Tema 1

Drumuri minime în graf
Costul minim între oricare 2 noduri

Bobocu Alexandra-Florentina, 321CA

Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA București
alexandra.bobocu@stud.acs.upb.ro
Noiembrie 2021

1 Introducere

1.1 Descrierea problemei rezolvate

În teoria grafurilor, problema drumului minim în graf se referă la a găsi drumul între două noduri ale grafului, astfel încât suma totală a ponderilor muchiilor să fie minimă. Într-un graf orientat G = (V, E) (V = mulțimea de noduri, E = mulțimea de muchii) cu n noduri, fiecărui arc îi este asociat un număr întreg numit cost. Dacă toate muchiile ar avea același cost, această problemă ar putea fi rezolvată cu ușurință folosind parcurgerea în lățime a grafului (BFS: Breadth First Search), însă având grafuri cu muchii de costuri diferite, sunt necesari algoritmi mai avansați, dintre care voi prezenta 3 în cele ce urmează.
Semnificaţia acestui cost poate fi foarte variată, în funcţie de domeniul pe care îl descrie graful. De exemplu, dacă graful reprezintă harta unui oraş în care arcele sunt străzile, iar nodurile sunt intersecţiile dintre străzi, atunci putem vorbi despre costul deplasării unui automobil între două intersecţii, de-a lungul unei străzi. Acesta s-ar putea măsura în cantitatea de benzină consumată, calculată prin prisma lungimii străzii în metrii sau în kilometrii.

1.2 Exemple de aplicații practice pentru problema aleasă

Grafurile au devenit astăzi foarte răspândite datorită ariei largi de aplicabilitate a acestora, de la aplicații atât software cât și hardware, la diverse aplicații în modelarea sistemelor economice, în științele inginerești și în cele sociale.
Google Maps, Moovit sau Waze sunt doar câteva din aplicațiile practice pe care le folosim în viața de zi cu zi pentru a găsi o rută posibilă, de preferat minimă între două locații pe hartă. Aceste aplicații arată utilizatorului timpul minim în care ajunge la destinație, respectiv traseul optim pe care să îl parcurgă.
Un alt exemplu îl reprezintă aplicațiile de rețele sociale, care sugerează o listă de prieteni pe care un anumit utilizator îi poate cunoaște. Nodurile sunt utilizatorii, iar muchiile reprezintă relațiile de prietenie care se stabilesc între aceștia.
În plus, utilități practice ale problemei enunțate ar mai fi agenda zborurilor, în

care cunoscându-se locația și ora de plecare, agentul de zboruri ar dori să determine ora de sosire cea mai favorabilă ajungerii la destinația propusă.
Într-o rețea locală de internet LAN(Local Area Network), se urmărește minimizarea întârzierii transmiterii datelor între calculatoare.

1.3 Specificarea soluțiilor alese

Floyd-Warshall este un algoritm care găsește drumul minim într-un graf, comparând toate căile posibile dintre fiecare 2 noduri. Poate fi aplicat pe grafurile orientate sau neorientate ce au muchii de cost pozitiv sau negativ, însă nu funcționează pentru grafurile cu cicluri în care costul total este negativ.

Dijkstra este un algoritm ce există în multe variante. Varianta originală a acestui algoritm a găsit cea mai scurtă cale între două noduri date, dar alternativa mai comună fixează un singur nod drept sursă și găsește cele mai scurte căi de la sursă la toate celelalte noduri din graf. Dijkstra poate fi folosit doar în grafuri care au toate muchiile nenegative, deoarece algoritmul se bazează pe un fapt simplu: dacă costurile muchiilor sunt nenegative, adăugând o muchie grafului nu putem găsi o cale mai scurtă decât cea pe care am ales-o deja. Alegerea de la început a unei muchii ca fiind calea cea mai scurtă de la un nod la alt nod (optim local) ne conduce către rezultatul corect (optim global).

Johnson este un algoritm ce se aplică pe grafuri orientate și găsește cele mai scurte căi între toate perechile de noduri într-un astfel de graf. Permite ca unele dintre costurile muchiilor să fie negative, dar nu pot exista cicluri negative.
Ideea algoritmului Johnson este de a repondera muchiile și de a le face pozitive, utilizând în acest sens algoritmul Bellman-Ford, după care folosește algoritmul Dijkstra pe graful transformat.

