Drumuri minime în graf

Problema găsirii drumului de cost minim dintre două noduri ale unui graf

Călin Jugănaru, 324CA

Universitatea POLITEHNICA din Bucureşti

Facultatea de Automatică şi Calculatoare

calin_vlad.juganaru@stud.acs.upb.ro

April 22, 2019

Drumuri minime în graf

1 Descrierea problemei

1.1 Noţiuni introductive

Dat fiind un graf **G = (V, E)**, adică o pereche de mulţimi de noduri (**V**), respectiv legături dintre acestea (**E**), numite şi muchii sau arce, SSSP *(Single Source Shortest Path)* înseamnă problema găsirii drumului de cost minim de la un nod ales la toate celelalte din graf. Acest cost minim reprezintă suma ponderilor arcelor care formează drumul optim găsit între două noduri. Numim graf ponderat un graf ce are asociat câte un cost (o pondere) pentru fiecare arc din mulţimea **E**. Pentru grafurile neponderate putem considera că toate arcele au cost unitar şi acesta va fi un caz particular al problemei studiate.

Pe de altă parte, există şi problema APSP *(All Pairs Shortest Path)*, unde ne propunem să găsim drumul de cost minim dintre oricare două noduri dintr-un graf, dar în lucrarea de faţă ne vom ocupa doar de primul caz, acela al unui nod de pornire fixat. Este evident faptul că problema SSSP este un caz particular al problemei APSP: dintre toate perechile de noduri le agelem doar pe acelea ce conţin nodul fixat. De asemenea, un caz particular al problemei SSSP este aflarea drumului de cost minim dintre două noduri fixate.

Vom considera toate grafurile la care ne vom referi în această lucrare ca fiind orientate, fiindcă orice graf neorientat poate fi privit ca unul orientat, dar bidirecţional. Acest detaliu este relevant pentru al treilea algoritm ce va fi prezentat, fiindcă acela funcţionează numai pe grafuri orientate aciclice.

1.2 Soluţiile alese şi aplicaţia practică

Considerând o hartă a unei ţări, formată dintr-o mulţime **V** de localităţi şi o mulţime **E** de drumuri ce le leagă, apoi asociind câte o pondere fiecărui drum, ce va reprezenta lungimea sa reală, această structură poate fi descrisă de un graf ponderat **G = (V, E)**. Dacă ne propunem să găsim drumul de cost minim (în cazul de faţă, cel cu distanţa parcursă cea mai mică) dintre capitală şi toate celelalte mari oraşe din ţară, un algoritm de SSSP ne va da soluţia problemei căutate. De asemenea, în cazul în care căutăm doar traseul cel mai scurt dintre două oraşe alese, putem obţine soluţia folosind un astfel de algoritm.

Algoritmii de tip SSSP pe care îi vom prezenta şi compara în lucrarea de faţă sunt: algoritmul lui Edsger Dijkstra, algoritmul Bellman-Ford (întâlnit uneori ca Bellman-Ford-Moore) şi o variantă modificată celui din urmă, optimizată pentru grafuri orientate aciclice. Pentru primii doi algoritmi, vom studia şi câte o variantă uşor modificată a lor, prin folosirea unei anumite structuri de date în locul unei iteraţii prin mulţimea nodurilor.

Ca exeplu practic des întâlnit în viaţa de zi cu zi, Google Maps foloseşte pentru calcularea rutelor optime un algoritm SSSP ce are la bază algoritmul lui Dijkstra.

1.3 Criterii de evaluare

Cei trei algoritmi aleşi vor fi implementaţi pentru testare în limbajul C++17 şi nu vor avea optimizări semnificative ce să depindă de limbajul folosit, ci doar de structurile de date auxiliare,

pentru a putea compara obiectiv eficienţa algoritmilor şi nu a implementării acestora.

