UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Ana Pavlišič

Porazdeljeni algoritem za problem maksimalnega pokritja grafa s trikotniki

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2009

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisana Ana Pavlišič,

z vpisno številko 63000335,

sem avtorica diplomskega dela z naslovom:

Porazdeljeni algoritem za problem maksimalnega pokritja grafa s trikotniki

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom
  doc. dr. Boštjana Slivnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 16.09.2009 Podpis avtorice:

Zahvala

Na prvem mestu se iskreno zahvaljujem mentorju doc. dr. Boštjanu Slivniku za vso pomoč in usmerjanje pri izdelavi tega dela.

Andražu in Blažu gre zahvala za moralno podporo in razjasnitev nekaterih nejasnosti. Za lekturo se zahvaljujem mami, Andreju pa za pomoč pri prevodu povzetka v angleščino. Zahvala gre tudi dr. Urošu Čibeju, ki mi je nevede prijazno posodil literaturo na temo porazdeljenih algoritmov.

Na koncu se zahvaljujem mami in očetu za vso podporo med celotnim študijem.

Kazalo

Povzetek **2**

Abstract **3**

1 Uvod **4**

1.1 Opis problema . . . 4
1.2 Obstoječe rešitve . . . 6
1.3 Cilji . . . 6

2 Porazdeljeni algoritem **8**

2.1 Opis porazdeljenega modela . . . 8
2.2 Opis algoritma . . . 9
2.2.1 Uvod . . . 9
2.2.2 Izmenjava podatkov o sosedih . . . 10
2.2.3 Naključno generiranje trikotnikov . . . 12
2.2.4 Optimizacija prvotne rešitve . . . 20
2.2.5 O prehodu med fazami . . . 40
2.3 Implementacija . . . 40
2.3.1 Programska oprema . . . 40
2.3.2 Parametri . . . 42
2.3.3 Rezultat . . . 43
2.3.4 Problemi . . . 43
2.4 Meritve . . . 44
2.4.1 Graf 1 . . . 45
2.4.2 Graf 2 . . . 46
2.4.3 Graf 3 . . . 48
2.4.4 Graf 4 . . . 49

3 Zaključek **50**

A Implementacija v Javi **1**
A.1 Funkcije 1

Seznam slik **13**
Seznam tabel **14**

Seznam algoritmov **15**

Literatura **16**

Povzetek

V uvodu definiramo optimizacijski problem maksimalnega pokritja grafa s trikotniki in ga umestimo v enega od razredov problemov v teoriji grafov. Na kratko povzamemo obstoječe rešitve in predlagamo porazdeljeni algoritem kot možnost za reševanje tega problema ter zaključimo uvodni del s predstavitvijo ciljev tega dela.

Nato opišemo predpostavljeni porazdeljeni model, za katerega naj bi algoritem deloval in predstavimo omenjeni porazdeljeni algoritem. Sledi opis implementacije s predstavitvijo simulatorja porazdeljenih sistemov DAJ, ki je bil uporabljen pri razvoju algoritma, ter opis vhodnih parametrov, s katerimi poženemo nastali program. Nato izpostavimo nekatere najbolj pogoste probleme, s katerimi smo se srečevali tekom sestave algoritma.

Na koncu predstavimo rezultate opravljenih meritev in podamo oceno uspešnosti algoritma. Zaključimo s predlogi za nadaljnji razvoj in izboljšave ter nekaterimi sklepnimi ugotovitvami.

Ključne besede:

maksimalno pokritje grafa s trikotniki, NP-težek problem, porazdeljeni algoritem, optimizacijski problem

Abstract

In the introduction we define the maximum triangle packing optimization problem and situate it within a particular class of problems in the graph theory. We summarize presently existing solutions and propose a distributed algorithm as one possible solution to the problem. We conclude the introductory part with the presentation of objectives of this work.

We continue with pressuposed distributed model for which the algorithm should be functional and present the said distributed algorithm. Then follows the description of the implementation accompanied with the presentation of a toolkit for the simulation of distributed algorithms in Java that was used in the development of the algorithm and the description of the input parameters on which the developed program is being applied. Then we expose some of the most common problems, that we have faced during the process of algorithm development.

Finally we present the results of the conducted measurements and evaluate the successfulness of the algorithm. We conclude with suggestions for future development, possible enhancements and with some conclusive discoveries.

Key words:

maximal triangle packing, NP-hard problem, distributed algorithm, optimization problem

Poglavje 1

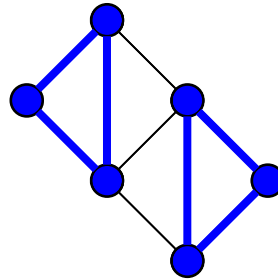
Uvod

1.1 Opis problema

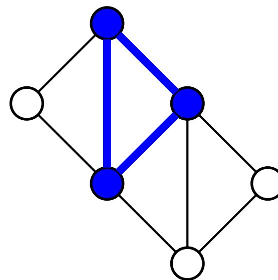
Optimizacijski problem maksimalnega pokritja grafa s trikotniki (angl. Maximum triangle packing, GT11 [1]) je definiran z naslednjimi podatki:

> NALOGA: Graf $G = (V, E)$, kjer je V množica vozlišč, E pa množica neusmerjenih povezav med njimi.
>
> DOPUSTNA REŠITEV: Pokritje grafa G s trikotniki - množica T disjunktnih podmnožic $T_1, T_2, ..., T_k$, kjer vsaka podmnožica $T_i = \{u_i, v_i, w_i\}$, $1 \leq i \leq k$ vsebuje natanko 3 različna vozlišča iz V, vse tri povezave $\{u_i, v_i\}$,$\{u_i, w_i\}$ in $\{v_i, w_i\}$ pa pripadajo množici E.
>
> KRITERIJ: Moč množice T - število trikotnikov, ki predstavljajo pokritje grafa G.
>
> CILJ: Maksimizacija.

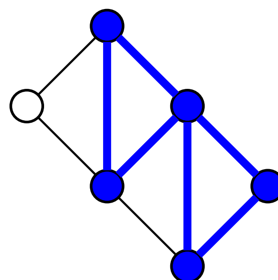
Za ilustracijo si poglejmo tri rešitve opisanega problema na primeru nekega grafa s 6 vozlišči. Na sliki 1.1 je prikazana optimalna rešitev, slika 1.2 prikazuje dopustno, vendar ne optimalno rešitev. Neveljavna rešitev, kjer je neko vozlišče del dveh trikotnikov (podmnožici množice T nista disjunktni) pa je prikazana na sliki 1.3.



Slika 1.1: Optimalna rešitev problema



Slika 1.2: Dopustna rešitev problema, ki ni optimalna



Slika 1.3: Neveljavna rešitev

Problem v teoriji grafov spada v širši razred optimizacijskih problemov pokritja grafa. Znano je, da tako splošni problem pokritja kot pokritje s trikotniki spadata med NP-težke probleme, za katere velja, da ne poznamo polinomskega algoritma za eksaktno rešitev problema. Prav tako pa ne poznamo dokaza, da takšnega algoritma ni.

Kot ena izmed zanimivih možnosti reševanja tega problema se nam je zdela uporaba porazdeljenega algoritma, kjer obstaja bijektivna preslikava med računskimi vozlišči porazdeljenega sistema in grafom. Ker takega algoritma nismo zasledili v nobenem viru, smo sestavili asinhron porazdeljeni algoritem za reševanje tega optimizacijskega problema in ga bomo v tem delu podrobno predstavili. Naprej pa omenimo še nekatere obstoječe rešitve za ta problem, ki vse predpostavljajo zaporedno izvajanje na enem samem procesorju.

1.2 Obstoječe rešitve

V članku iz leta 1989 sta Hurkens in Schijver predstavila algoritem, ki sicer rešuje bolj splošen problem pokritja grafa [3, 4]. Njuna aproksimacijska rešitev za problem maksimalnega pokritja grafa s trikotniki je do sedaj najboljša znana in zagotavlja aproksimacijski radij $\frac{3}{2}$.

Zasledimo pa še boljše aproksimacijske rešitve, ki se osredotočajo le na omejen razred teh problemov. Eden teh je algoritem, ki zagotavlja aproksimacijski radij malo manjši od $\frac{6}{5}$, in se omeji na grafe z maksimalno stopnjo vozlišča 4 [3].

Omenimo še, da v nekaterih virih obravnavajo nekoliko drugačne verzije problema maksimalnega pokritja grafa s trikotniki. Ena različica je ta, da so povezave v grafu obtežene, kriterij pa potem ni moč množice T, pač pa vsota vseh uteži, katerih povezave spadajo v pokritje grafa [2]. Druga različica pa elementov množice T ne definira kot množico vozlišč, pač pa kot množico povezav, ki sestavljajo trikotnik [3]. V tem primeru je lahko neko vozlišče del več trikotnikov hkrati.

1.3 Cilji

Namen tega dela je sestava porazdeljenega algoritma za rešitev optimizacijskega problema maksimalnega pokritja grafa s trikotniki. Najprej bi radi prišli do neke naključne rešitve, ki bi jo potem s postopkom lokalne optimizacije izboljšali. Algoritem bomo implementirali z uporabo knjižnice DAJ, ki omogoča simulacijo in vizualizacijo porazdeljega sistema. Ob tem bomo

pozorni na posebnosti, do katerih prihaja med sestavo in implementacijo algoritma.

Poglavje 2

Porazdeljeni algoritem

2.1 Opis porazdeljenega modela

Pred samim opisom algoritma najprej natačno opišimo lastnosti modela, ki jih predpostavljamo pri izvajanju le tega.

Porazdeljeni model sestavlja množico avtonomnih računskih vozlišč, ki med sabo komunicirajo s pošiljanjem sporočil po usmerjenih komunikacijskih kanalih. Na vsakem vozlišču teče enak program, v katerem je natančno definirano, katere operacije se bodo izvedle, ko vozlišče sprejme neko sporočilo. Predpostavimo pa še sledeče lastnosti modela, kjer vozlišče, ki sporočilo pošilja, imenujemo pošiljatelj, vozlišče na drugi strani kanala, ki sporočilo sprejme, pa sprejemnik.

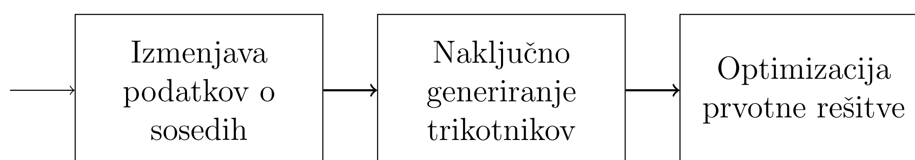
- Vrstni red sporočil, ki jih je pošiljatelj poslal po nekem kanalu, bo enak vrstnemu redu sporočil, ki jih bo sprejemnik sprejel.
- Časovni model je asinhron - vozlišče pri izvajanju in komunikaciji ne upošteva nobenih časovnih parametrov.
- Sistem je zanesljiv, pravilen in konsistenten - vsa sporočila, ki jih je pošiljatelj poslal po nekem kanalu, bodo zagotovo prišla do sprejemnika in to točno taka, kot so bila poslana. Do izpadov vozlišč pa ne prihaja.

Struktura porazdeljenega sistema je identična grafu G. Vsako vozlišče v grafu G predstavlja eno računsko vozlišče v sistemu in vsaka povezava v G dva nasprotno usmerjena komunikacijska kanala med vozliščema.

2.2 Opis algoritma

2.2.1 Uvod

Porazdeljeni algoritem, predstavljen v tem delu, je ločen na tri faze, ki si zaporedno sledijo pri izvajanju, prikazane pa so v diagramu na sliki 2.1. Najprej poskrbimo, da vsako vozlišča obdrži in pridobi samo tiste podatke, ki so relevantni za opisane postopke v naslednjih fazah. Vozlišča, ki ne morejo biti del nobenega trikotnika, končajo izvajanje v prvi fazi, z ostalimi pa potem v fazi naključnega generiranja trikotnikov pridemo do neke naključne dopustne rešitve, ki jo v fazi optimizacije poskušamo izboljšati.



Slika 2.1: Faze algoritma

Na začetku izvajanja programa ima vsako vozlišče samo seznam svojih sosedov. Začetno stanje vsakega vozlišča je FREE. Ob koncu izvajanja programa pa je vozlišče v stanju, ki je bodisi IN_TRIANGLE bodisi ALONE. Pomen omenjenih stanj je opisan kasneje pri podrobnem opisu pozameznih faz.

Pred predstavitvijo posameznih faz definirajmo nekaj izrazov, ki jih bomo uporabljali pri opisu algoritma. Pri tem upoštevamo definicijo grafa $G = (E, V)$, kjer je V množica vozlišč, E pa množica neusmerjenih povezav med vozlišči.

- Sosed (angl. neighbor) - v je sosed od u, če med njima obstaja povezava, torej če velja: $\{v, u\} \in E$.
- Skupni sosed (angl. neighbor incommon) - v je skupni sosed od u in w, če velja: v je sosed od u in v je sosed od w.
- Aktualni sosed (angl. promising neighbor) - v je aktualni sosed od u, če zanj obstaja možnost, da skupaj tvorita trikotnik - bolj podrobno je ta relacija opisana v drugi fazi algoritma.
- Potencialni trikotnik - množica {u,v,w} različnih vozlišč z močjo tri, za katera velja $\{\{u, v\}, \{u, w\}, \{v, w\}\} \subseteq E$.

V nadaljevanju podrobno predstavimo vse tri faze algoritma.

2.2.2 Izmenjava podatkov o sosedih

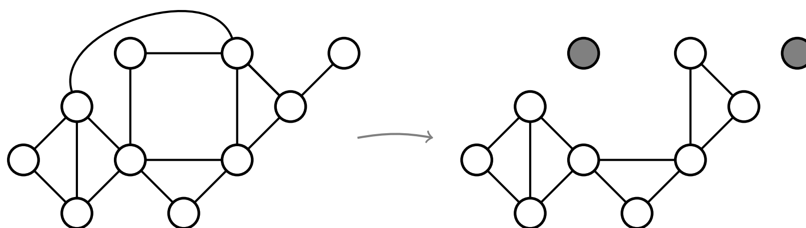
V prvi fazi algoritma vsako vozlišče najprej vsem sosedom pošlje sporočilo LIST s priloženim seznamom svojih sosedov ter preide v stanje LIST, v katerem ostane, dokler ne dobi sporočila s seznamom od vseh sosedov. V kolikor neko vozlišče po sprejetju sporočila LIST od vseh sosedov ugotovi, da ni del nobenega potencialnega trikotnika, na tej točki zaključi izvajanje programa. Ostala vozlišča preidejo nazaj v stanje FREE ter nadaljujejo izvajanje v naslednji fazi algoritma. Še prej pa vsakega soseda, s katerim vozlišče nima skupnih sosedov, izbriše iz seznama sosedov.

Primer, ki prikazuje stanje grafa po koncu izvajanja te faze, je prikazan na sliki 2.2. Podroben potek algoritma pa predstavlja Algoritem 1.

Analiza kompleksnosti

Če označimo z m_i število sosedov, ki jih ima neko vozlišče, je časovna zahtevnost za posamezno vozlišče, ko dobi sporočilo LIST s seznam sosedov dolžine m_k, reda $O(m \log m)$, kjer velja $m = max(m_i, m_k)$. Toliko je namreč časovna zahtevnost za izračun preseka obeh seznamov sosedov.

