Analiza Algoritmilor
Drumuri minime în graf

Bogdan - Alexandru Simion

Universitatea Politehnică București
Facultatea de Automatică și Calculatoare
Grupa 323CD
bogdan.simion2706@stud.acs.upb.ro

13 decembrie 2019

Rezumat Prezentul proiect se referă la analiza a trei algoritmi specifici problemei de drum minim în graf: Floyd - Warshall, Johnson și Dijkstra, din punct de vedere al complexității lor. Pe parcursul redactării proiectului, voi redacta codul pentru acești algoritmi, cât și testele necesare pentru a determina care algoritm este mai eficient, cât și pe ce cazuri e mai bun un algoritm decât altul. La sfârșit, voi trage concluziile privind algoritmii analizați și voi prezenta avantajele și dezavantajele folosirii lor în practică.

Keywords: Graf · Cost minim · Drum.

1 Introducere

1.1 Descrierea problemei

În teoria grafurilor, problema drumului minim în graf se referă la găsirea acelui drum dintre două vârfuri care este cel mai puțin costisitor. Această problemă este specifică grafurilor ponderate, ce au costuri specifice între două vârfuri. Problema drumului minim poate fi privită din perspectiva atât a grafurilor orientate cât și a celor neorientate, scopul temei o va face analiza pe grafuri orientate, aciclice, ponderate.

1.2 Utilizarea problemei alese in practică

În practică, aceasta problemă este pusă, într-un mod mai subtil, în traficul rutier. În aplicații precum Waze sau Google Maps această problemă trebuie rezolvată prin algoritmul cel mai rapid din punct de vedere temporal. Redactarea celei mai bune soluții este esențială, deoarece, în ziua de azi, traficul suprateran din marile metropole ale lumii este din ce în ce mai aglomerat, fiind destul de importantă ghidarea șoferilor către rute alternative, mai rapide. De asemenea, algoritmii care rezolvă problema sunt utilizați și în controlul traficului aerian, dar și în telecomunicații sau în circuitele VLSI.

1.3 Soluțiile alese pentru rezolvarea problemei

În rezolvarea acestei probleme, sunt propuși trei algoritmi: Floyd - Warshall, Johnson și Dijkstra. În rândurile ce urmează, voi explica pe scurt funcționalitatea algoritmilor.

FLOYD - WARSHALL

Algoritmul Floyd - Warshall a fost dezvoltat de Robert Floyd în anul 1962, fiind asemănător cu ceea ce au propus Bernard Roy în 1959 și Stephen Warshall în 1962.
Algoritmul întoarce un rezultat de tip matrice, cu dimensiunile V * V (unde V reprezintă numărul de vârfuri ale grafului). De asemenea, acest algoritm este corect din punct de vedere al rezultatului întors și pentru costuri negative pe o muchiile grafului.
Astfel, fie datele de input: numărul de vârfuri ale grafului și matricea de costuri asociată grafului. Algoritmul actualizează informațiile din matricea de costuri, dată la intrare, dacă este indeplinită condiția ca distanța dintre două vârfuri (cunoscută deja) este mai mare decât distanța dintre primul vârf și un vârf intermediar, însumată cu distanța dintre acel vârf intermediar și al doilea vârf. Din punct de vedere al programării, condiția arată astfel:

$$dist[i][j] > dist[i][k] + dist[k][j], \tag{1}$$

unde i este primul vârf, j este al doilea vârf si k este vârful intermediar. Algoritmul nu este foarte folosit, deoarece el are o complexitate foarte mare ($\Theta(V^3)$), cu toate că el este corect și pentru costuri negative. Acea complexitate provine din cele 3 for-uri imbricate, cărora nu le putem schimba ordinea fără să le schimbăm codul din interior. Explicația pentru acest lucru este destul de simplă, în sensul că se rupe legătura printr-un nod intermediar a celor două noduri i și j.