Pentru testarea funcţionalităţii algoritmilor vom genera un set de fişiere ce conţin de date de intrare sub forma unor liste de adiacenţă cât mai variate: grafuri cu număr de noduri şi arce de ordinul zecilor, sutelor şi chiar miilor, grafuri dense şi rare, conexe şi neconexe, grafuri cu ponderi ale arcelor foarte distanţate sau apropiate de o medie, şi cazuri cu ponderi unitare sau negative. Ultimul exemplu este important deoarece algoritmul lui Dijkstra nu funcţionează atunci când există costuri negative în graf (nu va da răspunsul corect), pe când algoritmul Bellman-Ford funcţionează de obicei în acest caz, dar nu şi atunci când există bucle cu costul total negativ (ar însemna ca la fiecare parcurgere a buclei să obţinem un cost mai mic decât la cea anterioară). Într-un asemenea caz, vom considera că nu există un drum de cost minim între oricare două noduri din graf, la fel şi atunci când graful nu este conex, deoarece este imposibil să ajungem în anumite noduri pornind din cel stabilit.

Mai mult decât atât, vom testa variantele originale ale algoritmilor menţionaţi şi variante optimizate, care folosesc anumite structuri de date (cozi / cozi de prioritate) şi reduc ordinul de complexitate, iar grafurile vor fi reţinute în memorie atât sub forma unor liste de adiacenţă, cât şi ca matrici, pentru realizarea unor comparaţii ale diferitelor abordări.

Pentru verificarea corectitudinii algoritmilor putem genera toate drumurile posibile în graf pornind de la un nod fixat şi să adunăm costurile de pe arcele parcurse, găsind drumurile de cost minim şi comparându-le cu rezultatele obţinute după executarea algoritmilor implementaţi. Totuşi, această metodă poate fi folosită doar pentru date de intrare suficient de mici, având în vedere resursele hardware limitate.

Această metodă 'empirică', totuşi, nu demonstrează corectitudinea globală a algoritmilor, ci doar locală, pe datele de intrare generate de noi. Din rularea a suficient de multe astfel de teste putem considera că aceşti algoritmi funcţionează şi în celelalte cazuri, dar, pentru demonstrarea corectitudinii lor globale vom folosi metode teoretice de analiză a corectitudinii.

2 Prezentarea soluţiilor problemei

2.1 Descrierea algoritmilor şi analiza complexităţii

1. **Algoritmul lui Dijkstra clasic**

   Acest algoritm se bazează pe două concepte: acela de nod finalizat, adică un nod aflat la distanţa minimă faţă de cel de pornire, care va fi folosit apoi la calcularea distanţelor până la cele adiacente lui, şi de muchie relaxată, o muchie ce porneşte dintr-un nod finalizat şi ponderea acesteia este folosită la procedeul de relaxare. Pentru a calcula distanţa minimă de la un nod fixat la toate celelate dintr-un graf $\mathbf{G = (V, E)}$, algoritmul va efectua $|\mathbf{V}|$ (numărul de noduri ale grafului) iteraţii, în care va căuta un nod nefinalizat (nefolosit la calculul altor distanţe) din mulţimea nodurilor, având distanţa minimă faţă de sursă şi va actualiza distanţele minime calculate (va relaxa muchiile) până la nodurile adiacente acestuia dacă distanţa minimă până la acest nod + costul muchiei dintre acesta şi un vecin este mai mică decât distanţa actuală[1].
   Vom deduce complexitatea algoritmului analizând liniile 6, 9 şi 14 din pseudocodul de mai jos. Dacă desfacem bucla de la linia 9 şi o ataşam celorlalte două ca şi cum ar lucra independent una de cealaltă, vom obţine pentru prima o complexitate $\mathbf{O(|V|\cdot|V|) = O(|V|^2)}$.
   Pentru cea de-a doua este necesară o menţiune: a itera prin toată mulţimea nodurilor ($\mathbf{V}$), şi, pentru fiecare nod, prin toate muchiile (arcele) care pornesc din acesta, înseamnă, de fapt, a itera prin mulţimea tuturor muchiilor ($\mathbf{E}$). Astfel, complexitatea celei de-a doua bucle va fi $\mathbf{O(|E|)}$, deci algoritmul are complexitatea totală $\mathbf{O(|V|^2 + |E|)}$.