1.4 Specificarea criteriilor de evaluare alese pentru validarea soluțiilor

Pentru a testa corectitudinea, eficiența și performanța algoritmilor aleși, o să pun la dispoziție teste cât mai numeroase și cât mai variate, cu grafuri rare sau complete, cu costuri/cicluri pozitive sau negative. Voi trata și cazurile particulare, cum ar fi grafurile cu costuri negative pentru algoritmul Dijkstra, grafurile cu cicluri negative pentru algoritmii Floyd-Warshall, respectiv Johnson. În cazul în care un algoritm va da de un astfel de caz particular, se va afișa un mesaj sugestiv.
În fișierele de intrare, pe prima linie vor fi numărul de noduri și de muchii, după care următoarele linii vor fi reprezentate de modelul: nod sursă - nod destinație - cost muchie. În fiecare fișier de ieșire corespunzător unui anumit fișier de intrare, îmi propun să afișez matricile de adiacență rezultate în urma aplicării celor 3 algoritmi descriși pe graful considerat ca input.
O să încep cu câteva teste cu date puține de intrare, astfel încât să fie asigurat

faptul că algoritmii pot rezolva corect un graf simplu. Pe parcurs, dimensiunea inputului va crește, deoarece voi genera grafuri cu un număr semnificativ mai mare de noduri și muchii, pentru a evidenția o deosebire mai vizibilă în ceea ce privește eficiența fiecărui algoritm.

2 Prezentarea soluțiilor

2.1 Descrierea modului în care funcționează algoritmii aleși

Floyd-Warshall este un exemplu al programării dinamice. Descompune problema în subprobleme mai mici, apoi combină răspunsurile la acele subprobleme pentru a rezolva marea problemă inițială.

```
Create a |V| x |V| matrix, M, that will describe the distances between vertices
For each cell (i, j) in M:
    if i == j:
        M[i][j] = 0
    if (i, j) is an edge in E:
        M[i][j] = weight(i, j)
    else:
        M[i][j] = infinity
for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            if M[i][j] > M[i][k] + M[k][j]:
                M[i][j] = M[i][k] + M[k][j]
```

Pseudocodul algoritmului Floyd-Warshall

Se inițializează matricea de adiacență cu costul muchiilor formate de nodurile date în input. Pe diagonala principală din matrice costul este 0.
Modificăm matricea de adiacență în felul următor:
Pornind cu valori ale lui k de la 1 la numărul de noduri, ne interesează să găsim cea mai scurtă cale de la fiecare nod sursă i la fiecare nod destinație j folosind doar noduri intermediare din mulțimea {1,...,k}. De fiecare dată, comparăm costul deja estimat al drumului de la nodul i la nodul j obținut la pasul anterior cu costul drumurilor de la i la k și de la k la j, obținute la pasul anterior. Dacă k nu este nod intermediar, nu actualizăm rezultatul; în caz contrar modificăm rezultatul în matrice dacă distanța folosindu-ne de acest nod intermediar este mai mică decât distanța inițială.

Ordinea for-urilor în algoritmul Floyd-Warshall este importantă, deoarece determină de câte ori găsim distanța dintre 2 noduri date. Orice modificare în interiorul acestor for-uri ar duce la rezultate greșite; dacă am itera cu i în primul for, cu j în al doilea și cu k în al treilea, vom obține distanța între i și j folosindune de un singur nod intermediar, iar acea distanță să nu fie neapărat cea mai mică posibilă. Scopul ar fi să îmbunătățim distanțele; în acest sens, nu luăm în

considerare oricare 2 noduri i și j, ci încercăm să găsim un drum optim între ele, fixând mai întâi un nod k și îmbunătățind toate celelealte drumuri relativ la k.

Algoritmul Floyd-Warshall nu funcționează pe grafuri cu ciclu negativ. Motivul constă în faptul că nu există o cale cea mai scurtă între nicio pereche de noduri care fac parte dintr-un ciclu negativ, deoarece lungimea căii între 2 noduri poate fi arbitrar de mică (negativă). Pentru o ieșire semnificativă numeric, algoritmul Floyd-Warshall presupune că nu există cicluri negative. Dacă totuși există, algoritmul poate fi folosit pentru a le detecta.
Este important să verificăm existența ciclurilor negative în graf, pentru a evita un rezultat eronat. Pentru aceasta, comparăm ca nu cumva costul de pe diagonala principală din matrice să fie mai mic decât 0.

