DISTRIBUIRANI PROCESI

PROJEKTNI ZADATAK

Distribuirani algoritmi za traženje najkraćih putova

Dominik Mikulčić

Petra Škrabo

Zagreb, lipanj 2021.

Sadržaj

1 Uvod

Grafove susrećemo svuda oko nas. Razni problemi iz naše okoline mogu se modelirati grafovima, a rješavanje istih tada je ekvivalentno traženju najkraćeg puta u grafu.

Ovaj rad bavi se algoritmima za pronalazak najkraćeg puta u težinskom grafu, a fokus je stavljen na *Bellman-Fordov* odnosno *Floyd-Warshallov* algoritam. Za razliku od uobičajnih sekvencijalnih verzija spomenutih algoritama, u radu se opisuju, implementiraju i testiraju njihove distribuirane varijante. Za implementaciju algoritama korišten je Java programski jezik.

Na početku se navodi teorijska podloga područja teorije grafova kao baza za rješavanje problema najkraćih putova u grafu.

Teorija grafova

Definicija 1. *Graf* definiramo kao uređeni par skupova (V, E), gdje je V skup vrhova, a E skup 2–podskupova od V, koje zovemo bridovi. *Podgraf* grafa $G = (V, E)$ je graf kojem su skup vrhova i skup bridova podskupovi od V i E, redom.

Definicija 2. *Usmjereni graf* D je uređeni par skupova (V, E'), gdje je V skup vrhova, a E' skup uređenih parova elemenata od V.

Definicija 3. *Težinski graf* je uređeni par (G, ω) gdje je G graf, a $\omega\colon E \to \mathbb{R}$ funkcija koju nazivamo *težinska funkcija*.

Definicija 4. *Put* od vrha v_1 do vrha v_n u grafu G je niz vrhova $v_1, v_2, \dots v_n$, gdje su $(v_i, v_{i+1}) \in E$ za $i \in \{1, 2 \dots, n-1\}$.

Definicija 5. Neka je $v_0 e_0 v_1 \dots v_{k-1} e_{k-1} v_k$ put i $k \geq 2$. Ako od vrha v_k prema vrhu v_0 postoji brid e_k, tada se graf $C := v_0 e_0 v_1 \dots v_k e_k v_0$ zove *ciklus*.

Budući da se problem kojim se bavimo bazira na grafovima, potrebno ih je reprezentirati u obliku koji je pogodan za spremanje u računalu. Najčešće se tu radi o matrici susjedstva i njenim težinskim varijantama.

Definicija 6. *Težinsku matricu* W definiramo kao $W = [w_{i,j}]_{n\times n},\ i, j = 1, \dots, n$, gdje je

$$w_{i,j} = \begin{cases} 0, & \text{ako } i = j \\ \text{težina usmjerenog brida } (i,j), & \text{ako } i \neq j \text{ i } (i,j) \in E(G) \\ \infty, & \text{ako } i \neq j \text{ i } (i,j) \notin E(G) \end{cases}$$

Definicija 7. *Matricu najkraćih puteva* D definiramo kao $D = [d_{i,j}^{(k)}]_{n\times n}$, gdje je $d_{i,j}^{(k)}$ težina najkraćeg puta od vrha i do vrha j, pri čemu se svi posredni[1] vrhovi nalaze u

[1]Posredni vrh puta $W = v_0 e_0 v_1 \dots v_{k-1} e_{k-1} v_k$ je bilo koji vrh osim v_0 i v_k, tj. vrh koji se nalazi u skupu $\{1, 2, \dots, k\}$.

skupu $\{1, 2, \ldots, k\}$. *Težinu najkraćeg puta* $d_{i,j}^{(k)}$ definiramo kao

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j}, & \text{ako } k = 0 \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}), & \text{ako } k \geq 1 \end{cases}$$

budući da za najkraći put od vrha i do vrha j, takav da su svi posredni vrhovi iz skupa $\{1, 2, \ldots, k\}$, postoje dvije mogućnosti:

- k nije vrh na danom putu, pa je najkraći takav put duljine $d_{i,j}^{(k-1)}$
- k je vrh na danom putu, pa je najkraći takav put duljine $d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$.

Definicija 8. *Matricu prethodnika* $P^{(k)}$ definiramo kao $P^{(k)} = [p_{i,j}^{(k)}]_{n \times n}$, gdje za prethodnika vrha j na najkraćem putu od vrha i, pri čemu su svi posredni vrhovi iz skupa $\{1, 2, \ldots, k\}$, vrijedi

$$p_{i,j}^{(0)} = \begin{cases} \perp, & \text{ako } i = j \text{ ili } w_{i,j} = \infty \\ i, & \text{ako } i \neq j \text{ i } w_{i,j} < \infty \end{cases} \qquad p_{i,j}^{(k)} = \begin{cases} w_{i,j}^{(k-1)}, & \text{ako } d_{i,j}^{(k-1)} \leq p_{i,k}^{(k-1)} + p_{k,j}^{(k-1)} \\ w_{k,j}^{(k-1)}, & \text{ako } d_{i,j}^{(k-1)} > p_{i,k}^{(k-1)} + p_{k,j}^{(k-1)} \end{cases}.$$

U teoriji grafova, problem najkraćeg puta je problem pronalaženja puta između dva čvora u grafu tako da je zbroj težina njegovih bridova minimalan.