Število sporočil, ki pretečejo po posameznem komunikacijskem kanalu, je 1. Celotno število sporočil, ki pretečejo po vseh kanalih, je torej $2|E|$. Velikost sporočila je sorazmerna s številom sosedov pošiljatelja.



Slika 2.2: Slika na levi prikazuje graf, kakršnega obravnavamo v fazi izmenjave podatkov o sosedih. Slika na desni pa prikazuje graf, kakršnega si predstavljamo v nadaljevanju. Vozlišča, ki so potemnjena, so končala izvajanje, saj ne morejo biti del nobenega trikotnika. Prav tako so bile izbrisane povezave med sosedi(oba sta izbrisala drugega v seznamu sosedov), ki ne morejo skupaj tvoriti trikotnika in so zato nepotrebne v nadaljevanju algoritma.

Algoritem 1: Izmenjava podatkov o sosedih

global variables:
$ARRAY\ neighbors$
$ARRAY\ OF\ ARRAY\ neighbors_incommon$

local variables:
$count \leftarrow neighbors.length$
$received \leftarrow 0$

$SetState(LIST)$
$SendMsg(\text{all}, LIST\ with\ neighbors)$
while $received < count$ **do**
  Wait for the message
  $msg \leftarrow receive_message()$
  $sender \leftarrow msg.sender$
  if $msg \neq LIST$ **then**
    if $msg = ALONE$ **then** $UpdateStates(sender, ALONE)$
    else if $msg = IN_TRIANGLE$ **then**
      $UpdateStates(sender, IN_TRIANGLE)$
    else
      $SendMsg(sender, DENIED)$
    continue
  $i \leftarrow index(sender)$
  $neighbors_incommon[i] \leftarrow neighbors \bigcap msg.neighbors$
  if $neighbors_incommon[i].length\ =\ 0$ **then**
    $neighbors.RemoveElement(sender)$
  $received \leftarrow received + 1$
end
if $neighbors.length = 0$ **then** **return**
$SetState(FREE)$
Go to the next phase

Opomba: Za funkcije $SetState$(state), $UpdateStates$(neighbor,state) in $SendMsg$(to,msg) podajamo v dodatku A.1 verzijo implementirano v programskem jeziku Java

2.2.3 Naključno generiranje trikotnikov

Glavni cilj v tej fazi algoritma je najti naključno pokritje grafa s trikotniki. Najprej bomo spoznali stanja vozlišč in sporočila, ki so v tej fazi uporabljena, nato pa podrobno opisali postopek generiranja posameznih trikotnikov, ki predstavljajo naključno pokritje. Primer takega pokritja prikazuje slika 2.5 na koncu opisa te faze.

Posamezna stanja, v katera neko vozlišče lahko prehaja, so našteta in na kratko opisana v tabeli 2.1, sporočila s kratkim opisom, ki si jih vozlišča pošiljajo med sabo, pa so predstavljena v tabeli 2.2.

Stanje	Opis
FREE	prost; začetno stanje
JOIN_ME	poslal povabilo k sodelovanju
JOIN_US	poslal povpraševanje skupnim sosedom in čaka na odgovor
WAITING	čaka na potrditev, po tem, ko se je pozitivno odzval na povabilo k trikotniku
IN_TRIANGLE	v trikotniku; končno stanje
ALONE	ne more biti del nobenega trikotnika; končno stanje

Tabela 2.1: Možna stanja vozlišča s kratkimi opisi v fazi naključnega generiranja trikotnikov

Sporočilo	Opis
JOIN_ME	povabilo k sodelovanju
JOIN_US	povabilo k trikotniku
DENIED	zavrnitev
CONFIRMED	potrditev trikotnika; priložen podatek: ostali dve vozlišči v trikotniku
OK	sprejeto povabilo k trikotniku
IN_TRIANGLE	obvestilo o končnem stanju IN_TRIANGLE
ALONE	obvestilo o končnem stanju ALONE

Tabela 2.2: Možna sporočila s kratkimi opisi v fazi naključnega generiranja trikotnikov

Na začetku ima vsako vozlišče podatke o sosedih in skupnih sosedih, ki jih ima z vsakim izmed njih. Med samim izvajanjem pa dopolnjuje še seznam stanj za posameznega soseda.

Vedno, ko vozlišče preide v končno stanje, to sporoči vsem svojim sosedom, ti pa si ta podatek zabeležijo v seznam stanj sosedov. Glede na te podatke potem vozlišče odloča o aktualnosti nekega soseda na podlagi sledečih kriterijev:

- Če s sosedom nimata skupnih sosedov, potem ta sosed ni aktualen.
- Če je sosed v končnem stanju IN_TRIANGLE ali ALONE, potem ta sosed ni aktualen.
- Če so vsi skupni sosedi, ki jih imata s sosedom u, v končnem stanju, potem ta sosed ni aktualen.
- Vsi ostali sosedi so aktualni.

Postopek

Najprej vsako vozlišče sestavi seznam aktualnih sosedov, nato začne postopek generiranja novega trikotnika. Ta je prikazan na sliki 2.3 in se za vozlišče v začne z izbiro naključnega aktualnega soseda u, kateremu pošlje povabilo k sodelovanju - sporočilo JOIN_ME. Vozlišče u, ki to vabilo prejme pošlje povabila k trikotniku - sporočilo JOIN_US - vsem skupnim aktualnim sosedom ter preide v stanje JOIN_US. Ko eden (w) od skupnih sosedov pošlje nazaj odgovor CONFIRMED in ob tem preide v stanje WAITING, vozlišče u sprejme to sporočilo, preide v stanje IN_TRIANGLE ter pošlje vozliščema v in w potrditev. Tudi ti vozlišči ob sprejetju potrditve - sporočila CONFIRMED - preideta v stanje IN_TRIANGLE. Ob vsakem prehodu v stanje IN_TRIANGLE posamezna vozlišča to sporočijo vsem svojim sosedom.

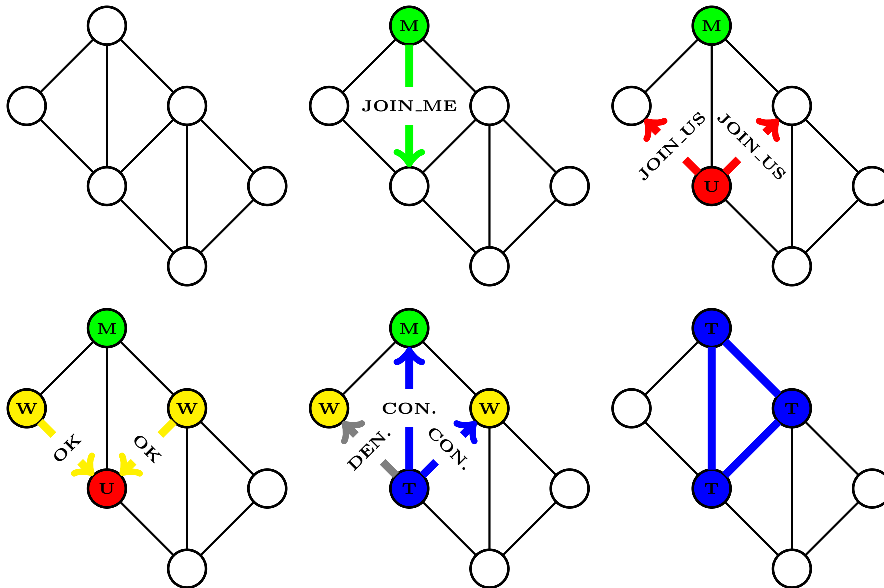
Opisali smo, kako pride do generiranja novega trikotnika, kar pa se zgodi samo v primeru, če ne pride do nobene od sledečih dveh situacij:

- vozlišče u, ki je dobilo povabilo k sodelovanju, se odzove s sporočilom DENIED. Do tega pride takrat, ko velja vsaj eden od primerov:
  - vozlišče u je v stanju, ki je različno od FREE in JOIN_ME
  - vozlišče u je v stanju JOIN_ME, njegov partner - tisti, ki mu je poslal sporočilo JOIN_ME - pa ni vozlišče v
  - vozlišči v in u nimata skupnih aktualnih sosedov

  - pride do situacije, opisane v naslednji točki

- vsa vozlišča (w_i), ki so dobila povabilo k trikotniku, se odzovejo s sporočilom DENIED. Do tega pride takrat, ko za vse povabljene velja eden od primerov:
  - vozlišče w_i je v stanju, ki je različno od FREE in JOIN_ME
  - vozlišče w_i je v stanju JOIN_ME, njegov partner pa ni vozlišče u

V obeh situacijah se ob zavrnitvi vozlišči v stanju JOIN_ME ali JOIN_US znajdeta pred odločitvijo, kako nadaljevati izvajanje. Enako velja za vozlišče v stanju WAITING, ki je dobilo zavrnitev, saj je bil trikotnik že generiran. Ker je potrebno zagotoviti, da bo vozlišče, ki preide v stanje FREE in počaka na naslednje sporočilo, zagotovo dobilo še kakšno sporočilo - saj bi v nasprotnem primeru ostalo za vedno v tem stanju - se izvajanje nadaljuje na enega od navedenih načinov:



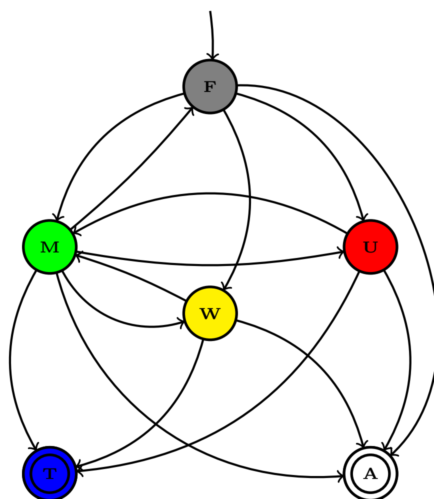
Slika 2.3: Zaporedje slik predstavlja postopek generiranja trikotnika. Predpostavljeno je, da je vozlišče v stanju JOIN_US sprejelo najprej sporočilo od vozlišča na desni, s katerim potem tvori trikotnik. V posameznih vozliščih so s črko označena njihova stanja, ki so relevantna pri postopku generiranja nastalega trikotnika. Pomen oznak je sledeč: F-FREE, M-JOIN_ME, U-JOIN_US, W-WAITING, T-IN_TRIANGLE.

- vozlišče preide v stanje FREE in počaka na naslednje sporočilo - če je bilo vozlišče neposredno pred tem v stanju INVITE_ME, njegov indeks pa je manjši od indeksa njegovega bivšega partnerja, ki je še vedno njegov aktualen sosed (če ni, potem nadaljuje na drug način)

- vozlišče začne postopek generiranja trikotnika od začetka in pošlje naključnemu aktualnemu sosedu sporočilo JOIN_ME in preide v ustrezno stanje - v ostalih primerih

Omenimo še kdaj preide vozlišče v končno stanje ALONE. To se zgodi takrat, ko je v nekem nekončnem stanju in ugotovi, da nima več aktualnih sosedov. To se lahko zgodi ob sprejemu sporočila ALONE ali IN_TRIANGLE.

Sedaj, ko smo za vsa posamezna stanja povedali, kdaj vozlišča prehajajo med njimi, te prehode predstavimo s končnim avtomatom na sliki 2.4.



Slika 2.4: Končni avtomat predstavlja prehode vozlišč med stanji v fazi naključnega generiranja trikotnikov. Pomen oznak je sledeč: F-FREE, M-JOIN_ME, U-JOIN_US, W-WAITING, T-IN_TRIANGLE, A-ALONE

Ostane nam še odgovor na vprašanje: Kdaj vozlišče preide v naslednjo fazo algoritma? To se zgodi takrat, ko dobi od vseh sosedov, s katerimi lahko tvori trikotnik, sporočilo o končnem stanju.

Psevdokodo te faze predstavlja Algoritem 2.

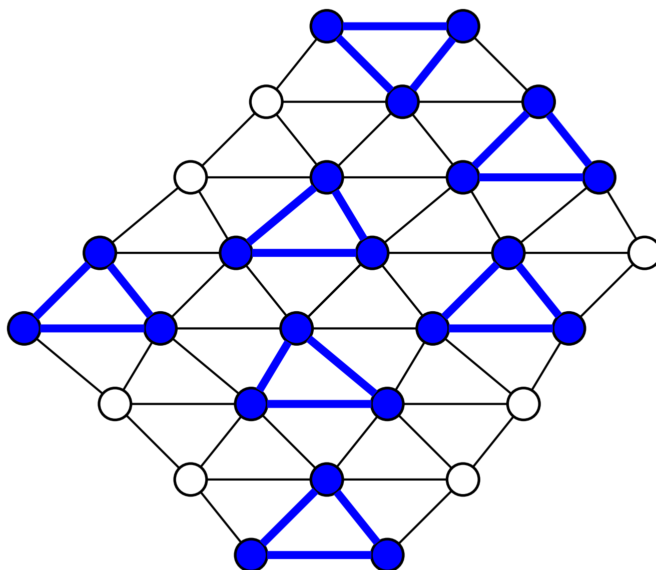
Analiza kompleksnosti

Obseg analize kompleksnosti na tem mestu presega obseg tega dela. Za boljšo predstavo si poglejmo, kolikšna je spodnja meja števila sporočil, ki pretečejo po komunikacijskih kanalih v fazi naključnega generiranja trikotnikov.

Pri posameznem generiranju trikotnika, ki je bilo uspešno zaključeno, po kanalu steče $3sa - d + 2$ sporočil, če ne upoštevamo sporočil o končnem stanju. Pri tem je sa število skupnih aktualnih sosedov, ki jih imata vozlišči, ki sta poslala sporočili JOIN_ME in JOIN_US, d pa število zavrnitev na povabilo k trikotniku. Če upoštevamo še, da vsako vozlišče ob prehodu v končno stanje to sporoči svojim sosedom, lahko pridemo do minimalnega števila sporočil, ki stečejo po vseh kanalih v tej fazi: $5|T| + 2|E|$, kjer je $|T|$ število trikotnikov v pokritju, $|E|$ pa število obstoječih povezav. Zaradi predpostavke minimalnosti smo upoštevali da je $sa = 1$ in $d = 0$.

Komentar

Glede na to, da vozlišča naključno izbirajo aktualne sosede, s katerimi bodo sodelovala, pridemo do več različnih rešitev. Tako lahko že s ponovitvijo izvršitve programa izboljšamo rezultat.



Slika 2.5: Primer naključnega pokritja grafa s trikotniki

Algoritem 2: Naključno generiranje trikotnikov

```
global variables:
ARRAY neighbors,states,phases
ARRAY OF ARRAY neighbors_incommon
triangle, state

local variables:
ARRAY waiting, promising_neighbor
partner,sender,msg

UpdateNeighbors()
partner ← GetRandomPromisingNeighbor()
SendMsg(partner, JOIN_ME)
SetState(JOIN_ME)
while at least one neighbor is not in final state do
    Wait for next message
    msg ← ReceiveMessage()
    sender ← msg.sender
    switch msg.value do

        * obravnava sporočil 1.del *
        * obravnava sporočil 2.del *

    end
end
Go to the next phase
```

Opomba: Za funkcije *UpdateNeighbors*(), *GetRandomPromisingNeighbor*(), *SetState*(state), *UpdateStates*(neighbor,state) in *SendMsg*(to,msg) podajamo v dodatku A.1 verzijo implementirano v programskem jeziku Java.