Dijkstra - Într-un graf dat, luând pe rând fiecare nod și considerându-l nod sursă, putem aplica algoritmul Dijkstra pentru a găsi distanța minimă.

```
High-level pseudocode of Dijkstra's algorithm

dijkstra(G, s):
    dist = list of length n initialized with INF everywhere except for a 0 at position s
    mark every node as unvisited
    while there are unvisited nodes:
        u = unvisited node with smallest distance in dist
        mark u as visited
        for each edge (u,v):
            relax((u,v))
```

Pseudocodul algoritmului Dijkstra

Se creează un vector de vizite care ține evidența nodurilor incluse în SPT (Shortest Path Tree). Inițial, niciun nod nu este încă vizitat, așa că toate elementele din el sunt marcate cu "false". În plus, asignăm fiecărui nod din graf o valoare a distanței calculate. Inițial, aceasta este marcată cu infinit. Distanța de la nodul sursă la el însuși este marcată cu 0.
Odată ce algoritmul a găsit cea mai scurtă cale între nodul sursă și un alt nod, acel nod este marcat ca "vizitat" și adăugat la calea minimă. În momentul în care un nod a fost marcat ca "vizitat", calea curentă către acel nod este marcată ca fiind cea mai scurtă cale pentru a ajunge la acel nod. Procedeul continuă până când toate nodurile din graf au fost adăugate la cale. În acest fel, am format un arborele de drumuri minime SPT, care conectează nodul sursă la toate celelalte noduri, urmând cea mai scurtă cale posibilă pentru a ajunge la fiecare nod.
Pentru a actualiza valorile distanței, se parcurg nodurile adiacente v nodului ales drept sursă u și "relaxăm" muchia (u, v). Astfel, pentru fiecare nod adiacent v, dacă valoarea distanței din nodul sursă u adunată cu costul muchiei formate de u și v este mai mică decât valoarea distanței din nodul adiacent v, atunci muchia este "relaxată" și actualizăm valoarea distanței din nodul v cu suma.

Johnson folosește alți 2 algoritmi de drumuri minime, și anume algoritmul Bellman-Ford și algoritmul Dijkstra. Prima oară aplică Bellman-Ford pentru a detecta ciclurile negative și pentru a recalcula costurile muchiilor grafului inițial, eliminându-se în acest mod muchiile cu cost negativ. Acesta este și motivul pentru care algoritmul Johnson poate fi folosit pe grafuri cu muchii de cost negativ. În continuare, aplică Dijkstra pe graful modificat pentru a calcula drumul minim.

```
johnson(graph_t G) {
  create G' where G' is the same as G except there is an additional
  vertex 's' added that is connected to all other vertices with 0 weight edges

  if Bellman-Ford(G', s) == False
    return "Negative Cycle"

  for every vertex v in G'
    h(v) = distance(s, v) computed by Bellman-Ford
  for every edge (u, v) in G
    weight'(u, v) = weight(u,v) + h(u) - h(v)

  D = new matrix of distances initialized to infinity
  for every vertex u in G
    run Dijkstra(G, weight', u) to compute distance'
    for each vertex v in G
      D(u,v) = distance'(u, v) + h(v) - h(u)
  return D
}
```

Pseudocodul algoritmului Johnson

Algoritmul Johnson are 3 pași:

1. Se începe prin selectarea unui nod sursă, însă pentru a ne asigura că un singur nod sursă poate atinge toate celelealte noduri din graf, este introdus un nou nod, care se conectează la toate celelealte prin muchii cu costul 0.
2. Se recalculează toate costurile muchiilor pe noul graf, aplicând algoritmul Bellman-Ford luând nodul nou adăugat drept sursă. În urma acestui pas, costurile muchiilor devin nenegative.
3. Nodul adăugat în plus la primul pas este eliminat, urmând să aplicăm algoritmul Dijkstra pentru fiecare nod.