DIJKSTRA

Algoritmul lui Dijkstra a fost creat de Edsgar W. Dijkstra în 1956, dar a apărut trei ani mai târziu.
Acesta este specific pentru aflarea celui mai scurt drum de la un nod la celelalte, dar poate fi modificat pentru a afla costul minim între oricare două noduri, ideea algoritmului ramânând aceeași. De asemenea, acest algoritm are o problemă: nu merge pentru costuri ale muchiilor mai mici de 0.
Astfel, la input avem matricea de costuri a grafului, dar și numărul de vârfuri ale grafului. La început, inițializăm un vector de tip boolean unde toate elementele din el sunt false, acestea reprezentând dacă vârfurile grafului au fost vizitate sau nu și inițializăm, într-un vector toate distanțele dintre noduri cu nodul de referință cu infinit.
Algortimul funcționează în felul următor: pentru fiecare nod care urmează să fie vizitat (începând de la nodul de referință), aflu distanța minimă dintre el și celelalte noduri și apoi actualizez informația din matricea finală de costuri (pe care o creez la început, asociindu-i toate valorile cu infinit). Condiția pentru

care actualizez matricea finală de costuri este dacă nodul nu este vizitat și acel nod are o legătură cu nodul curent din bucla for (adică implicit și cu nodul de referință) și dacă distanța dintre el și nodul de referință e mai mare decât distanța de la nodul de referință la vârful vecin plus costul de la vârful vecin până la nod. Din punct de vedere al programării, condiția arată astfel:

$$(visited[v] == 0)\&\&(cost[u][v]! = 0)\&\&(dist[u] + cost[u][v] < dist[v]), \quad (2)$$

unde visited e vectorul de noduri vizitate, cost este matricea de costuri și dist este vectorul de distanțe finale de la nodul de referință.
În final, aș vrea să adaug câteva mențiuni la acest algoritm.
Algoritmul clasic al lui Djikstra este specific grafurilor neorientate ponderate cu costuri pozitive, dar poate fi ușor adaptat pentru grafuri orientate (programul nici măcar nu se modifică). În plus, acest algoritm este specific problemei drumului minim de la un punct la toate celelalte, nu între două puncte oarecare. Acest neajuns poate fi rezolvat într-un mod ușor din punct de vedere al înțelegerii programului, dar mai costisitor din punct de vedere al timpului folosit la rulare. Din punct de vedere al complexității, Dijkstra este mai bun, deoarece are o complexitate de $\Theta(V^2)$, unde V este numărul de vârfuri ale grafului. În cazul în care se folosește heap-ul pentru dijkstra, complexitatea devine $\Theta((E + V^2) * logV)$, unde E este numărul de muchii ale grafului și V numărul de vârfuri. Problema folosirii acestui algoritm o constituie faptul că el nu este corect pentru costuri negative.

JOHNSON

Algoritmul lui Johnson a fost creat de către Donald B. Johnson, apărând pentru prima dată în 1977.
Acesta este specific pentru grafurile ponderate ce au și costuri negative și combină algoritmii Dijkstra și Bellman - Ford pentru a obține un algoritm cu o complexitate mai bună decât cea a algoritmului Floyd - Warshall pe anumite implementări (cozi de prioritate și liste de adiacență).
Acesta este specific grafurilor orientate sau neorientate și ponderate cu ponderi atât pozitive cât și negative.
Astfel, la început avem numărul de vârfuri, dar și matricea de costuri a grafului. Problema care se ridică în momentul de față este "Cum se transformă un graf dat la un graf cu toate ponderile pozitive?". La această întrebare va răspunde prima parte a acestui algoritm.
Prima parte a algoritmului utilizeaza Bellman - Ford pentru a transforma graful ce are și ponderi negative într-unul cu toate ponderile pozitive și constă în adăugarea unui nou vârf la graful existent, fără să modific costurile celorlalte muchii, dar și găsirea celui mai mic cost de la vârful nou adăugat până la fiecare nod din vechiul graf.
A doua parte a algoritmului constă în recalcularea ponderilor muchiilor din graful vechi după următoarea formulă:

$$new\_cost[u][v] = cost[u][v] + new\_vertex\_cost[u] - new\_vertex\_cost[v], \quad (3)$$

