Distribuirane metaheuristike za rješavanje NP-teških problema

Sunara, Nikola

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Nikola Sunara

DISTRIBUIRANE METAHEURISTIKE ZA RJEŠAVANJE NP-TEŠKIH PROBLEMA

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, veljača, 2022.

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

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

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Rješavanje NP-teških problema je veliki izazov u matematici i računarstvu. Neki primjeri NP-teških problema su: SAT, problem trgovačkog putnika i problem sume podskupa. Za egzaktno rješavanje većih primjeraka NP-teških problema potrebno je utrošiti mnogo računalnih resursa i vremena, a najčešće je slučaj da nam ti računalni resursi i vrijeme nisu dostupni. Iz tog razloga primorani smo primjerke takvih problema rješavati metodama koje daju približno točna rješenja. Metaheuristika je skup algoritamskih koncepata koje koristimo za definiranje heurističnih metoda primjenjivih na širok skup problema. Dakle, pomoću metaheuristika možemo približno točno i brzo rješavati probleme, te kao takve dobar su kandidat za rješavanje NP-teških problema. Distribuirani sustav se sastoji od više procesa koji komuniciraju razmjenom poruka preko komunikacijske mreže. Korištenjem distribuiranih sustava i algoritmima pisanim za njih nastojimo ubrzati rješavanje problema. U ovom radu bavit ćemo se distribuiranim metaheuristikama za rješavanje NP-teških problema.

U poglavlju 1 opisujemo kako analizirati performanse distribuiranih metaheuristika i uvodimo razne mjere performansi distribuiranih metaheuristika. U poglavlju 2 uvodimo prvu distribuiranu metaheuristiku - distribuirani genetski algoritam. Također, opisujemo razne načine strukturiranja populacije distribuiranog genetskog algoritma i modele distribuiranog genetskog algoritma. U poglavlju 3 opisujemo metaheuristiku mravlje optimizacije i njenu distribuiranu verziju, multikolonijalnu mravlju optimizaciju. U poglavlju 4 opisujemo metaheuristiku simulirano kaljenje, te navodimo načine kako možemo distribuirati simulirano kaljenje. U poglavlju 5 opisujemo metaheuristiku tabu-traženje i taksonomiju distribuiranih algoritama tabu-traženja. U poglavlju 6 opisujemo MAXSAT problem. Nakon toga opisuje se implementacija distribuiranog genetskog algoritma za rješavanje MAXSAT problema i analiziraju se performanse te implementacije. U završnom poglavlju 7 donosimo zaključak rada uz prijedloge daljnjeg istraživanja.

Poglavlje 1

Analiziranje performansi distribuiranih metaheuristika

Kod analize performansi distribuiranih metaheuristika možemo promatrati koliko brzo dolazimo do rješenja i koliko je dobiveno rješenje blizu optimalnog rješenja. Razlikujemo dva pristupa analizi: teorijski i eksperimentalni pristup. Teorijski pristup nam može dati bolju usporedbu metaheuristika (neovisno o implementacijskim detaljima i računalu na kojem se provodi analiza), ali ju je u praksi vrlo teško provesti zbog kompleksnosti praktičnih problema. Dakle, performanse metaheuristika kod rješavanja praktičnih problema najčešće analiziramo eksperimentalno. Eksperimentalna analiza se najčešće provodi tako da primjenimo metaheuristike na skup instanci problema i usporedimo rezultate kao što su brzina dobivanja rješenja ili blizina optimalnog rješenja.

1.1 Mjere performansi distribuiranih metaheuristika

U nastavku opisujemo nekoliko mjera performansi distribuiranja metaheuristika.

Ubrzanje (Speedup)

Ubrzanje uspoređuje vrijeme sekvencijalnog i distribuiranog izvođenja programa. Vrijeme koje ćemo koristiti u definiciji ubrzanja je vrijeme između početka i završetka cijelog algoritma.

Definicija 1. Označimo sa T_m vrijeme izvođenja algoritma na m distribuiranih računala i T_1 vrijeme izvođenja algoritma na jednom računalu. Tada ubrzanje (speedup), označeno s s_m, definiramo s

$$s_m = \frac{T_1}{T_m}. \tag{1.1}$$

Kod nedeterminističkih algoritama koristimo prosječno vrijeme, dakle ubrzanje tada definiramo s

$$s_m = \frac{E[T_1]}{E[T_m]}. \tag{1.2}$$

Razlikujemo tri tipa ubrzanja:

- sublinearno ubrzanje ($s_m < m$),
- linearno ubrzanje ($s_m = m$),
- superlinearno ubrzanje ($s_m > m$).