Definicija problema

Problem pronalaska najkraćih putova u grafu, sukladno definiciji 3, svodi se upravo na rješavanje problema traženja puta koji spaja promatrane vrhove s najmanjim mogućim zbrojem težina bridova na tom putu.

2 Rješenja problema

Postoje različite verzije rješavanja definiranog problema. Kako je prethodno naglašeno, ovaj rad fokusiran je na distribuirane varijante *Bellman-Fordovog* odnosno *Floyd–Warshallovog* algoritma.

Varijante će se razlikovati po tome što pretpostavljaju sinkronu ili asinkronu komunikaciju. Svaka od navedenih varijanti pretpostavlja postojanje jednog procesa u svakom čvoru grafa, a procesi mogu međusobno komunicirati samo preko bridova u grafu. Dakle, nijedan proces nema punu informaciju o grafu, već jedino zna tko su mu susjedi i kolike su težine bridova do tih susjeda.

Kao osnovu za rješavanje promatranih problema koriste se bazne klase pisane u Java programskom jeziku autora Vijay K. Garg*a* i detaljno opisane u knjizi [3]. Uz to, dodaju se vlastite klase kojima se implementiraju algoritmi, te pomoćne klase potrebne za pravilno izvršavanje algoritama.

2.1 Reprezentacija grafova

Za algoritamsku obradu nekog grafa potreban je sustavan i praktičan način za prikaz njegovih vrhova i bridova. U tu svrhu graf se opisuje u tekstualnoj datoteci gdje mu se zapravo pridružuje težinska matrica dana definicijom 6, uz malu korekciju kojom je u *.txt* datoteci vrijednost ∞ reprezentirana znakom x.

Pri zadavanju grafa važno je naglasiti kako on ne smije imati cikluse negativnih duljina. Tada problem najkraćeg puta nema smisla jer se kruženjem po negativnom ciklusu može postići po volji mala duljina puta. Dakle, u tom slučaju distribuirani algoritmi ne prepoznaju postojanje takvog ciklusa zbog čega ne prekidaju svoj rad.

2.1.1 Implementacija

Izgled datoteke koja sadrži podatke o grafu prilagođen je potrebama određenog algoritma. U nastavku slijedi detaljan opis klasa korištenih za čitanje svih potrebnih podataka o grafu.

Graph.java U ovoj datoteci se pomoću klase `Graph` čita tekstualna datoteka iz koje se konstruira pripadna težinska matrica. Ukoliko želimo prikazati graf s n vrhova, tada *.txt* datoteka treba sadržavati $n + 1$ redaka i to tako da se u prvom redu nalazi broj čvora (procesa) od kojeg se traže najkraći putevi prema ostalim čvorovima. U ostalim recima treba biti reprezentacija težinske matrice i to na način da se u i-tom retku i j-tom stupcu nalazi težina brida e_{ij} ukoliko taj brid postoji. Inače se na tom mjestu treba nalaziti slovo x.

Klasa `Graph` zatim metodom `readFile()` prolazi po tekstualnoj datoteci zadanoj pomoću konstruktora te prvo sprema izvor - *source*. Zatim prolazi po ostalih n redova i sprema težine u listu listi na način da brojeve spremi kao brojeve, a svaki znak x spremi kao `Integer.MAX_VALUE`.

GraphFloydWarshall.java Navedena klasa čita tekstualnu datoteku koja za graf od n vrhova sadrži n redaka tako da i-ti redak i j-ti stupac sadrži težinu brida e_{ij} ukoliko taj brid postoji, a inače slovo x. Stupci u datoteci odvojeni su razmacima.

Iz postojeće datoteke funkcija `readFile()` *otkriva* broj čvorova grafa i težine bridova te ih pamti. Uz to klasa sadrži i `findNeighbours()`, `findParents()` i `setUpWeights()` funkcije portebne u daljnem radu programa pri otkrivanju susjeda odnosno roditelja pojedinog čvora te inicijalizaciji početnih udaljenosti među vrhovima.

2.2 Bellman-Fordov algoritam

Bellman-Fordov algoritam služi za nalaženje najkraćeg puta od jednog izvora odnosno čvora do svih ostalih čvorova u težinskom grafu. Iako je sporiji od npr. Dijkstrinog algoritma, velika je prednost ovog algoritma što se može primijeniti i na grafove koji sadrže negativne težine. Međutim, ukoliko graf sadrži negativan ciklus do kojeg se može

doći iz izvora, algoritam ne daje rješenje jer tada najkraći put niti ne postoji. U sljedećem prikazu vidimo kako funkcionira ovaj algoritam.

Algoritam 1 *Bellman-Fordov* algoritam

```
1: za svaki vrh radi
2:    duljine[vrh] := ∞
3:    roditelji[vrh] := null
4: kraj za
5: duljine[izvor] = 0
6: za i = 0 do N radi
7:    za svaki brid (u, v) s težinom w radi
8:       ako duljine[u] + w < duljine[v] onda
9:          duljine[v] = duljine[u] + w
10:         roditelji[v] = u
11:      kraj ako
12:   kraj za
13: kraj za
14: vrati
```

Lokalne varijable

- N - ukupan broj čvorova
- $izvor$ - čvor od kojeg se traže najkraći putevi prema ostalim čvorovima
- $duljine[i]$ - duljina najkraćeg puta od izvora do vrha i
- $roditelji[i]$ - roditelj čvora i u najkraćem putu od izvora do čvora i

Distribuirani Bellman-Fordov algoritam funkcionira na analogan način, međutim pretpostavlja da je svaki čvor grafa zaseban proces, a procesi mogu komunicirati isključivo sa susjednim procesima. Tako se temeljem slanja poruka među procesima dijele informacije o duljinama puteva u danom trenutku i ažuriraju lokalne duljine puteva. Usto, niti jedan proces ne zna cijelu topolgiju (V, E).