Observăm că funcţia depinde de valorile ambelor cardinale, dar, în cazul unui graf dens, $|\mathbf{E}|$ are cam acelaşi ordin de mărime cu de $\frac{|\mathbf{V}|^2}{2}$. Cum în analiza complexităţii nu luăm în considerare constantele, putem concluziona apartenenţa la $\mathbf{O(|V|^2)}$. În cazul unui graf rar, $|\mathbf{E}|$ este neglijabil faţă de $|\mathbf{V}|^2$, deci nu-l vom lua în considerare.

```
1   dijkstra_classic(G = (V, E), start)
2       N = |V|
3       dist[1...N] = ∞
4       finalizat[1...N] = false
5       dist[start] = 0
6       repetă de N ori
7           dist_min = ∞
8           nod_min = start
9           pentru ∀ nod ∈ V
10              dacă !finalizat[nod] & (dist_min > dist[nod])
11                  dist_min = dist[nod]
12                  nod_min = nod
13              finalizat[nod] = true
14              pentru ∀ arc = (nod, vecin, cost) ∈ E
15                  dacă dist[vecin] > dist[nod_min] + cost
16                      dist[vecin] = dist[nod_min] + cost
17      return dist
```

2. **Algoritmul lui Dijkstra cu heap**

Pentru a optimiza complexitatea de timp a algoritmului precedent, în locul căutării liniare (parcurgerea de la 1 la $|\mathbf{V}|$) a nodului nefinalizat de distanţă minimă, putem folosi o structură de date de tip *min_heap*, sau o coadă de priorităţi bazată pe acesta, în care inserăm iniţial nodul de pornire, apoi, cât timp heap-ul nu este gol, extragem un nod şi actualizăm distanţele până la vecinii săi (relaxăm muchiile), la fel ca mai sus, iar dacă am găsit o nouă distanţă minimă vom folosi acest nod vecin la calculul următoarelor distanţe, adăugându-l în heap. Astfel, folosind un *min_heap*, după fiecare inserare a unui nod, structura arborescentă se va actualiza automat în timp logaritmic, punând ca rădăcină nodul cu distanţa minimă faţă de origine, deci am redus acea complexitate de $\mathbf{O(|V|)}$ a căutării la $\mathbf{O}(log(|\mathbf{V}|))$. În final, am redus complexitatea totală a algoritmului la $\mathbf{O}(|\mathbf{V}|\cdot log(|\mathbf{V}|))$.

```
1   dijkstra_heap(G = (V, E), start)
2       N = |V|
3       dist[1...N] = ∞
4       heap = ∅
5       dist[start] = 0
6       heap.push(start)
7       cât timp heap-ul nu este gol
8           nod = heap.pop()
9           pentru ∀ arc = (nod, vecin, cost) ∈ E
10              dacă dist[vecin] > dist[nod] + cost
11                  dist[vecin] = dist[nod] + cost
12                  heap.push(vecin)
13      return dist
```

3. **Algoritmul Bellman-Ford clasic[2]**

Despre acest algoritm putem spune că este mai simplist, naiv, intuitiv sau chiar *brute-force* decât oricare dintre cei prezentaţi în această lucrare: faţă de algoritmul lui Dijkstra clasic, nu vom căuta un nod nefinalizat sau de distanţă minimă, ci, pur şi simplu, de $|\mathbf{V}|$ ori vom itera prin mulţimea de noduri şi vom încerca actualizarea distanţelor până la orice nod adiacent celui curent. În concluzie, efectuând această iterare de $|\mathbf{V}|$ ori, obţinem complexitatea algoritmului $\mathbf{O(|V|\cdot|E|)}$.

