Distribuirani algoritmi za bojenje grafa
Distribuirani procesi

Matej Duvnjak
Tea Poljanić
Jelena Rajič

izv. prof. dr. sc. Robert Manger

July 25, 2020

Sadržaj

1 Uvod

Bojanje grafova je važan problem u Teoriji grafova s kojim se susrećemo kada modeliramo probleme na razini aplikacije. Problem se sastoji od dodijeljivanja vrijednosti (boje) svakom vrhu tako da:

- ne postoje dva susjedna vrha grafa koje imaju istu boju - **Zakonito bojanje**
- broj boja je „razmjerno mali"

Kada je broj boja najmanji mogući, problem je NP- potpun. Neka je Δ maksimalan stupanj grafa (pretpostavljamo da su bilo koja dva vrha grafa povezana najviše jednim bridom, a stupanj vrha je broj njegovih susjeda). Uvijek je moguće obojiti vrhove grafa u Δ+1 boja. Ovo proizlazi iz jednostavnog dokaza indukcijom. Tvrdnja je trivijalno istinita za bilo koji graf s najviše Δ vrhova. Zatim, pod pretpostavkom da to vrijedi za svaki graf sastavljen od n Δ vrhova čiji je maksimalan stupanj najviše Δ, dodajemo novi vrh na graf t.d. je Δ još uvijek maksimalan stupanj grafa. Kako (prema pretpostavci) Δ je maksimalan stupanj grafa slijed ima najviše Δ susjeda. Dakle, vrh se može obojiti najmanje jednom od Δ+1 boja da zadovolji tražene uvjete.

2 Sekvencijalni algoritam za bojanje grafa

U ovom poglavlju pokazati ćemo Sekvencijalni algoritam za bojanje grafa u najviše (Δ+1) boja. Algoritam je opisan na slici ispod. Niz COLOR od n varijabli inicijaliziran je na $[\perp, \perp, \perp, ..., \perp]$. Kada algoritam odradi $i - ti$ korak $COLOR[i]$ će sadržavati boju pridruženu procesu p_i. Boje su reprezentiranje brojevima od 1 do $\Delta + 1$. Algoritam uzima u obzir svaki vrh i te mu pridružuje prvu boju koja nije pridružena njegovim susjedima.

```
(1)  for i from 1 to n do
(2)    c ← 1;
(3)    while (COLOR[i] = ⊥) do
(4)      if (⋀_{j∈neighbors_i} COLOR[j] ≠ c) then COLOR[i] ← c else c ← c + 1 end if
(5)    end while
(6)  end for.
```

Slika 1: Pseudo kod skevencijalnog bojanja grafa

3 Distribuirano Δ+1 bojanje

U ovom odjeljku predstavljamo distribuirani algoritam koji boji u najviše (Δ+1) boja procese na takav način da nijedna dva susjeda nemaju istu boju. Bojanje na distribuiranim sustavima se susreće u praktičnim problemima kao što su raspodjela resursa ili raspoređivanje procesora. Općenitije, distribuirani algoritmi za bojanje su algoritmi za razbijanje simetrije u smislu da oni podijele skup procesa na podskupove (podskup po boji) tako da nijedna dva procesa u istom podskupu nisu susjedi.

Distribuirani algoritam za bojanje grafova opisan je ispod. Ovaj algoritam pretpostavlja da su procesi već obojeni u $m \geq \Delta + 1$ boja na takav način da niti jedna dva susjeda nemaju istu boju. Opažamo da je, sa stanovišta računarstva, ova pretpostavka besplatna (jer uzimanje $m = n$ i trivijalno definiranje boje procesa p kao njegov indeks zadovoljava ovu početnu pretpostavku o bojanju). Za razliku od toga, uzimanje $m = \Delta + 1$ pretpostavlja da je problem već riješen.

Lokalne varijable. Svaki proces p upravlja lokalnom varijablom $color_i[i]$ koja predstavlja početnu boju, a na kraju algoritma će sadržavati konačnu boju procesa. Proces p_i također upravlja lokalnom varijablom $color_i[j]$ za svakog od svojih susjeda p_j. Kako je algoritam asinkron i baziran na rundama, lokalna varijabla r_i kojim upravlja p_i označava njezin trenutni lokalni broj runde.

Ponašanje procesa p_i. Procesi se izvršavaju u uzastopnim asinkronim rundama i prije svake runde svaki proces sinkronizira svoj napredak sa susjedima. Kako su runde asinkrone, brojevi runde moraju izričito biti upravljani unutar samog procesa. Dakle, svaki proces p_i upravlja lokalnom varijablom r_i koja se povećava kada započinje nova asinkrona runda (*linija 5*). Prva runda (*linije*

