muchii negative
   - gaseste ciclurile de cost negativ si intrerupe executia programului
   - intuitiv, se aseamana cu algoritmul lui Lee si BFS
   - mai eficient din punct de vedere al timpului decat Floyd-Warshall in majoritatea cazurilor (in special pe grafuri rare)
2. Dezavantaje
   - are o complexitate temporala mai mare decat Dijkstra pe grafurile cu muchii pozitive

Algoritmul Johnson

1. Avantaje
   - trateaza ciclurile negative
   - functioneaza pe grafurile cu muchii negative
   - cea mai buna complexitate temporala in majoritatea cazurilor dintre solutiile prezentate pe grafurile cu muchii negative/pozitive, daca este implementata corespunzator
   - exista o structura de date specializata pentru acest algoritm, Fibonacci heap
2. Dezavantaje
   - greu de implementat
   - cel mai lung cod sursa dintre algoritmii prezentati

Algoritmul Dijkstra

1. Avantaje
   - pe grafuri cu muchii pozitive are o complexitate mai buna decat Bellman-Ford si Floyd-Warshall
   - exista o structura de date specializata pentru acest algoritm, Fibonacci heap
2. Dezavantaje
   - nu functioneaza pe grafurile cu muchii negative

3 Evaluare

3.1 Descrierea Modalitatii de Construire a Setului de Teste

In primul rand, este important sa avem un numar cat mai variat de teste pentru a compara rezultatele obtinute de algoritmi, in vederea testarii corectitudinii. Din acest motiv am ales sa preiau cateva teste de pe un site de programare competitiva [12]. Pentru restul testelor am folosit 3 generatoare de teste[9] [10][11] si am creat grafuri cu diverse proprietati. O parte dintre teste demonstreaza faptul ca Dijkstra nu functioneaza pe grafurile cu muchii negative. Mai exista si un test cu un graf cu ciclu de cost negativ, care arata faptul ca Floyd-Warshall in varianta descrisa nu gaseste ciclul, spre deosebire de celelalte programe a caror executie e intrerupta.
Pentru testarea eficientei am creat grafuri orientate cu muchii pozitive si negative, dense si rare. Pe langa acestea, am folosit si grafuri cu structura arborescenta, aciclice si cu $|V|-1$ muchii. De asemenea, am testat pe grafuri neorientate (cu arce pozitive, pentru a nu avea cicluri negative) si grafuri orientate cu arce de cost 1. Am observat ca diferenta de viteza relevanta este intre grafurile dense si grafurile rare si de aceea am creat mai multe teste cu 10, 100 si 1000 de varfuri si numar de muchii specific unui graf dens/rar. La un moment dat am observat o crestere semnificativa a timpului de executie si pentru a putea vedea mai bine ce se intampla la cresterea numarului de noduri am creat niste teste intermediare, unul dens cu 500 de varfuri si unul rar cu 1500 varfuri. In final, m-a interesat cum s-ar comporta algoritmii pe grafuri "random" si am generat un numar de muchii aleator pentru 10, 100 si 1000 de varfuri.

3.2 Specficatiile Sistemului de Calcul

Am folosit o masina virtuala pe 64 de biti cu 4GB de RAM, 4 CPU-uri virtuale si 100GB de memorie. Procesorul calculatorului pe care a fost folosita masina virtuala este Intel i5-3470 3.20GHz.

3.3 Ilustrarea rezultatelor si Prezentarea Valorilor Obtinute

In continuare vor fi ilustrare rezultatele pentru algoritmii alesi prin grafic si tabel. Intrucat am ales algoritmul Dijkstra ca bonus rezultatele lui vor fi reprezentate doar in tabel, iar prin grafice vor fi comparati ceilalti 3 algoritmi.

Test	Floyd-Warshall	Bellman-Ford	Johnson	Dijkstra
test0.in	0.009385	0.012206	0.035309	0.019538
test1.in	0.029227	0.022749	0.155515	0.048039
test2.in	16328.9	293.304	1094.36	1068.25
test3.in	161.802	8.77577	28.1846	-
test4.in	0.079566	0.046539	0.251022	-
test5.in	85.3204	9.49695	27.98109	25.4258
test6.in	375.501	15.4856	67.3934	64.4117
test7.in	0.759289	0.483902	1.209	0.93562
test8.in	8.93483	9.18301	12.6923	10.4822

Fig. 1. Timp comparativ in milisecunde.