2.2 Analiza complexității soluțiilor

Floyd-Warshall - Complexitatea temporală a algoritmului Floyd-Warshall este O(n^3), unde n = numărul de noduri din graf. Această complexitate se datorează celor 3 cicluri de for care sunt parcurse pentru fiecare nod în parte, în cele 3 ipostaze de: nod intermediar, nod sursă, nod destinație. Acest algoritm este o alegere bună pentru calcularea căilor între toate perechile de noduri în grafurile dense, în care majoritatea sau toate perechile de noduri sunt conectate prin muchii, deoarece complexitatea sa depinde doar de numărul de noduri din graful dat.
În general, preferăm reprezentarea grafurilor prin liste de adiacență deoarece au o complexitate mai bună în cazul parcurgerilor. Totuși, în cazul algoritmului Floyd-Warshall, alegerea reprezentării prin matrice de adiacență simplifică

mult rezolvarea unei probleme, deoarece putem obține ușor distanța dintre două noduri pe baza matricei în care reținem, adițional, și costurile muchiilor.

Dijkstra - În funcție de structura de date folosită pentru găsirea nodului cu valoarea distanței minime ce se regăsește printre nodurile ce nu au fost încă incluse în SPT, algoritmul Dijkstra are diferite complexități.
Cea mai simplă implementare a algoritmului Dijkstra, dar și cea mai puțin eficientă se realizează prin folosirea unui vector nesortat. Astfel, iterăm prin toate nodurile nevizitate pentru a-l găsi pe cel care conține distanța minimă. Complexitatea temporală inițială pentru această căutare este O(n), unde n = numărul de noduri din graf. În final, vom ajunge la o complexitate temporală O(n^2), deoarece, pentru fiecare nod în parte, acesta este ales drept sursă și trebuie calculată distanța minimă de la nodul sursă la toate celelelate noduri rămase. Complexitatea spațială este O(n), nefavorabilă dacă avem un graf complet, deoarece ar însemna să "relaxăm" toate muchiile.
Un graf orientat cu n noduri, ce are aproape n*(n - 1) muchii se numește dens. Această implementare a algoritmului Dijkstra se dovedește a fi optimă pentru grafurile dense, deoarece pentru "relaxarea" tuturor muchiilor, avem o complexitate O(m), unde m = numărul de muchii - este proporțional cu n^2.

Timpul de rulare pentru algoritmul Dijkstra folosind un Min-Heap binar și o coadă de prioritate în locul unui vector nesortat se încadrează într-o complexitate temporală O(m*log n). Complexitatea spațială este O(m), unde n = numărul de noduri, iar m = numărul de muchii.

Cu o structură de date mai avansată numită Fibonacci Heap, timpul de rulare al algoritmului Dijkstra poate fi redus la o complexitate temporală O(m + n * log n) și una spațială O(n), unde n = numărul de noduri, iar m = numărul de muchii.

Johnson - apelează algoritmul Bellman-Ford o singură dată pentru detectarea ciclurilor negative și relaxarea muchiilor, după care algoritmul Dijkstra este apelat de n ori pentru calculul drumului minim, unde n = numărul de noduri din graf. Complexitatea temporală a algoritmului Bellman-Ford este O(n * m), unde n = numărul de noduri din graf, iar m = numărul de muchii. În cazul favorabil, complexitatea algoritmului Dijkstra este O(n * log n). Deci, complexitatea algoritmului Johnson este O(n^2 * log n + n * m). În cel mai defavorabil caz, complexitatea temporală a algoritmului Johnson este O(n^3 + n * m), dacă algoritmul Dijkstra are complexitatea O(n^2).
Când graful este complet, complexitatea algoritmului Johnson devine aceeași cu cea a lui Floyd-Warshall. Pentru grafuri rare însă (unde numărul de muchii este redus), Johnson este mai eficient decât Floyd-Warshall, deoarece complexitatea sa temporală depinde de numărul de muchii din graf, în timp ce complexitatea algoritmului Floyd-Warshall nu.

2.3 Prezentarea principalelor avantaje și dezavantaje pentru soluțiile luate în considerare

Floyd-Warshall are următoarele avantaje și dezavantaje:
Avantaje:

1. Poate fi aplicat pe grafuri cu costuri negative
2. O singură execuție este suficientă pentru a găsi drumul minim între toate perechile de noduri din graf
3. Este ușor să scriem codul acestui algoritm într-un program
4. Evidențiază toate perechile de căi cele mai scurte dintr-un graf
5. Algoritmul este ușor de modificat și de folosit pentru a reconstrui căile

Dezavantaje:

1. Nu se poate aplica pe un graf care are cicluri negative
2. Nu reține detalii despre drum
3. Este mai lent decât alți algoritmi care rezolvă problema drumului minim în graf