Razlikujemo dvije vrste ubrzanja s obzirom na to što uzimamo za vrijeme T_1:

- jako uzbrzanje - za T_1 uzimamo vrijeme najboljeg trenutno poznatog sekvencijalnog algoritma,
- slabo ubrzanje - za T_1 uzimamo vrijeme sekvencijalne verzije distribuiranog algoritma čije ubrzanje određujemo.

Naglasimo da je kod uspoređivanja algoritama bitno osigurati da sekvencijalni i distribuirani algoritam daju rezultate iste ili slične točnosti.

Efikasnost (efficiency)

Definicija 2. Efikasnost je normalizacija ubrzanja, odnosno efikasnost u oznaci e_m definiramo s

$$e_m = \frac{s_m}{m}. \tag{1.3}$$

Efikasnost nam omogućava usporedbu raznih algoritama. Primijetimo da $e_m = 1$ znači linearno ubrzanje.

Inkrementalno ubrzanje

Definicija 3. Inkrementalno ubrzanje, u oznaci ie_m, definiramo s

$$ie_m = \frac{(m-1) \cdot E[T_{m-1}]}{m \cdot E[T_m]}, \tag{1.4}$$

gdje $E[T_n]$ označava prosječno vrijeme izvođenja algoritma na distribuiranom sustavu s n procesora. Također, možemo generalizirati inkrementalno ubrzanje tako da povećamo broj procesora s n na m:

$$gie_{n,m} = \frac{n \cdot E[T_n]}{m \cdot E[T_m]}. \tag{1.5}$$

Iz inkrementalnog ubrzanja vidimo kolika je korist dodavanja jednog procesora. Mjera inkrementalnog ubrzanja se najčešće koristi kad nam nije poznato sekvencijalno vrijeme izvođenja algoritma, pa ne možemo promatrati ubrzanje.

Scaleup

Prethodne mjere ne uzimaju u obzir da bi ubrzanje moglo doći zbog povećanja dostupne memorije, nego samo od povećanja procesorskog vremena, stoga definiramo mjeru scaleup.

Definicija 4. Scaleup, u oznaci $su_{m,n}$, definiramo s

$$su_{m,n} = \frac{\text{Vrijeme rješavanja } k \text{ problema na } m \text{ procesora}}{\text{Vrijeme rješavanja } nk \text{ problema na } nm \text{ procesora}}. \tag{1.6}$$

1.2 Kako analizirati distribuirane metaheuristeke

Najčešći ciljevi analize su usporedba algoritma s ostalim poznatim algoritmima za rješavanje nekog problema ili razumijevanje algoritma. Kada analiziramo distribuirane metaheuristike, prvo trebamo odrediti problem i instance problema na kojima ćemo testirati algoritam. Instance problema moraju biti dovoljno raznolike kako bi mogli generalizirati zaključke. Uvijek je dobro koristiti standardne instance problema (benchmark instance problema) koji su zbog široke uporabe u eksperimentima postale standardne u literaturi. Izbor takvog skupa testnih podataka nam omogućuje usporedbu algoritma s ostalim algoritmima. Nakon što smo odredili problem i instance problema, sljedeći korak u analizi algoritma je dizajniranje računalnog eksperimenta. Kod dizajna eksperimenta trebamo odrediti veličinu problema, parametre algoritma itd. (faktori koje analiziramo). Kod dizajna eksperimenta moramo paziti da se eksperiment može izvršiti u odgovarajućem vremenskom okviru. Sljedeći korak analize je izvršenje eksperimenta, izbor mjera performansi i analiza dobivenih rezultata.

Cilj metaheuristika je naći zadovoljavajuće rješenje u odgovarajućem vremenu, dakle kod analize metaheuristika moramo imati mjere za kvalitetu rješenja i brzinu dobivanja rješenja. Zbog stohastičke prirode metaheuristika, trebamo izvršiti više nezavisnih pokusa da bi dobili zadovoljavajuće eksperimentalne rezultate.