Teste 0, 1, 2 sunt teste cu grafuri orientate care au numai arce pozitive. Testele 3 si 4 au grafuri cu muchii negative, dar nu au cicluri de cost negativ. Testele 5 si 6 au grafuri orientate cu arce de cost 1. Testele 7 si 8 au grafuri neorientate, cu arce pozitive.

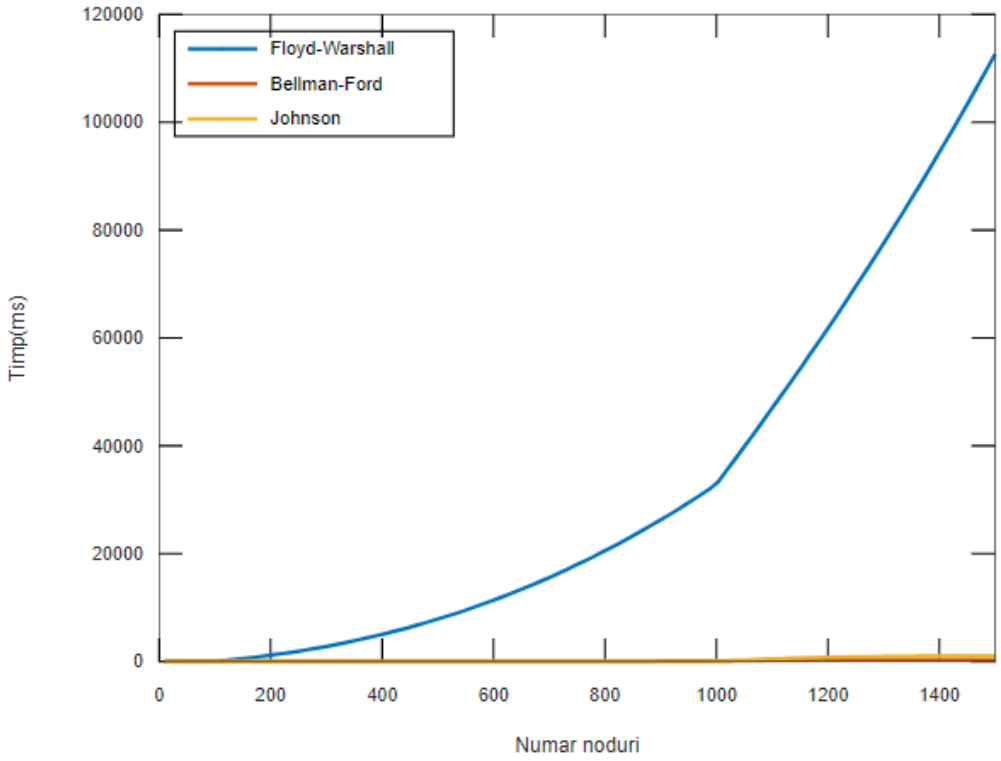
Pe langa aceste 9 teste de baza am folosit alte 14 teste pentru a observa mai bine diferentele.

Test	Floyd-Warshall	Bellman-Ford	Johnson	Dijkstra
test0.in	0.017136	0.053961	0.05656	-
test1.in	0.118323	0.206953	0.465829	0.537412
test2.in	113.569	429.157	194.075	179.699
test3.in	102244	506366	82801.3	80810.2
test4.in	0.050694	0.025109	0.089798	0.044377
test5.in	45.2426	1.34566	2.56599	2.28702
test6.in	32887.6	84.1544	69.9242	77.9295
test7.in	12841.3	60141.5	11984.2	11744.5
test8.in	112623	340.886	1030.91	1024.3
test9.in	0.127309	0.154358	0.429449	0.403039
test10.in	106.295	134.959	109.397	103.593
test11.in	100142	36846.8	17775.4	17304
test12.in	39.0579	1.43348	2.28723	-
test13.in	32758.2	73.6722	44.5367	-