unde new_cost[u][v] reprezintă noul cost de la vârful u la vârful v, cost[u][v] reprezintă costul vechi de la u la v, new_vertex_cost[u] reprezintă ponderea de la noul vârf la vârful de la care pleacă muchia, iar new_vertex_cost[v] reprezintă ponderea de la noul vârf la vârful la care sosește muchia.
Ultima parte a algoritmului șterge vârful nou adăugat în graf și aplică Dijkstra pe graful cu noile ponderi, ce sunt întotdeauna pozitive (deci putem aplica Dijkstra fără nicio grijă).
Datele de intrare și de ieșire sunt aceleași ca la ceilalți algoritmi. Complexitatea programului depinde de implementarea lui. În cazul analizat, ea este $\Theta(V^3 +$ VE), adică la fel ca Floyd - Warshall, dar cu implementarea cu cozi de prioritate și liste de adiacență complexitatea scade la $\Theta(V^2$ * logV + VE), adică mai bine ca Floyd - Warshall. În cazul în care graful este complet (E = V^2), atunci complexitatea devine $\Theta(V^3 + V^2$ * logV), adică la fel ca Floyd - Warshall, așadar Johnson se descurcă mai bine pe anumite cazuri ale grafului. Johnson merge pe grafuri cu costuri negative, chiar dacă se folosește Dijkstra, deoarece, înainte de a fi aplicat, toate costurile devin mai mari sau egale cu 0, deci toate sunt pozitive, ceea ce face posibilă aplicarea algoritmului Dijkstra.

1.4 Criterii de evaluare

Pentru a evalua algoritmii din punct de vedere obiectiv, voi întocmi o serie de teste, cu diverse date de intrare, cum ar fi: grafuri complete sau mai goale, grafuri cu ponderi pozitive / negative (pentru negative, algoritmul lui Dijkstra nu va fi corect, ceea ce va fi subliniat în etapa 2), grafuri ciclice / aciclice. Îmi propun să realizez teste pentru grafuri cu diferite dimensiuni, de la 2 noduri până la grafuri cu 1000 de noduri. Toate grafurile de test vor fi orientate și voi avea ca reprezentare a grafurilor o matrice de ponderi.

2 Prezentarea soluțiilor

2.1 Descrierea funcționalității algoritmilor

Deoarece am prezentat algoritmii mai abstract, din punct de vedere al exprimării, dar explicându-i în detaliu, în cele ce urmează, voi redacta doar pseudocodul pentru fiecare dintre algoritmi analizați.

FLOYD-WARSHALL

Algorithm 1: Floyd-Warshall algorithm

```
Input: cost, V
Output: dist
dist ← new int[V][V]
for i ≤ V do
    for j ≤ V do
        if cost[i][j] exists then
            dist[i][j] ← cost[i][j]
        else
            dist[i][j] ← ∞
        end if
    end for
end for
for k ≤ V do
    for i ≤ V do
        for j ≤ V do
            if dist[i][j] > dist[i][k] + dist[k][j] then
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
        end for
    end for
end for
return dist
```

DIJKSTRA

Voi scrie pseudocodul pentru implementarea cu liste de adiacență și cozi de prioritate.

Algorithm 2: Dijkstra algorithm

Input: graph
Output: dist

```
1: dist ← new int[graph.V][graph.V]
2: for i ≤ graph.V do
3:    for j ≤ graph.V do
4:       dist[i][j] ← ∞
5:    end for
6: end for
7: for i ≤ graph.V do
8:    q ← new PriorityQueue
9:    u ← new vertex
10:   for each vertex v in graph do
11:      if v! = i then
12:         add v to q
13:      end if
14:   end for
15:   dist[i][i] ← 0
16:   while q not empty do
17:      u ← minimum from q
18:      for each unvisited neighbour v of u do
19:         if dist[i][v] > dist[i][u] + graph.cost[u][v] then
20:            dist[i][v] ← dist[i][u] + graph.cost[u][v] {graph has it's associated
               costs}
21:         end if
22:      end for
23:   end while
24: end for
25: return dist
```

JOHNSON

Același lucru ca la Dijkstra, pseudocodul îl scriu cu liste de adiacență și cozi de prioritate.