2.2.1 Distribuirani Bellman-Fordov algoritam u sinkronoj mreži

U ovoj verziji algoritma se, uz maločas navedene uvjete, još pretpostavlja da svaki proces zna koliki je ukupan broj procesa odnosno čvorova $N = |V|$. Ovaj je uvjet potreban za zaustavljanje algoritma. Na sljedećem prikazu se vidi kako izgleda ova varijanta algoritma.

Algoritam 2 Sinkroni *Bellman-Fordov* algoritam za proces P_i

$duljina = \infty$
$roditelj = -1$

tip poruke: UPDATE
1: **ako** $i = i_0$ **onda**
2: $duljina = 0$
3: **kraj ako**
4: **za** $puls = 1$ **do** $N - 1$ **radi**
5: **pošalji** UPDATE(i, $duljina$) svim susjedima
6: **čekaj** UPDATE(j, $duljina_j$) **za svaki** $j \in Susjedi$
7: **za svaki** $j \in Susjedi$ **radi**
8: **ako** $duljina > duljina_j + težina_{i,j}$ **onda**
9: $duljina = duljina_j + težina_{i,j}$
10: $roditelj = j$
11: **kraj ako**
12: **kraj za**
13: **kraj za**

Lokalne varijable

- N - ukupan broj čvorova
- $duljina$ - duljina najkraćeg puta od izvora do vrha i
- $roditelj$ - roditelj čvora i u najkraćem putu od izvora do čvora i
- $težina_{i,j}$ - težina puta od procesa i do procesa j izvora do čvora i
- $Susjedi$, skup svih susjednih procesa procesu i

Implementacija

U implementaciji su korišteni programi s predavanja. Glavna je ideja bila prilagoditi klasu `Process` tako da po potrebi ažurira duljinu najkraćeg puta i pamti roditelja, tj. da obrađuje poruke ovisno o *tag*-u i sadržaju. Usto je bilo potrebno napraviti novu klasu `SynchBellmanFord` koja zapravo služi za pokretanje jednog procesa i inicijalizaciju podataka. Potom je bilo potrebno koristiti sinkronizator (u našem slučaju alfa sinkronizator) kako bismo mogli testirati valjanost algoritma. Za te potrebe je osmišljena i klasa `SynchBellmanFordTester`. Nadalje se navode značajne promjene napravljene na postojećem kodu.

Process.java

U ovu klasu dodajemo atribute:

- `length` pohranjuje duljinu najkraćeg puta od izvora do ovog procesa
- `parent` pohranjuje ID roditelja čvora koji sadrži ovaj proces
- `path` pohranjuje najkraći put od izvora do ovog procesa

U funkciji `receiveMsg()` dodaje se prikladna obrada poruka podijeljena na dva slučaja u ovisnosti o tipu poruke:

- *length* služi kao *tag* za tip poruke UPDATE iz navedenog algoritma. U slučaju da se preko procesa koji šalje poruku može konstruirati najkraći put od izvora do trenutnog procesa, ažurira se trenutna duljina
- *parent* služi kao *tag* za situaciju u kojoj tražimo točno koji je najkraći put po čvorovima od izvora do trenutnog procesa. Svaki proces šalje susjedima poruku *parent* s trenutnim `path`-om pa susjedi, ovisno o sadržaju i odnosu s tim procesom, ažuriraju svoj `path` ako je potrebno

Msg.java
U ovu klasu dodani su atributi `length` i `path` koji pohranjuju upravo duljinu i/ili točno najkraći put.

SynchBellmanFord.java
Ova klasa služi upravo za inicijalizaciju i pokretanje procesa. Sadrži atribute `parent` koji čuva ID procesa koji mu je roditelj i `s` koji je zapravo alfa sinkronizator. Prilikom pozivanja konstruktora, poziva se konstruktor nadklase `Process` te se postavlja sinkronizator. U funkciji `initiate()` se u pulsevima šalju poruke s *tag*om *length* (tj. UPDATE poruke), a kad prođu sve runde, susjedima se šalju poruke s *tag*om *parent* kako bi se dohvatio cijeli put. U funkciji `handleMsg()` se prihvaćaju poruke te se, ovisno o *tag*u i potrebi, odašilju nove poruke s prikladnim *tag*om.

SynchBellmanFordTester.java
Ova klasa služi za testiranje samog algoritma. U `main()` funkciji se obrađuje graf iz odabrane tekstualne datoteke i čitaju argumenti komandne linije. Potom se inicijaliziraju linker, sinkronizator i instanca klase `SynchBellmanFord` te se ta instanca *pokreće*.

Složenost

Vremenska **složenost** ovog algoritma je $n-1$ rundi. Broj poslanih poruke je pak $(n-1)l$, gdje je l broj bridova. Naime, u svakoj od $(n-1)$ rundi se pošalje točno l poruka.

