Sinkroni mrežni algoritmi

Petrović, Katarina

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Katarina Petrović

SINKRONI MREžNI ALGORITMI

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, veljača, 2020

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Sinkrona mreža je računalna mreža u kojoj postoji gornja ograda za vrijeme potrebno za slanje i primanje poruke.

Sinkroni mrežni algoritmi su distribuirani algoritmi predviđeni za rad na sinkronoj mreži. Njihova korektnost ili efikasnost zasniva se na pretpostavci da je mreža na kojoj rade zaista sinkrona. Rad sinkronog algoritma na asinkronoj mreži moguć je jedino ako uključimo tzv. sinkronizator, softver koji simulira sinkronu mrežu nad asinkronom.

U radu je prvo opisan model sinkrone mreže i sinkronizator. Zatim su oblikovani i analizirani sinkroni algoritmi u općenitoj mreži: algoritam izbora vođe, problem najkraćeg puta i maksimalni nezavisni skup. Također, algoritmi su implementirani i testirani na stvarnoj mreži uz korištenje odgovarajućeg sinkronizatora te su prikazane pogreške u odsustvu sinkronizatora.

Poglavlje 1

Modeliranje: Model sinkrone mreže

Distribuirani algoritmi su algoritmi dizajnirani za izvođenje na hardveru koji se sastoji od mnogo međusobno povezanih procesora. Dijelovi distribuiranog algoritma rade istodobno i nezavisno, svaki sa samo ograničenom količinom informacija.
Algoritmi trebaju raditi ispravno pa čak ako pojedini procesori i komunikacijski kanali djeluju različitim brzinama ili ako neke komponente ne uspiju.
Distribuirani algoritmi nastaju u širokom rasponu aplikacija, uključujući telekomunikacije, distribuiranu obradu informacija, znanstveno računanje. Na primjer, današnji telefonski sustavi, sustavi banaka, globalni informacijski sustavi, sustavi upravljanja zrakoplovima i nuklearnim elektranama, svi oni ovise o distribuiranim algoritmima.
Jako je važno da se algoritmi izvode pravilno i efikasno. Međutim, zbog kompliciranih postavki u kojima se izvršavaju, dizajn takvih algoritama može biti izuzetno težak zadatak.

1.1 Sinkrona mreža

Pretpostavka da je mreža sinkrona olakšava oblikovanje distribuiranih algoritama.

- Algoritam za sinkronu mrežu (sinkroni algoritam) ima potpunu kontrolu nad vremenom slanja i primanja poruka.
- Akcije pojedinih procesa mogu u potpunosti biti usklađene.
- Algoritam možemo oblikovati tako da u svakom pulsu svaki proces: najprije primi sve poruke koje bi u tom pulsu morao primiti, zatim nešto računa, na kraju pošalje sve poruke koje bi u tom pulsu trebao poslati.

Model sinkrone mreže je najjednostavniji model (to jest onaj s najmanje nepouzdanosti), u kojem svi procesori komuniciraju i računaju u sinkronim pulsevima.

1.2 Sinkroni mrežni sustav

Sinkroni mrežni sustav sastoji se od skupa računalnih elemenata smještenih na čvorovima usmjerenog mrežnog grafa. Ti računalni elementi su zapravo procesori, što znači da su dio hardvera. Često je korisno razmišljati o njima kao o *procesima* logičkog softvera koji rade na (ali nisu identični) stvarnim hardverskim procesorima.

Da bismo formalno definirali sustav sinkrone mreže, započnimo s usmjerenim grafom $G = (V, E)$. Koristimo slovo n za označavanje $|V|$, broja čvorova u mrežnom grafu.
Za svaki čvor i od G koristimo notaciju $out\text{-}nbrs_i$ kako bi označili "odlazeće susjede" od i, tj. one čvorove kojima u grafu G postoje lukovi iz čvora i. Notacijom $in\text{-}nbrs_i$ označavamo "dolazne susjede" od i, tj. one čvorove od kojih postoje lukovi do čvora i u G.
Označavamo $distance(i, j)$ kao duljinu najkraćeg puta od i do j u G, ako takva postoji; inače je $distance(i, j) = \infty$. Definiramo $diam$, dijametar[1], kao maksimalnu $distance(i, j)$, po svim parovima (i, j). Također smo pretpostavili da imamo neki fiksni alfabet M i označili smo da je *null* oznaka odsutnosti poruke.

Povezan sa svakim čvorom $i \in V$, imamo *proces* koji se formalno sastoji od sljedećih komponenti:

- $states_i$, (ne nužno konačan) skup stanja
- $start_i$, neprazni podskup $states_i$, poznat kao podskup početnih stanja ili inicijalnih stanja
- $msgs_i$, funkcija generiranja poruka koja preslikava $states_i$ x $out\text{-}nbrs_i$ na elemente od $M \cup null$
- $trans_i$, funkcija stanja tranzicije koja preslikava $states_i$ i vektore (indeksirane s $in - nbrs_i$) elemenata od $M \cup null$ na $states_i$

Svaki proces ima skup stanja, među kojima se razlikuje podskup početnih stanja. Skup stanja ne mora biti konačan. Ova općenitost je važna, budući da nam dopušta modeliranje sustava koji uključuje neograničene podatkovne strukture poput brojača. Za svako stanje i odlaznog susjeda, funkcija generiranja poruka određuje poruku (ako postoji) koju proces i šalje naznačenom susjedu, počevši od danog stanja. Funkcija tranzicije stanja određuje, za svako stanje i kolekciju poruka svih dolazećih susjeda, neko novo stanje s obzirom na koje se proces i pokreće. Povezan sa svakim lukom (i, j) u G, postoji kanal, također poznat kao *veza*, što je samo lokacija koja u svakom trenutku može sadržavati najviše jednu poruku u M. Izvođenje cijelog sustava započinje sa svim procesima u proizvoljnom početnom stanju

[1] Dijametar je maksimalna duljina najkraćeg puta između bilo koja dva čvora mjerena brojem bridova.

i sa svim praznim kanalima. Zatim procesi, u lock-stepu[2], uzastopce izvode sljedeća dva koraka:

1. Primjenjuju funkciju generiranja poruka na trenutno stanje kako bi se generirale poruke koje se šalju svim odlazećim susjedima. Stavljaju ove poruke u odgovarajuće kanale.

2. Primjenjuju funkciju tranzicije stanja na trenutno stanje i nadolazeće poruke da bi dobili novo stanje. Uklanjaju sve poruke iz kanala.

Kombinacija ova dva koraka se naziva *puls* (runda). Općenito nemamo postavljena ograničenja na količinu izračuna koji proces obavlja da bi izračunao vrijednosti svojih funkcija generiranja poruka i tranzicije stanja. Predstavljeni model je deterministički u smislu da su funkcija generiranja poruka i funkcija tranzicije stanja zapravo jednoznačne funkcije. Prema tome, s obzirom na određenu kolekciju startnih stanja, računanje se razvija na jedinstven način.

Halting. Do sada nismo predvidjeli zaustavljanje procesa. Lako je, međutim, istaknuti neke od procesnih stanja kao stanja zaustavljanja i specificirati da se iz tih stanja ne može dogoditi daljnja aktivnost. Odnosno, nikakve poruke nisu generirane i jedini prijelaz stanja je self-petlja. Ova stanja zaustavljanja ne igraju istu ulogu u tim sustavima kao u tradicionalnom konačnom automatu. Tamo uglavnom rade kao stanja prihvaćanja, koja odlučuju koji stringovi su u jeziku koji stroj izračunava. Ovdje nam služe samo za zaustavljanje procesa, a što proces izračunava mora se odrediti prema nekoj drugoj konvenciji. Pojam prihvaćanja stanja se obično ne koristi za distribuirane algoritme.

Promjenjivo vrijeme početka. Napomenimo da postoji mogućnost sinkronog sustava u kojem se procesi mogu započeti izvršavati u različitim pulsevima.

Neusmjereni grafovi. Razmotrimo slučaj gdje je temeljni mrežni graf neusmjeren. Modeliramo ovu situaciju unutar modela koji smo već definirali za usmjerene grafove jednostavno razmatranjem usmjerenog mrežnog graf s dvosmjernim komunikacijskim kanalima između svih parova susjeda. U ovom slučaju ćemo koristiti oznaku $nbrs_i$ za označavanje susjeda u grafu.

[2]Izvršavanje u lock-stepu znači da se istodobno, paralelno izvodi isti skup operacija.

1.3 Neuspjesi

Razmotrit ćemo različite vrste kvarova za sinkrone sustave, uključujući *neuspjehe procesa* i *pogreške veza (kanala)*.

Proces može pokazati neuspjeh zaustavljanja jednostavnim zaustavljanjem negdje usred izvršenja. U pogledu modela, proces može pasti prije ili nakon izvođenja neke instance prvog ili drugog koraka gore. Pored toga, dopuštamo neuspjeh negdje usred izvođenja koraka 1. To znači da je proces mogao uspjeti u stavljanju samo podskupa poruka koje bi trebao proizvesti u kanalima poruka. Pretpostavit ćemo da to može biti bilo koji podskup.

Proces također može pokazati Bizantinsku pogrešku, što znači da može generirati svoje sljedeće poruke i sljedeće stanje na neki proizvoljan način, bez slijeđenja pravila koja su određena funkcijama generiranja poruka i tranzicije stanja.

Veza može pasti gubitkom poruka. U pogledu modela, proces bi mogao pokušati smjestiti poruku u kanal tijekom koraka 1, ali neispravna veza može ne snimiti poruku.

1.4 Ulazi i izlazi

Još uvijek nismo osigurali mogućnost za modeliranje ulaza i izlaza. Koristimo jednostavnu konvenciju kodiranja ulaza i izlaza sa stanjima. Posebno, ulazi se stavljaju u određene ulazne varijable u početnim stanjima. Činjenica da proces može imati više početnih stanja jako je važna pa stoga možemo imati različite moguće ulaze. Zapravo, obično pretpostavljamo da je jedini izvor mnogostrukosti početnih stanja mogućnost različitog unosa vrijednosti u ulaznim varijablama. Izlazi se pojavljuju u određenim izlaznim varijablama. Svaki od tih zapisa bilježi rezultat samo prve operacije pisanja koja se izvodi (tj., to je write-once varijabla). Međutim, izlazne varijable se mogu očitati bilo koji broj puta.

1.5 Izvršenje(egzekucije)

Da bismo objasnili ponašanje sinkronog mrežnog sustava, trebamo formalni pojam *izvršenja* sustava.

Dodjela stanja sustava definira se kao dodjela stanja svakom procesu u sustavu. Također, *dodjela poruka* je dodjela (moguće null) poruke svakom kanalu. Definiramo izvršenje sustava kao beskonačan niz

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, ...,$$

(stanje, poslana poruka, primljena poruka) pri čemu je svaki C_r dodjela stanja, a svaki M_r i N_r dodjela poruke. C_r predstavlja stanje sustava nakon r rundi, dok M_r i N_r predstavljaju poruke koje se šalju i primaju u rundi r (mogu biti različite jer kanali mogu izgubiti poruke).

C_r često nazivamo dodjela stanja koja se događa u vremenu r, to jest, vrijeme r se odnosi na točku neposredno nakon r pulseva.