Algorithm 3: Johnson algorithm

Input: graph
Output: dist

```
1: dist ← new int[graph.V][graph.V]
2: graph' ← graph
3: add vertex s to graph'
4: for each vertex v in graph' that is not s do
5:    graph.cost[s][v] ← 0
6: end for
7: if BELLMAN − FORD(graph', s) == false then
8:    print Graph contains negative-weighted cycles!
9: else
10:   for each vertex v in graph' do
11:      h(v) ← graph'.cost[s][v] {graph'.cost[s][v] is computed in
         Bellman-Ford algorithm}
12:   end for
13:   for each edge(u, v) in graph' do
14:      graph.cost[u][v] = graph'.cost[u][v] + h(u) − h(v)
15:   end for
16:   dist ← DIJKSTRA(graph)
17:   for each vertex u in graph do
18:      for each vertex v in graph do
19:         dist[u][v] = dist[u][v] + h(v) − h(u)
20:      end for
21:   end for
22:   return dist
23: end if
```

Deoarece pentru Johnson am avut nevoie de algoritmul Bellman-Ford, am decis să-i scriu mai jos implementarea în pseudocod, după cum urmează.

Algorithm 4: Bellman-Ford algorithm

Input: graph, s
Output: boolean

```
1: for each vertex v in graph do
2:    graph'.cost[s][v] ← ∞
3: end for
4: graph.cost[s][s] ← 0
5: for i < |graph.V| − 1 do
6:    for each edge(u, v) in graph do
7:       if graph.cost[s][v] > graph.cost[s][u] + graph.cost[u][v] then
8:          graph.cost[s][v] = graph.cost[s][u] + graph.cost[u][v]
9:       end if
10:   end for
11: end for
12: for each edge(u, v) in graph do
13:   if graph.cost[s][v] > graph.cost[s][u] + graph.cost[u][v] then
14:      return false
15:   end if
16: end for
17: return true
```

2.2 Analiza complexităților și a corectitudinii algoritmilor folosiți

În cele ce urmează voi analiza atat complexitatea, cât și corectitudinea pentru fiecare dintre soluțiile propuse. Ca o completare, toate soluțiile pică în momentul folosirii unui graf cu cicluri negative, deoarece atunci când mă întorc în nodul sursă (ales de mine), costul total este negativ, deci voi prefera să merg pe acea cale la infinit, ceea ce nu este de dorit. Voi începe mai întâi cu complexitatea și apoi cu corectitudinea.

FLOYD-WARSHALL

Complexitate

După cum se vede din pseudocod, acest algoritm necesită 2 foruri imbricate, unul pentru scriere inițială în matrice și celălalt pentru aflarea distanței minime între fiecare dintre nodurile din graf. Astfel, primul for imbricat (liniile 2 - 10) este compus din 2 cicluri, ce merg amândouă până la numărul de noduri din graf, deci complexitatea este de $\Theta(V^2)$, iar al doilea for imbricat (liniile 11 - 19) este compus din 3 cicluri ce merg până la numărul de noduri din graf, complexitatea acesteia fiind de $\Theta(V^3)$. Așadar, complexitatea algoritmului Floyd- Warshall este de $\Theta(V^2 + V^3)$, ce se poate scrie ca $\Theta(V^3)$ (se ia cea mai mare complexitate separată).

Corectitudine

Din punct de vedere al corectitudinii, algoritmul prezentat este corect, acesta

dând rezultatele corecte atât pentru ponderi pozitive, cât și pentru ponderi negative.