2.2.2 Distribuirani Bellman-Fordov algoritam u asinkronoj mreži

U ovoj verziji algoritma se radi jednostavnosti također pretpostavlja da svaki proces zna koliki je ukupan broj procesa $N = |V|$. Taj uvjet nije bio potreban za zaustavljanje, već je olakšao samu implementaciju jednostavnijim prolaskom po susjedima. Međutim, budući da je riječ o asinkronom algoritmu, onda je glavna razlika da ne postoji

ograničenja na proces slanja poruka. U tom slučaju također nije potreban sinkronizator jer se poruke ne šalju u pulsevima, već se određena poruka šalje kao svojevrstan *odgovor* na neku prethodno primljenu poruku, ali se taj *odgovor* šalje svim susjedima. Na sljedećem prikazu je upravo ova varijanta algoritma.

Algoritam 3 Sinkroni *Bellman-Fordov* algoritam za proces P_i

$duljina = \infty$

tip poruke: UPDATE

1: **ako** $i = i_0$ **onda**
2: $duljina = 0$
3: **kraj ako**

kada UPDATE$(i_0, duljina_j)$ stigne od čvora j:

4: **ako** $duljina > duljina_j + težina_{i,j}$ **onda**
5: $duljina = duljina_j + težina_{i,j}$
6: **pošalji** UPDATE$(i_0, duljina)$ svakom susjedu
7: **kraj ako**

Lokalne varijable

- $duljina$ - duljina najkraćeg puta od izvora do vrha i
- $težina_{i,j}$ - težina puta od procesa i do procesa j

Implementacija

Implementacija ovog algoritma također koristi programe s predavanja. Sama ideja i prinicip slični su kao u sinkronoj verziji, međutim dvije su glavne razlike. Prva je dakako da nema sinkronizatora. Druga se glavna razlika odnosi na klasu `Process.java` u kojoj su atributi analogni kao i u sinkronoj verziji, ali ne i tipovi poruka. Naime, ovdje smo se odlučili cijelo vrijeme, kako ažuriramo duljinu puta, ažurirati i sam put. Tako se informacija o *parentu* i toku puta prosljeđivala istovremeno s informacijom o duljini puta.

U skladu s time, modificirana je i klasa `Msg.java` na način da također sadrži atribute `length` i `path`, ali ovaj put imamo i konstruktor kojim se omogućuje istovremeno pohranjivanje obje informacije (za razliku od sinkronog algoritma gdje smo slali **ili** jednu **ili** drugu informaciju). Stoga je i prilagođena datoteka `Linker.java` gdje smo u metodi `receiveMsg()` upravo omogućili opciju kreiranja poruke s dvije informacije.

Kao i u sinkronoj verziji, imamo još dvije datoteke: `AsynchBellmanFord.java` i `AsynchBellmanFordTester.java` koje implementiraju dvije klase s funkcionalnostima analognim svojim sinkronim parnjacima. Međutim, ovdje još valja napomenuti zašto smo *komplicirali* s istovremenim slanjem *parent* i *length* informacija kad smo već imali

'provjereni' sustav. Naime, nismo se baš dobro snašli sa samim zaustavljanjem algoritma, tj. nismo uspjeli detektirati kad je algoritam doista uspješno našao put koji je najkraći. Stoga se naš algoritam zapravo nikad ne zaustavlja, iako dosegne trenutak kada više ne šalje poruke. Ali zbog toga nismo uspjeli niti detektirati trenutak kada su poslane sve *length* poruke i kada možemo započeti s *parent* porukama. Upravo zato ih spajamo u jedan tip poruke, kako bismo traženu informaciju dobivali kontinuirano.

Složenost

Za ovaj algoritam se pokazalo da ima eksponencijalnu vremensku **složenost** te da je broj poruka koje se šalju također eksponencijalne složenosti.

2.3 Floyd–Warshallov algoritam

Floyd–Warshallov algoritam rješenje je problema najkraćih putova među svim parovima vrhova usmjerenog težinskog grafa s pozitivnim ili negativnim težinama bridova bez negativnih ciklusa. Dakle, za dani graf G rješenje problema je pronaći težinsku matricu udaljenosti D definiranu sa 7.

Ovaj algoritam se često još naziva i Floydov algoritam budući da ga je, u danas uvriježenom obliku, objavio Robert Floyd 1962. godine. Međutim on je suštinski ekvivalentan algoritmima koje su ranije objavili Bernard Roy 1959. godine i Stephen Warshall 1962. godine za traženje tranzitivnog zatvarača grafa[2].

2.3.1 Distribuirani Floyd–Warshallov algoritam u asinkronoj mreži

---

Lokalne varijable

- $LENGTH[1..n]$
  $LENGTH[j]$ - duljina najkraćeg poznatog puta od čvora i do čvora j
- $PARENT[1..n]$
  $PARENT[j]$ - prethodnik čvora j na najkraćem poznatom putu od čvora i
- $neighbours$ - svi čvorovi povezani bridom s čvorom i

Tipovi poruka

- IN_TREE($pivot$), NOT_IN_TREE($pivot$) - identifikacija djece pivotnog čvora
- PIV_ROW($pivot$, $PIV\_LEN[1..n]$, $PIV\_PAR[1..n]$) - prijenos $PARENT[\,]$ i $LENGTH[\,]$ nizova pivotnog elementa

---

[2]Problem se svodi na provjeru postojanja puta između neka dva vrha, ne obazirući se na težine.