Ako su α i α' dvije egzekucije sustava, kažemo da je α nerazlučiv iz α' s obzirom na proces i, u oznaci $\alpha \sim_i \alpha'$, ako i ima isti slijed stanja, isti slijed odlaznih poruka i isti slijed dolaznih poruka u α i α'. Također kažemo da su α i α' nerazlučivi za proces i kroz r pulseva ako i ima isti slijed stanja, isti slijed odlaznih poruka i isti slijed dolaznih poruka do kraja pulsa r, u α i α'. Te definicije proširujemo i na situacije u kojima se izvršavanja uspoređuju, a to su izvršavanja na dva različita sinkrona sustava.

1.6 Metode dokazivanja

Najvažnija metoda dokazivanja za rasuđivanje o sinkronim sustavima uključuje dokazivanje *invarijantnih tvrdnji*. Invarijantna tvrdnja je svojstvo stanja sustava (posebno stanja svih procesa) koje je istinito nakon svake runde. Dopuštamo da se broj završenih pulseva spomene u tvrdnjama, kako bismo mogli iznositi tvrdnje o stanju nakon svakog pulsa r. Invarijantne tvrdnje za sinkrone sustave su općenito dokazane indukcijom po r, broju završenih pulseva, počevši s $r = 0$.

Druga važna metoda je *simulacija*. Grubo rečeno, cilj je pokazati da jedan sinkroni algoritam A "implementira" drugi sinkron algoritam B, u smislu stvaranja istog ulaza / izlaza. Korespondencija između A i B izražena je tvrdnjom koja se odnosi na stanja od A i B, kada se dva algoritma pokreću na istim ulazima i s istim obrascem neuspjeha za isti broj pulseva. Takva tvrdnja poznata je pod nazivom simulacijska relacija. Simulacijske relacije se uglavnom dokazuju indukcijom po broju završenih rundi.

1.7 Mjere složenosti

Dvije mjere složenosti obično se uzimaju u obzir za sinkrone distribuirane algoritme: vremenska složenost i složenost komunikacije.

Vremenska složenost sinkronog sustava mjeri se brojem rundi dok se ne proizvedu svi potrebni izlazi ili dok se svi procesi ne zaustave. Ako sustav dopušta promjenjivo početno vrijeme, vremenska je složenost mjerena od prve runde u kojoj se događa buđenje, u bilo kojem procesu.

Složenost komunikacije obično se mjeri ukupnim brojem non-null poruka koje se šalju. Povremeno ćemo uzeti u obzir i broj bitova u porukama.

Mjera vremena je važnija mjera u praksi, i to ne samo za sinkrone distribuirane algoritme, ali za sve distribuirane algoritme. Komunikacijska složenost je uglavnom značajna ako uzrokuje dovoljno zagušenja da bi usporila obradu. To sugerira da bismo je možda željeli ignorirati i samo razmatrati vremensku složenost. Međutim, utjecaj opterećenja

komunikacije na vremensku složenost nije samo funkcija pojedinačnog distribuiranog algoritma. U tipičnoj mreži mnogi distribuirani algoritmi istodobno pokreću i dijele istu mrežnu propusnost. Učitavanje poruke dodane u kanal u bilo kojim pojedinačnom algoritmu dodaje se ukupnom učitavanju poruka na toj vezi i time pridonosi do zagušenja koju vide svi algoritmi. Budući da je teško kvantificirati utjecaj koji poruke bilo kojeg algoritma imaju na vrijeme performansi ostalih algoritama, rješavamo jednostavnu analizu (i pokušaj minimiziranja) broja poruka generiranih pomoću pojedinih algoritama.

Poglavlje 2

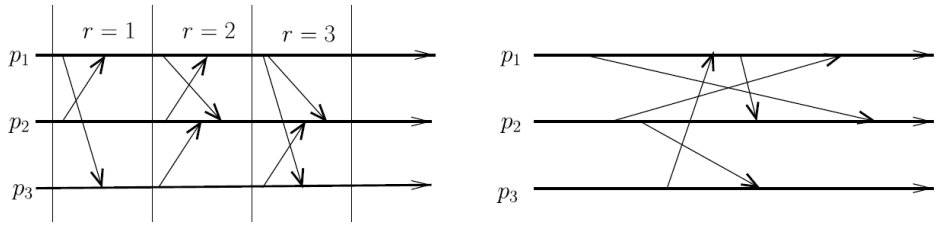
Sinkronizacija

Postavlja se pitanje: može li sinkroni algoritam ispravno raditi ako je mreža asinkrona? Odgovor glasi: može, ali pod nekim uvjetima. Da bi se sinkroni algoritam ispravno izvršavao na asinkronoj mreži, potreban je sinkronizator. Riječ je o pomoćnom algoritmu koji simulira sinkronu mrežu nad asinkronom.

2.1 Općenito o sinkronizaciji i sinkronizatorima

Najprije ćemo opisati parametre asinkrone mreže na kojoj radimo.

- Pretpostavljamo da se mreža sastoji od N procesa (čvorova).
- Ti procesi povezani su s E_b dvosmjernih komunikacijskih kanala.
- Mreža ima dijametar *diam*.
- Pretpostavljamo da su svi komunikacijski kanali pouzdani, nema gubitaka poruka.
- Uzimamo da računala rade bez pogreške, nema nepredviđenih prekida rada procesa.



Slika 2.1: Sinkrono (lijevo) vs. asinkrono (desno) izvršavanje [3]

Zatim opisujemo način izvršavanja sinkronog aplikacijskog algoritma na našoj asinkronoj mreži.

- Algoritam radi pod kontrolom sinkronizatora.
- Svaki proces koristi brojač koji broji korake algoritma i koji se zove puls.
- Puls se u svim procesima inicijalizira na 0 te se dalje povećava uz pomoć sinkronizatora na sinkronizirani način, dakle "istovremeno" u svim procesima.
- Unutar svakog pulsa, svaki proces radi sljedeće: najprije prima poruke koje su mu drugi procesi poslali u prethodnom pulsu, zatim obavlja interne radnje zasnovane na primljenim porukama. Dalje šalje poruke drugim procesima i na kraju čeka da sinkronizator dozvoli prelazak na idući puls.

Kao što vidimo, sinkronizator daje radni takt sinkronom aplikacijskom algoritmu, dakle on daje procesima dozvolu da zajedno krenu s izvršavanjem idućeg pulsa. Sam sinkronizator može se opisati sučeljem Synchronizer definirano u Synchronizer.java.

```
public interface Synchronizer extends MsgHandler {
    public void initialize(MsgHandler initProg);
    public void sendMessage(int destId, String tag, int msg);
    public void nextPulse(); // block for the next pulse
}
```

Aplikacija koji radi pod kontrolom sinkronizatora:

- najprije mora pozvati sinkronizatorovu metodu initialize()
- svaku svoju poruku mora slati pomoću sinkronizatorove metode sendMessage()
- da bi sačekala idući puls, mora pozvati sinkronizatorovu metodu nextPulse().

U nastavku raspravljamo o složenosti sinkronizacije, dakle o dodatnoj cijeni koju plaćamo kad sinkroni algoritam izvodimo na asinkronoj mreži.

- Postoje dva aspekta složenosti sinkronizacije:
    - Komunikacijska složenost: ukupan broj dodatnih poruka koje proizvodi sinkronizator u svrhu sinkronizacije.

  - Vremenska složenost: najveća duljina niza uzastopnih dodatnih poruka koje putuju po jednom kanalu u jednom smjeru.

- Neka je M_{init} broj poruka, a T_{init} vrijeme (duljina niza poruka) potrebno za inicijalizaciju sinkronizatora.

- Neka je M_{pulse} odnosno T_{pulse} broj poruka odnosno vrijeme potrebno sinkronizatoru za simuliranje jednog pulsa.

- Ako sinkroni aplikacijski algoritam zahtijeva ukupno M_{synch} aplikacijskih poruka i T_{synch} pulseva, tada stvarna složenost sinkrone aplikacije koji se izvodi pomoću sinkronizatora na asinkronoj mreži iznosi:

  $M_{asynch} = M_{init} + M_{synch} + M_{pulse}T_{synch}$
  $T_{asynch} = T_{init} + T_{pulse}T_{synch}$.

Poznati sinkronizatori su jednostavni sinkronizator te sinkronizatori α, β i γ. Imena je odabrao njihov izumitelj, Baruch Awerbuch. Glavna razlika između njih je mehanizam koji se koristi za određivanje dostavljanja poruka runde r. Kasnije u radu će biti implementirani algoritmi s jednostavnim sinkronizatorom i sinkronizatorom α,[1] stoga ih opišimo.

Jednostavni sinkronizator

U ovom odjeljku opisat ćemo vrlo jednostavni sinkronizator, koji može poslužiti svrsi, premda zahtijeva određene kompromise u pogledu načina izvođenja sinkrone aplikacije. Pretpostavljamo FIFO kanale. Uvodimo pravilo da svaki proces mora unutar svakog pulsa svakom svom susjedu poslati točno jednu poruku. Da bi realizirali ovakvo pravilo, u samoj sinkronoj aplikaciji moramo napraviti neke sitne izmjene.

- Ako u određenom pulsu aplikacija nije zahtijevala od P_i da šalje poruku P_j -u, tada P_i ipak mora P_j -u poslati null poruku.

- Ako je u određenom pulsu sinkrona aplikacija zahtijevala od P_i da šalje više poruka P_j -u, tada P_i mora te poruke zapakirati u jednu poruku i tako ih poslati P_j -u.

Sinkronizator dozvoljava procesu P_i idući puls nakon što je:

- P_i primio točno po jednu poruku od svakog susjednog procesa,

- P_i poslao točno po jednu poruku svakom susjednom procesu.

[1] Implementacija klasa SimpleSynch i AlphaSynch mogu se naći u [4].

Pokažimo složenost jednostavnog sinkronizatora.

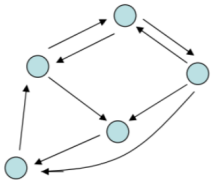
- Čim jedan proces starta puls 1, u roku od diam vremenskih jedinica svi ostali procesi će također startati puls 1. Zato vrijedi:
  $M_{init} = 0$, $T_{init} = diam$.
- Budući da svaki puls zahtijeva po jednu poruku duž svakog kanala u oba smjera, složenost simulacije jednog pulsa iznosi:
  $M_{pulse} = 2E_b$, $T_{pulse} = 1$.

Sinkronizator α

U ovom odjeljku obradit ćemo još jedan sinkronizator koji se zove sinkronizator α. Sinkronizator α sličan je jednostavnom sinkronizatoru. Zasniva se na pojmu sigurnosti procesa (safety). Proces P_i je siguran za puls k ako on zna da su sve poruke koje je on poslao u pulsu k bile primljene.

Slijedi opis sinkronizatora α.

- Sinkronizator α generira idući puls u procesu P_i ako je P_i siguran i ako su svi njegovi susjedi također sigurni.
- Da bi implementirali sinkronizator α, dovoljno je da svaki proces prati je li je siguran te da obavještava svoje susjede kad je siguran.
- Proces će znati da je siguran ako za svaku poslanu poruku prima potvrdu (ack) od primatelja.
- Za sinkronizator α vrijede slične ocjene kao za jednostavni sinkronizator:
  $M_{init} = 0$, $T_{init} = diam$,
  $M_{pulse} = O(E_b)$, $T_{pulse} = O(1)$.