Dijkstra are următoarele avantaje și dezavantaje:
Avantaje:

1. În cazul unei implementări cu Min-Heap binar sau Fibonacci Heap, complexitatea sa este mică, aproape liniară
2. Poate fi folosit pentru a calcula cea mai scurtă cale de la un singur nod la toate celelalte noduri și de la un singur nod sursă la un singur nod destinație prin oprirea algoritmului odată ce se atinge cea mai scurtă distanță pentru nodul de destinație

Dezavantaje:

1. Nu se poate aplica pe un graf care are cicluri și costuri negative
2. Funcționează doar pe grafuri orientate
3. Face o explorare obscură care consumă multe resurse în timpul procesării
4. Trebuie ținută evidența nodurilor care au fost vizitate

Johnson are următoarele avantaje și dezavantaje:
Avantaje:

1. Funcționează pe grafuri ce au costuri negative
2. Este mai eficient decât algoritmul Floyd-Warshall în cazul grafurilor rare

Dezavantaje:

1. Nu se poate aplica pe un graf care are cicluri negative
2. Funcționează doar pe grafuri orientate

Deși fiecare algoritm are avantajele și dezavantajele sale, situația este cea care ne îndrumă să decidem ce algoritm să utilizăm având dat un anumit graf. Spre exemplu, în situația unei hărți cu orașe, unde orașele sunt nodurile iar drumurile sunt muchiile, graful este unul cu costuri pozitive deci putem aplica un algoritm eficient, dar care are și neajunsul să nu funcționeze pe grafuri cu cost negativ, de pildă Dijkstra. În contextul unei afaceri care arată creșterile/scăderile profitului, avem un graf cu costuri negative deci aplicăm un algoritm ce se adaptează acestui caz. Nu întotdeauna contează cât de rapid este un algoritm, ci sfera mare de aplicabilitate a acestuia (de pildă Johnson: chiar dacă este un algoritm mai lent, acoperă marea majoritate a cazurilor din viața reală cu succes).

3 Evaluare

3.1 Descrierea modalității de construire a setului de teste folosite pentru validare

Am folosit următorul generator de grafuri: https://github.com/dimitriedavid/graph-generator, care având la dispoziție numărul de noduri, procentul densității grafului și alegerea de a insera costuri negative, generează un fișier care conține un input random ce respectă specificațiile din enunț. Fișierele generate sub această formă au fost incluse în folderul "in". În total, am generat 25 de teste de diferite dimensiuni (14 de dimensiuni mici cu grafuri până în 100 de noduri , 11 de dimensiuni mari cu grafuri până în 1000 de noduri), dar care verifică și cum se comportă algoritmii în cazul existenței grafurilor rare, complete, grafurilor cu costuri pozitive, negative și cu ciclu negativ.

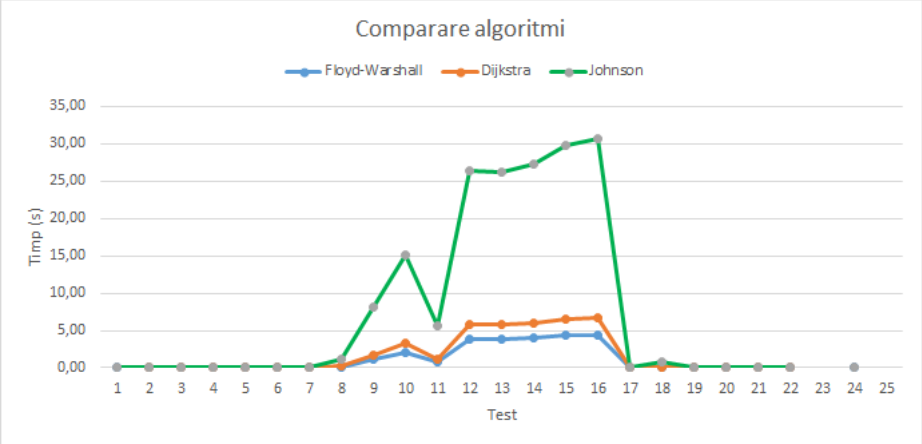
3.2 Menționarea specificațiilor sistemului de calcul pe care au fost rulate testele (procesor, memorie disponibilă)

Procesor: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Memorie: tip DDR4, cu o capacitate de 8GB