Algoritam 4 Asinkroni distribuirani *Floyd–Warshallov* algoritam za proces P_i

1: **za** $pivot = 1$ **do** n **radi**

2: **za svaki** susjed $nbh \in Neighbours$ **radi**
3: **ako** $PARENT[pivot] = nbh$ **onda**
4: ***pošalji*** IN_TREE($pivot$) susjedu nbh
inače *pošalji* NOT_IN_TREE($pivot$) susjedu nbh
5: **kraj ako**
6: **kraj za svaki**

7: **čekaj** IN_TREE ili NOT_IN_TREE od svakog susjeda

8: **ako** $LEN[pivot] \neq \infty$ **onda**

9: **ako** $pivot \neq i$ **onda**
10: ***primi*** PIV_ROW($pivot, PIV\_LEN[1 \ldots n], PIV\_PAR[1 \ldots n]$) od $pivot$
11: **kraj ako**

12: **za svaki** susjed $nbh \in Neighbours$ **radi**
13: **ako** je IN_TREE primljena od nbh **onda**
14: **ako** $pivot = i$ **onda**
15: ***pošalji*** PIV_ROW($pivot, LEN[1 \ldots n], PARENT[1 \ldots n]$) susjedu nbh
inače *pošalji* PIV_ROW($pivot, PIV\_LEN[1 \ldots n], PIV\_PAR[1 \ldots n]$) susjedu nbh
16: **kraj ako**
17: **kraj ako**
18: **kraj za svaki**

19: **za** $t = 1$ **do** n **radi**
20: **ako** $LEN[pivot] + PIVOT\_LEN[t] < LEN[t]$ **onda**
21: $LEN[t] \leftarrow LEN[pivot] + PIV\_LEN[t]$
22: $PARENT[t] \leftarrow PIV\_PAR[t]$
23: **kraj ako**
24: **kraj za**

25: **kraj ako**

26: **kraj za**

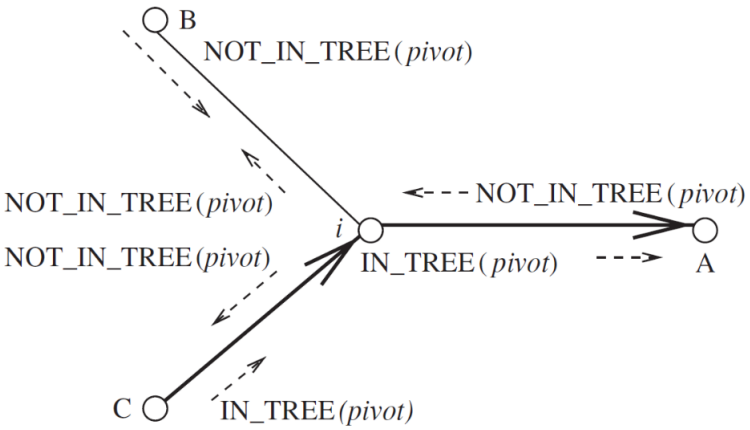
27: **vrati**

Kako je već prije navedeno, sve verzije pretpostavljaju postojanje jednog procesa u svakom čvoru grafa gdje se komunikacija ostvaruje razmjenom poruka isključivo preko bridova. Isto vrijedi i za distribuiranu asinkronu verziju *Floyd-Warshallovog* algoritma.

Glavna ideja algoritma je kroz niz iteracija ažurirati početne vrijednosti lokalnih varijabli $PARENT[\,]$ i $LENGTH[\,]$. Niz $LENGTH[\,]$ će se naposljetku sadržavati upravo najkraći mogući put od promatranog čvora tj. i-tog procesa do svakog preostalog čvora grafa, dok će se u varijabli $PARENT[\,]$ nalaziti čvorovi prethodnici $1 \ldots n$ čvorova na tom putu.

Realizacija ideje vidljiva je iz pseudokoda algoritma 4. Prikazan je algoritam za i-ti proces kojim je reprezentiran jedan čvor grafa. Svaki proces u n glavnih iteracija elementa *pivot provjerava* može li se do ostalih čvorova u grafu preko pivotnog elementa doći *kraćim* od dosad poznatog puta.

Provjera se odvija na način da *pivot* šalje svoje $LENGTH[\,]$ i $PARENT[\,]$ vektore duž njegovog minimalnog razapinjućeg stabla. Kako bi vrijednosti bile poslane djeci, pivotni element prvo mora identificirati svu svoju djecu. Identifikacija se odvija komunikacijom procesa preko IN_TREE odnosno NOT_IN_TREE poruka i prikazana je linijama $2 - 7$. Naime, u svakom procesu svaki pivotni element dobiva po jednu od navedenih poruka od svakog susjeda (linije $2 - 6$). Po primitku poruka od svih susjeda procesa (linija 7) *pivot* je svjestan svoje djece.



Slika 1: Prikaz toka IN_TREE i NOT_IN_TREE poruka u grafu

Nastavak algoritma, odnosno provjera je li put preko *pivota* od i-tog do ostalih čvorova kraći od dosad pronađenog najkraćeg puta, ima smisla jedino u slučaju da je put od i-tog do *pivot* čvora poznat. Ta provjera izvršava se osmom linijom algoritma.