Poglavlje 3

Algoritmi u općenitoj mreži

U ovom ćemo poglavlju razmatrati zbirku problema u većoj klasi sinkronih mreža. Konkretno, predstavljamo algoritme za izbor vođe, pronalaženje najkraćih putova i pronalaženje maksimalnog nezavisnog skupa (MIS) u mrežama na temelju proizvoljnih grafova. Problem izbora vođa nastaje kada mora biti izabran proces koji će preuzeti ulogu "vođe" u mrežnom računanju. Problem pronalaženja najkraćih putova potreban je za izgradnju struktura pogodnih za podršku učinkovite komunikacije. Problem pronalaska maksimalnog nezavisnog skupa proizlazi iz problema mrežne alokacije resursa (ti algoritmi se mogu analizirati i u kontekstu asinkronih mreža).

Promatramo jako povezani[1] mrežni graf $G = (V, E)$ s n čvorova. Kao i obično za sinkrone sustave, pretpostavljamo da procesi komuniciraju samo preko lukova grafa. Da bismo imenovali čvorove, dodijelili smo im indekse $1, ..., n$. Procesi ne znaju njihove indekse, niti one od svojih susjeda, već susjede označavaju lokalnim imenima. Pretpostavimo da ako proces i ima isti proces j i za dolaznog i za odlazećeg susjeda, onda znamo da su ta dva procesa ista.

U narednim potpoglavljima, implementirat ćemo algoritme koji koriste softversku infrastrukturu definiranu u [4]. Za pokretanje njihovih programa SynchFloodMax.java, SynchBellmanFord.java i SynchLubyMIS.java, uz sinkronizatorove datoteke i pripadne testne datoteke, potrebne su nam i sljedeće datoteke: Connector.java, ListenerThread.java, Msg.java, NameServer.java, Process.java, Util.java, IntLinkedList.java, MsgHandler.java, NameTable.java, Symbols.java, Linker.java, Name.java, PortAddr.java i Topology.java.

[1] Usmjeren graf je jako povezan ako za svaka njegova dva čvora postoji put od jednog do drugog čvora, ali i obrnut put od drugog do prvog čvora.

3.1 Algoritam izbora vođe u općenitoj mreži

Problem izbora vođe ima nekih sličnosti s problemom međusobnog isključivanja.
Naime, u oba slučaja je riječ o dogovoru između procesa te o isticanju jednog procesa u odnosu na druge. Pretpostavljamo da procesi imaju jedinstvene identifikatore (UID), izabrane iz nekog potpuno uređenog skupa identifikatora. Preciznije, želimo pokrenuti N procesa $P_0, P_1, ..., P_{N-1}$ te uspostaviti komunikacijske veze među njima. Procesi pritom ne moraju znati ni adrese ni imena računala ni portove, već trebaju slati i primati poruke služeći se isključivo identifikatorima procesa $0, 1, ..., N-1$.
UID svakog procesa se razlikuje od svakog drugog u mreži.

Postoji uvjet da bi na kraju točno jedan proces trebao sebe proglasiti vođom, mijenjajući posebnu *status* komponentu svoga stanja u *leader* vrijednost. Postoji nekoliko verzija problema:

1. Zahtjeva se da svi neizabrani procesi kao output daju činjenicu da oni nisu vođe, mijenjajući njihovu status komponentu u *non-leader*.

2. Broj n čvorova i dijametar diam, mogu biti poznati ili nepoznati procesima.

Jednostavni Flooding algoritam

Dajemo jednostavan algoritam koji uzrokuje identifikaciju leadera i non-leadera. Algoritam zahtijeva da procesi znaju diam. Algoritam propagira (eng. *floods*[2]) maksimalni UID u cijeloj mreži, pa ga nazivamo FloodMax algoritam.

Algoritam za izbor vođe opisivat ćemo klasom koja implementira sučelje Election. Dakle klasa koja opisuje algoritam za izbor vođe mora implementirati dvije metode koje su predviđene sučeljem. Bilo koji proces može pokrenuti izbor vođe tako da pozove metodu startElection(). Također, proces može saznati identifikator vođe tako da pozove metodu getLeader(). Ako postupak izbora vođe još nije završen, tada metoda getLeader() blokira proces i vraća vrijednost tek kad vođa bude poznat.

```
public interface Election extends MsgHandler {
    void startElection();
    int getLeader();//blocks till the leader is known
}
```

[2]Flooding je jednostavna tehnika usmjeravanja u računalnim mrežama gdje izvor ili čvor šalje pakete kroz svaku odlaznu vezu.

FloodMax algoritam (neformalno):
Svaki proces zna svoj identifikator, no ne zna koliko ukupno ima procesa te koji od njih ima najveći identifikator. Također, procesi održavaju evidenciju maksimalnog identifikatora kojega su dosad vidjeli (u početku je to vlastiti). Algoritam osigurava da proces s maksimalnim identifikatorom bude izabran za vođu. U svakoj rundi, svaki proces šalje taj maksimum svim svojim odlaznim susjedima. Ako je nakon diam rundi maksimalna vrijednost vlastiti UID procesa, proces bira sebe kao leder-a; inače je non-leader.

FloodMax algoritam(formalno):

Alfabet poruka je skup UID-ova.

$states_i$ se sastoje od komponenata:
UID, inicijalno UID procesa i
max-uid, UID, inicijalno UID procesa i
$status \in \{unknown, leader, non\text{-}leader\}$, inicijalno unknown
rounds, integer, inicijalno 0

$msgs_i$:
if $rounds < diam$ then
šalji max-uid svim $j \in out\text{-}nbrs$

$trans_i$:
rounds := $rounds + 1$
neka je U skup UID-eva koji dolaze od procesa u in-nbrs
$max\text{-}uid := max(\{max\text{-}uid\} \cup U)$
if rounds = diam then
if $max\text{-}uid = UID$ then $status := leader$
else $status := non\text{-}leader$

Lako je vidjeti da FloodMax odabire proces s maksimalnim UID-om. Definirajte i_{max} kao indeks procesa s maksimalnim UID-om. Slijedi teorem.

Teorem 1. *U FloodMax algoritmu, u diam rundi, proces i_{max} kao output daje leader, a svi ostali procesi kao output daju non-leader.*

Analiza složenosti. Lako je vidjeti da je vrijeme do izbora vođe (i kad svi drugi procesi znaju da oni nisu vođa) zapravo *diam* rundi. Broj poruka je $diam \cdot |E|$, gdje je $|E|$ broj lukova u grafu, jer se poruka šalje na svakom luku u svakoj od prvih diam rundi.

Gornja granica na dijametar. Algoritam također ispravno radi ako svi procesi znaju gornju granicu d na dijametru, a ne sami dijametar. Mjere složenosti tada se povećavaju tako da ovise o d, a ne o $diam$.

Opis klase SynchFloodMax uz pomoć sinkronizatora

Upotrebom sinkronizatora, sinkroni algoritam izbora vođe u prstenu, može se pokrenuti u asinkronom prstenu, ali to nije zanimljivo jer ovi algoritmi već rade na asinkronoj mreži, bez troškova koje uvode sinkronizatori.

U asinkronoj mreži zasnovanoj na proizvoljnom usmjerenom grafu s poznatim dijametrom, $diam$, sinkronizator se može koristiti za pokretanje SynchFloodMax sinkronog algoritma izbora vođe. Korištenjem sinkronizatora $Alpha$, rezultirajući algoritam šalje $O(|E| \cdot diam)$ poruka i za simulaciju mu je potrebno $O(diam \cdot k)$ vremena da simulira potrebnih diam sinkronih rundi, gdje je k gornja granica na vrijeme za dostavljanje najstarije poruke u kanalu.

Sinkronizator se također može koristiti za pokretanje OptFloodMax[3], sinkronog algoritma za izbor vođe, koji je sličan FloodMaxa algoritmu, osim što čvorovi šalju poruke samo kad imaju nove podatke za slanje. Ako se koristi sinkronizator Alpha, prednost optimizacije se gubi budući da sam sinkronizator šalje poruke svim kanalima u svim rundama.

Slijedi detaljan opis klase SynchFloodMax.

- Klasa SynchFloodMax proširuje klasu Process i implementira sučelje Election. Dakle, RingLeader mora implementirati metode *startElection*(), *getLeader*() i *handleMsg*().
- U ovoj implementaciji dodatno zahtjevamo da procesi znaju n, broj čvorova u grafu.
- Varijable su sljedeće.
  - *UID* označava identifikator lokalnog procesa.
  - *leaderId* poprima redni broj vođe, kad vođa postane poznat. U početku je -1.
  - *diam* označava dijametar mreže koji treba biti poznat kako bi algoritam ispravno radio.
  - *status* varijabla poprima vrijednost iz skupa {unknown, leader, non-leader}. Početna vrijednost je unknown.
  - *s* je sinkronizator, dakle objekt sa sučeljem Synchronizer.
  - *UIDs* je lista u koju dodamo sve identifikatore susjeda i lokalni identifikator.

[3]Opis ovog algoritma i algoritma izbora vođe u prstenu može se pronaći u [1].

  - *maxUID* je pomoćna varijabla koja poprima vrijednost najvećeg identifikatora iz liste UIDs
  - *stringNeighbours* je varijabla tipa Stringa. Redni broj susjeda, odvojenih zarezom.
  - *neighbours* je objekt tipa IntLinkedList u koji spremimo susjede lokalnog procesa.

- Metoda *startElection*() upravlja cijelim tijekom algoritma.
  - Definira se objekt *neighbours*.
  - Inicijalizira se sinkronizator pozivom njegove metode *initialize*().
  - Poziva se metoda *invite*() koja inicira postupak izbora vođe slanjem odgovarajuće poruke tipa invite.
  - Vrti se petlja čiji koraci odgovaraju pulsevima.
  - Zbog blokirajućeg poziva *nextPulse*(), takt izvršavanja petlje određuje sinkronizator.
  - Pri nastavku izvršavanja petlje, lokalni identifikator se doda u listu UIDs, izabere se maksimalan među svim identifikatorima pristiglim od susjeda i spremi se u varijablu maxUID.
  - Poziva se metoda *check*(), u kojoj ako nije dosegnuto diam pulseva, šalje se poruka *invite* s *maxUID* svim susjedima lokalnog procesa. Inače, ako je diam jednak broju pulseva i maxUID jednak lokalnom identifikatoru, pronašli smo vođu i status varijabla postaje leader ili u suprotnom definiramo status varijablu kao non-leader.
  - Isprazni se lista UIDs i ide se u novi puls.
  - I invite() i check() za slanje poruka koriste sinkronizatorovu metodu sendMessage().

- Metoda handleMsg() prima sve poruke susjeda u kojima se nalazi identifikator i sprema ih u listu UIDs.

- Metoda getLeader() čeka dok vođa ne bude izabran, a onda vraća sadržaj varijable leaderId.