DIJKSTRA

Complexitate

După cum se vede și din pseudocod, acest algoritm este mai complicat decât cel anterior, având mai multe cicluri imbricate. Astfel, primul ciclu imbricat (liniile 2 - 6) realizează scrierea în matrice și are complexitatea de $\Theta(V^2)$, unde V este numărul de noduri din graf. Al doilea ciclu imbricat (liniile 7 - 24) este puțin mai complicat, fiind format dintr-un for după toate nodurile, având, la rândul lui, mai multe cicluri. Primul ciclu (liniile 10 - 14) este un for după toate nodurile din graf, deci complexitatea operației este de $\Theta(V^2 * logV)$. În al doilea ciclu (liniile 16 - 23), problema principală se „desface" în subprobleme, deoarece în ultimul for (liniile 18 - 22) se iau toți vecinii nevizitați a nodului respectiv, iar împreună cu while-ul de deasupra formează o complexitate de $\Theta(E * logV)$, unde E este numărul de muchii din graf. Așadar, complexitatea per total a algoritmului Dijkstra este de $\Theta((V^2 + E) * logV)$. În cel mai rău caz ($E = V^2$), complexitatea algoritmului este de $\Theta(V^2 * logV)$, iar în cel mai bun caz ($E \leq V$), complexitatea algoritmului este de $\Theta(V^2 * logV)$, ceea ce rezultă în complexitatea generală de $\Theta(V^2 * logV)$.

Corectitudine

Din punct de vedere al corectitudinii, algoritmul nu este corect pentru ponderi negative, el dând rezultate corecte doar pentru ponderi pozitive.

JOHNSON

Complexitate

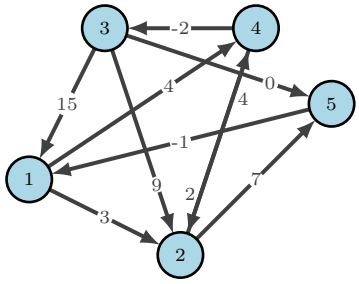
După cum se vede din pseudocod, algoritmul Johnson este format din mai multe cicluri separate, cărora le voi afla complexitatea în cele ce urmează. Primul ciclu (liniile 4 - 6) inițializează distanțele dintre nodul adăugat și celelalte noduri din graf și are complexitatea de $\Theta(V)$. După cum se poate oberva și din pseudocod, la linia 7 este apelată funcția Bellman- Ford, care este chiar algoritmul cu pricina, ce are complexitatea de $\Theta(E * V)$. Dacă se intră pe ramura else, atunci complexitatea algoritmului se va schimba. Astfel, primul ciclu (liniile 10 - 12) are complexitatea de $\Theta(V)$, al doilea ciclu (liniile 13 - 15) are complexitatea de $\Theta(E)$, asignarea distanțelor se face apelând Dijkstra, deci are o complexitate de $\Theta((V^2 * logV)$, iar ultimul ciclu are complexitatea de $\Theta(V^2)$. Așadar, însumând toate complexitățile, vom obține complexitatea algoritmului de $\Theta(V^2 * logV + E * V)$. În cel mai rău caz ($E = V^2$) avem complexitatea $\Theta(V^3)$, iar în cel mai bun caz ($E \leq V$) avem complexitatea $\Theta(V^2 * logV)$.

Corectitudine

Din punct de vedere al corectitudinii, algoritmul prezentat este corect, acesta dând rezultatele corecte atât pentru ponderi pozitive, cât și pentru ponderi negative, spre deosebire de Dijkstra, unde era corect doar pentru ponderi pozitive.

2.3 Avantaje și dezavantaje

Fiecare algoritm prezentat în această temă are avantajele și dezavantajele ei, iar pentru a ilustra mai bine acest lucru, voi încerca pe un exemplu ușor de vizualizat. Mai jos am desenat graful asociat exemplului pe care vreau să-l fac.



Matricea asociată acestui graf este reprezentată mai jos.

$$\begin{bmatrix} 0 & 3 & \infty & 4 & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 15 & 9 & 0 & \infty & 0 \\ \infty & 2 & -2 & 0 & \infty \\ -1 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Vom încerca să găsim soluția problemei drumului minim prin toți algoritmii prezentați până acum.

