Analiza Algoritmilor - Tema 1
Etapa finala

Tudorache Bogdan Mihai 324CA

Universitatea Politehnica Bucureşti
Facultatea de Automatica şi Calculatoare
bogdanmihait10@gmail.com

Abstract. Acest studiu prezintă problema celor mai scurte drumuri de la un nod la toate celelalte intr-un graf orientat ponderat. Documentul conţine prezentarea soluţilor care rezolvă problema, evaluarea soluţiilor si concluzii care reies din urma efectuării studiului.

Keywords: Drumuri minime · Dijkstra · Bellman Ford · Grafuri aciclice

1 Introducere

1.1 Descrierea Problemei rezolvate

Problema aleasă pentru rezolvare este cea a **drumurilor minime in graf**, mai exact **costul minim de la un nod la toate celelalte**. Pentru un **graf orientat ponderat** $G = (V, E)$ cu V noduri si E muchii, unde fiecare muchie are asociată o pondere w, ne propunem sa aflăm distanta de la un nod v la toate celelalte noduri. Aceasta este una dintre problemele clasice in procesarea de grafuri, si are o mulţime de utilizări.

1.2 Exemple de aplicaţii practice pentru problema aleasa

Aplicaţii imediat evidente ar fi modelarea unei hărti, unde nodurile sunt intersecţii şi muchiile sunt drumuri si ne dorim sa aflăm cea mai scurtă cale dintr-un punct anume, sau in retelistică, unde nodurile pot fi diferite aparate precum routere si switch-uri si muchiile sunt conexiunile. O altă aplicaţie poate fi gestionarea proceselor de catre sistemul de operare (Job Scheduler), unde nodurile sunt procesele, iar muchiile modeleaza o constrangere de ordine.

Seam Carving, un algoritm recent descoperit de redimensionare a imaginilor, poate fi modelat ca o problema de drumuri minime de la un nod la toate celelalte.

1.3 Specificarea soluţiilor alese

Am ales pentru studiu 3 algoritmi care rezolvă problema drumurilor minime in graf:

- **Dijkstra**, care rezolva problema in cazul in care nu avem muchii **negative**. Algoritmul foloseşte spaţiu suplimentar proporţional cu $O(V)$ si rezolva problema in timp proporţional cu $O(E \log V)$ (pentru implementarea cu coadă);
- **Bellman-Ford**, care functioneaza pentru orice tip de graf orientat. Foloseşte spaţiu suplimentar proporţional cu $O(V)$ si are complexitatea de timp $O(EV)$;
- **Algoritm eficient pentru graf orientat aciclic (DAG)**, care utilizeaza **sortarea topologica** a nodurilor. Problema se poate rezolva in timp proporţional cu $O(E + V)$.

Fiecare algoritm va fi studiat in detaliu, specificand pentru fiecare **complexitatea**, **viteza**, **limitări**, **structuri de date** folosite şi **detalii de implementare**.

1.4 Specificarea criteriilor de evaluare alese pentru validarea soluţiilor

Algoritmii vor fi evaluati in functie de mai multe criterii, cum ar fi **viteza**, **utilitate practica**, **dificultatea implementarii**. Astfel, trebuie intocmit un set de date ce va verifica doua proprietăţi principale: **cazuri limita** şi **viteza de executie**.

Pentru prima categorie ne dorim ca algoritmii să funcţioneze pentru cat mai multe situaţii si cat mai variate. Voi testa corectitudinea implementării, trecand algoritmii prin toate situaţiile limită. Ne dorim ca algoritmii sa acţioneze bine in situaţii reale, unde de cele mai multe ori ne vom intalni cu **worst case scenarios**.

Pentru a doua situaţie vrem sa vedem cat de rapid ruleaza algoritmii pe seturi de date foarte mari. Avand siguranţa ca implementarea este corecta, ne dorim sa testam eficienţa.

2 Prezentarea soluţiilor

2.1 Descrierea modului în care funcţionează algoritmii aleşi

Toţi algoritmii prezentaţi folosesc operaţia de **relaxare** a muchiilor.

Această operaţie simplă funcţionează in felul urmator: considerăm ca fiind sursă nodul s. Pentru a relaxa muchia de la nodul v la nodul w, trebuie sa verificăm daca cel mai scurt drum de la nodul s la nodul w este să mergem din nodul s la nodul v, si din nodul v la nodul w. În caz afirmativ, actualizăm structura de date in care memorăm dinstanţele.