Totuşi, acest algoritm şi următorii doi au un avantaj faţă de primii doi: în cazul în care graful conţine muchii (arce) cu ponderi negative, algoritmii Dijkstra nu vor genera rezultatele corecte, dar acestea vor funcţiona. Mai mult decât atât, dacă în urma iteraţiilor descrise mai sus mai efectuează la final o iterare prin mulţimea nodurilor, încearcă actualizarea distanţelor minime calculate şi găseşte o distanţă şi mai mică, înseamnă că graful conţine (cel puţin) o buclă de cost total negativ, deci nu pot fi calculate distanţele minime, iar problema noastră nu are soluţie. Acest caz ar însemna că la fiecare trecere printr-o astfel de buclă vom obţine o distanţă mai mică decât cea anterioară, până la -∞, iar formularea problemei studiate aici nu impune sau restricţionează în vreun fel traseul urmat prin graf pentru obţinerea distanţelor minime, ci doar nodul de pornire.

```
1   bellman_ford_classic(G = (V, E), start)
2       N = |V|
3       dist[1...N] = ∞
4       dist[start] = 0

5       repetă de N ori
6           pentru ∀ arc = (nod, vecin, cost) ∈ E
7               dacă dist[vecin] > dist[nod] + cost
8                   dist[vecin] = dist[nod] + cost

9       pentru ∀ arc = (nod, vecin, cost) ∈ E
10          dacă dist[vecin] > dist[nod] + cost
11              ∃ un ciclu negativ

12      return dist
```

4. **Algoritmul Bellman-Ford cu coadă**

La fel ca în cazul Dijkstra, putem optimiza algoritmul folosind o anumită structură de date pentru înlocuirea unor iteraţii costisitoare. Dacă în locul parcurgerii mulţimii nodurilor de $|\mathbf{V}|$ ori, folosim o structură de tip coadă, complexitatea în cazul cel mai defavorabil va rămâne aceeaşi, dar în practică se va dovedi mult mai bună. Algoritmul clasic va parcurge întotdeauna şi pentru orice date de intrare întreaga mulţime de noduri, dar, îl putem optimiza adăugând în coadă numai nodurile folosite pentru relaxarea muchiilor, situaţie asemănătoare nodurilor finalizate din algoritmul lui Dijkstra. Pentru a conserva abilitatea algoritmului de găsire a unei bucle negative, vom contoriza fiecare extragere a unui nod din coadă; dacă acest contor pentru un nod ajunge să fie egal cu $|\mathbf{V}|$, înseamnă că am găsit o buclă negativă.
Complexitatea finală rămâne de $\mathbf{O(|V|\cdot|E|)}$, dar comportamentul real se dovedeşte a fi mult mai eficient[4] şi, în unele cazuri, cel mai bun dintre cei cinci algoritmi prezentaţi.

```
1   bellman_ford_queue(G = (V, E), start)
2       N = |V|
3       dist[1...N] = ∞
4       vizitat[1...N] = 0
5       coadă = ∅

6       dist[start] = 0
7       coadă.push(start)

8       cât timp coada nu este goală
9           nod = coadă.pop()
10          vizitat[nod] ++

11          dacă vizitat[nod] == N
12              ∃ un ciclu negativ

13          pentru ∀ arc = (nod, vecin, cost) ∈ E
14              dacă dist[vecin] > dist[nod] + cost
15                  dist[vecin] = dist[nod] + cost
16                  coadă.push(vecin)

17      return dist
```