FLOYD-WARSHALL

Are complexitatea temporală de $\theta(V^3)$ și este corect pentru orice fel de ponderi, cu excepția ciclurilor negative. Calcularea distanțelor minime se face prin condiția ca distanțele intermediare dintre 2 noduri să fie mai mici decât distanța directă dintre respectivele noduri pentru ca matricea să fie actualizată. Astfel, matricea rezultată în urma aplicării algoritmului este:

$$\begin{bmatrix} 0 & 3 & 2 & 4 & 2 \\ 1 & 0 & 2 & 4 & 2 \\ -1 & 2 & 0 & 3 & 0 \\ -3 & 0 & -2 & 0 & -2 \\ -1 & 2 & 1 & 3 & 0 \end{bmatrix}$$

Într-adevăr, este matricea corectă. Principalele avantaje pe care le aduce acest algoritm sunt faptul că este foarte intuitiv și accesibil de scris, dar și

faptul că este corect în toate cazurile în care problema drumului minim este decidabilă. De asemenea, ocupă puține resurse de memorie, deoarece pentru acest algoritm avem nevoie doar de 2 matrici și 3 variabile, dar acest lucru nu mai reprezintă un avantaj important, deoarece spațiul consumat poate fi considerat nul în comparație cu spațiul disponibil pe o mașină de calcul obișnuită. Dezavantajul esențial pe care acest algoritm îl prezintă este faptul că are o complexitate temporală foarte mare, de $\Theta(V^3)$, ceea ce îl face de evitat.

DIJKSTRA

Are complexitatea temporală de $\theta(V^2 * logV)$ și este corect doar pentru ponderi pozitive. Calcularea distanțelor minime se face folosind liste de adiacență și cozi de prioritate. Matricea rezultată în urma aplicării algoritmului este:

$$\begin{bmatrix} 0 & 3 & 2 & 4 & 2 \\ 1 & 0 & 2 & 4 & 2 \\ -1 & 2 & 0 & 3 & 0 \\ -3 & 0 & -2 & 0 & -2 \\ -1 & 2 & 1 & 3 & 0 \end{bmatrix}$$

În mod straniu, matricea este corectă. Motivul pentru care Dijkstra nu funcțio -nează pe costuri negative este că se face o presupunere greșită de la bun început: atunci când se adună costurile, ele nu pot să scadă. O explicație pentru care în acest caz Dijkstra a fost corect este faptul că nu au fost decât 2 muchii cu costuri negative, iar acelea au fost destul de mici în modul, în comparație cu celelalte ponderi. Principalul avantaj pe care îl are Dijkstra se referă la complexitatea mai coborâtă ($\Theta(V^2 * logV)$) în comparație cu algoritmul Floyd-Warshall. În plus, este destul de inuitiv în comparație cu următorul algoritm ce trebuie discutat. Principalul dezavantaj pe care îl are acest algoritm se referă la incorectitudinea lui pentru muchiile cu ponderi negative (nu întotdeauna se întâmplă asta, după cum am văzut anterior, dar de obicei este incorect în aceste cazuri). De asemenea, el consumă mai multe resurse spațiale decât Floyd-Warshall, dar acum, acest incovenient nu mai este relevant.

JOHNSON

Are complexitatea temporală de $\theta(V^2 * logV + E * V)$ și este corect pentru orice fel de ponderi, cu excepția ciclurilor negative. Calcularea distanțelor minime se face la fel ca la Dijkstra, dar mai întâi se face o reponderare a muchiilor, astfel încât să se poata rula Dijkstra. Matricea care rezultă din algoritm este:

$$\begin{bmatrix} 0 & 3 & 2 & 4 & 2 \\ 1 & 0 & 2 & 4 & 2 \\ -1 & 2 & 0 & 3 & 0 \\ -3 & 0 & -2 & 0 & -2 \\ -1 & 2 & 1 & 3 & 0 \end{bmatrix}$$

Într-adevăr, este matricea corectă. Principalele avantaje pe care îl aduce acest algoritm sunt complexitatea temporală ($\theta(V^2 * logV + E * V)$), ceea ce în cazurile

bune este mai performant decât Floyd-Warshall, dar în cazul prost este la fel, dar și corectitudinea, deoarece este la fel de corect ca Floyd-Warshall. Un dezavantaj pe care îl prezintă este faptul că nu este atât de ușor de scris ca Floyd-Warshall, sau chiar Dijsktra, el apelând 2 algoritmi diferiți, necesitând astfel mai mult spațiu decât ceilalți doi algoritmi.