SynchFloodMax.java

```
public class SynchFloodMax extends Process {
   int UID;
   int maxUID = UID;
   int leaderId = -1;
   int diam;
   String status = "unknown";
   Synchronizer s;
   List UIDs = new ArrayList();
   String stringNeighbours;
   IntLinkedList neighbours = new IntLinkedList();

   public SynchFloodMax(Linker initComm, int UID, String stringNeighbours,
       int diam, Synchronizer initS) {
      super(initComm);
      this.UID = UID;
      this.stringNeighbours = stringNeighbours;
      this.diam = diam;
      s = initS;
    }

   public void startElection() {
      StringTokenizer st = new StringTokenizer(stringNeighbours, ",");
      while (st.hasMoreTokens()) {
         int n = Integer.parseInt(st.nextToken());
         neighbours.add(n);
      }
      s.initialize(this);
      invite();
      for (int pulse = 1; pulse <= diam; pulse++) {
         s.nextPulse();
         UIDs.add(UID);
         maxUID = Collections.max(UIDs);
         check(pulse);
         UIDs.clear();
      }
   }
```

```
  public synchronized void invite() {
    for (int i = 0; i < N; i++)
      if (isNeighbor(i) && neighbours.contains(i))
        s.sendMessage(i, "invite", UID);
  }

  public synchronized int getLeader(){
    while (status == "unknown")
      myWait();
    return leaderId;
  }

   public void handleMsg(Msg m, int src, String tag) {
    int j = m.getMessageInt();
    UIDs.add(j);
  }

  public void check(int pulse){
    if (pulse < diam){
      for (int i = 0; i < N; i++)
        if (isNeighbor(i) && neighbours.contains(i))
          s.sendMessage(i, "invite", maxUID);
    } else if (pulse == diam){ // I won!
      if (maxUID == UID){
        status = "leader";
        leaderId = myId;
      }
      else status = "nonLeader";
    }
    System.out.println("Status: " + status);
  }
}
```

Testiranje cijelog opisanog postupka, dakle izbor vođe na asinkronoj mreži sinkronim algoritmom uz pomoć sinkronizatora, omogućeno je programom SynchronizerFloodMaxTester.java. Slijede objašnjenja.
Prvi argument naredbenog retka kojim se pokreće program nakon SynchronizerFloodMaxTester je bazno ime pod kojim NameServer evidentira izvođenje programa. Zatim kao argumenti slijede redni broj dotičnog procesa i ukupan broj procesa. Dalje imamo identifikator dotičnog procesa, redni broj susjeda odvojenih zarezom i dijametar mreže. Zatim slijedi naziv sinkronizatora kojeg koristimo. Argument naredbenog retka koji služi za oda-

bir sinkronizatora ima: vrijednost ”simple” znači da biramo jednostavni sinkronizator, a vrijednost ”alpha” da biramo sinkronizator α.

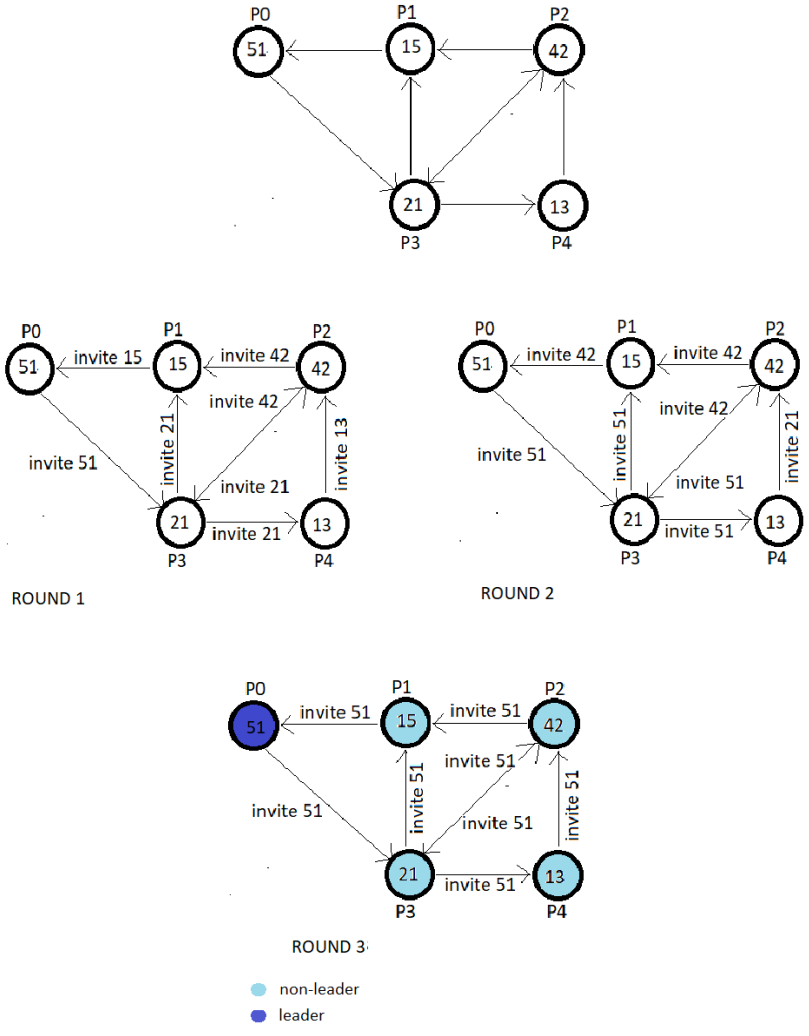
Tijekom rada programa, njegovi procesi ispisuju razne obavijesti, posebno poruke send i recieve. Iz tih ispisa moguće je pratiti plusove algoritma te dobiti vođu u općenitoj mreži.

SynchronizerFloodMaxTester.java.

```
public class SynchronizerFloodMaxTester {

    public static void main(String[] args) throws Exception {
        String baseName = args[0];
        int myId = Integer.parseInt(args[1]);
        int numProc = Integer.parseInt(args[2]);
        Linker comm = new Linker(baseName, myId, numProc);
        int UID = Integer.parseInt(args[3]);
        String neighbours = args[4];
        int diam = args[5]; //diam = 3
        Synchronizer pulser = null;
        if (args[6].equals("simple"))
            pulser = new SimpleSynch(comm);
        else if (args[6].equals("alpha"))
            pulser = new AlphaSynch(comm);

         SynchFloodMax g = new SynchFloodMax(comm, UID, neighbours, diam,
            pulser);

        for (int i = 0; i < numProc; i++)
             if (i != myId) (new ListenerThread(i, pulser)).start();

        g.startElection();

        int leader = g.getLeader();

        System.out.println("ET The leader is " + leader);
    }
}
```

Primjer

U nastavku slijedi konkretni primjer rada algoritma SynchFloodMax. Promatramo općenitu mrežu od 5 procesa, dijametra 3, povezanu jednosmjernim ili dvosmjernim kanalima. Identifikatori UID redom iznose: 51, 15, 42, 21 i 13. Postupak izbora vođe istovremeno pokreću svi procesi. Tijek algoritma vidi se na nizu grafova na slici 3.1



Slika 3.1: Primjer rada algoritma za izbor vođe

3.2 Problem najkraćeg puta u mreži

Algoritmi za traženje najkraćeg puta bili su predmetom opsežnih istraživanja dugi niz godina što rezultira velikim brojem algoritama s različitim uvjetima i ograničenjima. Velika većina njih bavi se fiksnim grafovima, odnosno fiksnom topologijom i fiksnim težinama. Napredak računalnih mreža i distribuirana obrada donijeli su novo zanimanje za temu, bazirajući se na ovisnost o vremenu.

Ispitujemo generalizaciju BFS (eng. *breadth-first search*) problema. Koristimo jako povezan usmjereni graf, s mogućnošću jednosmjerne komunikacije između parova susjeda. Ovaj put, međutim, pretpostavljamo da svaki luk $e = (i, j)$ ima pripadajuću realnu vrijednost, težinu, koju označavamo $weight(e)$ ili $weight_{i,j}$. Težina puta je definirana kao suma težina na njegovim lukovima. Problem je pronaći najkraći put od izdvojenog izvornog čvora i_0 u grafu do svakog drugog čvora u grafu, gdje je najkraći put definiran kao put s minimalnom težinom. Zbirka najkraćih putova od i_0 do svih ostalih čvorova u grafu čini podstablo grafa, čiji su lukovi orijentirani od roditelja prema djetetu.

Kod pretraživanja po širini dobivamo motivaciju za izgradnju stabla zbog želje da imamo prikladnu strukturu koja će se koristiti za broadcast komunikaciju[4]. Težine predstavljaju troškove koji mogu biti povezani s postupkom posjećivanja (provjere ili ažuriranja) svih lukova u grafu, kao na primjer, kašnjenje u komunikaciji. Stablo najkraćih putova minimizira maksimum troškova komunikacije u najgorem slučaju s bilo kojim procesom u mreži.

Pretpostavljamo da svaki proces u početku zna težinu svih svojih ulaznih kanala. Također pretpostavljamo da svaki proces zna broj n čvorova u grafu. Zahtijevamo da svaki proces treba odrediti roditelja u danom stablu najkraćih putova, kao i njegovu udaljenost (tj. ukupnu težinu njegovih najkraćih putova) iz i_0. Ako su svi lukovi jednake težine, tada je zapravo BFS stablo, stablo najkraćih putova. Dakle, u ovom slučaju, trivijalna modifikacija jednostavne konstrukcije *SynchBFS*[5] stabla se može načiniti kako bi se dobile informacije o udaljenosti kao i *parent* pokazivači. Zanimljiviji je slučaj u kojem težine mogu biti nejednake. Jedan od načina kako riješiti problem je sljedećim algoritmom, inačicom Bellman-Ford algoritma.

[4] Broadcast omogućuje da se poruka iz korijena dostavi svim procesima.

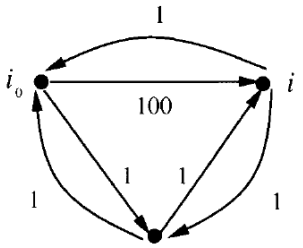
[5] Opis ovog algoritma može se pronaći u [2].

Algoritam BellmanFord

Svaki proces i prati $dist$, najkraću udaljenost od i_0 koju poznaje do sada, zajedno s *parent*, dolaznim susjedom koji prethodi i u stazi, čija je težina $dist$. U početku je $dist_{i_0} = 0$, $dist_i = \infty$ za $i \neq i_0$ te su parent komponente nedefinirane. U svakoj rundi, svaki proces šalje $dist$ svim svojim odlazećim susjedima. Zatim svaki proces i ažurira svoj $dist$ tako da uzima minimum svoje prethodne $dist$ vrijednosti i svih vrijednosti $dist_j + weight_{j,i}$, gdje je j dolazni susjed. Ako se $dist$ promijeni, *parent* komponenta se također ažurira. Nakon $n - 1$ rundi, $dist$ sadrži najkraću udaljenost i *parent*, roditelja u stablu najkraćih putova.

Nije teško vidjeti da nakon $n - 1$ rundi, vrijednosti $dist$ konvergira u točnu udaljenost. Indukcijom na r se može ispitati točnost BellmanFord algoritma tako da pokažemo da nakon r rundi algoritam radi sljedeće: svaki proces i ima svoje $dist$ i *parent* komponente koje odgovaraju najkraćem putu između tih putova od i_0 do i koje se sastoje od najviše r lukova. (Ako takvih staza nema, tada je $dist = \infty$ i *parent* je nedefiniran.)

Analiza složenosti. Vremenska složenost algoritma BellmanFord je $n - 1$, a broj poruka je $(n - 1)|E|$.



Slika 3.2: Najkraća staza se stabilizira za samo 2 runde, iako je diam = 1.

Primjer vremenske složenosti BellmanForda. Mogli biste posumnjati po analogiji sa SynchBFS da je složenost vremena BellmanFord algoritma zapravo diam. Primjer koji ukazuje zašto to nije slučaj prikazan je na slici 3.2. U ovom primjeru potrebne su nam 2 runde kako bismo našli ispravnu udaljenost, 2, od i_0 do i, budući da put kojim se realizira ta udaljenost ima dva luka. Međutim, dijametar je 1.
Algoritam BellmanFord također radi koristeći gornju granicu na n, umjesto samog broja n.