1.3 Mjere kvalitete rješenja

Kad određujemo performanse metaheuristike, trebamo posvetiti pažnju kvaliteti dobivenih rješenja. Kvalitetu rješenja možemo mjeriti na razne načine u ovisnosti o konkretnom problemu. Kada znamo optimalno rješenje, jedna od najboljih mjera kvalitete je broj pogodaka

(% hits) optimalnog rješenja. Postoje problemi kod kojih optimalno rješenje nije poznato ili zbog zahtjevnosti problema optimalno rješenje nije moguće pronaći u realnom vremenu. U takvim slučajevima promatramo prosječne ili medijalne vrijednosti dobivenih rješenja. Ponekad u praksi jednostavna usporedba prosječnih (ili medijalnih) vrijednosti nije dovoljna, nego je potrebno napraviti usporedbu distribucija. Dakle, potrebno je računati vrijednosti poput varijance da bi ustanovili da je neki algoritam stvarno bolji od drugog.

1.4 Brzina računanja

Brzina računanja najčešće se mjeri brojem evaluacija ili vremenom izvršavanja programa. Broj evaluacija se obično definira kao broj elemenata u prostoru pretraživanja koji su posjećeni. Broj evaluacija eliminira ovisnost usporedbe algoritama o implementacijskim detaljima, softveru i hardveru. Evaluacije kod raznih algoritama se odvijaju u različitim vremenima, pa ni broj evaluacija ne daje savršenu usporedbu algoritama. Najčešće je potrebno kombinirati dvije prethodno navedene mjere. Za usporedbu algoritama možemo koristiti prosječno vrijeme (broj evaluacija) izvršavanja koja završe s optimalnim rješenjem ili rješenjem koje zadovoljava unaprijed definiranu točnost. Također, kad uspoređujemo algoritme možemo unaprijed definirati vrijeme (broj evaluacija) i uspoređivati kvalitetu dobivenih rješenja.

Poglavlje 2

Distribuirani genetski algoritam

Genetski algoritam koristi populaciju rješenja kako bi pretražio mnoge različite dijelove prostora rješenja. Prostor rješenja se pretražuje na način da na populaciju primjenjujemo stohastičke operatore. Pseudokod klasičnog genetskog algoritma iskazan je u 1.

Algorithm 1 Pseudokod klasičnog genetskog algoritma

```
Generiraj(P(0))
Evaluiraj(P(0))
t = 0
while not KriterijZaustavljanja(P(t)) do
    P'(t) =Selekcija(P(t))
    P''(t) =Rekombinacija(P'(t))
    P'''(t) =Mutacija(P''(t))
    Evaluiraj(P'''(t))
    P(t + 1) = P'''(t)
    t = t + 1
end while
```

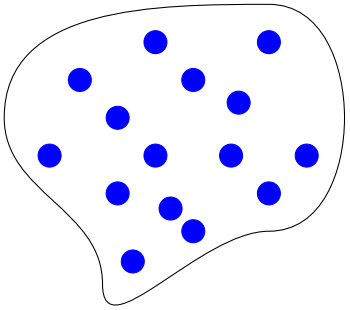
Na početku algoritma generiramo slučajnu populaciju $P(0)$. Nakon toga iterativno dobivamo populaciju $P(t)$ iz populacije $P(t-1)$, $t \in \{1, 2, 3, \dots\}$ primjenom stohastičkih operatora selekcije, rekombinacije i mutacije. Funkcija evaluacije pridjeljuje vrijednosti članovima populacije, te na taj način određuje koji članovi imaju veće šanse opstati u sljedećoj populaciji. Kriterij zaustavljanja je najčešće unaprijed definirani broj ponavljanja stohastičkih operacija ili pronalazak rješenja unutar dozvoljene pogreške u slučaju da je optimalno rješenje poznato.

Reproduktivni ciklus (selekcija, rekombinacija, mutacija) može zahtijevati dosta računalnih resursa. Zbog toga se razmatraju razni načini kako dobiti efikasnije evolucijske algoritme. Jedan od načina je da koristimo distribuirano računanje. Također, možemo promatrati dis-

tribuirane genetske algoritme s obzirom je li koriste strukturiranu populaciju. U nastavku promatramo genetske algoritme koji koriste panmiktičku ili strukturiranu populaciju.