Algoritem 2: Naključno generiranje trikotnikov - obravnava sporočil 1.del

case $JOIN_ME$:
  if $state = FREE$ **then**
    if not $SendInvitations(sender)$ **then**
      $SendMsg(sender, DENIED)$
    break
  else if $state = JOIN_ME$ **and** $partner = sender$ **then**
    if $index > sender$ **then**
      if not $SendInvitations(sender)$ **then**
        $SetState(FREE)$
        $SendMsg(sender, DENIED)$
    break
  else if $state = JOIN_US$ **then**
    if $sender \in waiting$ **then break;**
  $SendMsg(sender, DENIED)$
  break

case $JOIN_US$:
  if $state = FREE$ **or** $(state = JOIN_ME$ **and** $partner = sender)$ **then**
    $partner \leftarrow sender$
    $SendMsg(sender, OK)$
    $SetState(WAITING)$
  else $SendMsg(sender, DENIED)$
  break

case OK :
  if $state = JOIN_US$ **then**
    $SetState(IN_TRIANGLE)$
    $SendMsg(sender, CONFIRMED$ with $index, partner)$
    $SendMsg(partner, CONFIRMED$ with $index, sender)$
    $SetTriangle(partner, sender)$
  else $SendMsg(sender, DENIED)$

case $IN_TRIANGLE$:
  $UpdateStates(sender, IN_TRIANGLE)$
  $UpdatePromisingNeighbors()$
  if *no promising nodes left* **then**
    $SetState(ALONE)$
    $SendMsg($all$, ALONE)$
  if $state == FREE$ **then**
    $partner \leftarrow GetRandomPromisingNeighbor()$
    $SendMsg(partner, JOIN_ME)$
    $SetState(JOIN_ME)$

Algoritem 2: Naključno generiranje trikotnikov - obravnava sporočil 2.del

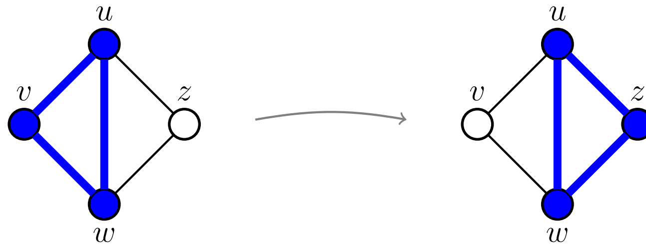
```
case CONFIRMED :
    SetTriangle(msg.triangle)
    SetState(IN_TRIANGLE)

case DENIED :
    if state = JOIN_ME then
        if partner < me or partner ∉ promising_neighbors then
            partner ← GetRandomPromisingNeighbor()
            SendMsg(partner, JOIN_ME)
        else  SetState(FREE)
    else if state = JOIN_US then
        waiting.remove(sender)
        if Empty(waiting) then
            SendMsg(partner, DENIED)
            partner ← GetRandomPromisingNeighbor()
            SendMsg(partner, JOIN_ME)
            SetState(JOIN_ME)
    else if state = WAITING then
        partner ← GetRandomPromisingNeighbor()
        SendMsg(partner, JOIN_ME)
        SetState(JOIN_ME)

case ALONE :
    UpdateStates(sender, ALONE)
    UpdatePromisingNeighbors()
    if no promising nodes left then
        SetState(ALONE)
    if state == FREE then
        partner ← GetRandomPromisingNeighbor()
        SendMsg(partner, JOIN_ME)
        SetState(JOIN_ME)

case NEXT_PHASE :
    phases[sender] ← phases[sender] + 1
    break
```

2.2.4 Optimizacija prvotne rešitve

Po tem, ko smo dobili prvotno rešitev, nas zanima, ali lahko to rešitev izboljšamo tako, da pri tem celotno število trikotnikov monotono narašča.

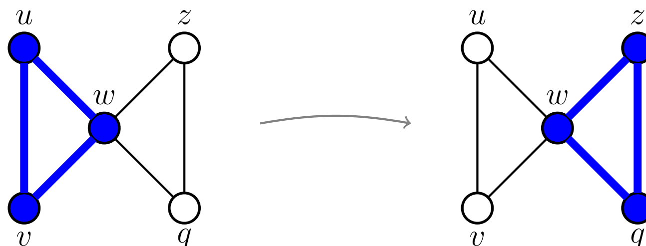
Najprej definiramo nekatere izraze, ki so v tej fazi algoritma prvič uporabljeni:

- Zrcaljenje trikotnika - operacija na grafu G, ki jo označimo z $Zr(G, T, v, z)$, kjer vozlišče v v nekem trikotniku iz T zamenja vozlišče z, ki prej ni bilo v nobenem trikotniku. Operacija je prikazana na sliki 2.6.
  Formalni zapis:
  $Zr(G, T, v, z) \iff G = (V, E) \wedge (T$ je množica trikotnikov v G$) \wedge (\exists\{u, v, w\} \in T : (z \in V : \forall T_i \in T : z \notin T_i) \wedge \{\{z, v\}, \{z, w\}\} \subset E) \implies (T \leftarrow (T - \{u, v, w\}) \bigcup \{\{z, v, w\}\})$



Slika 2.6: Operacija zrcaljenja trikotnika

- Zasuk trikotnika - operacija na grafu G, ki jo označimo z $Za(G, T, \{u, v\}, \{z, q\})$, kjer vozlišči u in v v pripadajočem trikotniku iz T zamenjata vozlišči q in z, ki prej nista bili v nobenem trikotniku. Operacija je prikazana na sliki 2.7.
  Formalni zapis:
  $Za(G, T, \{u, v\}, \{z, q\}) \iff G = (V, E) \wedge (T$ je množica trikotnikov v $G) \wedge (\exists\{u, v, w\} \in T : (\{z, q\} \subset V : \forall T_i \in T : \{z, q\} \bigcap T_i = \emptyset) \wedge \{\{z, w\}, \{q, w\}, \{z, q\}\} \subset E) \implies (T \leftarrow (T - \{u, v, w\}) \bigcup \{\{z, q, w\}\})$

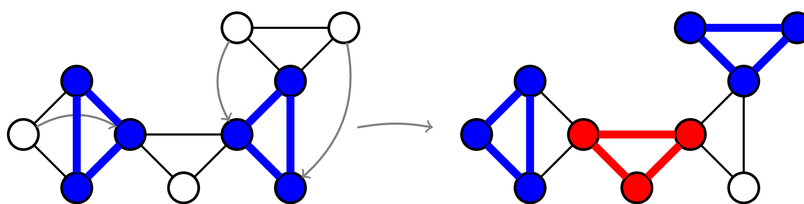


Slika 2.7: Operacija zasuka trikotnika

- Premik trikotnika - zajema operaciji zrcaljenja in zasuka trikotnika.
- Nezasedeno stanje - vsa možna stanja nekega vozlišča, ko to ni del nobenega trikotnika. V tabeli 2.4 je v opisu stanj natačno določeno za vsako stanje posebej, ali je stanje nezasedeno.
- Zasedeno stanje - vsa možna stanja nekega vozlišča, ko je to del nekega trikotnika. V tabeli 2.4 je v opisu stanj natačno določeno za vsako stanje posebej, ali je stanje zasedeno.
- Prenos stanja - ob izvršitvi neke operacije vozlišče preide v stanje, v katerem je bilo neko drugo vozlišče pred tem in obratno.
- Dogodek potencialnega trikotnika - dogodek, ko obstajajo tri vozlišča, ki potencialno lahko tvorijo trikotnik in so vsa v tistem trenutku v nezasedenem stanju.

Operaciji premika trikotnika imata lastnost, da ohranjata število trikotnikov, zato njuna uporaba zadovolji zahtevo po monotonem naraščanju števila trikotnikov. Poleg te lastnosti pa je zanimiva tudi lastnost, kjer se stanje nezasedenosti vozlišča prenaša po grafu. To lahko privede do dogodka potencialnega trikotnika in posledično generiranja novega trikotnika, kar je prikazano na sliki 2.8 na primeru nekega grafa. Ker je ravno tak dogodek ključnega pomena, saj se število trikotnikov takrat poveča, se na tem mestu pojavlja vprašanje, na katero odgovorimo v nadaljevanju:

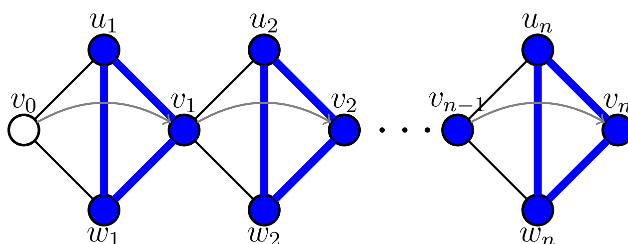
- Kdaj lahko pride do dogodka potencialnega trikotnika?



Slika 2.8: Slika prikazuje, kako bi v tem grafu in postavitvi trikotnikov lahko prišli z eno operacijo zrcaljenja in eno operacijo zasuka do dogodka potencialnega trikotnika.

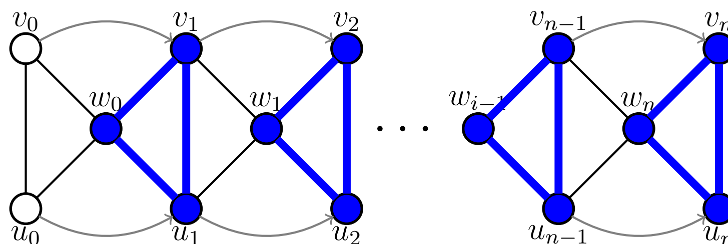
Za lažji odgovor na vprašanje definirajmo naslednje izraze, ki se nanašajo na stanje vozlišč v nekem trenutku:

- Zrcalna pot - zaporedje vozlišč $P_{zr} = (v_0, \ldots, v_n)$, na katera bi se pri zaporednem izvajanju operacije zrcaljenja trikotnika prenašalo nezasedeno stanje prvega vozlišča, če bi to bilo v nezasedenem stanju. Tako pot prikazuje slika 2.9.
  Formalni zapis, kjer upoštevamo, da je T množica trikotnikov, E množica povezav in V množica vozlišč:
  $P_{zr} = (v_0, \ldots, v_n)$, kjer velja $((v_0 \in V : \forall T_i \in T : v_0 \notin T_i) \wedge (v_j \in V, 0 < j \leq n, \wedge \exists \{u_j, w_j\} \subset V) : \{u_j, v_j, w_j\} \in T \wedge \{\{v_{i-1}, u_i\}, \{v_{i-1}, w_i\}\} \subset E)$



Slika 2.9: Zrcalna pot

- Zasučna pot - zaporedje parov vozlišč $P_{za} = ((u_0, v_0), \ldots, (u_n, v_n))$, na katera bi se pri zaporednem izvajanju operacije zasuka trikotnika prenašali nezasedeni stanji vozlišč prvega elementa, če bi vozlišči bili v nezasedenem stanju. Tako pot prikazuje slika 2.10.
  Formalni zapis, kjer upoštevamo, da je T množica trikotnikov, E množica povezav in V množica vozlišč:
  $P_{za} = ((u_0, v_0), \ldots, (u_n, v_n))$, kjer velja $((\{u_0, v_0\} \subset V : \forall T_i \in T : \{u_0, v_0\} \bigcap T_i = \emptyset) \wedge (\{u_j, v_j\} \subset V, 0 < j \leq n, \exists w_j \in V) : (\{u_j, v_j, w_j\} \in T) \wedge (\{\{v_{j-1}, u_{j-1}\}, \{v_{j-1}, w_j\}, \{u_{j-1}, w_j\}\} \subset E))$.



Slika 2.10: Zasučna pot

- Sledenje poti - zaporedno izvajanje operacij nad trikotniki, tako da se nezasedeno stanje prenaša od začetka do konca vozlišč seznama, ki predstavlja bodisi zrcalno bodisi zasučno pot.
  Sledenje zrcalne poti $P_{zr} = (v_0, \ldots, v_n)$: Izvajanje operacije $Zr(G, T, v_i, v_{i-1})$ za vsak i, ki po vrsti zavzema vrednosti od 1 do n.
  Sledenje zasučne poti $P_{za} = ((u_0, v_0), \ldots, (u_n, v_n))$: Izvajanje operacije $Za(G, T, (v_i, u_i), (v_{i-1}, u_{i-1}))$ za vsak i, ki po vrsti zavzema vrednosti od 1 do n.

Sedaj lahko odgovorimo na vprašanje :
Do dogodka potencialnega trikotnika lahko pride, če obstaja takšno pravilno (pri sledenju se vse operacije izvedejo uspešno) zaporedje poti, za katerega velja

- množica vozlišč, ki so po koncu zaporednega sledenja vseh poti v nezasedenem stanju, vsebuje podmnožico {u,v,w}, ki predstavlja nek potencialni trikotnik

in zaporedno izvedemo vse operacije, ki jih sledenje teh poti zahteva.

Pomen tega odgovora je, da predstavlja vodilo za nadaljnji razvoj algoritma. Osredotočili se bomo na to, kako priti do takšnega zaporedja poti, da bi prišlo do dogodka potencialnega trikotnika ter uspešno izvesti vse operacije. Pri tem je potrebno imeti v mislih, da je lahko takšnih zaporedij poti več, njihovo križanje (pri obeh je zahtevana operacija na istem trikotniku) pa se zato zaključi z neuspešnim sledenjem poti. Zato pričakovan rezultat ni optimalen, pač pa samo približek tega.

Idejo, kako se približati omenjenemu cilju na našem porazdeljenem modelu, predstavlja naslednji postopek:

Postopek
Vsako nezasedeno vozlišče ima težnjo po tem, da bi bila sosednja vozlišča v nezasedenem stanju in bi lahko skupaj tvorili trikotnik. Zaradi tega pošlje vsem nezasedenim sosednjim vozliščem sporočilo SIGNAL z objektom *signal* (v nadaljevanju signal), ki vsebuje *oznako*, *indeks* začetnega vozlišča in *razdaljo* od začetnega vozlišča. Vozlišče, ki ta signal sprejme, glede na *oznako* signala bodisi označi sebe kot eno od vozlišč na neki poti, bodisi ne označi, ker je samo prehodno vozlišče. Vsako zasedeno vozlišče, ki sprejme signal, tega shrani v ustrezen seznam, če ustreza pogojem, opisanim kasneje v poglavju, in posreduje naprej s tem, da upošteva pravila v tabeli 2.3. Ta določajo, komu posredovati signal in kako spremeniti *oznako*, ter ustrezno poveča podatek o *razdalji*. Na

tak način se označi bodisi zrcalna bodisi zasučna pot od končnega do prvega vozlišča, ki lahko neko nezasedeno stanje pripelje bližje k prvotnemu vozlišču. Ključno pri tem pa je, da signal pride do nezasedenega vozlišča, ki ob prejemu signala z ustrezno *oznako* sproži eno od operacij premika.