5. **Algoritmul Bellman-Ford cu sortare topologică[3]**

Pentru cazul special în care graful studiat este un DAG (*Directed Acyclic Graph* / graf orientat aciclic), pe acesta poate fi aplicată o sortare topologică, care, prin definiţie, înseamnă o ordonare liniară a nodurilor sale, astfel încât, pentru fiecare arc **(x, y)** din **E**, nodul **x** va precede nodul **y** în această ordonare. Definiţia este echivalentă cu: date fiind două noduri **x** şi **y** din graf, iar **x** precede **y** în orice sortare topologică a grafului, rezultă că nu există un drum de la nodul **y** la nodul **x**.
Din această proprietate rezultă că ar fi inutil să căutăm o distanţă minimă de la **y** la **x**, deci este suficient să parcurgem nodurile într-o ordine topologică şi numai atunci să încercăm relaxarea arcelor.
Sortarea topologică poate fi realizată în două moduri: prin metoda gradelor sau printr-o parcurgere în adâncime a tuturor nodurilor din graf şi reţinerea lor într-o listă în ordinea inversă a ieşirii din funcţia de parcurgere, variantă pe care am implementat-o pentru testarea algoritmilor. Complexitatea oricăreia dintre cele doua variante de sortare topologică este **O(|V|+|E|)**, iar a algoritmului în sine, analizând liniile 5 şi 6 de mai jos, este **O(|E|)**, deci obţinem o complexitate finală de **O(|V|+|E|)**.

```
1   bellman_ford_topo(G = (V, E), start)
2       ord_topo = sortare_topologică(G)
3       dist[1...N] = ∞
4       dist[start] = 0

5       pentru ∀ nod ∈ ord_topo
6           pentru ∀ arc = (nod, vecin, cost) ∈ E
7               dacă dist[vecin] > dist[nod] + cost
8                   dist[vecin] = dist[nod] + cost

9       return dist
```

2.2 Avantaje şi dezavantaje

Până să analizăm comportamentul în practică al algoritmilor prezentaţi, putem deduce şi menţiona câteva avantaje şi dezavantaje ale acestora.

Când graful conţine muchii de ponderi negative, algoritmul lui Dijkstra (optimizat sau nu) nu va da întotdeauna rezultatele corecte: acesta este construit pe baza axiomei că ponderile sunt pozitive, deci adăugând încă o muchie la un traseu, acesta nu poate deveni mai scurt. Fiindcă foloseşte optimul local pentru a ajunge la optimul global, după ce finalizează un nod, nu-l va mai folosi ulterior la relaxarea muchiilor, când există posibilitatea de a lua în calcul o muchie de cost negativ incidentă în acel nod şi a obţine o distanţă mai mică decât cea calculată anterior.

Algoritmul Bellman-Ford (toate cele trei variante prezentate) nu are această problemă fiindcă va lua în calcul toate nodurile şi toate muchiile. Datorită acestei calităţi poate descoperi existenţa unei bucle negative în graf, caz în care niciun algoritm dintre aceştia nu va da răspunsul corect, fiindcă problema nu are soluţie. Este vizibil din descrierile de mai sus că acest avantaj vine cu dezavantajul unei complexităţi mai mari.

Ultimul algoritm este indubitabil cel mai eficient dintre aceştia, din punct de vedere al complexităţii, dar poate fi aplicat pe o mulţime restrânsă de grafuri, neapărat orientate şi aciclice, cu orice fel de ponderi. De asemenea, toţi ceilalţi algoritmi pot funcţiona indiferent dacă graful este orientat sau nu.

Nu în ultimul rând, doi dintre algoritmi, cei 'clasici', sunt complet neoptimizaţi, deci au un clar dezavantaj în faţa celorlalţi trei, dar prezentarea lor este utilă în scop didactic.

3 Evaluarea algoritmilor

3.1 Setul de teste

Având în vedere restricţia impusă asupra numărului de teste (între 5 şi 10), am ales generarea a 10 teste care să cuprindă cazuri posibile pentru structura şi dimensiunea grafului, precum şi a ponderilor:

1. graf orientat aciclic rar, cu ponderi unitare
2. graf orientat aciclic rar, cu ponderi pozitive şi negative
3. graf orientat aciclic rar, cu ponderi pozitive mici
4. graf orientat aciclic rar, cu ponderi pozitive mari
5. graf orientat rar, cu cicluri şi ponderi pozitive
6. graf orientat aciclic dens, cu ponderi unitare
7. graf orientat aciclic dens, cu ponderi pozitive şi negative
8. graf orientat aciclic dens, cu ponderi pozitive mici
9. graf orientat aciclic dens, cu ponderi pozitive mari
10. graf orientat dens, cu cicluri şi ponderi pozitive