1-2) je početna runda tijekom koje procesi razmjenjuju svoju početnu boju kako bi se popunio njihov lokalni niz $colors_i[neighbours_i]$. Ako procesi znaju početne boje svojih susjeda, ova runda komunikacije može se izostaviti. Procesi zatim izvršavaju $m - (\Delta + 1)$ asinkrone runde gdje je m broj različitih boja koje su pridružene procesima (*linija 5*). Procesi čija početna boja pripada skupu boja $\{1, ., , , \Delta + 1\}$ zadržavaju svoju početnu boju zauvijek. Ostali procesi ažuriraju svoje boje kako bi dobili jednu od $\{1, ., , , \Delta+1\}$ boja. U tu svrhu, svi procesi izvršavaju uzastopne runde $\Delta+2$ do m, s obzirom na to svaka runda odgovara ažuriranju jedne od m inicijalnih boja, koja nije među prvih +1. Tijekom runde r, $\Delta+2 \leq r \leq m$, svaki postupak čija je početna boja r traži novu boju u $\{1, ., , , \Delta + 1\}$ koja nije boja njegovih susjeda i usvaja je kao svoju novu boju (*linija* 6–8). Zatim svaki proces zamijeni boje svojih susjeda koje su mu poznate u varijabli $color_i[j]$ gdje je j jedan od susjeda (*linije* 10–14) prije nego što se prebaci na sljedeću rundu. Dakle, runda izvršava sljedeću funckiju: kad se završi runda r, procesi čije su početne boje bile u $\{1, 2, .., r\}$ imaju boju u skupu $\{1, 2, .., \Delta + 1\}$ i različite boje ako su susjedi.

Vremenska složenost. Algoritam se izvršava u $m-$ rundi: početna runda plus $m - (\Delta + 1)$ rundi. Svaka poruka sadrži oznaku, boju i eventualno broj runde koji je ujedno i boja. Kako su početne boje u $\{1, 2, .., m\}$, složenost bita poruke je $\mathcal{O}(log_2 m)$. Konačno, tijekom svake runde, na svaki se kanal šalju dvije poruke. Složenost poruke je prema tome $2e(m - \Delta)$, gdje e označava broj kanala. Lako je vidjeti da, što je bolje početno bojanje procesa (tj. to je manja vrijednost m), to je algoritam učinkovitiji.

```
(1)  for each j ∈ neighbors_i do send INIT(color_i[i]) to p_j end for;
(2)  for each j ∈ neighbors_i
(3)        do wait (INIT(col_j) received from p_j); color_i[j] ← col_j
(4)  end for;
(5)  for r_i from (Δ + 2) to m do
       begin asynchronous round
(6)    if (color_i[i] = r_i)
(7)      then c ← smallest color in {1, ..., Δ + 1} such that ∀j ∈ neighbors_i : color_i[j] ≠ c;
(8)           color_i[i] ← c
(9)    end if;
(10)   for each j ∈ neighbors_i do send COLOR(r_i, color_i[i]) to p_j end for;
(11)   for each j ∈ neighbors_i do
(12)     wait (COLOR(r, col_j) with r = r_i received from p_j);
(13)     color_i[j] ← col_j
(14)   end for
       end asynchronous round
(15) end for.
```

Slika 2: Pseudo kod skevencijalnog bojanja grafa

Teorem 1. *Neka je $m \geq \Delta + 2$. Algoritam za distribuirano bojanje grafova zakonito boja graf procesa u $\Delta + 1$ boja.*

Dokaz. Primijetimo najprije da procesi kojima pripada početna boja $\{1, 2, .., \Delta + 1\}$ nikada ne mijenjaju svoju boju. Pretpostavimo da je, do runde r, procesi čije su početne boje bile u skupu $\{1, 2, .., r\}$ imaju nove boje u skup $\{1, ., , , \Delta + 1\}$ te imaju različite boje sa svim svojim susjedima. Zahvaljujući početnom m bojanju, ovo je u početku točno (tj. za izmišljenu rundu $r = \Delta + 1$). Pretpostavimo da je prethodna tvrdnja istinita do nekog kruga $r \geq \Delta + 1$. Iz algoritma proizlazi da se ažuriraju tijekom runde $r + 1$ samo oni procesi čija je trenutna boja $r + 1$. Štoviše, svaki proces čija je trenutna boja $r + 1$ ažurira se (*linija* 7) bojom koja pripada skupu $\{1, ., , , \Delta + 1\}$ i nije boja njegovih susjeda (vidjeli smo u uvodu da takva boja sigurno postoji). Slijedom toga, na kraju runde r + 1, procesi čije su početne boje bile u skupu $\{1, 2, ..., r + 1\}$ imaju nove boje u skupu $\{1, ..., \Delta + 1\}$ i nijedna od njih nema istu boju sa svojim susjedima. Iz toga slijedi da svaki proces ima konačnu boju u skup $\{1, ..., \Delta + 1\}$ i nijedna dva susjedna procesa nemaju istu boju.

Napomena o ponašanju komunikacijskih kanala. Jedina pretpostavka o komunikacijskim kanalima je da su pouzdani. Nema drugih pretpostavki za ponašanje kanala, stoga kanali implicitno mogu biti i ne-FIFO kanali. Razmotrimo dva susjedna procesa koji izvršavaju rundu r kao što je prikazano na slici ispod. Svaki od njih šalje svoju poruku *COLOR*(r, -) svojim susjedima (*linija* 10), i čeka poruku *COLOR()* od svakog od njih, izvodeći još uvijek rundu r (*linija* 12).