3 Evaluare

3.1 Modaliatea de construcție a testelor și specificațiile sistemului de calcul

Testele au fost construite folosind un generator proiectat exclusiv de mine, acestea pliindu-se pe mai multe cazuri de grafuri (cu costuri pozitive, negative, cu cicluri negative, etc.). Am întocmit testele astfel încât să ating toate cazurile posibile (grafuri rare, dense, cu mai multe sau putine vârfuri). Motivația pentru care am generat astfel de teste este de a vedea cum se comportă fiecare algoritm pe anumite date de intrare. Mai departe, voi scrie specificațiile fiecărui test:

- test0.in: 5 noduri, 13 muchii
- test1.in: 10 noduri, 33 muchii
- test2.in: 10 noduri, 32 muchii, costuri negative
- test3.in: 100 noduri, 2097 muchii
- test4.in: 100 noduri, 1245 muchii, costuri negative
- test5.in: 100 noduri, 2427 muchii
- test6.in: 1000 noduri, 376434 muchii
- test7.in: 1000 noduri, 113679 muchii, costuri negative

De asemenea, sistemul de calcul pe care am rulat algoritmii are un procesor Intel i7-4510U cu frecvența de 2.60 GHz, având memoria RAM de 8GB (7.89GB uitilizabili).

3.2 Rezultatele evaluării soluțiilor generate pe setul de teste

Rezultatele evaluării le voi reprezenta sub formă de tabele și sub formă de grafice.

Corectitudine

Număr criteriu test	Floyd-Warshall	Dijkstra	Johnson
test0.in	Corect	Corect	Corect
test1.in	Corect	Corect	Corect
test2.in	Corect	Incorect	Corect
test3.in	Corect	Incorect	Incorect
test4.in	Corect	Incorect	Incorect
test5.in	Corect	Incorect	Incorect
test6.in	Corect	Incorect	Incorect
test7.in	Inorect	Incorect	Incorect

Complexitate

Număr criteriu test	Floyd-Warshall	Dijkstra	Johnson
test0.in	0.015 ms	4 ms	4 ms
test1.in	0.1 ms	6 ms	6 ms
test2.in	0.1 ms	6 ms	6 ms
test3.in	25 ms	700 ms	700 ms
test4.in	25 ms	700 ms	700 ms
test5.in	25 ms	700 ms	700 ms
test6.in	2600 ms	5500000 ms	5500000 ms
test7.in	2600 ms	5500000 ms	5500000 ms

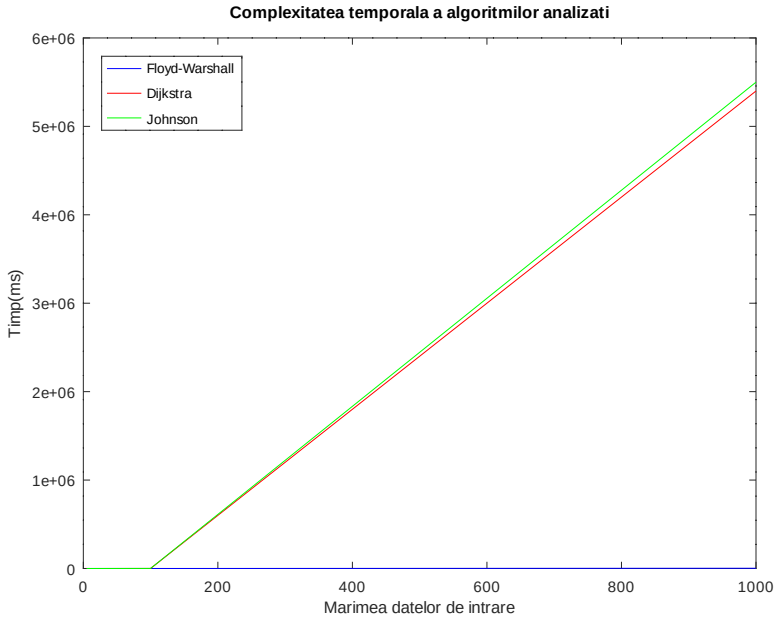


Figura 1. Graficul complexităților algoritmilor analizați

3.3 Prezentarea valorilor obținute

După cum se poate vedea și din tabelul pentru corectitudinea obținută, algoritmii nu sunt corecți întotdeauna, dintr-un motiv foarte simplu: ciclurile negative cauzează nedecidabilitatea problemei cu pricina (testul 7 a avut cicluri