Če je *oznaka* signala

- OZNAČI, sproži operacijo zasuka, če lahko; če ne pa operacijo zrcaljenja, tako da pošlje sporočilo za začetek operacije vozlišču, od katerega je prišel signal.
- OZNAČI-2, sproži operacijo zasuka, če lahko, tako, da bi nov trikotnik vseboval poleg njega še sosednje nezasedeno vozlišče.
- katera druga oznaka iz tabele 2.3, ne naredi nič.

Z uspešno izvedbo operacij se bo nezasedeno stanje preneslo na predhodno označeno vozlišče na poti in bo tako za en korak bližje vozlišču, ki je prvotno poslalo signal.

Ko se po prenosu stanja novo vozlišče znajde v nezasedenem stanju, preveri, ali je prišlo do dogodka potencialnega trikotnika. V tem primeru sproži proces za generiranje novega trikotnika. Drugače pa preveri med signali v seznamu, ali obstaja takšen z *oznako* OZNAČI ali OZNAČI-2 ter med takimi izbere tistega z najmanjšo *razdaljo*, s katerim lahko izvede ustrezno operacijo ter jo tudi sproži. Tako se nezasedeno stanje prenaša vse bližje prvotnemu vozlišču.

V kolikor ne pride do izvedbe operacije, ker je bila pot medtem že izbrisana, pošlje vozlišče signal vsem svojim sosedom kakor na začetku postopka in počaka na naslednje sporočilo.

Omenili smo že, da so v tabeli 2.3 pravila za pošiljanje signala naprej, slika 2.11 pa prikazuje primer označitve poti.

Prejeta oznaka	Komu poslati naprej	Poslana oznaka
(na začetku)	x je v trikotniku	OZNAČI
OZNAČI	$x = max(a, b)$	PREHOD-1
PREHOD-1	$x \in \{a, b\}$, $x \neq p$	PREHOD-2
	$y \notin skupni(\{a, b\} - \{p\},v)$	OZNAČI-2
	in $y \notin \{a, b\}$	
PREHOD-2	$x \in skupni(p,v)$	OZNAČI
	in $x \notin \{a, b\}$	
	$y \notin skupni(p,v)$	OZNAČI-2
	in $y \notin \{a, b\}$	
OZNAČI-2	če $a \in skupni(p,v)$: $y = b$	PREHOD-3
	če $b \in skupni(p,v)$: $y = a$	
	in obstaja signal	
	z oznako PREHOD-1 ali PREHOD-2	
	v seznamu signalov	
PREHOD-3	$y \notin \{a, b\}$ in $z \notin \{a, b\}$	OZNAČI-2
	$y \in skupni(z,v)$	

Tabela 2.3: Tabela predstavlja kako se signal posreduje naprej. a in b sta vozlišči, ki sta skupaj z vozliščem v trikotniku, p pa pošiljatelj sporočila SIGNAL. Pri tem predpostavljamo, da sta x in y soseda vozlišča, katerima se pošlje sporočilo, če je pogoj naveden in izpolnjen. z je nek sosed. $skupni(v_i, v_j)$ je funkcija, ki vrne množico skupnih sosedov v_i in v_j.

Opisan je bil povod za izvedbo ene od operacij na nekem trikotniku. V nadaljevanju pa je opisano, kako poteka sam proces izvedbe posamezne operacije ter kaj se pri teh spremembah dogaja s seznamov signalov. Najprej pa so predstavljena vsa stanja (Tabela 2.4) in sporočila (Tabela 2.5), uporabljena v tej fazi.

Slika 2.11: Primer, kako se označi pot, ko pošlje vozlišče v začetni signal vozlišču u. o označuje vozlišča, ki so prejela signal z oznako OZNAČI, o2 pa vozlišča, ki so prejela oznako OZNAČI-2. S p pa so označena prehodna vozlišča. Potovanje signala je nakazano z usmeritvijo povezav, po katerih se signal pošilja.

Stanje	Opis
ALONE	prost; nezasedeno stanje
IN_TRIANGLE	v trikotniku; zasedeno stanje
READY	povabilo k sodelovanju; nezasedeno stanje
INVITE	poslan je bil signal tretjemu, za skupno generiranje; nezasedeno stanje
MOVE	za začetek operacije zrcaljenja; nezasedeno stanje
MOVE_TRIANGLE	pri zrcaljenju trikotnika, čaka na blokado trikotnika; zasedeno stanje
INVITE_TRIANGLE	pri zrcaljenju trikotnika, čaka na blokado trikotnika; zasedeno stanje
BLOCK_TRIANGLE	blokiranje trikotnika; zasedeno stanje

Tabela 2.4: Možna stanja s kratkimi opisi v fazi optimizacije

Sporočilo	Opis
READY	povabilo k sodelovanju
YES	sprejeto povabilo k sodelovanju
INVITE	povabilo k trikotniku - generiranje ali operacija zasuka;
	priložen podatek: ostali vozlišči v morebitnem trikotniku
MOVE	pobuda za zrcaljenje trikotnika
BLOCK_TRIANGLE	blokiranje trikotnika
DENY	zavrnitev
DENY_INVITE	zavrnitev pri povabilu k generiranju
	trikotnika in operaciji zasuka
DENY_MOVE	zavrnitev operacije zrcaljenja trikotnika
CONFIRMED	potrditev trikotnika;
	priložen podatek: ostali dve vozlišči v trikotniku
LOST	obvestilo o izključitvi iz trikotnika;
	priložen podatek: ostali dve vozlišči v bivšem trikotniku
SWITCH	sprememba vozlišča trikotnika pri zrcaljenju;
	priložen podatek: ostali dve vozlišči v novem trikotniku
IN_TRIANGLE	obvestilo o stanju IN_TRIANGLE
ALONE	obvestilo o stanju ALONE
SIGNAL	signal za označevanje poti
	priložen podatek: objekt *signal*

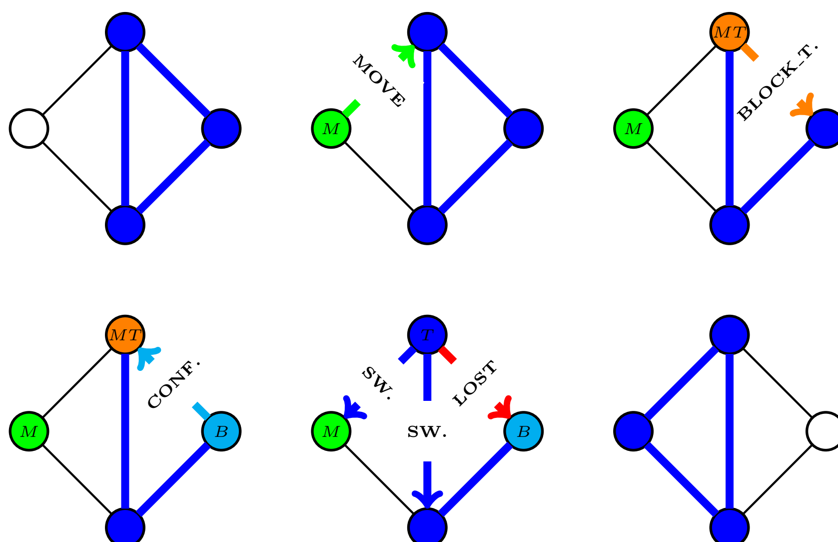
Tabela 2.5: Možna sporočila s kratkimi opisi v fazi optimizacije

Operacija zrcaljenja
Postopek operacije zrcaljenja prikazuje slika 2.12. Začne se s tem, da vozlišče v stanju ALONE pošlje sporočilo MOVE sosedu, ki je v stanju IN_TRIANGLE in ga imenuje za partnerja ter preide v stanje MOVE.

Ko partner sprejme sporočilo MOVE, pošlje sporočilo BLOCK_TRIANGLE vozlišču z največjim indeksom v trikotniku ter preide v stanje MOVE_TRIANGLE in počaka na potrditev blokade trikotnika. Če je to vozlišče z največjim indeksom v trikotniku, ta korak ni potreben.

Ob sprejemu signala CONFIRMED - v prvem primeru - ali pa takoj - v drugem primeru - vozlišče pošlje sporočili SWITCH s podatki o novem trikotniku vozlišču, ki ostane v trikotniku, in vozlišču, ki je začelo operacijo. Vozlišču, ki je izpadlo iz trikotnika, pošlje sporočilo LOST.

Obe vozlišči, ki sta prejeli sporočilo SWITCH nato preideta v stanje IN_TRI-



Slika 2.12: Postopek operacije zrcaljenja trikotnika. Predpostavljeno je, da je največje vozlišče v prvotnem trikotniku skrajno desno.

ANGLE. Izpadlo vozlišče pa ob prejetju sporočila LOST preide v stanje ALONE in ukrepa, kakor je opisano kasneje. Vsa štiri udeležena vozlišča na koncu pošljejo sporočilo o novem stanju vsem svojim sosedom.

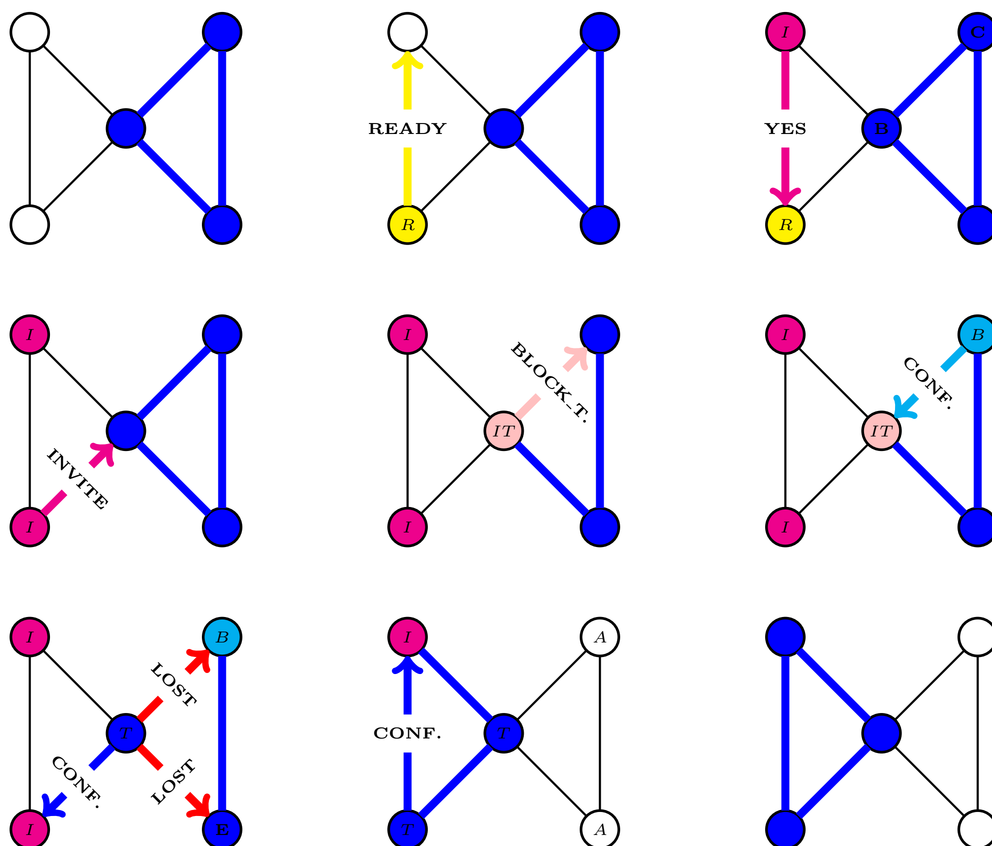
Operacija se ne izvede uspešno:

- če je trikotnik že blokiran - vozlišče v stanju MOVE_TRIANGLE dobi sporočilo DENY, in pošlje vozlišču v stanju MOVE sporočilo DENY_MOVE.
- če trikotnik ne obstaja več - vozlišče, ki je prejelo sporočilo MOVE, pošlje nazaj sporočilo DENY_MOVE.

V tem primeru vsa vozlišča preidejo v stanje, v katerem so bila pred pričetkom operacije.

Operacija zasuka

Postopek operacije zasuka prikazuje slika 2.13. Začne se tako, da vozlišče v stanju ALONE pošlje sosednjemu nezasedenemu vozlišču sporočilo READY in ga imenuje za svojega partnerja. Ta se odzove s sporočilom YES in ga sprejme kot partnerja ter preide v stanje INVITE. Ob sprejemu sporočila YES tudi prvo vozlišče preide v stanje INVITE in pošlje sporočilo INVITE skupnemu vozlišču, ki je v trikotniku. Nato se trikotnik blokira, tako da ta pošlje sporočilo



Slika 2.13: Postopek operacije zasuka trikotnika. Predpostavljeno je, da je največje vozlišče v prvotnem trikotniku zgoraj desno.

BLOCK_TRIANGLE največjemu vozlišču v trikotniku (razen, če ni ravno to največje) in počaka na potrditev s sporočilom CONFIRMED. Po uspešnem blokiranju, vozlišče pošlje sporočilo CONFIRMED s podatki o novem trikotniku vozlišču, od katerega je dobilo sporočilo INVITE ter sporočilo LOST obema izpadlima vozliščema, in preide nazaj (če je prej ob blokadi prešlo v BLOCK_TRIANGLE) v stanje IN_TRIANGLE. Prav tako začetno vozlišče ob sprejemu sporočila CONFIRMED pošlje partnerju sporočilo CONFIRMED in preide v stanje IN_TRIANGLE. Ob sprejemu tega sporočila preide v stanje IN_TRIANGLE še tretje vozlišče v trikotniku. Vozlišči, ki sta prejeli LOST, pa preideta v stanje ALONE in nadaljujeta izvajanje, kakor je že bilo opisano. Ob tem vsako vozlišče o prehodu v stanje ALONE ali IN_TRIANGLE pošlje svojim sosedom ustrezno sporočilo.

Operacija se ne izvede uspešno, če:

- se povabljeno vozlišče odzove s sporočilom DENY ali DENY_INVITE,
- je trikotnik že blokiran, ker na njem že poteka ena od operacij.

V tem primeru vsa vozlišče preidejo v stanje, v katerem so bila pred pričetkom operacije.

Če vozlišče, ki naj bi bilo v trikotniku, ni več del trikotnika, se namesto operacije zasuka izvede generiranje novega trikotnika.

Omenimo še nekaj situacij, do katerih lahko pride in kako se izvajanje nadaljuje.

- V primeru, ko dve vozlišči hkrati eno drugemu pošljeta sporočilo READY, večji ob prejetju sporočila nadaljuje, kot da bi dobilo sporočilo YES, manjši pa samo preide v stanje INVITE.
- V primeru, ko vozlišče v stanju READY dobi sporočilo INVITE od svojega partnerja oz. njegovega partnerja, pošlje nazaj sporočilo CONFIRMED, kakor, da bi že prišlo do uspešne blokade trikotnika. Postopek se nato konča z generiranjem novega trikotnika.
- Če vozlišče, ki je dobilo sporočilo LOST, čaka na odgovor za blokiranje trikotnika zaradi operacije zasuka (je v stanju INVITE_TRIANGLE), se zgodi isto kot v predhodnem primeru.