Na slici je p_j primio poruku iz runde r od p_i, nastavio prema sljedećoj rundi i poslao je poruku *COLOR*(r + 1, -) svojem susjedu p_i dok proces p_i još čeka za poruku iz runde r od p_j. Štoviše, kako kanal nije FIFO, slika ispod prikazuje slučaj u kojem poruka *COLOR*(r + 1, -) koju je p_j

poslao p_i stiže prije poruka *COLOR* (r, -) koju je prethodno poslala. Kao što je naznačeno u *liniji* 12, algoritam prisiljava p_i da pričeka poruku *COLOR* (r, -) kako bi prekinuo svoju rundu r. Kako u svakoj rundi, svaki postupak šalje poruku svakom od svojih susjeda, analiza obrasca razmjene poruka pokazuje da je sljedeći odnos:

$$\forall i, \forall j\text{: (ako su } p_i \text{ i } p_j \text{ susjedi)} \rightarrow (0 \leq |r_i - r_j| \leq 1)$$

3.1 Implementacija

Za komunikaciju između procesa koristimo implementaciju servera, topologije te sve komunikacijske kanale iz [**1**] te smo mijenjali samo implementaciju određenih procesa koji smo koristili za bojanje grafa.

Sa programskih argumenata učitavamo, ime servera na koji se spajaju procesi njegov indeks, te ukupan broj pokrenutih procesa te proces povezujemo sa ostalima pomoću klase Linker implementirane u [**1**] Nakon toga dodajemo sve susjede u LinkedList neighbors, te pronalazimo najveći stupanj vrha u grafu, pomoću funckije findMaxDeegre();

```
try {
    String baseName = args[0];
    myId = Integer.parseInt(args[1]);
    numProc = Integer.parseInt(args[2]);
    System.out.println(numProc);
    comm = new Linker(baseName, myId, numProc);
    neighbors.addAll(comm.neighbors);
    degree = neighbors.size();
    findMaxDeegre();
    System.out.println(degree);
    initColour();
    System.out.println(colour);
    bojanje();
    System.out.println(colour);
```

Slika 3: Implementacija procesa za Distribuirani algoritam bojanja grafa

Na slici ispod prikazujemo poziv nultog procesa u razvojnoj okolini IntelijCommunity 2020.1

Slika 4: Poziv nultog procesa sa ukupnu 8 procesa

Na slici ispod prikazujemo strukture podataka korištene za implementaciju Procesa

```
private static final LinkedList<Integer> neighbors = new LinkedList<~>();
private static final HashMap<Integer, Integer> colour = new HashMap<>();
private static int degree = 0;
private static int myId;
private static int numProc;
private static Linker comm = null;
private static Msg m;
private static HashMap<Integer, Boolean> selected = new HashMap<~>();
```

Slika 5: Strukture podataka korištene za implementaciju procesa

```
public static void findMaxDeegre() throws IOException {
    for (int i = 0; i < numProc; i++) {
        BufferedReader dIn = new BufferedReader(new FileReader( s: "topology" + i));
        StringTokenizer st = new StringTokenizer(dIn.readLine());
        if (st.countTokens() > degree) degree = st.countTokens();

    }


}
```

Slika 6: Pronalazak stupnja grafa iz tekstualnih datoteka topology.txt

Funkcija findMaxDeegre() koristi tekstualne datoteke topologyID.txt da pronađe broj susjeda čvora koji ih ima najviše.

Nakon pronalaska najvećeg stupnja grafa prelazimo na inicijalizaciju boja gdje smo odlučili koristiti inicijalno bojanje tako da svakom procesu pridružimo za boju njegov indeks. Funkcija *initColour*() postavlja vrijednost hashMape *colour* za index procesa *myId* na *myId*, te šalje svim susjedima svoju trenutnu boju, te od njih očekuje odgovor, ova funkcija odgovara linijama iz algoritma (1-4) Na slici prije prikazana funkcija initColour

```
private static void initColour() throws IOException, InterruptedException {

    colour.put(myId, myId);

    for (int i = 0; i < numProc; i++)
        if (i != myId && neighbors.contains(i))
            comm.sendMsg(i, tag: "init_colour", msg: myId + " " + myId);
    for (int i = 0; i < numProc; i++)
        if (i != myId && neighbors.contains(i)) {
            m = comm.receiveMsg(i);

            handleMessage(m);
        }


}
```

Slika 7: Pronalazak stupnja grafa iz tekstualnih datoteka topology.txt

Nakon funkcije *InitColor* proces ulazi u funkciju *bojanje*() koja odrađuje runde, te implementaciju redova (5-15) distribuiranog algoritma. Proces ulazi u if granu ukoliko je boja procesa jednaka broju runde, zatim provjeravamo je li jedna od $\{0, 2, .., \Delta\}$ (u implementaciji smo koristili brojanje od 0 da bi omogućili bojanje) slobodna, te u funkciji *nemaliSus*(i) provjeravamo da niti jedan od susjeda nije njome obojan.