2.1 Panmiktički genetski algoritmi

Prvo promatramo klasu genetskih algoritama kod kojih je populacija panmiktička, dakle svaki član populacije se može rekombinirati s bilo kojim drugim članom populacije i selekcija se odvija na čitavoj populaciji. Označimo sa λ broj novih članova populacije koji nastaju u jednom reproduktivnom ciklusu (iteraciji algoritma), a sa μ veličinu populacije, tada pridruživanjem različitih vrijednosti varijabli λ dobivamo razne algoritme iz klase panmiktičkih genetskih algoritama koji koriste "reproduktivni razmak". Za $\lambda = 1$ dobivamo "steady state" algoritam, a za $\lambda = \mu$ dobivamo generacijski algoritam, dakle kod generacijskog algoritma u svakoj iteraciji se sve stare individue zamjenjuju novim, odnosno dolazi do smjene generacija. Panmiktička populacija se uglavnom koristi kod sekvencijalnih genetskih algoritama. Kod distribuiranih panmiktičkih genetskih algoritama najčešće se distribuira evaluacija, a selekciju i rekombinaciju obavlja jedan proces.

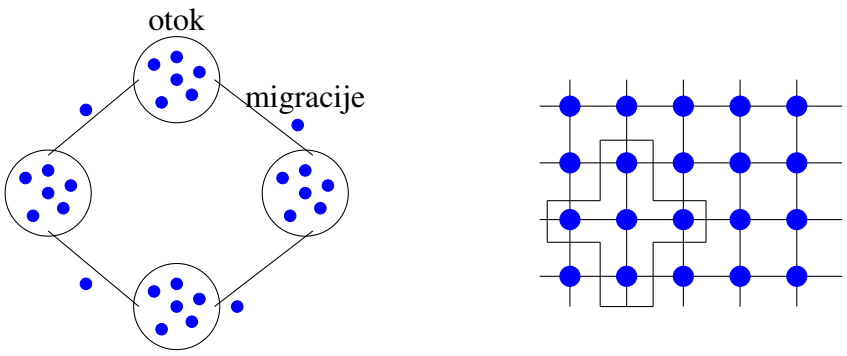


Slika 2.1: Panmiktička populacija genetskih algoritama.

2.2 Strukturirani genetski algoritmi

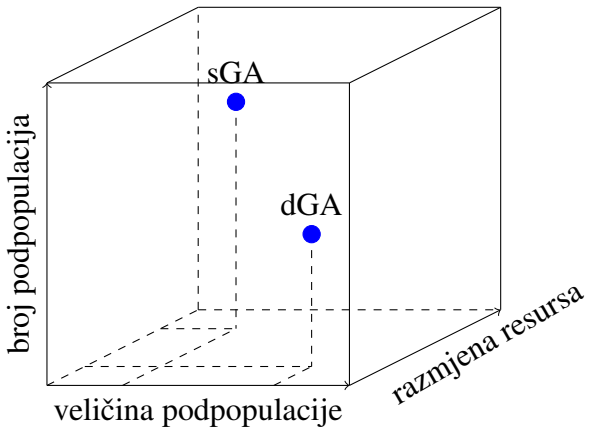
Populacija strukturiranih genetskih algoritama ima određenu strukturu, a reproduktivni ciklus (rekombinacija, selekcija) se odvija u skladu s tom strukturom. Najpoznatiji strukturirani genetski algoritmi su distribuirani genetski algoritam i stanični genetski algoritam. Populacija distribuiranog genetskog algoritma je particionirana u nekoliko manjih podpopulacija (otoka). Rekombinacija i selekcija se odvijaju unutar otoka. Da bi se postigla raznolikija pretraga prostora rješenja, uvode se migracije između otoka. Posebni parametri unutar algoritma kontroliraju gdje će se migracije odvijati i koliko često. Svaku jedinku

populacije staničnog genetskog algoritma, zajedno s njenim susjedima, možemo smatrati jednom podpopulacijom unutar koje se odvija rekombinacija i selekcija. Budući da se svaka jedinka nalazi u više podpopulacija, omogućeno je širenje dobrih svojstava jedinki kroz čitavu populaciju.



Slika 2.2: Distribuirani genetski algoritam (lijevo) i stanični genetski algoritam (desno).