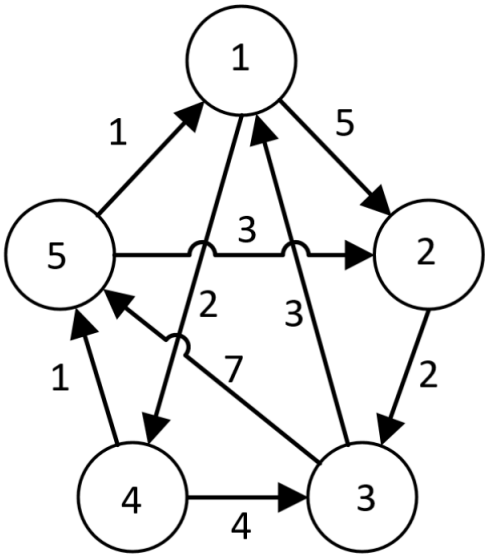
U slučaju potvrdnog odgovora algoritam se nastavlja i pokrenuti proces prima PIV_ROW poruku koja sadrži liste duljina i roditelja trenutnog pivota (linija 10). Poruku prosljeđuje svojoj, sada poznatoj, djeci koji su ujedno i susjedi i-tog procesa kojeg promatramo (linije $12 - 18$).

Konačno, ukoliko je preko trenutnog pivotnog elementa pronašao kraći od dosad poznatog puta do nekog čvora grafa, proces može ažurirati svoje liste roditelja i duljina budući da sad raspolaže sa svim potrebnim informacijama. To prikazuju linije $19 - 24$.

Nakon završetka svih n *pivot* iteracija u listi $LENGTH[\,]$ i-tog procesa nalazi se duljina najkraćeg **mogućeg** puta od čvora i do svakog čvora grafa, dok se u $PARENT[\,]$ listi nalazi prethodnik svakog čvora na najkraćem **mogućem** putu od čvora i. Iz ovakvih rezultata najkraći put iščitava se unatrag odnosno od krajnjeg do početnog čvora.

Primjer algoritma

Pronađimo najkraći put među svim parovima vrhova za težinski graf zadan slikom 2.



Slika 2: Težinski graf G

Do rješenja se dolazi kroz 5 iteracija *Floyd-Warshallovog* algoritma, informirajući susjede *pivota* iteracije o njegovoj najkraćoj dosad pronađenoj udaljenosti do svih vrhova.

Najprije se inicijalizira matrica udaljenosti $D^{(0)}$ među vrhovima i matrica prethodnika $P^{(0)}$. Inicijalno stanje grafa G obzirom na težine bridova zapravo je težinska matrica W iz definicije 6 dok je inicijalizacija matrice prethodnika dana definicijom 7.

$$D^{(0)} = W = \begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{bmatrix}, \quad P^{(0)} = \begin{bmatrix} \perp & 1 & \perp & 1 & \perp \\ \perp & \perp & 2 & \perp & \perp \\ 3 & \perp & \perp & \perp & 3 \\ \perp & \perp & 4 & \perp & 4 \\ 5 & 5 & \perp & \perp & \perp \end{bmatrix}$$

U sljedećih pet koraka ažuriraju se vrijednosti ovih matrica na način da se za svaka dva vrha provjerava postoji li put kraći od dosad poznatog najkraćeg puta preko *pivot* čvora. Za lakše praćenje koraka algoritma elementi matrica koji su ažurirani prethodnim prolaskom su uokvireni.

$$D^{(1)} = \begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \boxed{8} & 0 & \boxed{5} & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \boxed{3} & 0 \end{bmatrix}, \quad P^{(1)} = \begin{bmatrix} \bot & 1 & \bot & 1 & \bot \\ \bot & \bot & 2 & \bot & \bot \\ 3 & \boxed{1} & \bot & \boxed{1} & 3 \\ \bot & \bot & 4 & \bot & 4 \\ 5 & 5 & \bot & \boxed{1} & \bot \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 5 & \boxed{7} & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \boxed{5} & 3 & 0 \end{bmatrix}, \quad P^{(2)} = \begin{bmatrix} \bot & 1 & \boxed{2} & 1 & \bot \\ \bot & \bot & 2 & \bot & \bot \\ 3 & 1 & \bot & 1 & 3 \\ \bot & \bot & 4 & \bot & 4 \\ 5 & 5 & \boxed{2} & 1 & \bot \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 5 & 7 & 2 & \boxed{14} \\ \boxed{5} & 0 & 2 & \boxed{7} & \boxed{9} \\ 3 & 8 & 0 & 5 & 7 \\ \boxed{7} & \boxed{12} & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \quad P^{(3)} = \begin{bmatrix} \bot & 1 & 2 & 1 & \boxed{3} \\ \boxed{3} & \bot & 2 & \boxed{1} & \boxed{3} \\ 3 & 1 & \bot & 1 & 3 \\ \boxed{3} & \boxed{1} & 4 & \bot & 4 \\ 5 & 5 & 2 & 1 & \bot \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 5 & \boxed{6} & 2 & \boxed{3} \\ 5 & 0 & 2 & 7 & \boxed{8} \\ 3 & 8 & 0 & 5 & \boxed{6} \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \quad P^{(4)} = \begin{bmatrix} \bot & 1 & \boxed{4} & 1 & \boxed{4} \\ 3 & \bot & 2 & 1 & \boxed{4} \\ 3 & 1 & \bot & 1 & \boxed{4} \\ 3 & 1 & 4 & \bot & 4 \\ 5 & 5 & 2 & 1 & \bot \end{bmatrix}$$