Zatim šaljemo svim ostalim procesima novu postavljenu boju te čekamo primitak promjene boja od svih susjeda. Nakon primitka poruku propuštamo u funkciju *handelMassage*(msg) gdje ukoliko je primljena poruka s tagom bojanje, parsiramo poruku te postavljamo vrijednost hashMape *color* od ključa j na boju koju je poslao proces p_j.

```
private static Boolean nemaliSus(int i) {

    if (i == myId) return false;
    for (Integer neighbor : neighbors) {
        if (neighbor == myId) continue;
        if (colour.get(neighbor) == i) return false;
    }
    return true;
}


public static void handleMessage(Msg msg) {

    if (msg.getTag().equals("init_colour")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        int boja = Integer.parseInt(st.nextToken());
        colour.put(srcId, boja);
    } else if (msg.getTag().equals("bojanje")) {

        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        int boja = Integer.parseInt(st.nextToken());

          if(srcId<myId&&colour.get(myId)==boja)
        colour.put(srcId, boja);
    }
}
```

Slika 8: Funkcija handleMessage i nemaliSus

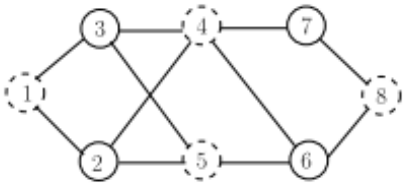
4 Maksimalan nezavisni skup

Sada ćemo nadograditi naš algoritam tako da u njemu koristimo maksimalan nezavisni skup. Najprije ćemo definirati što je to, a potom i kako smo ga koristili u našem algoritmu.

Definicija 2. Nezavisni skup je podskup skupa vrhova nekog grafa, takvih da niti jedna dva vrha u tom skupu nisu susjedna. ◁

Definicija 3. Nezavisni skup M je maksimalan ako nijedan od njegovih nadskupova M' (tj. $M \subset M'$ i $M \neq M'$) nije nezavisan. ◁

Sada kada imamo potrebne definicije, možemo vidjeti kako to izgleda na primjeru.
Promatrat ćemo graf na sljedećoj slici.



Slika 9: Primjer grafa

Primjer maksimalnog nezavisnog skupa je sljedeći skup vrhova: $\{1, 4, 5, 8\}$. Kao što možemo vidjeti iz slike, ne postoji samo jedan takav skup nego ih može biti i više. U konkretnom primjeru, to bi bili skup $\{1, 5, 7\}$ i $\{2, 3, 6, 7\}$.
Kada bi razmišljali kako izračunati maksimalan nezavisan skup na najjednostavniji način, dolazimo do pohlepnog algoritma. Odabiremo vrh, dodajemo ga u nezavisni skup ukoliko unutar njega nisu njegovi susjedi te isto radimo dok imamo još vrhova. Na taj način bi dobili jedan od maksimalanh

nezavisnih skupova. Međutim, takvo izračunavanje maksimalnog nezavisnog skupa pripada klasi problema koji se mogu riješiti algoritmima polinomne vremenske složenosti.

Osim maksimalnog nezavisnog skupa možemo definirati i maksimum nezavisnog skupa.

Definicija 4. Maksimum nezavisnog skupa je onaj maksimalan nezavisni skup s maksimalnom kardinalnošću. ◁

Ako ponovno pogledamo graf sa slike, možemo vidjeti da imamo dva takva skupa i to: $\{1, 4, 5, 8\}$ i $\{2, 3, 6, 7\}$. Ukoliko gledamo složenost vremena izračunavanja takvih skupova, izračunavanje maksimuma nezavisnog jednostavan je problem dok je izračunavanje maksimalnog nazavisnog skupa NP-težak problem.

4.1 Maksimalan nezavisan skup pomoću m - bojanja

U ovom odjeljku prikazat ćemo asinkroni algoritam koji izračunava maksimalan nezavisni skup. Pseudo kod algoritma prikazan je na sljedećoj slici.

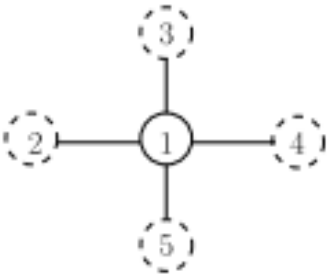
```
(1)  for r_i from 1 to m do
       begin asynchronous round
(2)    if (color_i = r_i) then
(3)      if (⋀_{j∈neighbors_i} (¬selected_i[j])) then selected_i[i] ← true end if;
(4)    end if;
(5)    for each j ∈ neighbors_i do send SELECTED(r_i, selected_i[i]) to p_j end for;
(6)    for each j ∈ neighbors_i do
(7)          wait (SELECTED(r, selected_j) with r = r_i received from p_j);
(8)          selected_i[j] ← selected_j
(9)    end for
       end asynchronous round
(10) end for.
```