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
   }
}
```

Fig. 1. Implementare a operaţiei de relaxare, imagine preluata din cartea Algorithms, 4th Edition[1]

Fiecare algoritm descris se foloseşte de această operaţie in moduri diferite.

Disjkstra

Pentru algoritmul lui Dijkstra, incepand de la nodul sursa, considerăm mulţimea nodurilor care sunt prelucrate, şi multimea nodurilor care urmeaza să fie prelucrate. Cat timp mulţimea nodurilor care sunt prelucrate nu este egala cu multimea tuturor nodurilor, căutăm nodul cel mai apropriat de mulţimea pe care vrem să o formam şi actualizăm distanţele.

Pentru Dijkstra exista doua implementări generale: **cu coada si fără coadă**.

Implementarea **cu coadă**, bazată pe o reprezentare a grafului prin liste de adiacenţă, foloseste o coada de priorităţi pentru muchii pentru a găsi rapid cel mai apropriat nod.

Implementarea **fara coadă**, bazată pe o reprezentarea a grafului prin matrice de adiacenţa, caută "manual" nodul cel mai apropriat.

Bellman-Ford

Bellman-Ford **fara coadă** nu face decat sa ia toate muchiile E si sa le relaxeze in ordine, de V ori.

Bellman-Ford cu **coadă** aduce o optimizare peste Bellman-Ford fara coadă, pentru a evita parcurgerea tuturor muchiilor de exact V ori. Pentru fiecare muchie, in timpul relaxării, dacă distanţa de la nodul sursă la nodul căutat s-a schimbat, adăugam nodul in coadă, si continuăm cu prelucrarea nodurilor in ordinea din coadă. Astfel, în cazul în care nu se mai poate relaxa nici o muchie, algoritmul se va termina mai rapid.

Algoritm pentru grafuri aciclice
Pentru acest algoritm, pur si simplu obţinem **sortarea topologică** a grafului printr-o parcurgere in adancime, dupa care relaxăm muchiile fiecărui nod din ordinea topologică.

2.2 Analiza complexitaţii algoritmilor

Analiza complexităţii pentru fiecare algoritm in parte:

Dijkstra
Pentru implementarea fara coadă, complexitatea reiese din cat de rapid găsim nodul cel mai apropriat de multimea nodurilor prelucrate. Pentru fiecare nod, o sa fie nevoie sa verificăm toate celelalte noduri pentru a vedea care este cel mai apropriat nod din distanţele cunoscute la acel moment. Suma rezultantă este:

$$(V-1)+(V-2)+(V-3)+...+1=\frac{(V-1)V}{2}$$

Astfel, complexitatea este $O(V^2)$.

Pentru implementarea cu coadă, complexitatea algoritmului depinde de modul in care este implementată coada. Această structura de date trebuie sa conţina 3 operaţii: **Insert**, **Extract-Min**, si **Decrease-Key**. In cazul unei cozi de prioritaţi implementate printr-un heap binar, timpul de constructie este $O(V)$, Extract-Min are loc in O(log V) iar Decrease-Key tot in $O(\log V)$. Extract-Min va fi apelat odată pentru fiecare nod, deci de V ori, iar Decrease-Key va fi apelat de cel mult E ori (pentru fiecare muchie). Astfel, complexitatea finala este $O(V\log V + E\log V)$ care este $O(E\log V)$[2].

Bellman-Ford
In cazul algoritmului fără coadă, complexitatea este simplu de calculat. Daca pentru fiecare nod V relaxăm fiecare muchie din graf E, complexitatea o sa fie $O(VE)$.

Complexitatea pentru Bellman-Ford cu coadă este aceeaşi, $O(VE)$, numai că în practică nu va fi nevoie să prelucrăm toate muchiile E de V ori. In mod sigur algoritmul va termina orice relaxare posibilă mult mai rapid.

Algoritm pentru grafuri aciclice
Pentru acest algoritm este usor de calculat complexitatea. Sortarea topologica se face in $O(V+E)$ iar relaxarea tuturor muchiilor in ordinea sortării topologice tot in $O(V+E)$. Complexitatea finală este $O(V+E)$.

2.3 Prezentarea principalelor avantaje si dezavantaje ale soluţiilor

Fiecare dintre algoritmii prezentaţi exceleaza in anumite situaţii specifice.

Algoritmul pentru grafuri aciclice este cel mai rapid ca si timp de execuţie, dar funcţioneaza numai pentru **grafuri aciclice**.

Dijkstra este urmatorul algoritm ca şi viteză, doar că poate fi folosit numai in grafuri cu **muchii pozitive**.

Al treilea algoritm ca şi viteză este Bellman-Ford, pentru care singura restrictie este că algoritmul nu funcţioneaza pe grafuri care au **cicluri negative**.

Ca şi dificultate de implementare, doar Dijkstra cu coadă poate impune probleme, pentru ca avem nevoie de o coadă de priorităţi specializată, care are in plus operaţia de **Decrease-Key**. Ceilalţi algoritmi sunt accesibili din punct de vedere al implementării

algorithm	restriction	path length compares (order of growth) typical	path length compares (order of growth) worst case	extra space	sweet spot
Dijkstra (eager)	positive edge weights	$E \log V$	$E \log V$	V	worst-case guarantee