3.2 Rezultatele obţinute

	test 1	test 2	test 3	test 4	test 5
dijkstra_classic	0.00453704	0.00865469	0.00854924	0.00447580	0.00448099
dijkstra_heap	0.00010744	0.00013690	0.00020867	0.00017749	0.00031419
bellman_ford_classic	0.02901970	0.04250350	0.05664500	0.03530740	0.04951330
bellman_ford_queue	0.00003947	0.00005196	0.00007507	0.00006429	0.00018871
bellman_ford_topo	0.00013729	0.00016273	0.00017764	0.00014562	———

	test 6	test 7	test 8	test 9	test 10
dijkstra_classic	0.00141310	0.0014464	0.00259722	0.00282810	0.00293342
dijkstra_heap	0.00166487	0.0653319	0.00210536	0.00293078	0.00787709
bellman_ford_classic	0.20363700	0.2059870	0.34725600	0.49071700	0.92628900
bellman_ford_queue	0.00204445	0.0190473	0.00285553	0.00445682	0.01327470
bellman_ford_topo	0.00073198	0.0009147	0.00109577	0.00149374	———

3.3 Sistemul de calcul folosit

Algoritmii au fost implementaţi şi testaţi pe o maşină cu procesor Intel Core i5, cu 2.6GHz şi 8GB RAM (7.85GB disponibili), folosind compilatorul GNU g++ verisunea 8.1.0, pe un sistem de operare Linux pe 64 de biţi.

4 Concluzii

Din analiza făcută asupra algoritmilor aleşi, în secţiunea 2, şi din evaluarea lor prezentată în secţiunea 3 putem trage următoarele concluzii:

1. În practică vom exclude din start algoritmul Bellman-Ford neoptimizat: este de departe cel mai ineficient, şi poate fi oricând înlocuit de varianta optimizată prin folosirea unei cozi.
2. Pentru grafurile rare, Bellman-Ford optimizat cu coadă se dovedeşte cel mai rapid, urmat de Dijkstra cu heap şi cel cu sortarea topologică (valorile acestora două sunt în medie foarte apropiate).
3. În cazul unui DAG dens, cel mai eficient algoritm este clar Bellman-Ford optimizat cu sortarea topologică. Cum verificarea unui graf dacă este sau nu DAG poate fi facută în maxim $\mathbf{O(|V|+|E|)}$, cu o parcurgere în adâncime şi cu ieşirea imediată din funcţie când găsim o buclă (considerăm inclusiv că o muchie bidirecţională este o buclă), este rezonabil să facem întâi această verificare şi apoi să aplicăm algoritmul cel mai eficient. Pentru determinarea densităţii unui graf este necesară o simplă comparaţie a numărului de noduri cu numărul de muchii.
4. Pentru grafurile dense şi fără ponderi negative, dar indiferent de mărimea lor, Dijkstra clasic, Dijkstra cu heap şi Bellman-Ford cu coadă par a se comporta fără diferenţe semnificative la timpul de execuţie.
5. Chiar dacă în teorie Dijsktra clasic are complexitatea mai mare decât cel cu heap, pe grafurile dense, cu cicluri şi, bineînţeles, cu ponderi pozitive, acesta este semnificativ mai rapid (de peste două ori).

Bibliografie

[1] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik. 1: 269-271 Addison-Wesley, Reading, Massachusetts, 1993.

[2] Bellman, Richard (1958). On a routing problem. Quarterly Applied Mathematics 16, 87-90.

[3] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. Introduction to Algorithms. 1990. MIT Press.

[4] Wei Zhang, Hao Chen, Chong Jiang, Lin Zhu. Improvement And Experimental Evaluation Bellman-Ford Algorithm. International Conference on Advanced Information and Communication Technology for Education (ICAICTE) 2013.