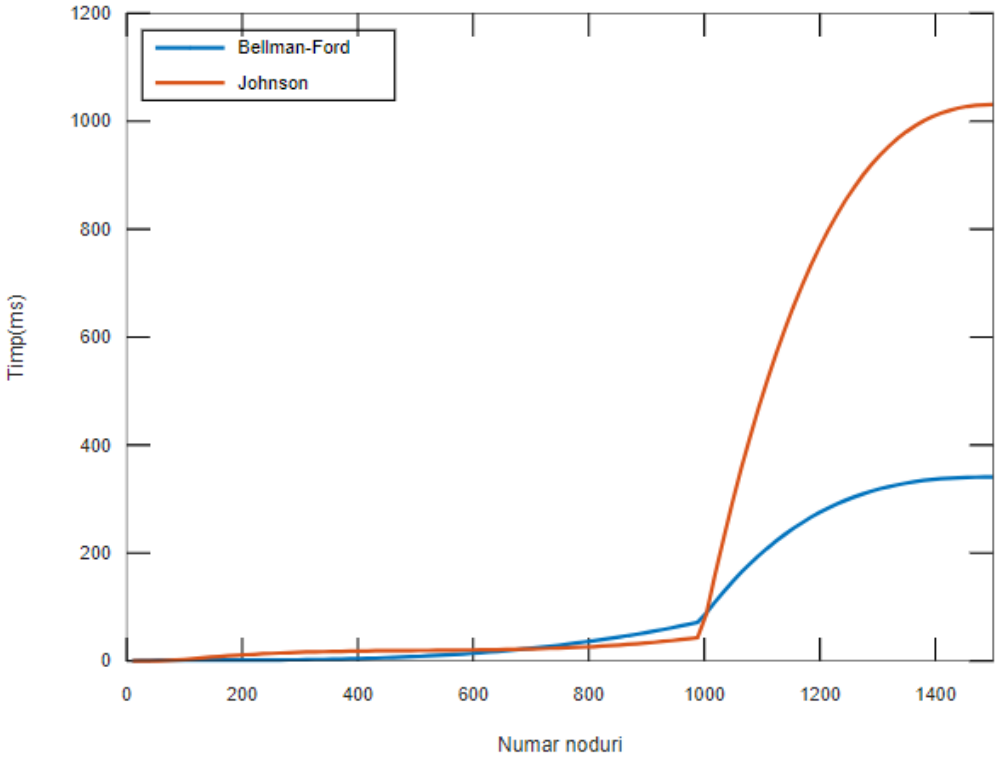
Fig. 2. Timp comparativ in milisecunde.

Testul 0 contine un ciclu de cost negativ. Testele 1, 2 si 3 contin grafuri dense orientate cu 10, 100, 1000 de varfuri. Testele 4, 5, 6 contin grafuri rare orientate cu 10, 100, 1000 de varfuri. Testul 7 contine un graf dens orientat cu 500 varfuri, testul 8 contine un graf rar orientat cu 1500 varfuri. Testele 9, 10 si 11 au un numar de arce generat aleator cu 10, 100, 1000 de varfuri. Testele 12 si 13 au grafuri aciclice, cu $|V| - 1$ arce.

Grafuri rare Mai jos este ilustrat printr-un grafic comportamentul celor 3 algoritmi, Floyd-Warshall, Bellman-Ford si Johnson pe grafuri rare cu pana la 1500 de noduri. Este evident ca timpul de rulare al lui Floyd-Warshall este mult mai mare decat cel al celorlaltor 2 algoritmi, deoarece complexitatea sa temporala depinde exclusiv de numarul de noduri ($O(V^3)$).