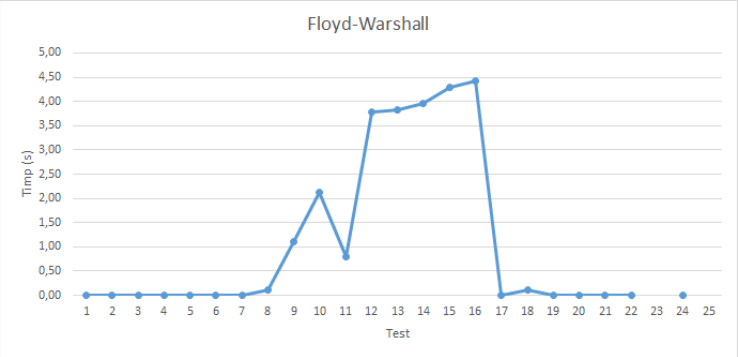
3.3 Ilustrarea, folosind grafice/tabele, a rezultatelor evaluării soluțiilor pe setul de teste

Test/Alg	Floyd-Warshall	Dijkstra	Johnson
1	0,0000015	-	0,1425849
2	0,0000604	0,0002512	0,0849756
3	0,0000040	0,0000109	0,0846478
4	0,0002044	0,0006966	0,0869250
5	0,0000483	0,0001119	0,0859029
6	0,0001519	0,0005208	0,0872925
7	0,0046566	0,0101643	0,1220336
8	0,1063539	0,2168382	1,1068018
9	1,0976545	1,7837573	8,2159560
10	2,1208425	3,3420036	15,1624328
11	0,8079991	1,1988705	5,5534337
12	3,7874157	5,8837140	26,3226566
13	3,8195181	5,7884965	26,1968403
14	3,9663208	5,9716039	27,2952771
15	4,2948456	6,4848499	29,7984694
16	4,4227888	6,6976523	30,6091544
17	0,0027134	0,0048548	0,1070687
18	0,1017962	0,1504924	0,7737192
19	0,0050374	0,0069215	0,1207418
20	0,0000019	0,0000037	0,0822303
21	0,0000041	0,0000080	0,0833580
22	0,0000263	0,0000523	0,0831540
23	-	-	-
24	0,0000092	-	0,0835513
25	-	-	-

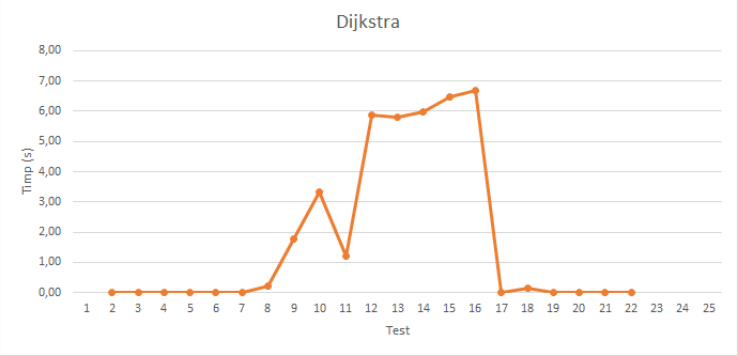
Timpii de execuție



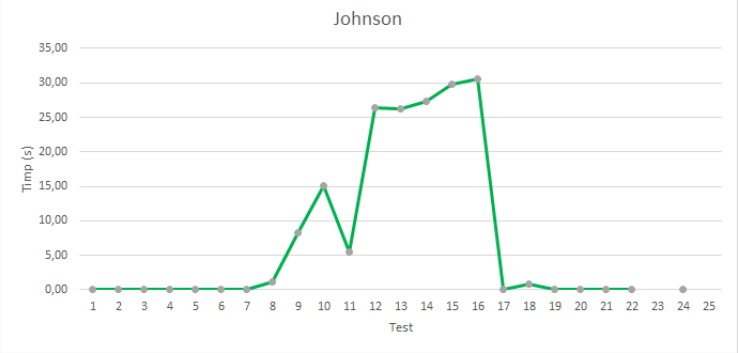
Compararea celor 3 algoritmi în funcție de eficiența timpilor de rulare