Na slici 2.3 možemo vidjeti trodimenzionalni prikaz strukturiranih genetskih algoritama s obzirom na veličinu podpopulacija, broj podpopulacija i stupanj razmjene resursa između podpopulacija. Iz slike 2.3 vidimo da stanični genetski algoritam ima više podpopulacija i veći stupanj razmjene resursa između podpopulacija od distribuiranog genetskog algoritma, ali manje individua u podpopulacijama.



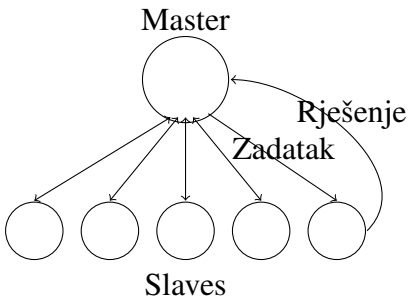
Slika 2.3: Kocka strukturiranih genetskih algoritama.

2.3 Modeli distribuiranih genetskih algoritama

U literaturi možemo pronaći sljedeće modele distribuiranih genetskih algoritama:

- model nezavisnih izvođenja,
- master-salve model,
- distribuirani model
- i stanični model.

Koncepte staničnog i distribuiranog modela smo opisali već ranije. Model nezavisnih izvođenja temelji se na tome da sekvencijalni genetski algoritam pokrećemo na raznim distribuiranim računalima. Budući da je genetski algoritam stohastički, dobit ćemo različite rezultate od svakog nezavisnog izvođenja. Master-slave model izvršava glavnu petlju genetskog algoritma u master čvoru distribuiranog sustava, a evaluacija se odvija u slave čvorovima. Master čvor šalje slave čvorovima resurse potrebne za evaluaciju, dakle master čvor upravlja zadatcima koje će obavljati svi čvorovi u sustavu. Master-slave model je efikasan kad je evaluacija vremenski zahtjevna, pa vrijeme koje se potroši za komunikaciju čvorova postane beznačajno.

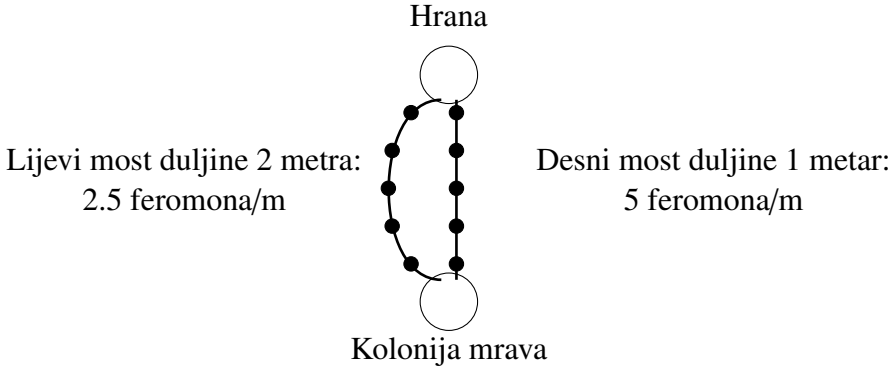


Slika 2.4: Master-slave model.

Poglavlje 3

Distribuirana mravlja optimizacija

Mravlja optimizacija je inspirirana ponašanjem mrava prilikom traženja hrane. Opišimo eksperiment "dva mosta" pomoću kojeg ćemo lakše shvatiti način rada mravlje optimizacije. U eksperimentu "dva mosta" između mravlje kolonije i hrane nalaze se dva mosta različite duljine. Nakon par minuta odvijanja eksperimenta, većina mrava se kreće po kraćem mostu. Ovakvo ponašanje je zanimljivo jer mravi koji se koriste u eksperimentu slabo vide pa na taj način nisu mogli odrediti kraći put. Objašnjenje ovakvog razvoja događaja je to da mravi ostavljaju feromone dok se kreću. Mravi koji putuju kraćim mostom će prije doći do hrane i vratiti se natrag na polazište. Tada, prilikom izbora mosta veća je vjerojatnost da će izabrati most kojim su mravi prošli dva puta (do hrane i natrag), nego da će izabrati most kojim su mravi prošli jedan cijeli put (do hrane) i eventualno dio puta prema polazištu. Odnosno, na kraćem mostu će biti veća koncentracija feromona. Zbog veće koncentracije feromona sve više mrava će se kretati kraćim putem.