Slika 10: Algoritam za određivanje maksimalnog nezavisnog skupa

Svaki proces p_i upravlja svojim lokalnim nizom $selected_i[j], j \in neighbors_i \cup i$. Takvi nizovi prikazuju pripadnost maksimalnom nezavisnom skupu. Na početku je svaki element svakog niza postavljen na $false$, dok na kraju znamo da proces p_i pripada maksimalnom nezavisnom skupu ako i samo ako je $selected_i[i] = true$.
U ovom algoritmu pretpostavljamo da već postoji neko bojanje grafa od kojeg počinjemo. U našem slučaju, koristi se prethodno opisani algoritam. Naš algoritam temelji se na jednostavnom promatranju i pohlepnoj strategiji.

- Jednostavno promatranje Procesi koji imaju istu poju definiraju nezavisan skup, no on nije nužno maksimalan.
- Pohlepna strategija Pošto prethodni skup nije nužno maksimalan, algoritam započinje s početnik nezavisnim skupom koji je definiran nekom bojom i izvšava određen broj krugova (svaki od krugova odgovara nekoj boji). U svakom krugu teži tome da doda što više mogućih procesa nezavisnom skupu. Uvjet da proces p_i upadne u nezavisni skup je da nijedan njegov susjed nije već u tom skupu.

Opisani algoritam simulira sinkroni algoritam. Boja procesa p_i sprema se u lokalnoj varijabli označenoj s $color_i$. Poruke imaju neki cijeli broj tj. boju. Procesi izvršavaju m asinkronih krugova što znači da izvršavaju jedan krug po boji. Kada izvrši krug za boju r, provjerava je li boja procesa r te pripada li ijedan njegov susjed nezavisnom skupu kojeg gradimo. Ukoliko je boja jednaka r te nema njegovih susjeda u skupu, proces p_i dodaje se skupu. Nakon što algoritam krene na sljedeći krug, procesi razmjenjuju poruke o članstvu u maksimalnom nezavisnom skupu. Nakon što procesi ažuriraju svoje lokalne varijable $selected_i[j]$, kreće novi krug. U redosljedu izvođenja nije bitno da idu redosljedom od 1 do m, ali je bitno da se krugovi izvršavaju nekim unaprijed određenim redosljedom.
Veličina maksimalnog nezavisnog skupa koji dobijemo ovisi o redosljedu u kojem algoritam posjeti određenu boju. Za primjer gledamo graf na sljedećoj slici.



Slika 11: Primjer grafa

Vidimo da su maksimalan nezavisni skupovi $\{1\}$ i $\{2, 3, 4, 5\}$. Proces p_1 obojan je bojom a dok su drugi procesi obojani bojom b. Ukoliko je $a = 1$ i $b = 2$, dobit ćemo da je maksimalan nezavisni skup jednak $\{1\}$. U suprotnom slučaju, dobijemo skup $\{2, 3, 4, 5\}$.

4.1.1 Implementacija

Sada ćemo prikazati kod pomoću kojeg smo implementirali navedeni algoritam. Pošto smo koristili već navedeni implementirani algoritam za bojanje grafa, samo smo nadopunjavali postojeći algoritam.
Najprije smo nadopunili metodu handleMessage(Msg msg) tako da bi mogli primati poruke o tome koji su procesi u nezavisnom skupu a koji ne.

```
public static void handleMessage(Msg msg) {
    if (msg.getTag().equals("init_colour")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        int boja = Integer.parseInt(st.nextToken());
        colour.put(srcId, boja);
    } else if (msg.getTag().equals("bojanje")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        int boja = Integer.parseInt(st.nextToken());
        colour.put(srcId, boja);
    } else if (msg.getTag().equals("init_selection")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        Boolean boja = Boolean.parseBoolean(st.nextToken());
        selected.put(srcId, boja);
    } else if (msg.getTag().equals("maxindset")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        Boolean boja = Boolean.parseBoolean(st.nextToken());
        selected.put(srcId, boja);
    }
}
```

Slika 12: Funkcija handleMessage()

Također, implementiramo fukciju init_selection(). Kao što smo naveli, na početku je svaki element *selected* liste postavljen na $false$. Ova funkcija nam služi za to napraviti.

Sada dolazimo do fukcije pomoću koje gradimo maksimalan nezavisni skup, maxindset().