Opis klase SynchBellmanFord uz pomoć sinkronizatora

Za problem pronalaska najkraćih putova od određenog izvora veliki je dobitak upotreba sinkronizatora. Algoritam *AsynchBellmanFord*[6] ima složenost vremena i komunikacije eksponencijalnu s obzirom na broj čvorova. Međutim, sinkroni algoritam *BellmanFord* ima složenost komunikacije "samo" $O(n \cdot |E|)$ i složenost rundi samo $O(n)$, za mrežu s poznatom veličinom n. Možemo pokrenuti sinkroni algoritam *SynchBellmanFord* koristeći, recimo, sinkronizator *Alpha*, dobivajući algoritam koji šalje $O(n \cdot |E|)$ poruka i koristi $O(n \cdot k)$ vrijeme za simulaciju potrebnih n rundi, gdje je k gornja granica na vrijeme za dostavljanje najstarije poruke u kanal. Sinkronizator SimpleSynch bi funkcionirao jednako dobro.

Slijedi detaljan opis klase SynchBellmanFord.

Imamo skup procesa $P_0, P_1, P_2, ..., P_{N-1}$. Između nekih od njih postoje jednosmjerni komunikacijski kanali. Treba utvrditi optimalni način slanja poruke od P_0 do bilo kojeg P_i tako da ukupno kašnjenje bude minimalno. Situacija se može opisati usmjerenim grafom, gdje su procesi čvorovi, kanali lukovi, a kašnjenja cijene (duljine) lukova. Optimalni način slanja poruke odgovara najkraćem putu u grafu. Opisana verzija problema zapravo je tzv. "single source shortest path problem" jer se najkraći put traži od jednog polaznog do bilo kojeg dolaznog čvora, a ne između svaka dva čvora.

- Varijable su sljedeće.
  - *parent* varijabla je neposredni prethodnik od P_i na najkraćem putu od P_0 do P_i koji je trenutno poznat P_i -u. Inicijalno je -1.
  - *cost* je cijena (kašnjenje) najkraćeg puta od P_0 do P_i koji je trenutno poznat P_i-u. Inicijalno je -1.
  - *weight* je pomoćna varijabla tipa String koja sadrži težine lokalnih ulaznih kanala ili -1 za nesusjedne procese i izlazne kanale, odvojene zarezom.
  - *edgeWeight* je polje intova. Svaki proces zna koliko iznosi prosječno kašnjenje (delay) po svakom od njegovih ulaznih kanala. Te vrijednosti proces P_i ima upisane u svom polju $edgeWeight[]$, gdje je $edgeWeight[k]$ kašnjenje za kanal od P_k do P_i .
  - *stringNeighbours* je pomoćna varijabla tipa String. Redni broj susjeda, odvojenih zarezom.
  - *neighbours* je objekt tipa IntLinkedList u koji spremimo susjede lokalnog procesa.

[6] Opis ovog algoritma može se pronaći u [1].

- *s* je sinkronizator, dakle objekt sa sučeljem Synchronizer.

- Procesi razmjenjuju poruke tipa path. Unutar te poruke P_i dojavljuje P_j -u svoju trenutnu vrijednost varijable cost. Dakle P_i dojavljuje P_j -u kolika je cijena najkraćeg puta od P_0 do P_i za kojeg zna P_i.

- Metoda *initiate*() upravlja cijelim tijekom algoritma.

  - Definiraju se objekti *edgeWeight* i *neighbours*.
  - Inicijalizira se sinkronizator pozivom njegove metode *initialize*().
  - U slučaju procesa P_0 varijable cost i parent se definiraju na vrijednost 0.
  - Vrti se petlja čiji koraci odgovaraju pulsevima.
  - U slučaju pulsa nula, P_0, proces koordinator, koji se ponaša kao okolina i započinje računanje, šalje poruku path sa sadržajem 0 po svim svojim izlaznim kanalima.
  - Zbog blokirajućeg poziva *nextPulse*(), takt izvršavanja petlje određuje sinkronizator.

- Klasa SynchBellmanFord proširuje klasu Process. Dakle, SynchBellmanFord mora implementirati metodu *handleMsg*().

- Metoda *handleMsg*(). Kad P_j dobije od P_i poruku path s cijenom c, gleda je li njegova trenutna cijena najkraćeg puta od P_0 do P_j veća od cijene najkraćeg puta od P_0 do P_i, plus cijena luka od P_i do P_j. Ako jest, tada je P_j otkrio kraći put od P_0 do P_j pa on ažurira svoje varijable *cost* i *parent*. Štoviše, nakon takvog ažuriranja P_j je dužan dojaviti susjedima svoju novu cijenu, tj. poslati nove poruke *path* po svim izlaznim kanalima.

SynchBellmanFord.java

```
public class SynchBellmanFord extends Process {
   int parent = -1;
   int cost = -1;
   String weight;
   int edgeWeight[] = new int[N];
   IntLinkedList neighbours = new IntLinkedList();
   String stringNeighbours;
   Synchronizer s;

   public SynchBellmanFord(Linker initComm, String weight, String
       stringNeighbours, Synchronizer initS) {
      super(initComm);
      this.weight = weight;
      this.stringNeighbours = stringNeighbours;
      s = initS;
   }

   public void initiate() {
      StringTokenizer st1 = new StringTokenizer(weight, ",");
      int i = 0;
      while (st1.hasMoreTokens()) {
         edgeWeight[i] = Integer.parseInt(st1.nextToken());
         ++i;
      }
      StringTokenizer st2 = new StringTokenizer(stringNeighbours, ",");
      while (st2.hasMoreTokens()) {
         neighbours.add(Integer.parseInt(st2.nextToken()));
      }
      s.initialize(this);
      if (myId == Symbols.coordinator) {
         cost = 0;
         parent = 0;
      }
      for (int pulse = 0; pulse < N-1; ++pulse) {
          if ((pulse == 0) && (myId == Symbols.coordinator))
            for (int j = 0; j < N; ++j)
                  if (isNeighbor(j) && neighbours.contains(j))
                     s.sendMessage(j, "path", cost);
```

```
        s.nextPulse();
      }
  }

  public void handleMsg(Msg m, int src, String tag){
    if (tag.equals("path")) {
      int dist = m.getMessageInt();
      if ((parent == -1) || (dist + edgeWeight[src] < cost)) {
        parent = src;
        cost = dist + edgeWeight[src];
        for (int p = 0; p < N; ++p)
              if (isNeighbor(p) && neighbours.contains(p))
                 s.sendMessage(p, "path", cost);
      }
    }
  }
}
```

SynchBellmanFord algoritam uz pomoć sinkronizatora testiramo programom SynchronizerBellmanFordTester. Slijede objašnjenja.
Prvi argument naredbenog retka kojim se pokreće program nakon SynchronizerBellmanFordTester je bazno ime pod kojim NameServer evidentira izvođenje programa. Zatim kao argumenti slijede redni broj te ukupan broj procesa. Zatim slijedi varijabla s težinama lokalnih ulaznih kanala, susjedi lokalnog procesa i naziv sinkronizatora kojeg koristimo.

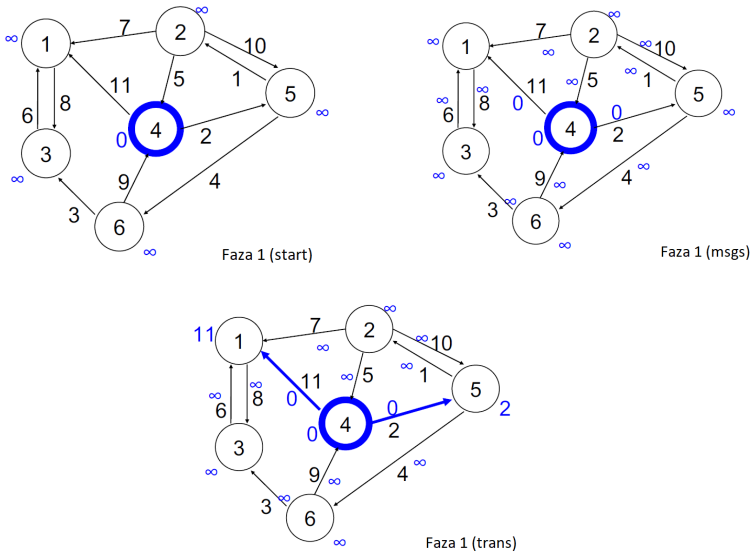
SynchronizerBellmanFordTester.java

```
public class SynchronizerBellmanFordTester {

   public static void main(String[] args) throws Exception {

     String baseName = args[0];
     int myId = Integer.parseInt(args[1]);
     int numProc = Integer.parseInt(args[2]);
     Linker comm = new Linker(args[0], myId, numProc);
     String edgeWeight = args[3];
     String neighbours = args[4];

     Synchronizer pulser = null;
```

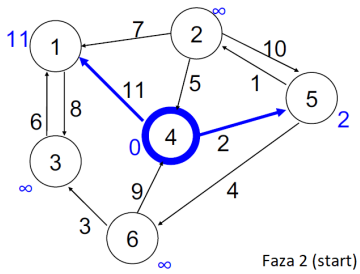
```
        if (args[5].equals("simple"))
            pulser = new SimpleSynch(comm);
        else if (args[5].equals("alpha"))
            pulser = new AlphaSynch(comm);
      SynchBellmanFord t = new SynchBellmanFord(comm, edgeWeight,
         neighbours, pulser);

      for (int i = 0; i < numProc; i++)
           if (i != myId) (new ListenerThread(i, pulser)).start();
      t.initiate();
    }
}
```

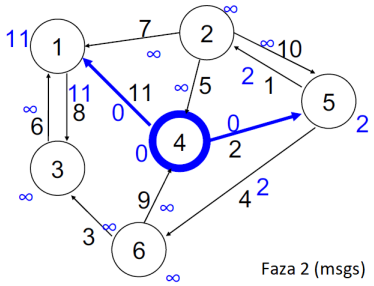
Primjer

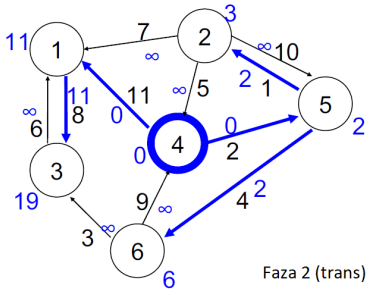
U nastavku slijedi konkretni primjer rada algoritma SynchBellmanFord. Promatramo općenitu mrežu od 6 procesa, povezanu jednosmjernim ili dvosmjernim kanalima. Čvor 4 je i_0 (P_0), izdvojeni izvorni čvor od kojeg tražimo najkraći put do ostalih čvorova u grafu. Postupak traženja najkraćeg puta, prvi pokreće proces koordinator. Tijek algoritma vidi se na nizu grafova na slici 3.3[7].