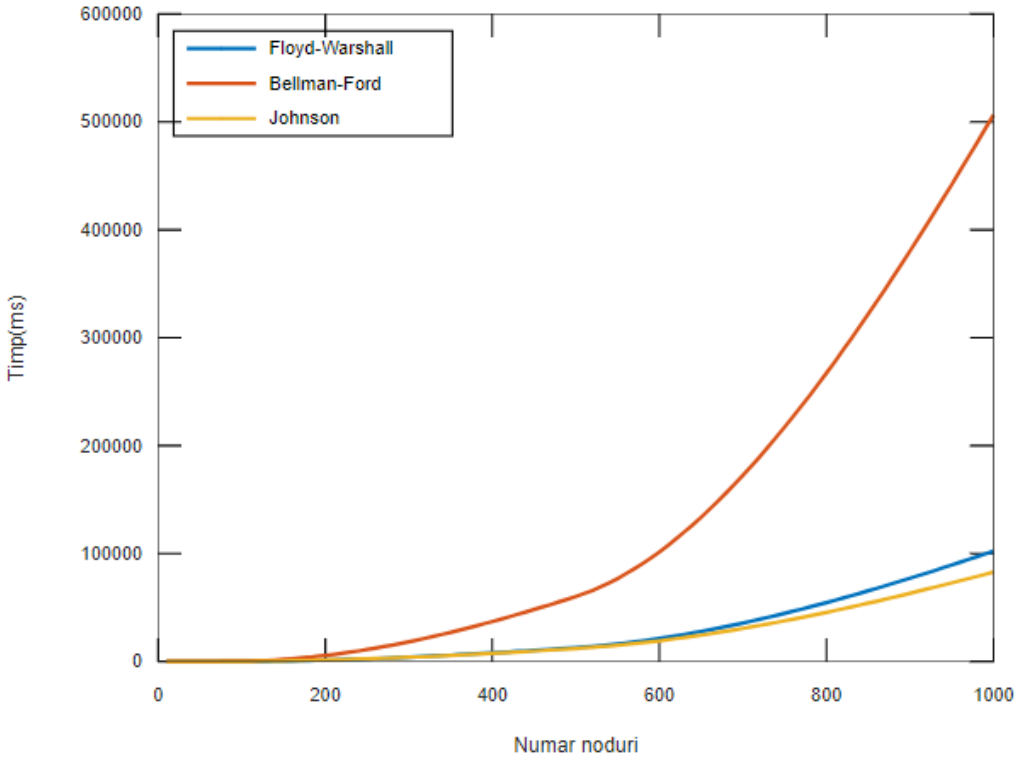


Deoarece dorim sa vedem ce se intampla mai exact cu Bellman-Ford si Johnson vom face un grafic separat doar cu acesti doi algoritmi.



Ambii algoritmi scot un timp foarte bun pe un graf rar. Desi diferenta de timp este relativ mica, aceasta exista deoarece algoritmul lui Johnson este format din mai multe etape decat algoritmul Bellman-Ford, printre care, in implementarea realizata, si de etapa copierii grafului. Totodata, algoritmul lui Johnson poate fi facut mai rapid prin utilizarea unui heap Fibonacci, varianta in care formula complexitatii se schimba. Din cauza operatiilor suplimentare si a implementarii cu o coada cu prioritate algoritmul lui Johnson dureaza cu putin mai mult timp decat algoritmul Bellman-Ford. Micile diferente pot fi cauzate si de functiile de masurare a timpului

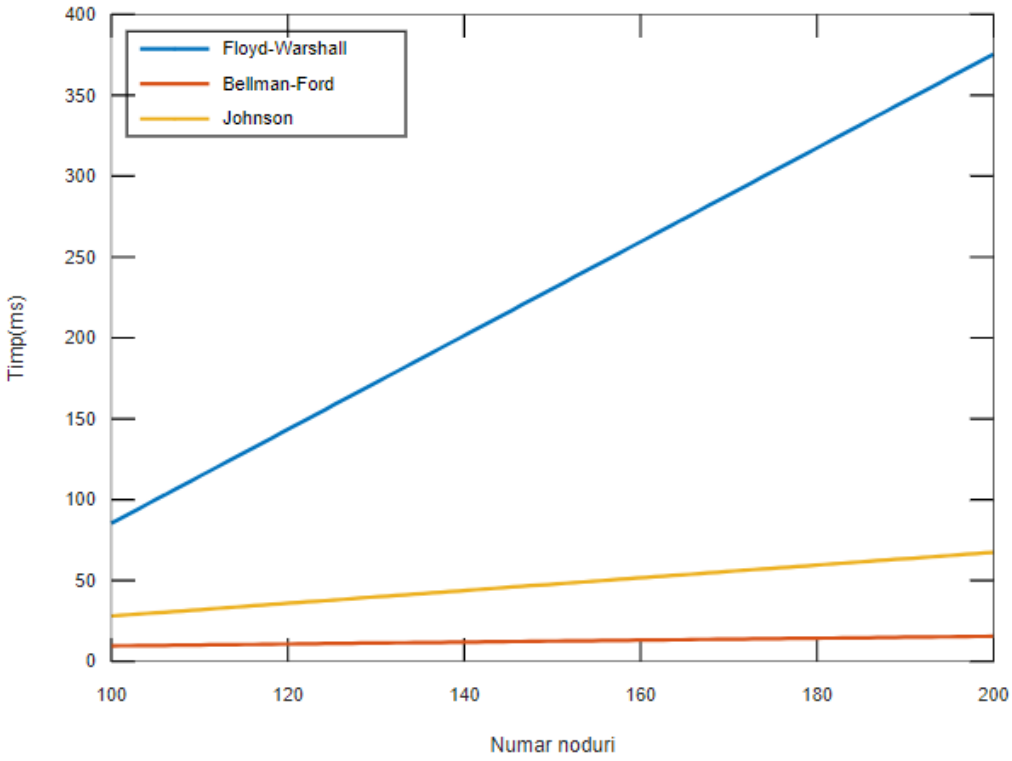
Grafuri dense Vom ilustra comportamentul celor 3 algoritmi pe testele cu grafuri dense cu pana la 1000 de noduri.