Generiranje trikotnika

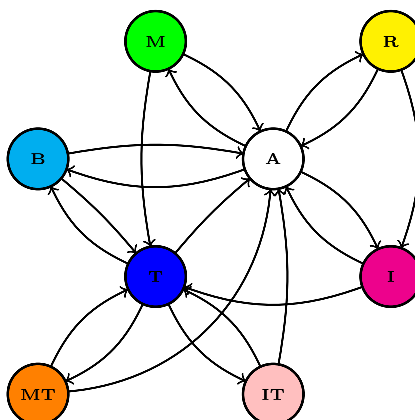
Če vozlišče ugotovi, da ima dva nezasedena soseda, s katerima lahko tvori nov trikotnik, začne proces generiranja novega trikotnika, kakor bi hotelo izvesti operacijo zasuka s tem, da vozlišče, ki sprejme sporočilo INVITE, sploh ni bilo v trikotniku.

Omenjeni so bili tudi ponesrečeni primeri operacije zasuka, ki pripeljejo do generiranja novega trikotnika.

Kako neko vozlišče prehaja med stanji v fazi optimizacije, je predstavljeno z avtomatom na sliki 2.14.

O signalih

Sedaj bolj podrobno opišimo, katere signale posamezno vozlišče shrani v seznam signalov in kako potem z obnavljanjem in brisanjem signalov zagotavlja konsistenco označbe posamezne poti.



Slika 2.14: Avtomat predstavlja možne prehode vozlišč med stanji v fazi optimizacije. Pomen oznak je sledeč: T-IN_TRIANGLE, A-ALONE, R-READY, I-INVITE, M-MOVE, B-BLOCK_TRIANGLE, MT-MOVE_TRIANGLE, IT-INVITE_TRIANGLE

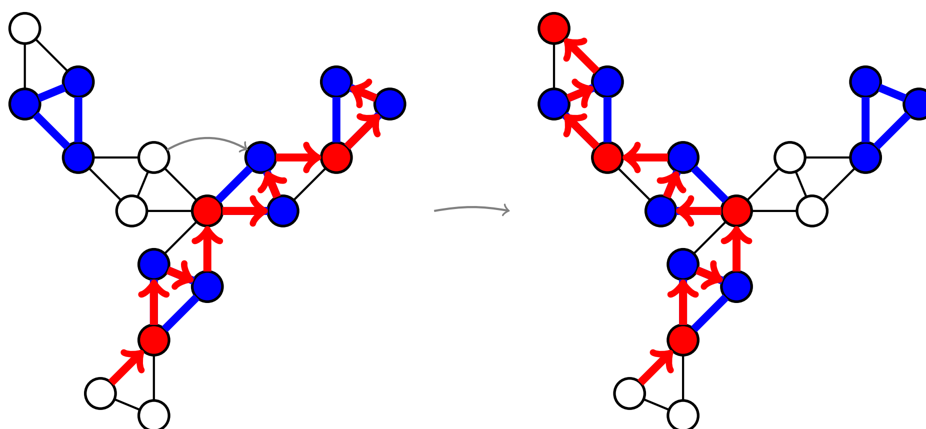
Nezasedeno vozlišče, ki nek signal pošilja, si tega najprej shrani v seznam signalov. Poleg tega si zabeleži vsa vozlišča, ki jim ta signal posreduje, kot to naredi vsako vozlišče pri posredovanju signala. Prav tako si ob sprejetju signala vsako vozlišče skupaj s signalom shrani še pošiljatelja.

Vozlišče bo prejet signal glede na obstoječe, ki jih ima v seznamu, tega shranilo in posredovalo naprej (samo, če je v zasedenem stanju), če:

- indeks vozlišča, ki je prvotno poslalo ta signal, ni enak indeksu trenutnega vozlišča,
- ne obstaja signal v seznamu, ki bi imel isto oznako in pošiljatelja, razdaljo pa manjšo ali enako,
- ima signal oznako OZNAČI-2 in ni v seznamu signala z oznako OZNAČI z istim pošiljateljem ter manjšo ali enako razdaljo,
- signala s takim ,indeksom prvotnega vozlišča in pošiljateljem ni v seznamu izbrisanih signalov,

pri tem pa se upoštevajo le oznake, ki jih najdemo v tabeli 2.3.

Med izvajanjem operacij se lahko zgodi, da je seznam signalov, ki ga ima neko vozlišče, zastarelo in ne predstavlja pravilne poti. Zato so potrebni ukrepi s katerimi seznam signalov ob izvedbi operacije nad nekim trikotnikom oziroma



Slika 2.15: Slika prikazuje, kako se potek neke zrcalne poti spremeni, če pride do operacije zasuka trikotnika na tej poti.

generiranjem novega trikotnika obnovimo novemu stanju primerno (primer spremembe poti prikazuje slika 2.15). Pri tem velja, da vedno, ko dobi vozlišče signal z oznako IZBRIS in indeksom prvotnega vozlišča, izbriše signal, ki je prišel od tega pošiljatelja s tem indeksom ter posreduje naprej signal za izbris vsem, ki jim je predhodno poslal ta signal.

- Če je vozlišče prešlo iz stanja nezasedenosti v stanje IN_TRIANGLE, gre skozi vse signale, ki jih je poslalo in
  - pošlje signal z oznako IZBRIS vsem prejemnikom prejšnjih signalov ter sam signal izbriše iz svojega seznama, če nima oznake OZNAČI ali OZNAČI-2.
  - pri signalih z oznako OZNAČI ali OZNAČI-2 ponovno naprej posreduje signal, tako da upošteva nov trikotnik in pravila v tabeli 2.3, razen, če tega signala prvotno ni poslal sosed, ki je sedaj z njim v trikotniku.

  in izbriše vse signale, ki nimajo oznake OZNAČI ali OZNAČI-2 in jih prvotno ni poslalo vozlišče, s katerim je skupaj v trikotniku.
- Če je vozlišče prešlo iz stanja IN_TRIANGLE v stanje nezasedenosti, in ne pride do izvedbe nobene operacije oz. generiranja trikotnika, pošlje sporočilo SIGNAL vsem sosedom kakor na začetku postopka. Pred tem

pa pošlje signal za izbris vsem, ki jim je posredovalo signale v seznamu in izbriše vse signale.

- Če vozlišče ostane v stanju IN_TRIANGLE, zamenjajo pa se ostala vozlišča v trikotniku
  - pošlje za vse signale, ki jih je posredovalo vozliščem izven prejšnjega trikotnika, signal z oznako IZBRIS tem vozliščem,
  - za vse signale z oznako OZNAČI ali OZNAČI-2 pošlje signal naprej, tako da upošteva podatke novega trikotnika in pravila iz tabele 2.3 razen, če tega signala prvotno ni poslal sosed, s katerim sedaj tvori trikotnik

  ter izbriše vse signale, ki nimajo oznake OZNAČI ali OZNAČI-2 in jih prvotno ni poslalo vozlišče s katerim je skupaj v trikotniku.

Ker se lahko zgodi, da se zaradi brisanja signala iz seznama nek signal pošilja v ciklu (signal za izbris zasleduje signal za označbo, ki pa bo vedno znova izbrisan in tako bo vedno tudi sprejet ob naslednjem prihodu do istega vozlišča) uvedemo seznam izbrisanih signalov, v katerega shranjujemo vse izbrisane signale, ter ga vedno, ko vozlišče preide v drugačno stanje, izbrišemo v celoti. Vedno, ko bi radi shranili nek signal in ga posredovali naprej, najprej preverimo, če je ta signal že prišel na to vozlišče in je bil morebiti že izbrisan. V tem primeru se signala ne shrani in ne posreduje naprej ter tako prekine zaciklano pot.

Psevdokodo faze optimizacije predstavlja Algoritem 3.

Analiza kompleksnosti
Analiza kompleksnosti te faze presega obseg tega dela.

O ustavljivosti
Na neki točki se globalno stanje sistema ne spreminja več. Sam algoritem pa nima lastnosti, da bi takšno stanje tudi prepoznal, zato bi bilo potrebno algoritem nadgraditi oziroma uporabiti kakšen dodaten algoritem, ki bi ugotovil, kdaj se je izvajanje končalo. To je takrat, ko vsa sporočila čakajo na novo sporočilo, sporočil pa ni več v nobenem kanalu.

Eden od znanih algoritmov, ki rešuje ta problem, je $Chandy - Lamportov$ algoritem [5, 6] v čigar opis se tukaj ne bomo spuščali.

Algoritem 3: Optimizacija naključne rešitve

global variables:
$ARRAY\ neighbors$
$ARRAY\ OF\ ARRAY\ neighbors_incommon$
$ARRAY\, states, phases, signals, old_signals$
$state$,$phase$,$triangle$

local variables:
$partner$,$another$,msg
$alone$,$sender$,$other$

$alone \leftarrow CalcAlone()$
if $state = ALONE$ **then** $SendSignal()$
while $true$ **do**
    if $Emtpy(waitlist)$ **then** Wait for next message
    else Wait for next message from $waitlist$
    $msg \leftarrow ReceiveMessage()$
    $sender \leftarrow msg.sender$
    $other \leftarrow msg.other$
    switch $msg.value$ **do**

        * obravnava sporočil 1.del *
        * obravnava sporočil 2.del *
        * obravnava sporočil 3.del *
        * obravnava sporočil 4.del *
        * obravnava sporočil 5.del *

    end
end

Opomba: Za funkcije $SetState$(state), $UpdateStates$(neighbor,state), $SendMsg$(to,msg), $RotateTriangle$(neighbor), $GetTriangle$(compare), $ICanMakeTriangle$(alone), $CheckSignals$(alone), $RotateTriangle()$, $MoveTriangle()$, $ReSendEraseSignals()$, $SendSignal()$, $DeletePath(signal)$ in $CleanSignals()$ podajamo v dodatku A.1 verzijo implementirano v programskem jeziku Java.

Algoritem 3: Optimizacija - obravnava sporočil 1.del

case $ALONE$:
  $alone \leftarrow UpdateStates(sender, ALONE, alone)$
  if $state = ALONE$ **and** $alone > 1$ **then**
    if $RotateTriangle(sender)$ **then** **break**
  break
case $IN_TRIANGLE$:
  $alone \leftarrow UpdateStates(sender, IN_TRIANGLE, alone)$
  break

case $READY$:
  if $state = ALONE$ **or** $(state = ALONE$ **and** $partner = sender)$ **then**
    $partner \leftarrow sender$
    $SetState(INVITE)$
    $SendMsg(sender, YES)$
  else if $state = READY$ **and** $partner = sender$ **then**
    if $GetTriangle(true)$ **then**
      $SetState(INVITE)$
    else
      $SendMsg(sender, DENY_INVITE)$
      $SetState(ALONE)$
      $CheckSignals(alone)$
  else $SendMsg(sender, DENY)$
  break

case $INVITE$:
  $alone \leftarrow UpdateStates(sender, ALONE)$
  $alone \leftarrow UpdateStates(other, ALONE)$
  if $state = IN_TRIANGLE$ **then**
    $partner = sender$
    $another = other$
    $SetState(INVITE_TRIANGLE)$
    if $index > max(triangle.a, triangle.b)$ **then**
      $RotateTriangle()$
    else
      $SendMsg(max(triangle.a, triangle.b), BLOCK_TRIANGLE)$
  else if $state = ALONE$ **or** $(state = READY$ **and** $(partner = sender$
  or $partner = other))$ **then**
    $SendMsg(sender, CONFIRMED, index, other)$
    $SetTriangle(partner, other)$ $ReSendEraseSignals()$;
    if $state = READY$**and**$partner = other$ **then**
      $deny \leftarrow partner$
  else $SendMsg(sender, DENY_INVITE)$
  break

Algoritem 3: Optimizacija - obravnava sporočil 2.del

```
case YES :
    if state = READY then
        if partner = sender then
            if GetTriangle(false) then
                SetState(INVITE)
            else
                SetState(ALONE)
                SendMsg(sender, DENY_INVITE)
                if alone > 1 then
                    if ICanMakeTriangle(alone) then break
                CheckSignals(alone)
case MOVE :
    alone ← UpdateStates(sender, ALONE)
    if  state = IN_TRIANGLE then
        if MoveOK() then
            partner = sender
            SetState(MOVE_TRIANGLE)
            if index > max(triangle.a, triangle.b) then
                MoveTriangle()
            else
                SendMsg(max(triangle.a, triangle.b), BLOCK_TRIANGLE)
            break
    SendMsg(sender, DENY_MOVE)
    break
case BLOCK_TRIANGLE :
    if state = IN_TRIANGLE and (triangle.a = sender or
    triangle.b = sender) then
        SetState(BLOCK_TRIANGLE)
        partner ← sender
        SendMsg(sender, CONFIRMED
    break
case DENY_MOVE :
    SetState(ALONE)
    if alone > 1 then
        if ICanMakeTriangle(alone) then break
    CheckSignals(alone)
```

Algoritem 3: Optimizacija - obravnava sporočil 3.del

```
case CONFIRMED :
    if state = INVITE_TRIANGLE then  RotateTriangle()
    else if state = INVITE and partner = other then
        if another > −1 then
            SendMsg(partner, CONFIRMED, sender, index)
            another = −1
        SendMsg(sender, CONFIRMED, index, other)
        SetNewTriangle(partner, other) ReSendEraseSignals();
    else if state = MOVE_TRIANGLE then
        for k ∈ neighbors_incommon[partner] do
            if MoveTriangle(partner, k) then
                break
        end
    else if state = MOVE then
        SetNewTriangle(partner, other)
        ReSendEraseSignals()
    break

case DENY :
    if sender = deny then
        deny ← −1
        break;
    if state = INVITE_TRIANGLE then
        SendMsg(partner, DENY_INVITE)
        SetState(IN_TRIANGLE)
        another ← −1
    else if state = READY and partner = sender then
        SetState(ALONE)
        if alone > 1 then
            if ICanMakeTriangle(alone) then break
        CheckSignals(alone)
    else if state = MOVE_TRIANGLE then
        SetState(IN_TRIANGLE)
        SendMsg(partner, DENY)
        partner ← −1
        another ← −1

case DENY_INVITE :
    if state = INVITE then
        if another > −1 then
            SendMsg(partner, DENY_INVITE)
            another ← −1
        SetState(ALONE)
        CheckSignals(alone)
    break
```

Algoritem 3: Optimizacija - obravnava sporočil 4.del

```
case SWITCH :
    if state = INVITE_TRIANGLE then
        SendMsg(partner, DENY_INVITE)
        SetState(IN_TRIANGLE)
    if state = MOVE_TRIANGLE then
        SendMsg(partner, DENY_MOVE)
        SetState(IN_TRIANGLE)
    SetNewTriangle(partner, other)
    ReSendEraseSignals()
    break;

case LOST :
    if other > −1 then
        alone ← UpdateStates(other, ALONE)
    if state = INVITE_TRIANGLE then
        SendMsg(partner, CONFIRMED, index, another)
        SetTriangle(partner, another)
        ReSendEraseSignals(false)
        break
    else
        if state = MOVE_TRIANGLE then
        SendMsg(partner, DENY_MOVE)
        SetState(ALONE)
        CleanSignals()
        triangle ← null if alone > 1 then
            if ICanMakeTriangle(alone) then break
        CheckSignals(alone)
    break
```

Algoritem 3: Optimizacija - obravnava sporočil 5.del

```
case SIGNAL :
    signal ← msg.index
    if index = signal.index then break
    if signal.mark = IZBRIS then
        DeletePath(signal)
        break
    if state ≠ ALONE then
        FwdSignal(sender, signal, false)
        break
    if state = ALONE and (signal.mark = OZNACI or
    signal.mark = OZNACI2) then
        if signal.index ∈ neighbors and alone = 1 then break
        if (not RotateTriangle(sender)) and signal.mark = OZNACI
        then
            StartMove(sender)

case NEXT_PHASE :
    phases[sender] ← phases[sender] + 1
    if not Empty(waitlist) then
        SendMsg(sender, msgwait.firstElement())
        waitlist.removeFirstElement()
        msgwait.removeFirstElement()
    break
```

2.2.5 O prehodu med fazami

Prehod iz prve faze na naslednjo je enostaven. Vozlišče v začetni fazi izmenjave podatkov o sosedih pošlje na vsa sporočila, ki niso LIST, ALONE ali IN_TRIANGLE, odgovor DENIED. V primeru, da dobi sporočili ALONE ali IN_TRIANGLE, to ustrezno zabeleži v seznamu stanj, ki ga potem upošteva ob prvi sestavi seznama aktualnih sosedov.