Slika 3.1: Eksperiment "dva mosta".

3.1 Algoritam mravlje optimizacije

Pretpostavimo da imamo problem kombinatorne optimizacije i pripadno stablo odluke. Kreiranje rješenja možemo promatrati kao kreiranje puta u stablu odluke. Cilj je da mravi traže putove po bridovima stabla odluke koji pripadaju prethodno pronađenim dobrim rješenjima. Cilj se postiže tako da mravi koji su pronašli dobro rješenje označe bridove svoga puta umjetnim feromonima. Mravi koji u sljedećim iteracijama traže rješenje koriste informacije koje su ostavljene u obliku umjetnih feromona. Da feromoni koji su dugo prisutni na stablu odluke ne bi utjecali previše na konačno rješenje, potrebno je smanjivati njihov utjecaj na odluku mrava nakon svake iteracije algoritma. Algoritam završava kada je zadovoljen kriterij zaustavljanja. Kriterij zaustavljanja je najčešće predefiniran broj iteracija algoritma ili pronalazak rješenja odgovarajuće točnosti.

Promotrimo kako prethodno opisani algoritam možemo primijeniti na problem trgovačkog putnika. Za dane skupove S (n gradova) i E (bridovi između gradova), trebamo pronaći najkraći put koji prolazi kroz svih n gradova i počinje i završava u istom gradu. Rješenje problema možemo zapisati kao permutaciju gradova π. Informacije koje mravi ostavljaju pomoću feromona možemo reprezentirati $n \times n$ matricom feromona $[\tau_{ij}], i, j \in \{1, 2, \ldots, n\}$. Što je vrijednost τ_{ij} veća, to je veća šansa da grad j bude odabran nakon grada i u određenom putu. Matrica $[\tau_{ij}]$ se obično inicijalizira tako da sve vrijednosti budu jednake. Prilikom kreiranja rješenja, mrav bira grad iz skupa gradova koji već nisu bili posjećeni. Vjerojatnost izbora grada j u slučaju da se mrav nalazi u gradu i je dana formulom

$$p_{ij} = \frac{\tau_{ij}}{\sum_{z \in S} \tau_{iz}}, \forall j \in S. \tag{3.1}$$

Nakon što svi mravi u određenoj iteraciji algoritma pronađu svoj put, usporedbom duljina pronađenih putova, određujemo najkraći put π^*. Zatim mijenjamo matricu feromona u sljedeća dva koraka:

- svi feromoni se smanjuju za određeni postotak $\rho \in (0, 1)$:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij}, \forall i, j \in \{1, 2, \ldots, n\}, \tag{3.2}$$

- svi elementi matrice feromona koji odgovaraju bridovima najkraćeg puta se povećavaju za određeni broj Δ:

$$\tau_{i\pi^*(i)} = \tau_{i\pi^*(i)} + \Delta, \forall i \in \{1, 2, \ldots, n\}. \tag{3.3}$$

Algorithm 2 Mravlja optimizacija za problem trgovačkog putnika

```
Inicijaliziraj matricu feromona
while Kriterij zaustavljanja nije zadovoljen do
    for mrav k ∈ {1, 2, ..., m} [konstrukcija rješenja] do
        S = {1, ..., n} [skup svih gradova]
        Izaberi početni grad i
        S = S − {i}
        while S ≠ ∅ do
            Izaberi grad j ∈ S s vjerojatnošću p_ij
            S = S − {j}
            i = j
        end while
    end for
    for all i, j do
        τ_ij = (1 − ρ) · τ_ij
    end for
    for all i, j u najboljem rješenju ove iteracije algoritma do
        τ_ij = τ_ij + Δ
    end for
end while
```

3.2 Multikolonijalna mravlja optimizacija

Grupa mrava koji su smješteni u jednom procesu naziva se kolonija. Algoritam multikolonijalne mravlje optimizacije sastoji se od više kolonija mrava, te svaka kolonija ima svoju matricu feromona.