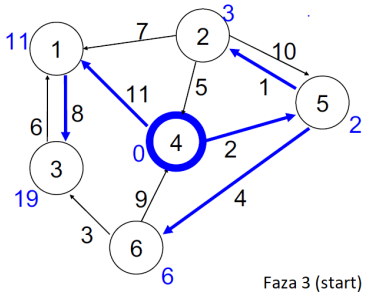


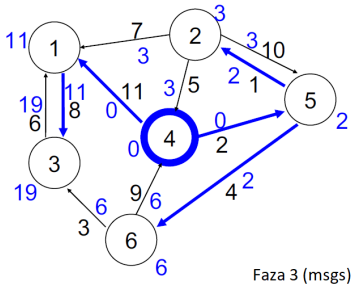
[7] Preuzeto s https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-852j-distributed-algorithms-fall-2009/lecture-notes/MIT6_852JF09_lec03.pdf

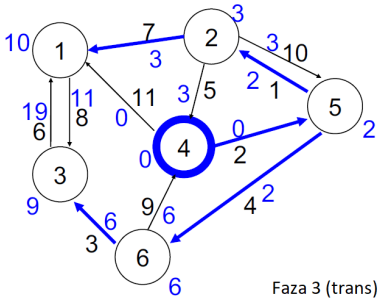

Faza 2 (start)

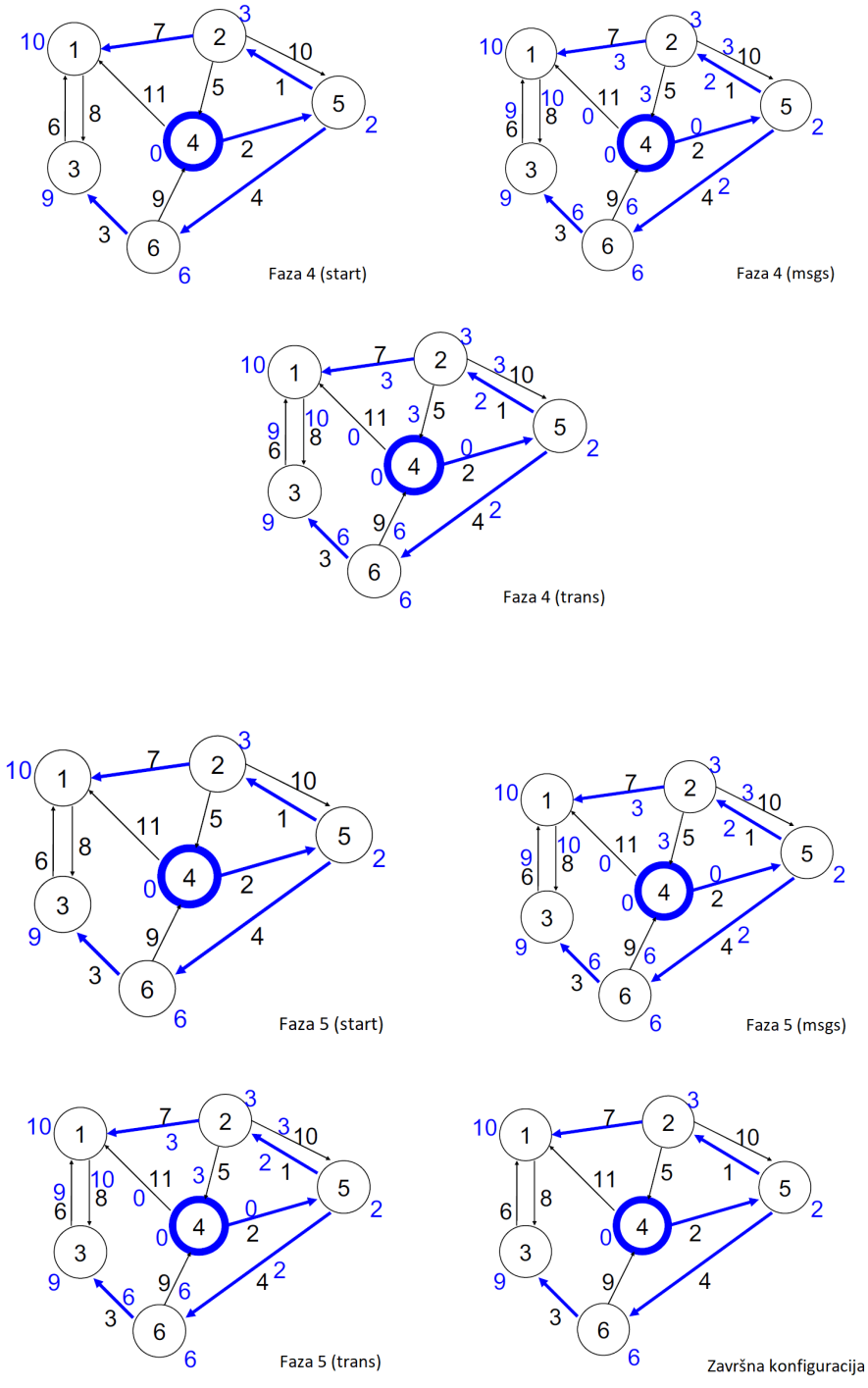

Faza 2 (msgs)


Faza 2 (trans)


Faza 3 (start)


Faza 3 (msgs)


Faza 3 (trans)



Slika 3.3: Primjer rada algoritma SynchBellmanForda-a

3.3 Maksimalni nezavisni skup

Problem koji razmatramo u ovom potpoglavlju je problem pronalaska maksimalnog nezavisnog skupa (MIS) čvorova usmjerenog grafa.
Skup čvorova se naziva *nezavisnim skupom* ako ne sadrži niti jedan par susjednih čvorova i kaže se da je nezavisni skup *maksimalan* ako ga nije moguće povećati da bi se dobio veći nezavisan skup dodavanjem nekog od ostalih čvorova.
Usmjereni graf može imati mnogo različitih maksimalnih nezavisnih skupova. Ne zahtijevamo najveći mogući maksimalni nezavisni skup - bilo koji je dobar.
Jedna od motivacija za MIS algoritam je problem raspodjele (alokacije) zajedničkih resursa za procese u mreži. Susjedi na grafu G predstavljaju procese koji ne mogu istovremeno obavljati neku aktivnost koja uključuje zajedničke resurse (na primjer, pristup bazi podataka). Ako želimo odabrati skup procesa kojima bi se moglo dopustiti da djeluju istovremeno tada bi ti procesi trebali sadržavati nezavisan skup G kako bi izbjegli sukobe. Nadalje, iz razloga izvedbe nepoželjno je blokirati proces ako nijedan od susjeda nije aktivan, stoga bi odabrani skup procesa trebao biti maksimalan.

Problem

Neka je $G = (V, E)$ neusmjereni graf.
Kaže se da je skup $I \subseteq V$ *nezavisan* ako je za sve čvorove $i, j \in I$, $(i, j) \notin E$.
Nezavisni skup I je *maksimalan* ako bilo koji skup I' koji strogo sadrži I nije nezavisan.
Cilj je izračunati maksimalni nezavisni skup od G. Točnije, svaki proces čiji se indeks nalazi u I, na kraju bi trebao postaviti posebnu status komponentu svog stanja na vrijednost *in*, a svaki proces čiji indeks nije u I, trebao bi kao izlaz dati *out*.
Pretpostavljamo da je n, broj čvorova, poznat svim procesima (mogli bi, alternativno, upotrijebiti gornju granicu na n). Ne pretpostavljamo postojanje UID-ova.

Randomizirani algoritam

Nije teško pokazati da se na nekim grafovima problem MIS-a ne može riješiti ako se zahtijeva da procesi budu deterministički. U ovom dijelu predstavljamo jednostavno rješenje koje koristi randomizaciju da bi prevladao ovo svojstveno ograničenje determinističkih sustava. Da budemo precizni, napominjemo da nasumični algoritam zapravo rješava slabiji problem od onog koji je gore naveden, u tome što će imati (vjerojatnost nula) mogućnost da nikad ne prestane. Ovaj algoritam nazivamo LubyMIS, po Lubyju njegovom pronalazaču.

LubyMIS se temelji na sljedećoj iterativnoj shemi u kojoj je proizvoljni neprazni nezavisni skup odabran iz danog grafa G, čvorovi u ovom skupu i svi njihovi susjedi uklonjeni su iz grafa, a proces je ponovljen. Ako je W podskup čvorova grafa, tada koristimo

$nbrs(W)$, što označava skup susjeda čvorova u W.

Neka *graph* bude zapis s poljima *nodes*, *edges* i *nbrs* inicijaliziranim na naznačene komponente izvornog grafa G. Neka je I skup čvorova, na početku prazan.

while $graph.nodes \neq \emptyset$ do
 izaberi neprazan skup $I' \subseteq graph.nodes$ koji je nezavisan u graph
 $I := I \cup I'$
 $graph :=$ inducirani podgraf[8] grafa na $graph.nodes - I' - graph.nbrs(I')$
end while

Nije teško vidjeti da ova shema uvijek daje maksimalan nezavisan skup. Da biste vidjeli zašto je to nezavisno, imajte na umu da je u svakoj fazi, odabrani skup I' nezavisan i izričito odbacujemo od preostalog grafa sve susjede vrhova koji se stavljaju u I. Da biste vidjeli zašto je to maksimalno, imajte na umu da su jedini čvorovi koji su uklonjeni iz razmatranja, susjedi čvorova stavljenih u I.

Ključno pitanje u primjeni ove opće sheme u distribuiranoj mreži je kako odabrati skup I' pri svakoj iteraciji. Ovdje se koristi slučajnost.

U svakoj fazi, svaki proces i odabire cjelobrojni val_i u rasponu $1, ..., n^4$ nasumično, koristeći jednoliku raspodjelu (eng. *uniform distribution*)[9]. Razlog upotrebe n^4 kao granice je da je dovoljno velik, tako da, s velikom vjerojatnošću, svi procesi u grafu će odabrati različite vrijednosti. Jednom kad procesi odaberu te vrijednosti, definiramo da se I' sastoji od svih čvorova i čija je status komponenta poprimila vrijednost *in*, to jest oni čvorovi i takvi da $val_i > val_j$ za sve susjede j od i. Ovo očito daje nezavisan skup, jer dva susjeda ne mogu istodobno poraziti jedan drugog.

U ovoj je provedbi moguće da skup I' bude prazan u nekim fazama zbog slučajnog izbora; te će faze biti "beskorisne". Pod uvjetom da algoritam ne dosegne točku nakon koje će neprestano izvoditi beskorisne faze, jednostavno možemo zanemariti beskorisne faze i tvrditi da LubyMIS ispravno slijedi opću shemu. Morat ćemo, međutim, uzeti u obzir beskorisne faze u analizi.

[8]Inducirani podgraf grafa G je podgraf G', gdje se E' sastoji od lukova grafa G čija su oba kraja u V'. Dakle, ako je zadan skup svih vrhova $V' \subset V$, onda je E' određen lukovima koji u grafu G spajaju te vrhove.

[9]Takva distribucija se koristi ako slučajna varijabla X može primiti samo vrijednosti iz skupa $R(X) = x_1, x_2, ..., x_n$ i to tako da je vjerojatnost realizacije svakog pojedinog ishoda ista, tj. $P\{X = x_i\} = 1/n$ za svaki $i \in \{1, ..., n\}$.

LubyMIS algoritam (neformalno):
Algoritam radi u fazama, a svaka se sastoji od tri kruga.

1. krug: U prvom krugu faze, procesi biraju val i šalju ih svojim susjedima. Do kraja prvog kruga, kada su sve val poruke primljene, procesi *in* - to jest procesi u I' - znaju tko su.

2. krug: U drugom krugu, procesi *in* obavještavaju svoje susjede. Pred kraj drugog kruga, procesi *out* - to jest procesi kojima su susjedi u I' - znaju tko su.