Tak način pa ni primeren za naslednji prehod med fazama, saj želimo konsistenco pri pošiljanju sporočila SIGNAL. Pri prehodu med fazo generiranja trikotnikov in optimizacijo zato vozlišče vsem svojim sosedom pošlje sporočilo NEXT_PHASE, ki se obravnava v obeh fazah.

Če bi neko vozlišče rado poslalo sporočilo vozlišču, ki je še v predhodni fazi, počaka na njegovo sporočilo o prehodu v naslednjo fazo in šele takrat pošlje sporočilo. Med čakanjem pa ne sprejema sporočil od drugih vozlišč. Ker lahko vozlišče hkrati pošlje sporočilo več različnim vozliščem, ki še niso v isti fazi kakor to vozlišče, se zato podatki o tem shranijo v ustrezen seznam. Ta se potem postopoma prazni, dokler ni več nobenega vozlišča v seznamu. Nato nadaljujemo izvajanje normalno, tako da sprejemamo sporočila od vseh vozlišč.

2.3 Implementacija

2.3.1 Programska oprema

Algoritem je bil razvit na odprtokodni razvojni platformi Eclipse v programskem jeziku Java s pomočjo orodja DAJ, ki je prosto dostopno na spletu [7]. DAJ je orodje, ki je bilo razvito s ciljem ponuditi splošno dostopno platformo za raziskovanje in izobraževanje na področju porazdeljenih algoritmov. Namenjen je dizajniranju, implementaciji, testiranju, simulaciji in vizualizaciji porazdeljenih sistemov v Javi. Sestavlja ga knjižnica razredov z enostavnim programskim vmesnikom, ki omogoča razvoj porazdeljenih algoritmov, baziranih na modelu pošiljanja sporočil. Omogoča simuliranje sinhronih in asinhronih modelov.

Posamezno računsko vozlišče je simulirano z nitjo, prav tako celotno omrežje, ki povezuje vsa vozlišča. Objekt *scheduler* skrbi, da vsako logično časovno enoto aktivna vozlišča(niti) dobijo enoto procesorskega časa.

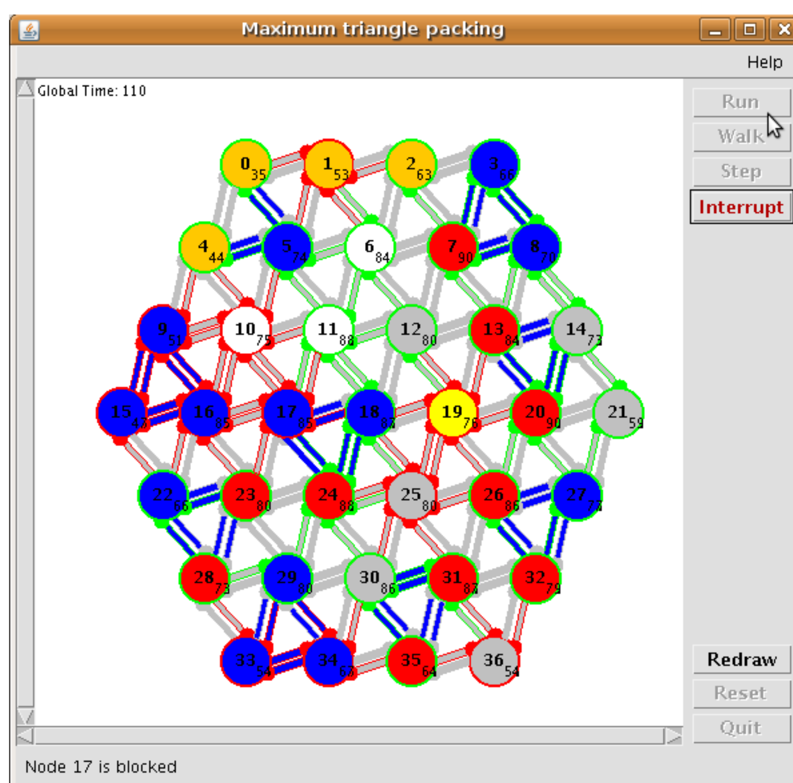
Knjižnica omogoča izvajanje programa za vsak korak posebej(step), počasno izvajanje(walk) in hitro izvajanje(run). Prikazuje globalen čas in aktiven čas za vsako vozlišče. Z barvami pa se ustrezno označuje, kdaj je neko vozlišče

aktivno in kdaj v kanalu čaka kakšno sporočilo.

Ker je koda knjižnice DAJ prosto dostopna, jo je možno za svoje potrebe tudi poljubno dopolnjevati. To smo izkoristili pri vizualizaciji modela, kjer je bila dodana še možnost barvanja posameznega vozlišča in povezav oz. kanalov med njimi, kar je močno olajšalo razvoj algoritma in razhroščevanje le tega. Prav tako je bilo dodano štetje sporočil in računanje povprečnega aktivnega časa za vozlišča. Primer vizualizacije z uporabo knjižnice DAJ je prikazan na sliki 2.16.

Orodje zna prepoznati, kdaj se je izvajanje končalo, čeprav vozlišča sama tega mogoče ne prepoznajo (odvisno od algoritma).

Eden od problemov, na katerega sem naletela pri uporabi te knjižnice, je ta, da selekcija kanala, iz katerega se bo prebralo naslednje sporočilo ni pravična (kar sem napačno predvidevala). Kanal se namreč izbere glede na njegovo mesto v seznamu kanalov. To pomeni, da bodo imeli vedno prednost tisti



Slika 2.16: Primer vizualizacije v orodju DAJ

kanali, ki vsebujejo sporočilo, ki imajo manjši indeks v seznamu.

Pogrešala sem tudi možnost blokiranja nekega komunikacijskega kanala, s čimer bi odklopili vozlišča, s katerimi ne bi radi komunicirali. Tako ne bi nastal odvečen promet pri pošiljanju sporočila, ki je namenjeno vsem. Obe pomanjkljivosti je mogoče z ustreznimi prijemi preprosto zaobiti.

Glavno pomankljivost samega orodja DAJ pa vidim v tem, da ne omogoča direktnega prenosa razvitega algoritma na obstoječ porazdeljen sistem. Zato bi bilo potrebno dodatno delo, če hočemo algoritem pognati na resničnem porazdeljenem sistemu. Po drugi strani pa je ravno zaradi tega primeren za uporabo, saj olajša sam razvoj in omogoča, da se osredotočimo le na reševanje določenega problema.

2.3.2 Parametri

Program lahko poženemo z naslednjimi vhodnimi parametri:

- $filename$ - Ime datoteke, kjer je v vsaki vrstici zapisana ena povezava: s presledkom ločeni dve števili, ki predstavljata indeksa vozlišč, ki predstavljata povezavo. Privzeto bo graf tak, kot ga prikazuje slika 2.17.

- *opt* - Vrednost 0 pomeni, da bo program končal izvajanje po koncu faze naključnega generiranja trikotnikov. Vrednost 1 pomeni izvajanje programa v vseh fazah. Privzeta vrednost je 1.

- *path* - Vrednost 1 pomeni, da bo signal potoval samo po zrcalni poti. Vrednost 2 pomeni, da bo signal potoval po obeh poteh. Privzeta je vrednost 2. Vrednost 1 je priporočljiva, ko je stopnja vozlišč velika in je omrežje preveč zasičeno s signali.

- $depth1$ - Največja razdalja signala (število označenih vozlišč) pri zrcalni poti. Privzeta vrednost je neskončno(-1). Vrednost je priporočljivo omejiti pri grafih z velikim številom vozlišč in povezav.

- $depth2$ - Največja razdalja signala (število označenih vozlišč) pri zasučni poti. Privzeta vrednost je neskončno(-1). Vrednost je priporočljivo omejiti pri grafih z velikim številom vozlišč in povezav, če *path* ni 1.

2.3.3 Rezultat

Ob vsakem generiranju novega trikotnika se na standardni izhod zapiše vrstica
$TRIKOTNIK\ u\ v\ w$
, kjer so u,v in w indeksi posameznih vozlišč, ki sestavljajo trikotnik.
Pri spremembi trikotnika (ob izvedbi neke operacije) se zapišeta vrstici
$\#TRIKOTNIK\ u_0\ v_0\ w_0$
TRIKOTNIK u_1 v_1 w_1
, kjer so u_0, v_0 in w_0 indeksi posameznih vozlišč, ki sestavljajo trikotnik, nad katerim se izvede ena od operacij, u_1,v_1 in w_1 pa indeksi vozlišč, ki sestavljajo novo nastali trikotnik.

Poleg podatkov na standardnem izhodu je rezultat razviden tudi na vizualizaciji. Tam so vozlišča in povezave, ki so del trikotnikov, pobarvana z modro barvo.

2.3.4 Problemi

V splošnem velja, da je programiranje za porazdeljene sisteme zahtevnejše kot programiranje, ki predpostavlja zaporedno izvajanje operacij na enem samem procesorju. Nekatere probleme, s katerimi smo se srečevali tekom tega dela (ki se ne nanašajo na programsko opremo) in so povzročili največ težav zato na tem mestu posebej izpostavimo:

- smrtni objem (dead lock) - vsako vozlišče v zaporedju $(v_1, ..., v_n)$ čaka na naslednje vozlišče in zadnje vozlišče čaka na prvo vozlišče. Vozlišče zato ne sme čakati samo na sporočilo, pač pa mora medtem podati odgovor vozlišču, ki ga pričakuje od njega.

- časovni zamiki - nekatera sporočila pridejo do sprejemnika prej,nekatera pa kasneje. Lahko se zgodi, da sprejemnik dobi sporočilo, ki ni več aktualno in ga lahko narobe razume. Zato je potrebno paziti, da do tega sploh ne more priti, najbolje tako, da vozlišče vedno počaka na pričakovano sporočilo preden začne izvajati druge akcije.

- zastareli podatki - na podatke, ki jih imamo o stanju nekega drugega vozlišča, se ne moremo zanesti, če vozlišče ni blokirano. V vsakem trenutku namreč lahko pride do spremembe, obvestilo o spremembi pa lahko pride mnogo kasneje. Zato je potrebno, kadar se moramo zanesti na te podatke, vozlišče blokirati in šele nato izvesti določene akcije.

- ustavljivost - vozlišče naj bi znalo prepoznati, kdaj se je izvajanje končalo, kar pa ni tako enostavno, saj nima globalne slike o omrežju.

- ciklanje - operacije, ki se ponavljajo v ciklu v neskončnost, moramo ustrezno zaznati in preprečiti. Uporabimo lahko dodatne spremenljivke, kamor shranimo ustrezne podatke iz preteklega dogajanja.

Ker naš porazdeljen model predpostavlja, da ne prihaja do izpadov vozlišč ter da deluje pravilno, se ni bilo potrebno ukvarjati s toleranco napak, kar tudi spada med glavne probleme, s katerimi se srečujejo(mo) pri porazdeljenem programiranju.

2.4 Meritve

Algoritem je bil pognan za nekatere specifične primere grafov, ki so bili v pomoč pri samemu razvoju algoritma ter za naključno generirane grafe z različnim številom vozlišč in povezav.

Za vsak primer grafa smo algoritem pognali večkrat z različnimi parametri ter opazovali sledeče podatke:

- število trikotnikov pri naključni rešitvi (N)

- največje in najmanjše število trikotnikov pri naključni rešitvi

- število trikotnikov po optimizaciji (**O**)

- največje in najmanjše število trikotnikov po optimizaciji

- optimalna rešitev

- globalni čas (g-čas)

- povprečen aktivni čas vozlišč (a-čas)

- celotno število sporočil, pretečenih po kanalih (g-msg)

- povprečno število sporočil, ki jih je prejelo vozlišče (msg)

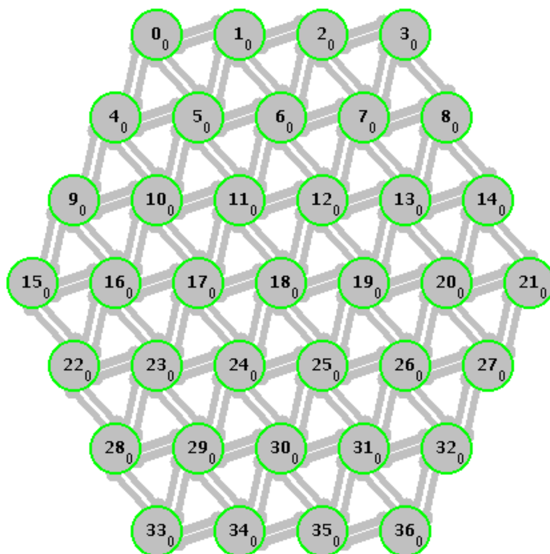
2.4.1 Graf 1

Najprej smo opravili meritve na primeru grafa, ki je bil v pomoč pri razvoju algoritma. Graf je prikazan na sliki 2.17 in ima sledeče lastnosti:

- **Optimalna rešitev je pokritje z 12 trikotniki.**
- Število vozlišč je 37.
- Število povezav je 90.
- Maksimalna stopnja vozlišča je 6.
- Minimalna stopnja vzozlišča je 3.
- Vsako vozlišče in povezava sta lahko del nekega trikotnika.

Opravili smo 10 meritev s privzetimi parametri. Rezultat je bil optimalen v 9 primerih, pri enem pa je bil rezultat pokritev z 11 trikotniki. V tabelah 2.6 in 2.7 so prikazani še nekateri ostali relevanti podatki:

Komentar
Algoritem se je na tem konkretnem primeru dobro izkazal, kar smo tudi predvidevali saj je bil to eden od primerov, ki je uporabljen pri razvoju. Zanimivo



Slika 2.17: Graf 1

	N	**O**	Št.pojavitev	g-čas	a-čas	g-msg	msg
Povprečje	9.3	**11.9**		359.1	190.6	2950.9	79.2
Največja naključna reš.	**10**	12	5	341.2	175.6	2752.8	73.2
Najmanjša naključan reš.	**7**	12	1	471	234	3611	97

Tabela 2.6: Graf 1 - Rezultati

	N	g-čas	g-msg	msg
Največji g-čas	10	**515**	4202	113
Najmanjši g-čas	10	**201**	3611	97

Tabela 2.7: Graf 1 - Rezultati

je, da je število trikotnikov po naključni postavitvi enako v primerih, ko algoritem porabi največ in najmanj časa.