Za svaki pojedini proces r_i provjeravamo najprije je li njegova boja je jednaka r_i. Ukoliko je, provjeravamo za svakog susjeda $j \in neighbors_i$ je li $selected_i[j] == false$. Ako je za svakog postavljeno

```
public static void init_selection() {
    for(Integer neighbor: neighbors){
        selected.put(neighbor,false);
    }
    selected.put(myId,false);

}
```

Slika 13: Funkcija init_selection()

```
public static void maxindset() throws InterruptedException {
    for (int ri = 1; ri < numProc; ri++) {
        System.out.println("--------------------"+" runda "+ ri+"--------------------");
        if(colour.get(myId) == ri) {
            Boolean selectedi = true;
            for(int i = 0; i < numProc; i++) {
                if (neighbors.contains(i) && selected.get(i)) {
                    selectedi = false;
                }
            }
            if(selectedi) {
                selected.put(myId, true);
            }
        }

        for (int i = 0; i < numProc; i++) {
            if (i != myId && neighbors.contains(i))
                comm.sendMsg(i, tag: "maxindset", msg: myId + " " + selected.get(myId));
        }

        for (int i = 0; i < numProc; i++) {
            if (i != myId && neighbors.contains(i)) {
                m = null;
                while (m == null) {
                    m = comm.receiveMsg(i);
                    Thread.sleep( l: 1000);
                }
                handleMessage(m);
            }
        }
    }
}
```

Slika 14: Funkcija maxindset()

$false$, za naš proces postavljamo selected na true. Nakon toga šaljemo svim susjedima poruku o tome kako nam je postavljen *selected* i također od njih primimo istu poruku. Nakon što prođemo sve procese, dobijemo jedan od maksimalnih nezavisnih skupova.

4.2 Algoritam Luby-a za maksimalan nezavisni skup

U ovom se dijelu nalazi algoritam M. Luby-a (1987) koji gradi maksimalan nezavisni skup. Ovaj algoritam koristi slučajnu funkciju označenu random () koja proizvodi slučajnu vrijednost svaki put kada je pozvana (korisnost toga pristupa motivirana je u daljnjem tekstu). Radi lakšeg izlaganja, ovaj algoritam izražen je u sinkronom modelu. Sjetimo se da glavno svojstvo sinkronog modela leži u činjenici da je poruka poslana u rundi primljena od strane odredišnog procesa u istoj toj rundi. (Ovaj algoritam se lako proširi tako da djeluje u asinkronom modelu.)

Svaki proces p_i upravlja sljedećim lokalnim varijablama:

- Lokalna varijabla $state_i$, čija je početna vrijednost proizvoljna, ažurira se samo jednom. Konačna vrijednost pokazuje je li p_i pripada ili ne pripada maksimalnom nezavisnom skupu koji se izračunava. Kada je ažurirao $state_i$ na konačnu vrijednost, proces p_i izvršava izraz return() koji zaustavlja njegovo sudjelovanje u algoritmu. Primijetimo da se procesi ne moraju nužno završiti tijekom iste runde.

- Lokalna varijabla com_with_i, koja se inicijalizira u $neighbors_i$, skup je koji sadrži procese s kojima će p_i nastaviti komunikaciju tijekom sljedeće runde.

- Svaka lokalna varijabla $random_i[j]$, gdje je $j \in neighbors_i \cup \{i\}$, predstavlja znanje p_i o posljednjem slučajnom broju koji koristi p_j.

Kao što je naznačeno, procesi izvršavaju niz sinkronih rundi. Šifra algoritma sastoji se u opisu tri uzastopne runde, posebice runde $r, r+1$ i $r+2$, gdje je $r = 1, 4, 7, 10, ...$ Poruke razmjenjene tijekom ove tri uzastopne runde prikazane su na slici. Ponašanje sinkronog algoritma tijekom ova tri uzastopna kruga je sljedeće:

- Runda r: *linije* 2–6. Svaki proces p_i prvo poziva funkciju $random()$ radi dobivanja slučajnog broja (*linija* 2) kojeg šalje svim svojim susjedima s kojima i dalje komunicira (*linija* 3). Zatim pohranjuje sve slučajne brojeve koje je primio, a svaki dolazi iz procesa u *com_with*.

- Runda $r + 1$: *linije* 7–11. Zatim p_i šalje poruku SELECTED (yes) svojim susjedima u *com_with* ako je njezin slučajni broj manji od njihovog (*linija* 8). U ovom slučaju, napreduje do lokalnog stanja *in* i zaustavlja se (*linija* 9). Inače, njegov slučajni broj nije najmanji. U ovom slučaju, p_i prvo šalje poruku SELECTED (no) svojim susjedima u *com_with* (*linija* 10), a zatim čeka poruku svakog od tih susjeda (*linija* 11).

- Runda $r + 2$: *linije* 12–26. Konačno, ako p_i nije ušao u maksimalan neovisni skup koji je u izradi, provjerava se je li jedan od njegovih susjeda u *com_with* dodan u ovaj skup (*linija* 12).

- Ako je jedan od njegovih susjeda dodan u nezavisni skup, ubuduće ga ne možete dodati. Slijedom toga, šalje poruku ELIMINATED (yes) svojim susjedima u *com_with* kako bi ih obavijestio da se više ne natječe za ulazak u nezavisni skup (*linija* 13). U tom slučaju ona također ulazi u lokalno stanje i vraća ga (*linija* 16). - Ako nitko od njegovih susjeda u *com_with* nije dodan u nezavisni skup, p_i im šalje poruku ELIMINATED (no) kako bi ih obavijestio da se još uvijek natječe za ulazak u nezavisni skup (*linija* 17). Zatim čeka poruku ELIMINATED (-) svakog od njih (*linija* 18) i suzbija iz skupa *com_with* svoje susjede koji se više ne natječu (to su procesi koji su mu poslali poruku ELIMINATED (yes), *linije* 21– 23).