3. krug: U trećem krugu svaki procesi *out* obavještava svoje susjede. Tada svi uključeni procesi - *in*, *out* i susjedi od *out* - uklanjaju odgovarajuće čvorove i lukove s grafa. Točnije, to znači da procesi *in* i *out* prekidaju sudjelovanje nakon ove faze, a susjedi od *out* uklanjaju sve lukove koji su incidentni s novo uklonjenim čvorovima.

Sada algoritam opisujemo formalnije u našem modelu. Kao što je već opisano, svaki proces koristi posebnu slučajnu funkciju $rand_i$ koja se primjenjuje svaki krug prije primjene $msgs_i$ i $trans_i$ funkcija. Ovdje koristimo *random* kako bi se naznačio slučajni izbor iz $\{1, ..., n^4\}$ koristeći jednoliku raspodjelu.

LubyMIS algoritam (formalno):

$states_i$:
round $\in \{1, 2, 3\}$, inicijalno 1
val $\in \{1, ..., n^4\}$
awake, Boolean, inicijalno *true*
rem-nbrs, skup vrhova, inicijalno susjedi početnog grafa G
status $\in \{unknown, winner, loser\}$, inicijalno unknown

$rand_i$
if *awake* i *round* = 1 then *val* := *random*

$msgs_i$:
if *awake* then
case
round = 1:
šalji val svim čvorovima u rem-nbrs
round = 2:
if *status* = *winner* then
šalji winner svim čvorovima u rem-nbrs

round = 3:
if *status* = *loser* then
šalji loser svim čvorovima u rem-nbrs
endcase

U sljedećem kodu identificiramo 3 s 0 modulo 3.

$trans_i$:
if *awake* then
case
round = 1:
if *val* > *v* za sve nadolazeće vrijednosti v then *status* := *winner*
round = 2:
if winner poruka stigne then *status* := *loser*
round = 3:
if $status \in \{winner, loser\}$ then *awake* := $false$
$rem\text{-}nbrs := rem\text{-}nbrs - \{j : loser$ poruka stigne od $j\}$
endcase
round : = (round + 1 mod 3)

Imajte na umu da LubyMIS i dalje radi ispravno ako u nekim fazama neki susjedni procesi odaberu iste slučajne vrijednosti.

Analiza

Pod uvjetom da se LubyMIS ne zaustavi izvršavajući zauvijek beskorisne faze, utvrđeno je da će se stvoriti MIS. S vjerojatnošću jedan tvrdimo da algoritam zapravo ne zastaje. Točnije, tvrdimo da u bilo kojem slučaju faze algoritma, očekivani broj lukova koji je uklonjen iz preostalog grafa je barem konstantan udio od ukupnog broja preostalih lukova. Iz ovoga slijedi da postoji stalna vjerojatnost da je barem konstantan udio lukova uklonjen. U suprotnom implicira da je očekivani broj rundi do završetaka $O(logn)$. Također implicira da s vjerojatnošću jedan, algoritam zapravo završi.

Teorem 2. *S vjerojatnošću jedan, LubyMIS se na kraju zaustavi. Štoviše, očekivani broj rundi do završetka je* $O(logn)$.

Randomizirani algoritmi. Tehnika randomizacije se često koristi u distribuiranim algoritmima. Njegova glavna upotreba je razbijanje simetrije. Na primjer, problemi izbora vođe i maksimalnog nezavisnog skupa ne mogu se riješiti općim grafovima s determinističkim procesima bez UID-ova zbog nemogućnosti prekida simetrije. Nasuprot tome,

ovi se problemi mogu riješiti pomoću randomizacije. Čak kada postoje UID-ovi, randomizacija može omogućiti brži prekid simetrije.

Jedan od problema s randomiziranim algoritmima je da se njihova jamstva ispravnosti i performansi mogu održati samo s velikom vjerojatnošću, ne sa sigurnošću. Pri dizajniranju takvih algoritama važno je osigurati da se ključna svojstva algoritma jamče sa sigurnošću, a ne s vjerojatnošću. Na primjer, svakom izvedbom LubyMIS algoritma zajamčeno je da se proizvede nezavisan skup, bez obzira na rezultate slučajnih izbora. Efikasnost izvođenja, međutim, ovisi o sreći slučajnih izbora. Postoji čak i (vjerojatnost nula) mogućnost da će svi procesi više puta odabrati istu vrijednost, čime će se zauvijek zaustaviti napredak.

Opis klase SynchLubyMIS uz pomoć sinkronizatora

Sinkronizatori se mogu koristiti i s randomiziranim sinkronim algoritmima kao što je SynchLubyMIS.

Slijedi detaljan opis klase SynchLubyMIS.

- Klasa SynchLubyMIS proširuje klasu Process. Dakle, SynchLubyMIS mora implementirati metodu *handleMsg*().
- Varijable su sljedeće.
  - *val* označava vrijednost lokalnog procesa.
  - *vals* je lista u koju dodamo sve vrijednosti susjeda i lokalnu vrijednost.
  - *maxVal* je pomoćna varijabla koja poprima najveću vrijednost iz liste vals
  - *status* je komponenta koja poprima vrijednosti iz skupa {inactive, in, out}. Početna vrijednost je inactive.
  - *s* je sinkronizator, dakle objekt sa sučeljem Synchronizer.
  - *update_nbrs* je pomoćni objekt klase IntLinkedList u kojem ažuriramo skup susjeda nakon izbacivanja. U početku su spremljeni svi susjedi lokalnog procesa.
- Metoda *initiate*() upravlja cijelim tijekom algoritma.
  - Dodamo sve susjede u *update_nbrs*.
  - Inicijalizira se sinkronizator pozivom njegove metode *initialize*().
  - Vrti se petlja čiji koraci odgovaraju pulsevima.

  - U svakoj fazi, u prvom krugu, dodjeljujemo val i pozivamo metodu startMIS() koja svim ažuriranim susjedima šalje poruku inactive s vrijednosti val.
  - Zbog blokirajućeg poziva *nextPulse*(), takt izvršavanja petlje određuje sinkronizator.
  - Pri nastavku izvršavanja petlje, također, u slučaju prvog kruga faze, pozivamo metodu check().
  - U check() metodi dodijelimo vrijednost maxVal varijabli i provjeravamo odnos lokalne vrijednosti val i maxVal. U slučaju da je val veća od maxVal, status komponentu postavljamo na in i svim preostalim susjedima javljamo da smo in i uklanjamo ih iz skupa susjeda.
  - I startMIS() i check() za slanje poruka koriste sinkronizatorovu metodu sendMessage().

- Metoda handleMsg().
  - Prima sve poruke susjeda i obrađuje slučajeve kad je poruka in, out ili inactive.
  - Pri primanju poruke in, status komponenta se postavlja na out, uklanja se pošiljatelj iz skupa susjeda, šalje se svim preostalim susjedima poruka out i uklanja ih se iz skupa preostalih lokalnih susjeda $update\_nbrs$.
  - U slučaju primljene poruke inactive, vrijednost susjeda spremimo u listu vals.
  - Ako je poruka out, izbrišemo pošiljatelja iz skupa lokalnih susjeda $update\_nbrs$.

SynchLubyMIS.java.

```
public class SynchLubyMIS extends Process {
   int val;
   int pulse;
   int maxVal = 0;
   String status = "inactive";
   Synchronizer s;
   List vals = new ArrayList();
   IntLinkedList update_nbrs = new IntLinkedList();

   public SynchLubyMIS(Linker initComm, Synchronizer initS) {
      super(initComm);
      s = initS;
   }

   public void initiate() {
      updateNeighbors.addAll(comm.neighbors);
      s.initialize(this);
      for (int pulse = 0; pulse < N; pulse++) {
         if((pulse % 3) == 0){
            val = (int)(Math.random() * Math.pow(N, 4)) + 1;
            startMIS();
         }
         s.nextPulse();
         if(status == "inactive" && (pulse % 3) == 0)
            check();
      }
   }

    public void handleMsg(Msg m, int src, String tag) {
      int j = m.getMessageInt();
      if (tag.equals("in")){
         status = "out";
         update_nbrs.removeObject(src);
         for (int i = 0; i < N; i++){
            if(update_nbrs.contains(i)){
               update_nbrs.removeObject(i);
               s.sendMessage(i, "out", myId);
            }
         }
```

```
      }
      else if (tag.equals("inactive"))
        vals.add(j);
      else if (tag.equals("out")){
        update_nbrs.removeObject(src);
      }
    }

    public void check(){
      if (!vals.isEmpty())
        maxVal = Collections.max(vals);
      else maxVal = 0;
      vals.clear();
      if(status == "inactive"){
        if (val > maxVal){
          status = "in";
          for (int i = 0; i < N; i++)
            if (update_nbrs.contains(i)){
              s.sendMessage(i, "in", val);
              update_nbrs.removeObject(i);
            }
         }
       }
    }

    public synchronized void startMIS() {
      for (int i = 0; i < N; i++)
        if (update_nbrs.contains(i))
          s.sendMessage(i, "inactive", val);
    }
}
```

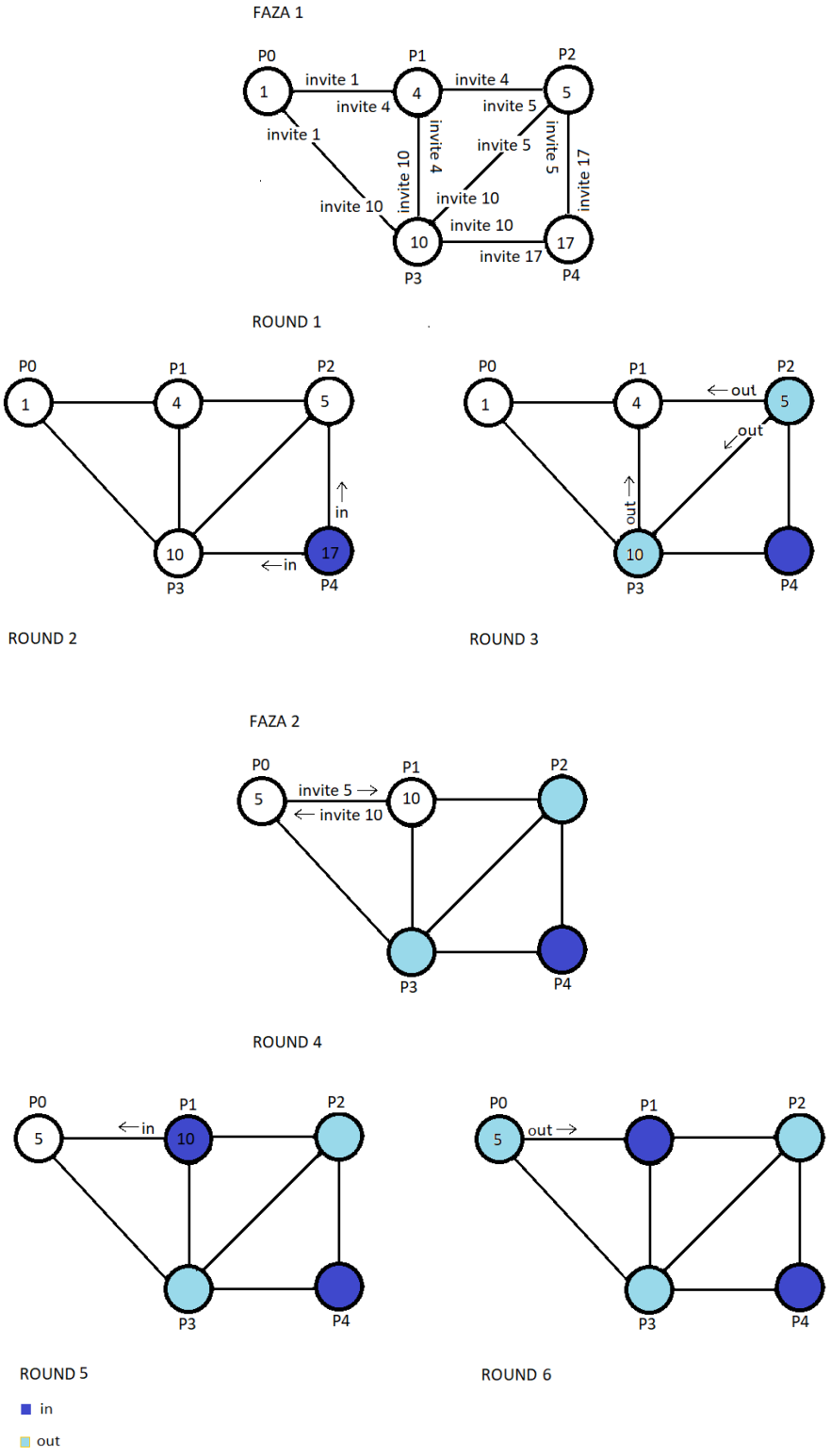
SynchLubyMIS algoritam uz pomoć sinkronizatora testiramo programom SynchronizerMISTester. Slijede objašnjenja.
Prvi argument naredbenog retka kojim se pokreće program nakon SynchronizerMISTester je bazno ime pod kojim NameServer evidentira izvođenje programa. Zatim kao argumenti slijede redni broj dotičnog procesa te ukupan broj procesa. Zatim slijedi naziv sinkronizatora kojeg koristimo.