$$D^{(5)} = \begin{bmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ \boxed{2} & \boxed{4} & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \quad P^{(5)} = \begin{bmatrix} \bot & 1 & 4 & 1 & 4 \\ 3 & \bot & 2 & 1 & 4 \\ 3 & 1 & \bot & 1 & 4 \\ \boxed{5} & \boxed{5} & 4 & \bot & 4 \\ 5 & 5 & 2 & 1 & \bot \end{bmatrix}$$

Duljina najkraćeg puta između svih parova čitama se iz rezultantne matrice $D^{(5)}$, dok se put identificira iz rezultantne matrice prethodnika $P^{(5)}$, unatrag od krajnjeg čvora j do početnog čvora i, prateći oznake čvorova u i-tom retku matrice.

Tako je primjerice najkraći put od čvora 2 do 1 duljine $d^{(5)}_{1,2} = 5$ i to preko čvora 3, budući da su $p^{(5)}_{2,1} = 3$ i $p^{(5)}_{3,1} = 2$. Analogno se čitaju putevi za svaka dva para preostalih čvorova.

Implementacija

Opisani algoritam 4 implementiran je pomoću osnovnih klasa za razvoj distribuiranih aplikacija pisanih u Javi i obrađenih na predavanju. Za potrebe *Floyd-Warshallovog* algoritma neke klase su prilagođene a neke su novouvedene. Slijedi njihov detaljniji opis.

Msg.java - *prijenos tri tipa poruka između procesa*
Za prijenos IN_TREE i NOT_IN_TREE poruka nije bila potrebna modifikacija postojeće klase. Međutim, za PIV_ROW tip poruka klasa je prilagođena s dvije nove članske varijable `pivotLength` i `pivotParent` koje služe za prijenos lokalnih nizova $PARENT[\,]$ i $LENGTH[\,]$ pojedinog procesa.

Linker.java - *međusobno povezivanje skupa procesa*
Dodana je nova varijabla tipa `Graph` koju `Linker` prima kroz konstruktor, a potrebna je pri inicijalizaciji listi roditelja i duljina. Sama inicijalizacija varijabli odvija se u klasi `Graph` na način opisan u odjeljku 2.1. Dakle, vidimo da je ovaj dodatak je ključan za pokretanje algoritma jer predstavlja sam *ulaz* algoritma.

Uz to, modificirana je metoda `receiveMsg()`, na temelju kojeg obrađuje dobivene poruke, u skladu s klasom `Msg` i ovisno o tagu poruke.

ProcessFloydWarshall.java - *izvršavanje algoritma*
U ovoj klasi implementiran je sam algoritam. Na prije opisan način proces šalje odnosno prima poruke određenog tipa te po potrebi ažurira varijable članice. Obrada primljenih poruka odvija se u *synchronized* metodi `handleMsg()` ovisno o *tagu* poruke.

FloydWarshallTester.java - *pokretanje algoritma*
Jednostavna testna klasa koja sadrži samo `main()` funkciju. Funkcija *čita* graf iz tekstualne datoteke odnosno pretvara ga u instancu klase `Graph` koju predaje klasi `Linker`.

Konačno, s tim podacima pokreće se program `ProcessFloydWarshall` koji predstavlja jedan čvor grafa. Svaki čvor je jedinstveno određen njegovim identifikatorom unesenim preko komandne linije.

Složenost

U svakoj od n iteracija vanjske petlje šalju se dvije IN_TREE ili NOT_IN_TREE poruke po svakom bridu, te najviše $n-1$ PIV_ROW poruka u slučaju da je *pivot* element promatrane iteracije susjed svim preostalim vrhovima grafa. Dakle, imamo da je ukupni broj poruka jednak $n \cdot (2l+n)$. Budući da je veličina PIV_ROW poruke n, dok su IN_TREE i NOT_IN_TREE poruke reda veličine $O(1)$ dobivamo da je vremenaska složenost izvršavanja algoritma po čvoru grafa jednaka $O(n^2)$, plus vrijeme potrebno za n *broadcast*[3] odnosno *convergecast*[4] faza.

[3]Čvor stabla prosljeđuje poruku svojoj djeci, sve do listova stabla.
[4]Roditelj prosljeđuje dobivenu poruku svojim roditeljima sve do korijena stabla.

3 Upute za pokretanje programa

U sklopu ovog projektnog zadatka razvijeni su programi koji pronalaze najkraći put prethodno opisani trima algoritmima te je za svaki od njih napisan testni program. Programi su razvijani u Java programskom jeziku u Eclipse razvojnom okruženju, a čitav programski kod nalazi se na *Github-u*.

Svaki od tih programa izvršava se kao skup od N procesa pri čemu svaki pojedini proces izvršava svoju kopiju istog programskog koda i razmjenjuje poruke s drugim procesima. Dakle, u pravilu je potrebno pokrenuti N testnih programa koje međusobno povezujemo osnovnom klasom `NameServer`.