Primerjajmo sedaj povprečen aktivni čas vozlišč z globalnim časom: $\frac{190.6}{359.1} = 0.53$. To pomeni, da je vozlišče v 47 % časa neaktivno in čaka na naslednje sporočilo, ki naj bi prišlo po enem od kanalov. Razlog za tako slab izkoristek računskega vozlišča je v nizki časovni kompleksnosti operacij, ki jih izvede vozlišče ob sprejemu nekega sporočila. Če primerjamo še povprečen aktivni čas za posamezno vozlišče in povprečno število sporočil, ki jih je prejelo, dobimo $\frac{190.6}{79.2} = 2.41$. To pomeni, da vozlišče v povprečju porabi 2.41 enot logičnega časa za izvedbo operacij, ki predstavljajo odziv na neko sporočilo.

Oglejmi si še, koliko sporočil je povprečno preteklo po nekem kanalu pri neki povprečni rešitvi: $\frac{2950.9}{180} = 16.39$. To pomeni, da je neko vozlišče v povprečju poslalo nekemu drugemu vozlišču malo več kot 16 sporočil.

2.4.2 Graf 2

Naslednja meritev je bila opravljena na grafu, ki ima takšno 'zgradbo', kot predhodni graf, le da ima 217 vozlišč. Slika 2.18 prikazuje tak graf po tem, ko je bilo izvajanje programa končano, rešitev pa optimalna.

Graf ima sledeče lastnosti:

- **Optimalna rešitev je pokritje s 72 trikotniki.**
- Število vozlišč je 217.
- Število povezav je 1200.
- Maksimalna stopnja vozlišča je 6.

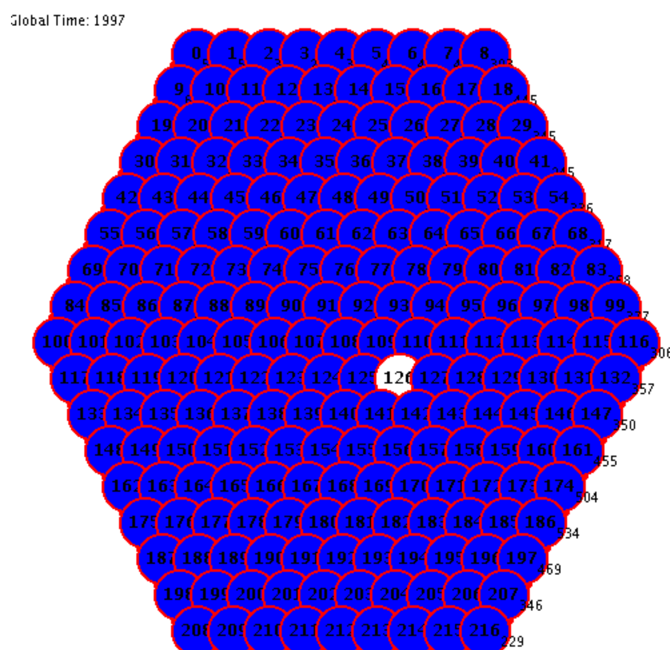
- Minimalna stopnja vozlišča je 3.
- Vsako vozlišče in povezava sta lahko del nekega trikotnika.

Opravili smo 15 meritev. 5 s privzetimi parametri, 5 z globino obeh poti 5 ($depth1 = 5$, $depth2 = 5$), ter 5 z globino zrcalne poti 5 brez možnosti zasučne poti ($depth1 = 5$, $path = 1$). Rezultati so v tabelah 2.8, 2.9 in 2.10. Podatki, na katere bi radi še posebej opozorili so pisani s krepko pisavo.

	N	**O**	št.pojavitev	g-čas	msg
Povprečje	52	**70.8**		1747.8	253.6
Največja naključna r.	**58**	70	1	2555	352
Najmanjša naključna r.	**48**	71	1	1369	247
Največja optimalna r.	52	**72**	1	1994	244
Namanjša optimalna r.	57.5	**70**	2	1874.5	273

Tabela 2.8: Graf 2 - Rezultati s privzetimi parametri

Komentar
Najmanjše število trikotnikov po koncu izvajanja je bilo od vseh treh primerov



Slika 2.18: Graf 2 - Po končanem izvajanju z optimalno rešitvijo

	N	**O**	št.pojavitev	g-čas	msg
Povprečje	57.4	**70.2**		1328.2	210
Največja naključna r.	**60**	71	1	1140	213
Najmanjša naključna r.	**51**	70	1	1475	212
Največja optimalna r.	59	**71**	1	1994	214
Najmanjša optimalna r.	57	**70**	4	1259.8	209

Tabela 2.9: Graf 2 - Rezultati z globino poti 5.

	N	**O**	št.pojavitev	g-čas	msg
Povprečje	54	**69.8**		804.8	90.8
Največja naključna r.	**56**	69	1	671	84
Najmanjša naključna r.	**52**	69	1	949	117
Največja optimalna r.	53	**72**	1	1163	88
Najmanjša optimalna r.	55	**68**	1	568	77

Tabela 2.10: Graf 2 - Rezultati z označevanjem le zrcalnih poti z globino 5.

68. To je torej za $\frac{1}{18}$ slabše od optimalne vrednosti.

Opazimo lahko, kako se optimalna rešitev slabša z omejevanjem poti označevanja. Občutno pa se zmanjša tudi povprečen globalni čas. V tretjem primeru se je izvajanje v povprečju ustavilo približno dvakrat prej, kot v prvem primeru. Rezultat pa, kot je razvidno iz tabel, ni bistveno slabši.

Povprečno število sporočil, ki ga neko vozlišče pošlje drugemu vozlišču, je v prvem primeru skoraj 23, v drugem primeru skoraj 19, v tretjem primeru pa malo več kot 8. Pri tem moramo upoštevati, da število sporočil za posamezen kanal močno variira in ni enakomerno porazdeljeno.

Opomba
Ker zaradi vizualizacije izvajanje takšnega programa lahko poteka izredno počasi, je potrebno vizualizacijo minimizirati, da se program izvede v sprejemljivem času. Včasih je program izvajanje predčasno zaključil, 'Java runtime Environment' pa je javil napako. Verjetno zaradi prevelikega števila niti.

2.4.3 Graf 3

Graf 3 ima naslednje lastnosti:

- **Optimalno število trikotnikov je 20.**
- Število vozlišč je 60.

- Število povezav je 150.
- Maksimalna stopnja vozlišča je 4.
- Vsaka povezava lahko pripada nekemu trikotniku.

Naredili smo 10 meritev s privzetimi parametri. Za vsako meritev smo graf ponovno generirali, tako da so bili grafi vedno različni. Rešitev je bila vedno optimalna. Ostali zanimivi podatki pa so zapisani v tabeli 2.11

	N	**O**	št.pojavitev	g-čas	msg
Povprečje	16.4	**20**		303	46.9
Največja naključna r.	**19**	20	2	195.5	352
Najmanjša naključna r.	**13**	20	1	456	247

Tabela 2.11: Graf 3 - Rezultati

2.4.4 Graf 4

Graf 4 je naključno generiran graf, za katerega naj bi veljalo, da nima prevelike stopnje vozlišča, prav tako pa ne premajhne in ima naslednje lastnosti:

- **Optimalna rešitev je pokritje s 33 trikotniki.**
- Število vozlišč je 101.
- Povprečna stopnja vozlišča je 8.
- Vsako vozlišče in povezava sta lahko del nekega trikotnika.

Opravili smo 10 meritev z **globino obeh poti 3** ($depth1 = 3$, $depth2 = 3$). Rezultat je bil optimalen v enem primeru, najmanjše število trikotnikov po koncu izvajanja pa je bilo 30. To je torej za $\frac{1}{11}$ slabše od optimalne vrednosti. V tabeli 2.4.4 so prikazani še nekateri ostali relevantni podatki.

	N	**O**	g-čas
Povprečje	25.5	**31.3**	2076

Tabela 2.12: Graf 4 - Rezultati

Komentar
Rezultat je bil glede na omejitev poti zelo dober.

Poglavje 3

Zaključek

V tej diplomski nalogi je opisan porazdeljeni algoritem za reševanje optimizacijskega problema maksimalnega pokritja grafa s trikotniki. Rešitev, ki jo algoritem najde, v splošnem ni optimalna, je pa glede na rezultate zadovoljiva. Ob tem je potrebno poudariti, da takega pristopa k rešitvi tega problema ni bilo zaslediti kje drugje ter da je rešitev predvidena za vse grafe.

Logičen naslednji korak v nadaljevanju bi bil pognati takšen algoritem na obstoječemu porazdeljenemu sistemu, poleg tega pa dodati postopke za ugotavljanje ustavitve procesa ter zagotoviti toleranco izpadov. Zanimivo bi bilo tudi realizirati možnost dodajanja novega vozlišča v graf v katerikoli fazi algoritma.

Kot je razvidno iz meritev, je obremenitev komunikacijskih kanalov mnogo večja od obremenitve posameznih vozlišč. Da bi izkoristili moč posameznega računskega vozlišča (procesorja) in razbremenili komunikacijske kanale, bi graf lahko razdelili na podgrafe in potem problem na posameznem podgrafu reševali kar znotraj enega vozlišča. Seveda bi to prineslo nove probleme, kot je, kako razdeliti graf na podgrafe, da bo celoten sistem čimbolj uravnoteženo obremenjen in odločitev o tem, kako se problema lotiti na posameznem podgrafu in tako naprej.

Prav tako vidim prostor za izboljšave v optimizacijski fazi algoritma, kjer bi z nekaterimi ukrepi lahko rezultate še izboljšali. Že sama implementacija algoritma ni optimalna, prav tako bi verjetno lahko zmanjšali število vseh sporočil in stanj. Vendar cilj ni bil najbolj optimalen algoritem, pač pa bolj pregleden in čimbolj razumljiv, predvsem pa algoritem, ki bi vrnil pravilno rešitev problema, ki bi bila boljša od neke naključne rešitve. To pa je bilo doseženo.

Samo porazdeljeno programiranje se je izkazalo za zanimiv in miselno zahteven proces, ki zahteva čim bolj analitičen pristop k reševanju problema, tudi takrat, ko se neka ideja zdi na prvi pogled čisto enostavna. Hitro lahko namreč pride do situacij, ki niso bile predvidene, saj je porazdeljen sistem izredno gibljiv in kompleksen. Tudi sam problem je bil za takšen način reševanja ustrezen, saj ga je bilo lahko vizualizirati in tako spremljati dogajanje, ob tem pa spoznavati lastnosti takšnega programiranja.

Ena od zanimivih lastnosti, ki jo razvoj porazdeljenega algoritma lahko prinese, je tudi ta, da smo primorani pogledati na problem z drugačnega zornega kota, kot pa, če bi hoteli razviti algoritem za zaporedno izvajanje enega samega programa. Ta drugačna perspektiva lahko pripomore tudi k boljši zaporedni rešitvi.