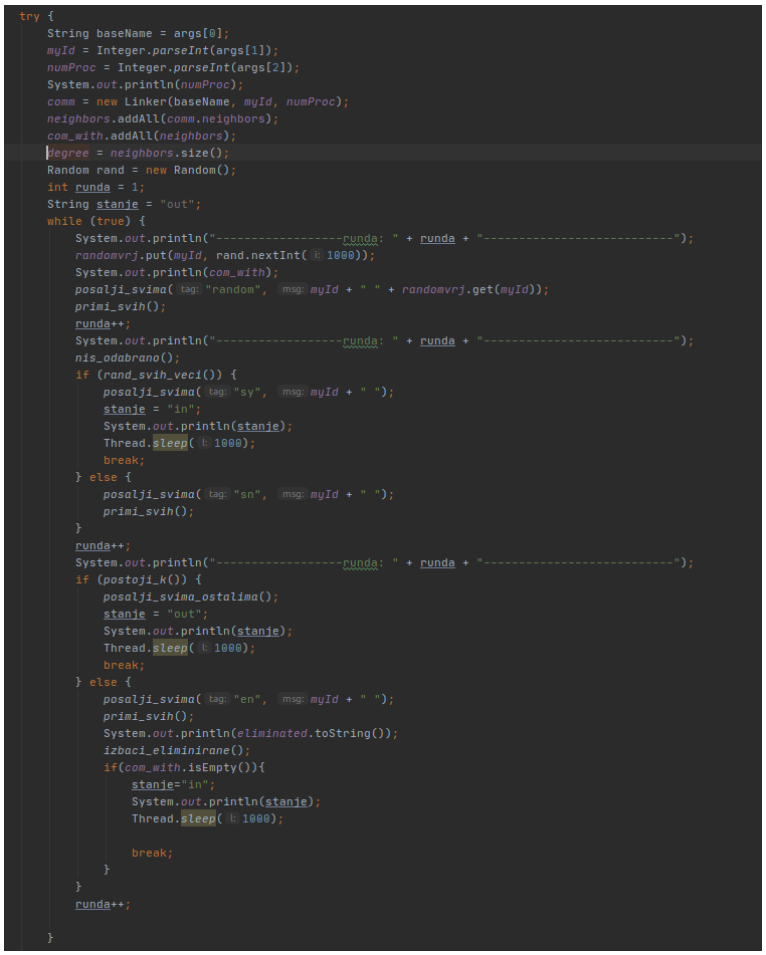
Na kraju, p_i provjerava je li $com\_with_i = \oslash$.

Ako je to slučaj, ulazi u nezavisni skup i vraća se (*linija* 24). U protivnom, nastavlja se u sljedećoj rundi.

Algoritam izračunava nezavisni skup jer kad se neki proces doda skupu, svi njegovi susjedi prestaju se natjecati u skupu (*linija* 12-15). Taj je skup maksimalan, jer kada proces uđe u nezavisni skup, kandidati se eliminiraju samo njegovi susjedi.

4.2.1 Implementacija

Na slici ispod nalazi se cijeli proces za Luby-ev algoritam za Maksimalan nezavisan skup, proces se u po 3 sinkrone runde. Zadamo procesu njegovu random vrijednost koje u runda r šaljemo svima procesima koji se nalaza u LinkedListi *com_with* koja je inicijalizirana na sve susjede, te od svih njih primi poruku o njihovoj vrijednosti random broja.

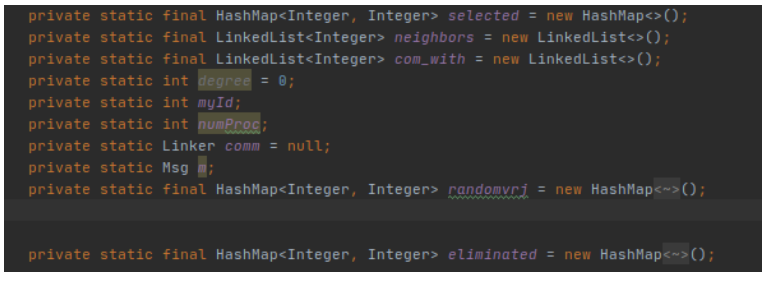


```
try {
    String baseName = args[0];
    myId = Integer.parseInt(args[1]);
    numProc = Integer.parseInt(args[2]);
    System.out.println(numProc);
    comm = new Linker(baseName, myId, numProc);
    neighbors.addAll(comm.neighbors);
    com_with.addAll(neighbors);
    degree = neighbors.size();
    Random rand = new Random();
    int runda = 1;
    String stanje = "out";
    while (true) {
        System.out.println("------------------runda: " + runda + "--------------------------");
        randomvrj.put(myId, rand.nextInt( i: 1000));
        System.out.println(com_with);
        posalji_svima( tag: "random",  msg: myId + " " + randomvrj.get(myId));
        primi_svih();
        runda++;
        System.out.println("------------------runda: " + runda + "--------------------------");
        nis_odabrano();
        if (rand_svih_veci()) {
            posalji_svima( tag: "sy",  msg: myId + " ");
            stanje = "in";
            System.out.println(stanje);
            Thread.sleep( l: 1000);
            break;
        } else {
            posalji_svima( tag: "sn",  msg: myId + " ");
            primi_svih();
        }
        runda++;
        System.out.println("------------------runda: " + runda + "--------------------------");
        if (postoji_k()) {
            posalji_svima_ostalima();
            stanje = "out";
            System.out.println(stanje);
            Thread.sleep( l: 1000);
            break;
        } else {
            posalji_svima( tag: "en",  msg: myId + " ");
            primi_svih();
            System.out.println(eliminated.toString());
            izbaci_eliminirane();
            if(com_with.isEmpty()){
                stanje="in";
                System.out.println(stanje);
                Thread.sleep( l: 1000);

                break;
            }
        }
        runda++;

    }
```