Observam imediat ca Bellman-Ford are un timp cu mult mai prost decat Floyd-Warshall si Johnson, deoarece gaseste constant noduri care ar putea imbunatati distantele anterioare, din cauza numarului ridicat de muchii. Pe de alta parte, algoritmul lui Johnson ia mereu cel mai apropiat nod de sursa si verifica daca poate imbunatati drumurile numai in cazul in care distanta din perechea extrasa din coada cu prioritati este mai mica decat distanta minima gasita pentru nod pana in acel moment.
Complexitatea temporala a algoritmului Floyd-Warshall este dependenta de numarul de noduri, fapt ce nu influenteaza prea mult rezultatele, indiferent daca graful este dens. Diferenta dintre Floyd-Warshall di Bellman-Ford este ca Bellman-Ford aduce in ecuatie si numarul de muchii, iar comparatia se reduce la diferenta intre $|E|$ si $|V|$. Deoarece $|E|$ este cu mult mai mare decat $|V|$, complexitatea $O(V^2 \cdot E)$ este mai mare decat $O(V^3)$.
Algoritmul lui Johnson isi pastreaza viteza si pe cazurile cu grafuri dense.

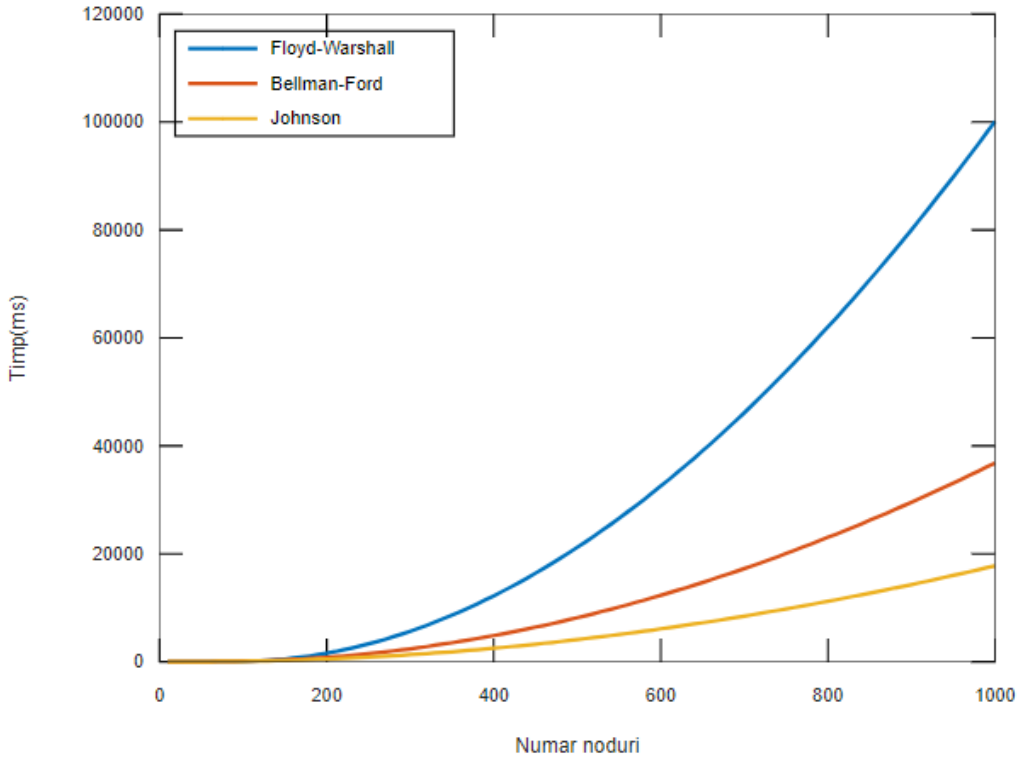
3.4 Grafuri cu arce fara cost

Cei 3 algoritmi pe arce fara cost au obtinut urmatoarele rezultate.