Dodatek A

Implementacija v Javi

A.1 Funkcije

```
private void SendMsg(int channel, Msg msg) {
  if (channel == -1) //all
    out().send(msg);
  else
    if (this.phase == 1) {
      if (this.phases[channel] == this.phase) {
        out(channel).send(msg);
    } else {
        this.iwait.addElement(channel);
        this.msgwait.addElement(msg);
    }
  } else
    out(channel).send(msg);
}

private void UpdateStates(int isender, int state) {
   this.states[isender] = state;
}

private void SetState(int s) {
  int tmp = this.state;
  this.state = s;
  switch (this.state) {
  case IN_TRIANGLE:
       if ((tmp != SIGNAL) && (tmp != MOVE\_TRIANGLE)
                         && (tmp != INVITE\_TRIANGLE) && (tmp != BLOCK\_TRIANGLE)) {
      SendMsg(-1,new Msg(IN_TRIANGLE, triangle.a, triangle.b));
      if (old != null)
```

```
          old.clear();
        this.ipar = -1;
        this.ianother = -1;
       }
       break;
  case ALONE:
       ianother = -1;
       ipar = -1;
       if ((tmp != SIGNAL) && (tmp != MOVE) && (tmp != READY)
                        && (tmp != READY) && (tmp != INVITE))
      SendMsg(-1,new Msg(ALONE));
    if (old != null)
      old.clear();
       break;
  }
}

/* izbere naključnega aktualnega soseda */
private int getRandomPromisingnode() {
  int i = Math.abs(random.nextInt() % promising.size());
  return n.indexOf(promising.elementAt(i));
}

/* odstrani vse neuporabne sosede in inicializira aktualne sosede */
private void UpdateNeighbors() {
  Vector<Integer> tmp = new Vector<Integer>();
  for (int i = 0; i < n.size(); i++) {
   if (incommon.get(i) != null)
    if (incommon.get(i).size() > 0)
     tmp.addElement(n.elementAt(i));
  }

  promising = new Vector<Integer>(tmp);
  for (int i = 0; i < n.size(); i++)
  {
   if ((this.states[i] == IN_TRIANGLE)||(this.states[i] == ALONE))
    UpdatePromisingNodes(n.elementAt(i));
  }
}

/* posodobi seznam aktualnih sosedov */
private void UpdatePromisingNeighbors(int k) {
  if (promising == null)
   return;
  if (promising.contains(k)) {
    promising.removeElement(k);
    for (int w : incommon.get(n.indexOf(k))) {
```

```
      boolean all_states = true;
      for (int v : incommon.get(n.indexOf(w)))
        if (promising.contains(v))
         all_states = false;
        if (all_states)
         promising.removeElement(w);
    }
  }
}

/* pošlje vabila k trikotniku vsem skupnim aktualnim sosedom */
private boolean SendInvitations(int isender) {
  waiting.removeAllElements();

  this.ipar = isender;
  if (!promising.contains(n.elementAt(isender)))
   return false;

  for (int k : incommon.get(isender)) {
    if (promising.contains(k)) {
      SendMsg(n.indexOf(k), new Msg(JOIN_US, this.index, this.n
       .elementAt(isender)));
      waiting.addElement(n.indexOf(k));
      SetState(JOIN_US);
    }
  }
  return true;
}

/* vrne število osamljeni sosedov */
private int CalcAlone(){
  int alone = 0;
  for (int i = 0; i < states.length; i++)
    if (this.states[i] == ALONE)
      alone++;
  return alone;
}

private void SendSignal() {
  Signal signal = null;

  for (Signal sig : signals)
    if (sig.number == 1) {
      signal = sig;
      break;
    }
```

```
  if (signal == null) {
    signal = new Signal(-1, this.index, 1, -1);
    signals.addElement(signal);
  }

  if (signal.index != this.index) {
    signal.index = index;
    signal.fwd.clear();
  }

  for (int i = 0; i < n.size(); i++) {
    if (states[i] == IN_TRIANGLE) {
      if (signal.fwd.contains(i))
      continue;
      signal.fwd.addElement(i);
      SendMsg(i, new Msg(SIGNAL, new Signal(-1, this.index, OZNAČI, 0)));
    }
  }
}

/* zacetek operacije zasuka */
private boolean StartRotate(int i) {
  for (int k : incommon.get(i)) {
    if (states[n.indexOf(k)] == ALONE) {
      this.ipar = n.indexOf(k);
      SetState(READY);
      this.ianother = i;
      SendMsg(ipar, new Msg(READY));
      return true;
    }
  }
  return false;
}

private int UpdateStates(int isender, int state, int alone)
{
  if (state == IN_TRIANGLE)
  {
    if (this.states[isender] == ALONE)
      alone--;
  }else
  if (state == ALONE){
    if (this.states[isender] == IN_TRIANGLE) {
      alone++;
      UpdateTriangles(false, isender, -1, -1);
    }
  }
```

```
  this.states[isender] = state;
  return alone;
}

private boolean GetTriangle(boolean compare) {
  if (compare)
    if (this.index > n.elementAt(ipar)) {
      ianother = -1;
      return true;
    }
    int i = -1;
    for (int k : this.incommon.get(ipar)) {
      if (states[n.indexOf(k)] == ALONE)
      i = n.indexOf(k);
    }
    if (i == -1) {
      if ((ianother > -1)&& (incommon.get(ipar).contains(n.elementAt(ianother))))
        i = ianother;
      else
        return false;
  }
  last = i;
  SendMsg(i, new Msg(INVITE, this.index, n.elementAt(this.ipar)));
  this.ianother = i;
  return true;
}

private void CheckSignals(int alone) {
  Signal min = null;
  do {
    for (Signal sig : signals) {
      if (sig.number == OZNACI)
         if ((min == null) || (min.ichannel == last))
           min = sig;
         else if ((min.distance < sig.distance)&& (min.ichannel != last))
            min = sig;
      if (alone > 0)
        if (sig.number == OZNACI2)
          if ((min == null) || (min.ichannel == last)|| (min.distance < sig.distance))
            for (int k : incommon.get(sig.ichannel))
              if (states[n.indexOf(k)] == ALONE) {
                min = sig;
                break;
              }
    }
    if (min == null)
     break;
```

```
    if (alone > 0) {
      if (RotateTriangle(min.ichannel)) {
        last = min.ichannel;
        return;
      }
    }
    if (min.number == 2) {
      if (triangles[min.ichannel][0] != -1)
        if (StartMove(min.ichannel)) {
          last = min.ichannel;
          return;
        }
    }
    for (int k : min.fwd)
      SendMsg(k, new Msg(SIGNAL, new Signal(-1, min.index, IZBRIS, -1)));

    signals.removeElement(min);
    boolean in = false;
    for (Signal s:old)
    {
      if (s.index == min.index && s.number == min.number && s.ichannel == min.ichannel)
      {
        if (s.distance > min.distance)
           s.distance = min.distance;
        in = true;
      }
    }
    if (!in){
      min.fwd = null;
      old.addElement(min);
    }

        min = null;
    } while (min != null);
    last = -1;
    SendSignal();
}

private void MoveTriangle(sender) {
      int lost = -1;
   int switch = -1;
    for (int k : incommon.get(sender)) {
    if (triangle.a == k) {
      lost = triangle.b;
      switch = triangle.a;
      break;
    }
```

```
    if (triangle.b == k) {
     lost = triangle.a;
     switch = triangle.b;
     break;
   }

   System.out.println("#TRIANGLE" + " " + this.index + " " + triangle.a
        + " " + triangle.b);
   SendMsg(n.indexOf(sender), new Msg(SWITCH, this.index, switch));
   SendMsg(n.indexOf(switch), new Msg(SWITCH, this.index, sender));
   SendMsg(n.indexOf(lost), new Msg(this.LOST, this.index, -1));

   this.triangle = new Triangle(a, b, n.indexOf(a), n.indexOf(b));
   System.out.println(" TRIANGLE" + " " + this.index + " " + triangle.a
        + " " + triangle.b);

   SetState(IN_TRIANGLE);
   ReSendEraseSignals();
}

private void ReSendEraseSignals() {
    for (int i = 0; i < signals.size(); i++) {
        Signal sig = signals.elementAt(i);
        if ((sig.number == OZNACI) || (sig.number == OZNACI2)) {
          for (int k : sig.fwd)
            SendMsg(k, new Msg(SIGNAL, new Signal(-1, sig.index, IZBRIS, -1)));

          if ((sig.index != triangle.a)&&(sig.index != triangle.b))
          {
            sig.fwd.clear();
            FwdSignal(sig.ichannel, sig, true);
          }
          else
          {
            signals.remove(sig);
            i--;
          }
          continue;
        }
        for (int k : sig.fwd)
          SendMsg(k, new Msg(SIGNAL, new Signal(-1, sig.index, IZBRIS, -1)));
        signals.remove(sig);
        i--;
    }
}

private void CleanSignals(){
```

```
  for (int i = 0; i < signals.size(); i++) {
    Signal sig = signals.elementAt(i);
    if ((sig.number == OZNACI)||(sig.number == OZNACI2))
      for (int k : sig.fwd)
        SendMsg(k, new Msg(SIGNAL, new Signal(-1, sig.index, IZBRIS, -1)));
    else if ((sig.number == PREHOD2)||(sig.number == PREHOD3))
    {
      for (int k : sig.fwd)
      SendMsg(k, new Msg(SIGNAL, new Signal(-1, sig.index, IZBRIS, -1)));
      signals.remove(sig);
      i--;
    }
  }
}

private void DeletePath(int isender, Signal signal){
  for (int i = 0; i < signals.size(); i++) {
    Signal sig = signals.elementAt(i);
    if ((sig.ichannel == isender)&&(sig.index == signal.index)){
      for (int k : sig.fwd)
        SendMsg(k, new Msg(SIGNAL, new Signal(-1, sig.index,IZBRIS, -1)));
      boolean in = false;
      for (Signal s:old){
        if (s.index == sig.index && s.number == sig.number && s.ichannel == sig.ichannel){
          if (s.distance > sig.distance)
            s.distance = sig.distance;
            in = true;
        }
      }
      if (!in){
        sig.fwd = null;
        old.addElement(sig);
      }
      signals.removeElementAt(i);
      i--;
    }
  }
}

private void FwdSignal(int isender, Signal signal, boolean resend) {
 if (!resend)
   for (Signal sig:old)
   {
     if ((sig.index == signal.index)&&(sig.number == signal.number)
             &&(sig.distance <= signal.distance)&&(sig.ichannel == isender))
       return;
   }
```

```
for (int i = 0; i < signals.size(); i++) {
  Signal sig = signals.elementAt(i);
  if (((sig.number == signal.number) || ((signal.number == OZNACI)
     && (sig.number == OZNACI))) && (sig.ichannel == isender)) {
    if (sig.index == signal.index)
      if (sig.distance <= signal.distance)
      return;
    else {
      signals.removeElementAt(i--);
      continue;
    }
  }
 }
 if (!resend) {
   signal = new Signal(isender, signal.index, signal.number,signal.distance);
   signals.addElement(signal);
 }
 if (triangle == null)
  return;
 if (signal.number == PREHOD2) {
  for (int k : incommon.get(isender)) {
   if (k == triangle.a)
    continue;
   if (k == triangle.b)
    continue;
   signal.fwd.addElement(n.indexOf(k));
   SendMsg(n.indexOf(k), new Msg(SIGNAL, new Signal(-1,
     signal.index, OZNACI, signal.distance + 1)));
  }
  if (this.path == 2)
   TwoPathSignal(isender,signal);

 } else if (signal.number == OZNACI) {
  signal.fwd.addElement(triangle.a > triangle.b ? triangle.ca: triangle.cb);
  SendMsg(triangle.a > triangle.b ? triangle.ca : triangle.cb,
    new Msg(SIGNAL, new Signal(-1, signal.index, PREHOD1, signal.distance + 1)));
 } else if (signal.number == PREHOD1) {
  signal.fwd.addElement(triangle.ca == isender ? triangle.cb: triangle.ca);
  SendMsg(triangle.ca == isender ? triangle.cb : triangle.ca,
    new Msg(SIGNAL, new Signal(-1, signal.index, PREHOD2, signal.distance + 1)));
  if (this.path == 2)
   TwoPathSignal(isender,signal);
 }
 else if (signal.number == OZNACI2) {
  if (incommon.get(isender).contains(triangle.a)) {
   if (triangle.a > triangle.b){
    boolean send = false;
```

```
     for (Signal sig : signals)
      if ((sig.ichannel == triangle.cb)&& ((sig.number == PREHOD2) || (sig.number == PREHOD1)))
        send = true;
      if (send) {
        signal.fwd.addElement(triangle.cb);
        SendMsg(triangle.cb, new Msg(SIGNAL, new Signal(-1,signal.index, PREHOD3,
              signal.distance + 2)));
     }
    }
   }
   if (incommon.get(isender).contains(triangle.b)) {
    if (triangle.b > triangle.a){
     boolean send = false;
     for (Signal sig : signals)
      if ((sig.ichannel == triangle.ca)&& ((sig.number == PREHOD1) || (sig.number == PREHOD2)))
       send = true;
     if (send) {
      signal.fwd.addElement(triangle.ca);
      SendMsg(triangle.ca, new Msg(SIGNAL, new Signal(-1, signal.index, PREHOD3,
              signal.distance + 2)));
     }
    }
   }
  }
  else if (signal.number == PREHOD3) {
   for (int i= 0; i < n.size(); i++)
   {
     if (incommon.get(i) != null)
     for (int k:incommon.get(i))
     {
       if (k > n.elementAt(i))continue;
       if (k == triangle.a) continue;
       if (k == triangle.b) continue;
       signal.fwd.addElement(i);
       SendMsg(i, new Msg(SIGNAL, new Signal(-1, signal.index, OZNACI2, signal.distance + 1)));
       break;
   }
  }
 }
}

/* oznacevanje zasucne poti */
private void TwoPathSignal(int isender, Signal sig) {
  int ic = -1;
  if (sig.number == PREHOD1)
    ic = (isender == triangle.ca ? triangle.cb : triangle.ca);
  else if (sig.number == 1)
```

```
    ic = (isender == triangle.ca ? triangle.ca : triangle.cb);
  for (int i = 0; i < n.size(); i++) {
    if (incommon.get(i) == null)
      continue;
    if (i == triangle.ca)
      continue;
    if (i == triangle.cb)
      continue;
    if (incommon.get(ic).contains(n.elementAt(i)))
      continue;
    sig.fwd.addElement(i);
    SendMsg(i, new Msg(SIGNAL, new Signal(-1, sig.index, OZNACI2, sig.distance + 1)));
  }
}

/* zacetek operacije zrcaljenja trikotnika */
private boolean StartMove(int i) {
  if (states[i] == ALONE)
    return false;
  if (triangles[i][0] == -1)
    return false;
  this.ipar = i;
  SendMsg(i, new Msg(MOVE, this.index));
  SetState(MOVE);
  last = i;
  return true;
}

private void RotateTriangle() {
  boolean change = false;
  if (triangle != null) {
    change = true;
 System.out.println("#TRIANGLE" + " " + this.index + " "
+ triangle.a + " " + triangle.b);
    SendMsg(triangle.ca, new Msg(this.LOST, this.index, triangle.b));
    SendMsg(triangle.cb, new Msg(this.LOST, this.index, triangle.a));
  }

  triangle = new Triangle(n.elementAt(ipar), n.elementAt(ianother), ipar,
ianother);
  System.out.println(" TRIANGLE" + " " + this.index + " " + triangle.a
+ " " + triangle.b);

  SendMsg(ipar, new Msg(CONFIRMED, this.index, n.elementAt(ianother)));
  SetState(IN_TRIANGLE);
  ReSendEraseSignals(change);
}
```

Slike

1.1 Primer optimalne rešitve problema 5
1.2 Primer neoptimalne rešitve problema 5
1.3 Primer neveljavne rešitve 5

2.1 Faze algoritma 9
2.2 Primer grafa po fazi izmenjave podatkov o sosedih 10
2.3 Postopek generiranja trikotnika 14
2.4 Avtomat stanj v fazi naključnega generiranja trikotnikov 15
2.5 Primer naključnga pokritja grafa s trikotniki 16
2.6 Operacija zrcaljenja trikotnika 20
2.7 Operacija zasuka trikotnika 20
2.8 Prikaz dogodka potencialnega trikotnika 21
2.9 Zrcalna pot 22
2.10 Zasučna pot 22
2.11 Označevanje vozlišč 26
2.12 Postopek operacije zrcaljenja trikotnika 28
2.13 Postopek operacije zasuka trikotnika 29
2.14 Avtomat stanj v fazi optimizacije 31
2.15 Sprememba poti po operaciji zasuka 32
2.16 Primer vizualizacije v orodju DAJ 41
2.17 Graf 1 45
2.18 Graf 2 - Po končanem izvajanju z optimalno rešitvijo 47

Tabele

2.1 Tabela stanj v fazi naključnega generiranja trikotnikov 12
2.2 Tabela sporočil v fazi naključnega generiranja trikotnikov 12
2.3 Pravila za posredovanje signala 25
2.4 Tabela stanj v fazi optimizacije 26
2.5 Tabela sporočil v fazi optimizacije 27
2.6 Graf 1 - Rezultati 1 46
2.7 Graf 1 - Rezultati 2 46
2.8 Graf 2 - Rezultati s privzetimi parametri 47
2.9 Graf 2 - Rezultati s globino poti 5 48
2.10 Graf 2 - Rezultati z označevanjem le zrcalnih poti z globino 5 . 48
2.11 Graf 3 - Rezultati 49
2.12 Graf 4 - Rezultati 49

Algoritmi

1 Izmenjava podatkov o sosedih 11
2 Algoritem 2: Naključno generiranje trikotnikov 17
3 Algoritem 2: Naključno generiranje trikotnikov - obravnava sporočil 1.del 18
4 Algoritem 2: Naključno generiranje trikotnikov - obravnava sporočil 2.del 19
5 Algoritem 3: Optimizacija naključne rešitve 34
6 Algoritem 3: Optimizacija - obravnava sporočil 1.del 35
7 Algoritem 3: Optimizacija - obravnava sporočil 2.del 36
8 Algoritem 3: Optimizacija - obravnava sporočil 3.del 37
9 Algoritem 3: Optimizacija - obravnava sporočil 4.del 38
10 Algoritem 3: Optimizacija - obravnava sporočil 5.del 39

Literatura

[1] M. R. Garey, D. S. Johnson, “Computers and Intractability: A Guide to the Theory of NP-Completeness,” Freeman, New York, 1979.

[2] R. Hassin, S. Rubinstein, “An approximation algorithm for maximum triangle packing,” *Discrete Applied Mathematics*, vol. 154, str. 971-979, 2006.

[3] G. Manić, Y. Wakabayashi, “Packing triangles in low degree graphs and indifference graphs,” v *2005 European Conference on Combinatorics, Graph Theory and Applications (EuroComb ’05), Stefan Felsner (ed.), Discrete Mathematics and Theoretical Computer Science Proceedings AE*, str. 251-256, 2005.

[4] C.A.J. Hurkens, A. Schrijver, “On the size of systems of sets every t of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems,” *SIAM J. Discrete Math.*, 2(1):68–72, 1989.

[5] A. Černivec, “Ogrodje za realizacijo porazdeljenih algoritmov v sistemu PlanetLab,” Diplomsko delo, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana 2007.

[6] K. Mani Chandy, L. Lamport, ”Distributed snapshots: Determining global states of distributed systems,” *ACM Transaction on Computer Systems*, 3(1): 63-75. Februar 1985.

[7] A Toolkit for the Simulation of Distributed Algorithms in Java , dostopno na: http://www.risc.uni-linz.ac.at/software/daj/ (2009)