Slika 15: Proces Luby-evog algoritma



```
private static final HashMap<Integer, Integer> selected = new HashMap<>();
private static final LinkedList<Integer> neighbors = new LinkedList<>();
private static final LinkedList<Integer> com_with = new LinkedList<>();
private static int degree = 0;
private static int myId;
private static int numProc;
private static Linker comm = null;
private static Msg m;
private static final HashMap<Integer, Integer> randomvrj = new HashMap<~>();


private static final HashMap<Integer, Integer> eliminated = new HashMap<~>();
```

Slika 16: Strukture podataka za implementaciju Luby-evog algoritma

Koristimo HashMapove *selected*, *com_with*, *eliminated* da bi zapamtili procese odabrane, te susjede koji su eliminirani, U rundi $r + 1$ gledamo jeli random vrijednost procesa manja od svih procesa u *com_with*, to radimo funkcijom *rand_svih_veci()*, ukoliko je šaljemo svim susjedima da je taj proces u ovoj rundi te da će biti u maksimalnom nezavisnom skupu, ispisujemo „in" što signalizira nalaženje u maksimalnom nezavisnom skupu, ukoliko nije šaljemo svima da nije u tome skupu i nastavljamo dalje.

U rundi $r + 2$ gledamo postoji li među procesima u *com_with* ukoliko postoji, ispisujemo „out" što signalizira da ovaj proces nije u MNS-u u maksimalnom nezavisnom skupu te gasimo ovaj proces. Ukoliko ne postoji odabrani proces, proces nastavlja dalje sa traženjem odgovora na to pitanje.

```
private static void izbaci_eliminirane() {
    for (Integer j : neighbors)
        if (com_with.contains(j) && eliminated.get(j) == 1) com_
}

private static void posalji_svima_ostalima() {
    for (Integer j : com_with)
        if (j != myId && selected.get(j) == 0) {

            comm.sendMsg(j, tag: "ey", msg: myId + " ");
        }
}

private static boolean postoji_k() {
    for (Integer j : com_with) if (selected.get(j) == 1) return
    return false;
}


private static boolean rand_svih_veci() {
    for (Integer j : com_with) {

        if (j != myId && randomvrj.get(j) <= randomvrj.get(myId)
    }
    return true;

}

private static void nis_odabrano() {
    for (Integer j : com_with) {
        if (j != myId) {
            selected.put(j, v: 0);
            eliminated.put(j, v: 0);
        }
    }
}
```

Slika 17: Pomočne funkcije za implementaciju Luby-evog algoritma

```
static void posalji_svima(String tag, String msg) {
    for (Integer j : com_with)
        if (j != myId) {
            try {
                comm.sendMsg(j, tag, msg);
                Thread.sleep( l: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}

static void primi_svih() {
    for (Integer j : com_with)
        if (j != myId) {
            m = null;
            while (m == null) {
                try {
                    m = comm.receiveMsg(j);
                    Thread.sleep( l: 1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

            }
            handleMessage(m);
        }

}
```

Slika 18: Pomočne funkcije za implementaciju Luby-evog algoritma

```
public static void handleMessage(Msg msg) {

    if (msg.getTag().equals("random")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        int boja = Integer.parseInt(st.nextToken());
        randomvrj.put(srcId, boja);
    } else if (msg.getTag().equals("sy")) {

        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        selected.put(srcId,  v: 1);
    } else if (msg.getTag().equals("ey")) {
        StringTokenizer st = new StringTokenizer(msg.getMessage());
        int srcId = Integer.parseInt(st.nextToken());
        eliminated.put(srcId,  v: 1);
    }

}
```

Slika 19: Pomočne funkcije za implementaciju Luby-evog algoritma

Literatura

[1] Robert Manger-Distribuirani Procesi, Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet Matematički odsjek; Ožujak 2017