O observatie importanta este ca algoritmul lui Johnson are un timp mai slab decat algoritmul Bellman-Ford deoarece executa operatii care nu sunt neaparat necesare, de extragere de minim si adaugare. Bellman-Ford foloseste o coada simpla, nu o coada cu prioritati si se comporta asemanator cu algoritmul BFS, care are complexitatea $O(E + V)$.

Grafuri random Am testat cei 3 algoritmi pe grafuri cu numar de muchii si cost alese random.



Aceste teste ne arata ca in cazul general, cel mai eficient din punct de vedere al timpului este algoritmul lui Johnson. Urmatorul este algoritmul Bellman-Ford aplicat din toate nodurile, iar pe ultimul loc se afla algoritmul Floyd-Warshall.

Grafuri cu muchii de cost negativ Pe testele cu muchii negative algoritmul lui Dijkstra nu functioneaza, deoarece incearca mereu sa viziteze nodul cel mai apropiat de sursa. Algoritmul lui Dijkstra pune nodurile vizitate intr-o lista si evita sa mai treaca prin ele inca o data bazandu-se pe faptul ca valoarea lor este minima. Cand exista muchii negative nu stim niciodata cand o valoare este minima deoarece mereu putem scadea din ea. Daca exista un arc de valoare 2 de la A la B, un arc de valoare 100 de la A la C si un arc de valoare -110 de la B la C, atunci plecand din varful A vom descoperi varful B, iar pe urma vom merge in C si nu vom mai vizita B deoarece era deja adaugat in multimea nodurilor vizitate.

Grafuri cu cicluri negative Exista un test cu un graf cu ciclu negativ (in folderul other_tests). Pe acest test algoritmul Floyd-Warshall obtine rezultate eronate. Bellman-Ford si algoritmul lui Johnson gasesc ciclul negativ si intrerup executia programului.

4 Concluzie

In urma analizei asupra implementarii algoritmilor si a complexitatii lor spatiale/temporale putem extrage cateva concluzii. Algoritmul lui Johnson pare sa se comporte cel mai bine in practica pe majoritatea tipurilor de grafuri. Singura problema este ca e un algoritm foarte greu de implementat, atat in varianta cu coada cu prioritati, cat si in varianta cu heap Fibonacci. Acest fapt face imposibila implementarea lui in concursurile de programare competitiva, in forma cea mai eficienta, cu heap. Totusi, ar putea fi folosit in diverse programe, precum cele descrise in sectiunea aplicatii practice. Algoritmul Floyd-Warshall este foarte usor de implementat si functioneaza si pe grafuri cu muchii negative, are o complexitate spatiala foarte buna, dar este mai lent in general. Algoritmul Bellman-Ford este intuitiv, se aseamana cu BFS si algoritmul lui Lee, si obtine timpi de rulare destul de buni. Acesta ar fi, dupa mine, alegerea cea mai buna in cazul concursurilor de programare cu timp limitat pentru implementarea unei solutii. Algoritmul lui Dijkstra este de asemenea destul de rapid, dar are dezavantajul ca nu functioneza pe grafuri cu muchii negative.

Bibliografie

1. http://www2.hawaii.edu/~nodari/teaching/f15/Notes/Topic-19.html. Last accessed 12 December 2018
2. Introduction to Algorithms, Book by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen.
3. Fibonacci heaps and their uses in improved network optimization algorithms. 25th Annual Symposium on Foundations of Computer Science, by Tarjan, Robert E.
4. https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture14.pdf. Last accessed 19 November 2018
5. http://jeffe.cs.illinois.edu/teaching/algorithms/. Last accessed 19 November 2018
6. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/. Last accessed 19 November 2018
7. https://www.geeksforgeeks.org/johnsons-algorithm/. Last accessed 19 November 2018
8. https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/. Last accessed 19 November 2018
9. http://spojtoolkit.com/TestCaseGenerator/. Last accessed 2 December 2018
10. https://sourceforge.net/projects/test-case-generator-tool/. Last accessed 2 December 2018
11. https://test-case-generator.herokuapp.com/. Last accessed 2 December 2018
12. https://infoarena.ro/problema/dijkstra. Last accessed 19 November 2018
13. https://infoarena.ro/problema/royfloyd. Last accessed 19 November 2018
14. https://infoarena.ro/problema/bellmanford. Last accessed 19 November 2018