Timpii de rulare pentru algoritmul Floyd-Warshall



Timpii de rulare pentru algoritmul Dijkstra



Timpii de rulare pentru algoritmul Johnson

3.4 Prezentarea valorilor obținute pe teste

Pentru fiecare algoritm în parte, am calculat timpii în care aceștia reușesc să găsească drumul minim într-un graf dat. Din primul grafic, se observă că algoritmul Johnson este cel mai lent, iar Floyd-Warshall cel mai eficient ca timp de găsire a drumului minim în graf. Aceste rezultate sunt însă influențate

și de natura testelor create, dar și de modalitatea în care algoritmii au fost implementați (de exemplu, Dijkstra este implementat folosind vector nesortat, mai ineficient decât o implementare folosind Heap). În plus, am creat câte un grafic pentru fiecare algoritm în parte și remarcăm evoluția asemănătoare a acestora în timp pentru fiecare test în parte, însă axa timpului este cea care face diferența, observând timpul maxim la care ajunge fiecare algoritm, și anume pe testul 16, care are 1049 de noduri și 10993 de muchii. În plus, pe grafic apar anumite porțiuni întrerupte, deoarece testele respective implică grafuri cu ciclu negativ. La Dijkstra, apar porțiuni întrerupte și pentru grafurile cu cost negativ (testele 1 și 24), deoarece în acel caz nu am calculat timpii de rulare.
Un alt factor important care a dus la valorile obținute pe teste îl reprezintă specificațiile sistemului de calcul pe care testele au fost rulate. Cu toate că testele de intrare sunt aceleași, la diferite rulări ale algoritmilor nu se obține același output, ca urmare a diferențelor cauzate de hardware. Astfel, pentru o acuratețe mai mare, pentru fiecare algoritm în parte, am rulat de câte 10 ori timpii de găsire a drumului minim din cele 25 de teste, după care am calculat media aritemtică a valorilor obținute.

4 Concluzii

În urma analizei făcute, într-o situație din viața reală, există multe soluții pentru rezolvarea problemei drumului minim, printre care se regăsesc aplicațiile de GPS. Dacă aș avea de dezvoltat o aplicație de acest tip, aș alege algoritmul Dijkstra datorită performanței sale destul de bune, indiferent de structura de date folosită pentru implementare, dar și datorită faptului că nu există muchii de cost negativ într-o hartă cu orașe. O alternativă bună ar fi reprezentată și de algoritmul Floyd-Warshall, al cărui cod este mai ușor de implementat. Utilitatea algoritmului Johnson ar putea fi remarcată într-o situație care ar implica un graf cu costuri negative, cum ar fi în domeniul afacerilor. Spre deosebire de algoritmul Floyd-Warshall, algoritmul Johnson recalculează costurile muchiilor astfel încât acestea să fie pozitive, deci rezultatele ar fi mai sugestive în cazul lui Johnson.

Referințe

1. https://www.adamconrad.dev/blog/shortest-paths/. Ultima accesare: 26 Oct 2021
2. https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/. Ultima accesare: 26 Oct 2021
3. https://www.programiz.com/dsa/floyd-warshall-algorithm. Ultima accesare: 29 Oct 2021
4. https://www.geeksforgeeks.org/johnsons-algorithm/. Ultima accesare: 30 Oct 2021
5. https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/. Ultima accesare: 9 Dec 2021.
6. https://www.sciencedirect.com/topics/computer-science/shortest-path-problem. Ultima accesare: 9 Dec 2021.
7. https://www.quora.com/Why-is-the-order-of-the-loops-in-Floyd-Warshall-algorithm-important-to-its-correctness. Ultima accesare: 9 Dec 2021

8. https://brilliant.org/wiki/floyd-warshall-algorithm/. Pseudocodul algoritmului Floyd-Warshall. Ultima accesare: 10 Dec 2021
9. http://nmamano.com/blog/dijkstra/dijkstra.html. Pseudocodul algoritmului Dijkstra. Ultima accesare: 10 Dec 2021
10. https://moorejs.github.io/APSP-in-parallel/. Pseudocodul algoritmului Johnson. Ultima accesare: 10 Dec 2021
11. https://iq.opengenus.org/time-and-space-complexity-of-dijkstra-algorithm/.Ultima accesare: 11 Dec 2021
12. https://iq.opengenus.org/johnson-algorithm. Ultima accesare: 11 Dec 2021
13. Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, Cliord Stein: Introduction to Algorithms, 3rd edition, The MIT Press, Massachusetts Institute of Techology (2009)