topological sort	edge-weighted DAGs	$E + V$	$E + V$	V	optimal for acyclic
Bellman-Ford (queue-based)	no negative cycles	$E + V$	VE	V	widely applicable

Performance characteristics of shortest-paths algorithms

Fig. 2. Tabel cu principalele caracteristici ale algoritmilor, imagine preluata din Algorithms, 4th Edition[1]

3 Evaluare

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

Pentru fiecare algoritm am alcătuit 10 teste, in total avand 30 de teste.

Primele 5 teste pentru fiecare algoritm sunt construite ”de mana” pentru a fi verificate usor. Sunt teste de dimensiuni mici, pentru a verifica concret corectitudinea algoritmilor.

Urmatoarele 5 teste pentru fiecare algoritm cresc progresiv in dimensiuni, atat in noduri cat si in muchii. Pentru formarea lor am scris un script in Python ce poate să genereze grafuri si care are mai multe opţiuni pentru diferite tipuri de grafuri.

Testele pentru Dijkstra conţin grafuri orientate cu cicluri si cu muchii pozitive, cele pentru Bellman-Ford conţin grafuri cu cicluri, cu muchii pozitive si negative, iar algoritmul pentru grafuri aciclice conţine numai grafuri aciclice.

3.2 Specificaţiile sistemului de calcul

Soluţiile au fost testate pe laptop-ul personal, un Lenovo T440s (vezi Fig. 3)

Processor:	Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz 2.69 GHz
Installed memory (RAM):	12.0 GB (11.9 GB usable)
System type:	64-bit Operating System, x64-based processor

Fig. 3. Specificaţiile sistemului pe care au fost testate soluţiile

3.3 Ilustrarea rezultatelor evaluării soluţiilor pe setul de test

Algoritmii au fost implementaţi in Java si am măsurat timpul de execuţie folosind funcţia $System.nanoTime()$. Table 1 conţine rezultatele cronometrării testelor.

Table 1. Timpul de rulare al algoritmilor in milisecunde

Nr. test	Dijkstra	Bellman-Ford	Acyclic
0	0.1041	0.0109	1.2234
1	0.0994	0.0144	0.0466
2	0.0767	0.0143	0.0249
3	2.1043	0.0143	0.0213
4	0.1545	0.1091	0.0255
5	0.0246	0.0293	0.1584
6	3.5975	0.1396	0.2976
7	2.3464	0.4503	0.9818
8	55.6837	19.7222	32.9567
9	40.1515	78.9837	47.0182

In urmatoarele tabele voi ilustra numărul de muchii şi numărul de noduri pentru fiecare categorie de teste.

Dijkstra

Testele de la Dijkstra conţin grafuri cu cicluri si cu muchii pozitive.

Table 2. Datele de intrare pentru testele de la Dijkstra

Nr. test	Nr. noduri	Nr. muchii
0	4	3
1	5	6
2	6	10
3	4	9
4	7	14
5	10	36
6	50	962
7	100	3917
8	500	74945
9	1000	300036

Bellman-Ford

Testele de la Bellman-Ford conţin grafuri cu cicluri si cu muchii pozitive si negative. Ultimile 2 teste nu conţin muchii negative.

Table 3. Datele de intrare pentru testele de la Bellman-Ford

Nr. test	Nr. noduri	Nr. muchii
0	4	3
1	5	6
2	6	10
3	4	9
4	7	14
5	10	15
6	50	499
7	100	904
8	500	75179
9	1000	300003

Algoritm Aciclic

Testele pentru algoritmul aciclic conţin grafuri aciclice.

Table 4. Datele de intrare pentru testele de la algoritmul pentru grafuri aciclice

Nr. test	Nr. noduri	Nr. muchii
0	4	3
1	5	6
2	7	11
3	7	11
4	7	11
5	10	17
6	50	355
7	100	1492
8	500	37718
9	1000	150188

3.4 Prezentarea valorilor obţinute

Tabelele ilustreaza eficienţa algoritmilor pe diferite teste, in situaţiile specifice fiecărui algoritm. Se poate observa diferenţa de viteză intre cei trei algoritmi. Astfel, algoritmul aciclic este cel mai rapid, urmat de Dijkstra şi de Bellman-Ford.

Există totuşi nişte valori atipice in tabelul de viteză. Testul 2 de la Dijkstra si testul 1 de la algoritmul aciclic au timpul de execuţie relativ mare pentru un numar mic de noduri si de muchii, insă nu am putut găsi o explicaţie pentru acest fapt.

Testul 9 de la Dijkstra se termina mai repede decat testul 8 de la Dijkstra, deşi datele de intrare de la testul 9 sunt mai mari decat la testul 8. O explicaţie ar fi că, fiind multe muchii, algoritmul găseste drumul minim final rapid, şi nu se mai efectueaza operaţii pe coada de priorităţi, reducand timpul de execuţie.

4 Concluzii

Fiecare dintre cei trei algoritmi este folositor intr-o anumită situaţie, nu se poate spune exact că unul este mai bun decat celălalt. In funcţie de restricţiile impuse de graful care trebuie prelucrat, in practică voi alege algoritmul corespunzător.

Fiecare dintre algoritmi pot fi adaptaţi pentru situaţii si mai specifice, pentru a rula cat mai rapid pe tipuri cat mai restrictive de grafuri.

References

1. *Algorithms, 4th Edition*,
   Robert Sedgewick, Kevin Wayne
2. *Introduction To Algorithms, 3rd Edition*,
   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
3. Articolul despre Dijkstra fără coadă de pe GeeksForGeeks,
   https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/