negative). Pe de altă parte, alte teste (2 si 4) au cauzat incorectidunea algoritmului Dijkstra, deoarece au avut ponderi negative. Spre deosebire de aceste teste ce au cauzat incorectitudine, s-au strecurat anumite erori în programele scrise (pentru Dijkstra, deci, implicit și pentru Johnson), ce au făcut ca algoritmii scriși să pice și pe alte teste (3, 4 pentru Johnson, 5 și 6). Singura explicație pe care o pot da ar fi aceea că am un bug nedetectat în algoritmi și nu am putut / nu am avut timp să le fixez.

De asemenea, complexitatea temporală a algoritmilor este diferită față de ce trebuia să obțin din punct de vedere teoretic. Într-adevăr, în testele de până la 5 inclusiv, pare că raportul dintre timpul rulat de Dijkstra (sau Johnson) și timpul rulat de Floyd-Warshall se micșorează, fapt ce mă determină să dau o explicație pertinentă cu privire la acest timp: se folosesc structuri de date complexe în cei 2 algoritmi (cozi de prioritate, TDA de tip graf, perechi) față de structurile de date folosite în algoritmul Floyd-Warshall (matrice). Pentru testele 6 și 7, totuși, nu am habar de ce durează atât de mult (aprox. o oră și jumătate), cel mai probabil de la faptul că rulez programul într-un IDE, decât în mod clasic, în terminal, ceea ce consumă mai multă memorie RAM.

4 Concluzii

După toate cele scrise mai sus, am tras mai multe concluzii, referitoare la problema drumului minim în graf și a algoritmilor corespunzători. Astfel, problema drumului minim încă rămâne în dezbatere și în zilele noastre, deoarece, cel puțin acești 3 algoritmi au avantajele și dezavantajele lor, programatorul trebuind să se orienteze în funcție de nevoile lui pentru un anume algoritm.

Din punctul meu de vedere, pentru această situație (de analizare a algoritmilor și de generare a testelor) prefer un algoritm, poate care are o complexitate mai ridicată dar corect și ușor de scris, decât un algoritm care poate e mai rapid, dar e mai greu de scris (ceea ce implică mai multe bug-uri) și poate nu e corect întotdeauna (cum este Dijkstra). Strict pentru această situație aș prefera algoritmul Floyd-Warshall, în defavoarea celorlalți. Pe de altă parte, pentru a afla drumul cel mai scurt pe care pot să-l pot parcurge până la facultate utilizând mașina personală sau transportul public, mă voi baza mai degrabă pe un algoritm ce este mai rapid, deoarece, în situația de față, nu voi avea de a face cu ponderile negative, deci aș putea să utilizez cu siguranță algoritmul Dijkstra (dacă este scris bine, în mod evident). Mă interesează un algoritm mai rapid deoarece vreau și trebuie să aflu instant noua rută, mai ales dacă, în mod neașteptat traficul devine aglomerat (din cauze diverse) și de aceea voi alege algoritmul mai rapid (Dijkstra sau alții neanalizați aici, de exemplu A*).

5 Bibliografie

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", Third Edition, The MIT Press

2. Quora, https://www.quora.com/How-do-algorithms-of-tracing-a-way-like-Waze-and-Googles-Maps-work. Ultima accesare: 29 Noiembrie 2019

3. Geeks for Geeks, https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/. Ultima accesare: 29 Noiembrie 2019

4. Geeks for Geeks, https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. Ultima accesare: 29 Noiembrie 2019

5. Geeks for Geeks, https://www.geeksforgeeks.org/johnsons-algorithm/. Ultima accesare: 29 Noiembrie 2019

6. Algorithmist, https://algorithmist.com/wiki/Floyd-Warshall's_Algorithm. Ultima accesare: 11 Decembrie 2019