Pokretanje preko Command Prompt i terminala

U nastavku su dane upute za pokretanje implementacije sinkronog *Bellman-Fordovog* algoritma preko `Command Prompt` u sustavu Microsoft Windows, odnosno terminala u `Linuxu` ili `Unixu`.

Ukoliko se koristi graf u datoteci `graph.txt` priloženoj među materijalima, tada nastavite na prvi korak. Ukoliko želite promijeniti taj graf, učinite to prema uputama u sekciji za grafove. Također, možete i stvoriti novu datoteku s novim grafom, ali tada trebate dodatno u datoteci `SynchBellmanFordTester` u desetoj liniji promijeniti ime (i po potrebi put) datoteke u kojoj se nalazi novi graf. Nadalje pretpostavljamo da graf ima N čvorova.

1. Otvorite $N + 1$ terminala.
2. U prvom terminalu idite u `/src/synch_bellman_ford` i izvršite naredbu: `javac *.java`. Potom naredbom `cd ..` idite u `src` direktorij i naredbom `java synch_bellman_ford.NameServer` pokrenite `NameServer`.
3. U ostalih N terminala za svaki $i = 0, \ldots, N-1$ i proizvoljnu komandnu riječ `komanda`, u pripadnom terminalu idite u `src` direktorij projekta i pokrenite naredbu: `java synch_bellman_ford.SynchBellmanFordTester komanda i N`

Tokom cijelog postupka će se za svaki proces ispisivati prikladne poruke o relevantnim koracima algoritma, npr. slanje ili primanje poruke određenog sadržaja, ažuriranje duljine ili sadržaja puta. Finalni rezultat za svaki od N procesa bit će zadnja poruka u pripadnom terminalu oblika:

```
-----------------------------------------------------
new path:  i_1/i_2/.../i_m weight:  w
-----------------------------------------------------.
```

Pokretanje asinkrone verzije ovog algoritma analogno je opisanoj sinkronoj varijanti, samo što svaku riječ `synch`/`Synch` zamjenimo s `asynch`/`Asynch`. Finalni je rezultat jednakog oblika kao u sinkronoj varijanti.

Pokretanje kroz Eclipse razvojno okruženje

Ulaz algoritma zadan je grafom zapisanim u tekstualnoj datoteci detaljno opisanoj u odjeljku 2.1. Ukoliko se inicijalno postavljen ulazni graf želi mijenjati potrebno je u testnom programu promijeniti putanju do željene datoteke. Nakon ove eventualne izmjene program je spreman za pokretanje.

Prvo se pokreće klasa `NameServer` na desni klik *Run as* → *Run Configurations* gdje se program jednostavno *run-a*. Slijedi pokretanje N testnih programa, primjerice neka je to `FloydWarshallTester`. Postupak je sličan, *Run as* → *Run Configurations*, uz dodatak unošenja argumenata programa u *tabu Arguments*. Ulazni argumenti programa su redom `test i N`, gdje `test` predstavlja ime po kojem `NameServer` evidentira izvođenje programa. To ime je proizvoljno, ali mora biti jednako za sve od N procesa. Argument i, $i = 0, \ldots, N-1$ predstavlja jedinstvenu oznaku procesa dok je N ukupni broj procesa.

Nakon unosa argumenata navedeni programi se pokreću. Za vrijeme izvršavanja programa `FloydWarshallTester`, u svakom od N prozora ispisuju se poruke koje taj proces razmjenjuje s ostalim procesima te trenutno stanje njegovih lokalnih $LENGTH$ i $PARENT$ varijabli dok će njihova vrijednost nakon N pivotnih iteracija predstavljati konačno rješenje.

Svi procesi oblikovani su kao beskonačne petlje odnosno neće se sami zaustaviti. Zbog toga je nakon pronalaska rješenja potrebno prekinuti sve pokrenute programe. U ovom okruženju to funkcionira na način: desni klik na konzolu → *Terminate/Disconnect All*.

4 Zaključak

U ovom smo radu proučavali distribuirane varijante algoritama za traženje najkraćih putova, preciznije *Bellman-Fordov* i *Floyd-Warshallov*. Vidjeli smo na koje načine oni funkcioniraju te koje su njihove karakteristike. Općenito znamo da distribuirani algoritmi imaju velike prednosti poput bolje tolerancije na eventualne greške u izvođenju, fleksibilnost i dobre performanse.

Ovdje primjerom pokazujemo da distribuirani algoritmi doista mogu imati i bolju složenost. Konkretno, sinkroni *Bellman-Fordov* algoritam ima linearnu vremensku složenost i broj poruka. Stoga možemo zaključiti da, iako ovakvi algoritmi imaju lošiju sigurnost i veće teškoće u dijagnosticiranju eventualnih pogreška u programu, kvaliteta rješavanja problema ih čini vrlo pouzdanima.

Literatura

[1] Mislav Karaula. "Pronalaženje najkraćih putova u grafu korištenjem hibridne CPU-GPU platforme". Mag. rad. Osijek, 2017. URL: https://repozitorij.mathos.hr/islandora/object/mathos%3A107/datastream/PDF/view.

[2] A.D. Kshemkalyani i Singal M. *Distributed Computing: Principles, Algorithms and Systems*. Cambridge University Press, 2011.

[3] Robert Manger. *Distribuirani procesi*. 2021.