SynchronizerMISTester.java

```
public class SynchronizerMISTester {

   public static void main(String[] args) throws Exception {

     String baseName = args[0];
       int myId = Integer.parseInt(args[1]);
       int numProc = Integer.parseInt(args[2]);
       Linker comm = new Linker(baseName, myId, numProc);
       Synchronizer pulser = null;
       if (args[3].equals("simple"))
           pulser = new SimpleSynch(comm);
       else if (args[3].equals("alpha"))
           pulser = new AlphaSynch(comm);
       else if (args[3].equals("beta"))
           pulser = new BetaSynch(comm, (myId == 0) );

       SynchLubyMIS m = new SynchLubyMIS(comm, pulser);

       for (int i = 0; i < numProc; i++)
            if (i != myId) (new ListenerThread(i, pulser)).start();

        m.initiate();
   }
}
```

Primjer

U nastavku slijedi konkretni primjer rada algoritma SynchLubyMIS. Promatramo općenitu mrežu od 5 procesa. Vrijednosti val redom iznose: 1, 4, 5, 10 i 17 u prvoj fazi, dok u drugoj fazi iznose 5 i 10. Postupak traženja maksimalnog nezavisnog skupa istovremeno pokreću svi procesi. Tijek algoritma vidi se na nizu grafova na slici 3.4.



Slika 3.4: Primjer rada algoritma LubyMIS-a

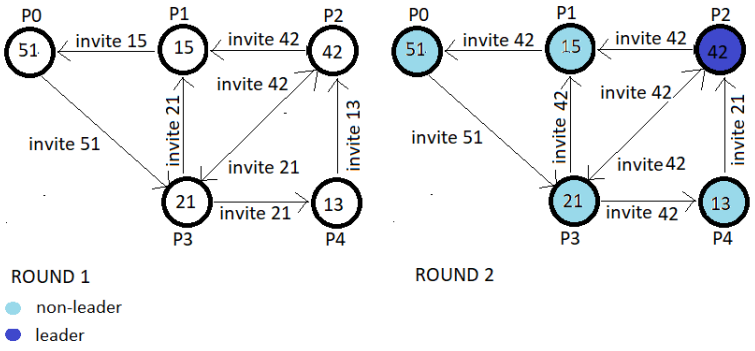
3.4 Pogreške u odsustvu sinkronizatora

Primijetimo da sinkroni algoritam ne mora ispravno raditi ako je mreža asinkrona. Naime, ako je vrijeme komunikacije nepredvidivo, radnje koje su se trebale obaviti u jednom pulsu mogle bi se odgoditi i pomiješati s radnjama koje već pripadaju idućem pulsu. Tako bi se mogao dobiti pogrešan rezultat.

Izbor vođe

Pogledajmo pogrešan izbor vođe u slučaju asinkronosti, promatramo istu mrežu kao na slici 3.1, no sad dobivamo rezultat kao na slici 3.5.

- Procesi šalju poruku invite svakom od svojih susjeda.
- Proces P_3 prima poruku invite 42 u drugoj rundi i šalje je procesima P_1, P_2 i P_4.
- Pogreška je nastala zato što je pozivna poruka od P_0 do P_3 zakasnila, a poruka od P_3 do P_2 uranila i time P_2 izabran vođom.



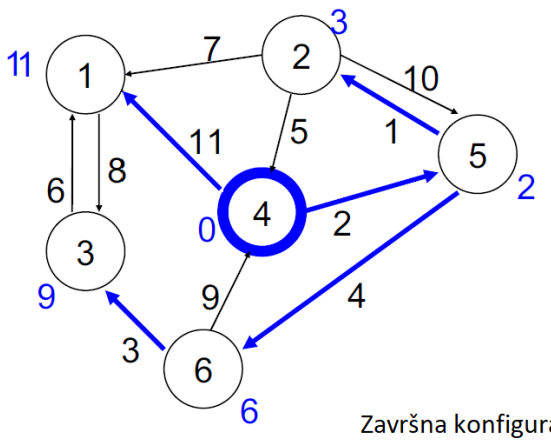
Slika 3.5: Primjer pogrešnog rada algoritma leader election u asinkronoj mreži bez korištenja sinkronizatora

Problem najkraćeg puta

Opišimo problem najkraćeg puta u slučaju asinkrone mreže bez sinkronizatora. U slučaju mreže sa slike 3.3, dobivamo pogrešku koju vidimo na slici 3.6.

Neka su redom čvorovi 4, 1, 2, 3, 5, 6 procesi i_0 (P_0), P_1, P_2, P_3, P_4, P_5.

- U trećoj fazi proces P_2 šalje poruku path 10 procesu P_1.
- Proces P_1 u sljedećoj fazi ne prima poslanu poruku procesa P_2.
- Pogreška je nastala zato što je poruka od P_2 do P_1 zakasnila,a ostale poruke sljedećih faza uranile i time je izabran pogrešan najkraći put do P_1 s troškom 11 umjesto 10.



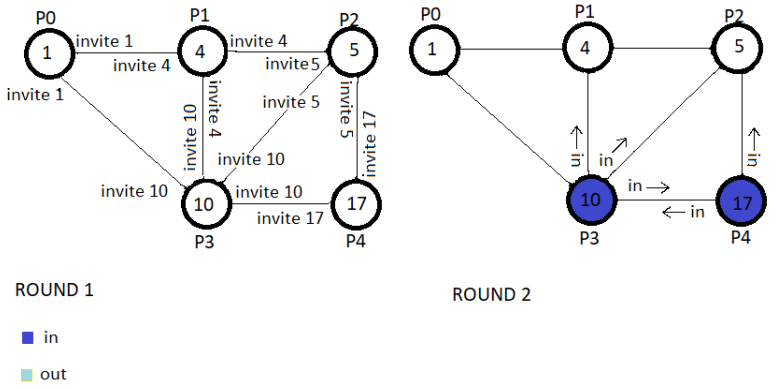
Slika 3.6: Primjer pogrešnog rada algoritma BellmanFord u asinkronoj mreži bez korištenja sinkronizatora

Maksimalni nezavisni skup

Pokažimo odabir maksimalnog nezavisnog skupa u slučaju asinkronosti. Uočimo da sada promatramo istu mrežu kao na slici 3.4, i dobivamo rezultat kao na slici 3.7.

- Procesi šalju poruku invite s identifikatorom svakom od svojih susjeda.
- Proces P_3 prima poruke od procesa P_1 i P_2 i šalje procesima P_1, P_2 i P_4.

- Proces P_3 se proglašava in zbog kašnjenja poruke procesa P_4.
- Proces P_4 se proglašava in također, čime gubimo nezavisnost skupa.



Slika 3.7: Primjer pogrešnog rada algoritma LubyMIS u asinkronoj mreži bez korištenja sinkronizatora

U prethodnim implementacijama ovog poglavlja, pokazali smo kako se sinkrona aplikacija povezuje sa sinkronizatorom. Konkretno, povezali smo sinkrone aplikacije za izbor vođe, traženje najkraćeg puta i maksimalnog nezavisnog skupa sa sinkronizatorom i dobili aplikacije koje ispravno rade i na asinkronoj mreži.

Bibliografija

[1] Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc. An Imprint of Elsevier, 1996.

[2] Robert Manger, *Distribuirani procesi*, PMF - Matematički odsjek, Zagreb, 2017.

[3] Michel Raynal, *Distributed Algorithms for Message-Passing Systems*, Springer-Verlag Berlin Heidelberg, 2013.

[4] Garg V.K., *Concurrent and Distributed Computing in Java*, Wiley – IEEE Press, Hoboken NY, 2004.

Sažetak

Ovaj rad opisuje model sinkrone mreže i obrađuje tri sinkrona algoritma u općenitoj mreži, algoritam za izbor vođe, problem najkraćeg puta i maksimalni nezavisni skup. Nabrojani algoritmi služe u rješavaju problema izbora ”vođe” u mrežnom računanju, za izgradnju struktura pogodnih za podršku učinkovite komunikacije, rješavanje problema mrežne alokacije resursa i slično. Nakon opisa i analize algoritama, slijedi njihova implementacija u asinkronoj mreži sa sinkronizatorom. U tu svrhu, korišteni su jednostavni sinkronizator i sinkronizator alpha, koji su prethodno opisani. Na kraju je na primjerima demonstriran rad algoritama u odsustvu sinkronizatora.

Summary

This thesis describes synchronous network model and addresses three algorithms in general synchronous network, leader election, shortest paths and maximal independent set. The above-mentioned algorithms are used to solve the problem of choosing a ”leader” in network computing, to build a structure suitable for support communication, to solve the problem of network resource allocations, et cetera. After describing and analyzing the algorithms, they are implemented in asynchronous network model with synchronizer. For this purpose, simple synchronizer and alpha synchronizer were used and described previously. Finally, it is demonstrated on examples how algorithms work in the absence of a synchronizer.

Životopis

Rođena sam 12. ožujka 1995. godine u Ljubuškom. Osnovnu školu završila sam u Klobuku, a opću gimnaziju u Ljubuškom, BiH, gdje sam maturirala 2013. godine. U ak. god. 2013./2014. upisala sam preddiplomski sveučilišni studij matematike na Prirodoslovno-matematičkom fakultetu Sveučilišta u Zagrebu, koji sam kao prvostupnica završila 2017. godine. Na istome sam fakultetu ak. god. 2017./2018. upisala diplomski sveučilišni studij Računarstvo i matematika. Tijekom školovanja sam pružala instrukcije iz matematike i programiranja. Od travnja 2019. godine zaposlena sam u Ericsson Nikola Tesla d.d. na odjelu kontinuirane integracije.