U sljedećih nekoliko tablica bit će opisani načini na koje možemo implementirati određene dijelove algoritma multikolonijalne mravlje optimizacije. U tablici 3.1 možemo vidjeti na koje načine možemo implementirati topologiju susjedstva. Topologija susjedstva nam daje informaciju jesu li određene dvije kolonije susjedne.

Algorithm 3 Algoritam multikolonijalne mravlje optimizacije

```
Dio algoritma za proces i ∈ {1, ..., p}
Inicijaliziraj matricu feromona
while Kriterij zaustavljanja nije zadovoljen do
    for mrav k ∈ {1, 2, ..., m/p} iz kolonije i do
        konstrukcija rješenja
    end for
    if komunikacijski korak then
        razmjeni informacije sa susjednim kolonijama
        obradi primljene informacije
    end if
    for all vrijednosti feromona do
        smanji sve vrijednosti feromona za određeni postotak
    end for
    for all vrijednosti feromona u dobrim rješenjima do
        povećaj vrijednost za neku konstantu
    end for
end while
```

All-to-all topologija	U ovoj topologiji svaka kolonija je susjedna sa svakom drugom kolonijom.
Prstenasta topologija	Pretpostavimo da imamo p kolonija i neka je $i \in \{0, \ldots, p-1\}$ jedna od njih. U usmjerenoj prstenastoj topologiji jedini susjed kolonije i je kolonija $(i+1) mod\ p$. U neusmjerenoj prstenastoj topologiji susjedi kolonije i su kolonije $(i+1) mod\ p$ i $(i-1) mod\ p$.
Topologija hiperkocke	Kod ove topologije uvjet je da imamo $p = 2^k$ kolonija. Susjedi kolonije i su sve kolonije čije se binarne reprezentacije razlikuju na samo jednom mjestu s binarnom reprezentacijom kolonije i. Stoga, svaka kolonija ima $log_2\ p = k$ susjeda.
Slučajna topologija	Susjedi kolonije i su neke slučajno odabrane kolonije $S \subseteq \{1, \ldots, p\}$. Postoji varijanta kada biramo samo jednog susjeda.

Tablica 3.1: Topologije susjedstva algoritma multikolonijalne mravlje optimizacije.

Tablica 3.2 prikazuje informacije koje kolonije mogu razmjenjivati.

Rješenja	Kolonije šalju dobra rješenja, koja su pronašle, ostalim kolonijama. Neke od strategija slanja dobrih rješenja su: • Migranti - Rješenje, koje je pronašao neki mrav u nekoj iteraciji, se šalje u ostale kolonije. Najčešće se šalje najbolje rješenje pronađeno u nekoj iteraciji. • Globalno najbolje rješenje - U ovoj strategiji najbolje globalno rješenje se određuje i šalje svim kolonijama. • Lokalno najbolje rješenje - U ovoj strategiji trenutno najbolje rješenje neke kolonije se šalje susjednim kolonijama. • Najbolje rješenje u susjedstvu - U ovoj strategiji određuje se najbolje rješenje u susjedstvu i šalje svim kolonijama u susjedstvu.
Vektor feromona	U ovoj strategiji vektor feromona se šalje susjednim kolonijama.
Matrica feromona	U ovoj strategiji matrica feromona se šalje susjednim kolonijama.

Tablica 3.2: Informacije koje kolonije razmjenjuju.

U tablici 3.3 prikazujemo nekoliko načina korištenja informacija koje se razmjenjuju između kolonija.

Usporedba s najboljim rješenjem	Rješenje, koje je primljeno, se uspoređuje s trenutnim najboljim rješenjem, te ako je bolje od njega, primljeno rješenje postaje novo najbolje rješenje.
Dodavanje matrici feromona	Kada je rješenje zaprimljeno, matrica feromona se osvježava pomoću tog rješenja. Kada primimo matricu feromona, kreira se nova matrica feromona pomoću stare i zaprimljene matrice feromona.
Dodavanje trenutnoj generaciji	Rješenje se dodaje svim rješenjima koje su pronašli mravi u određenoj iteraciji. Ako je primljeno rješenje bolje od svih rješenja trenutne iteracije, tada ono utječe na promjenu matrice feromona.

Tablica 3.3: Korištenje informacija dobivenih od ostalih kolonija.

Tablica 3.4 prikazuje kada se sve može odvijati komunikacija između kolonija.

Svaku iteraciju	Informacije se razmjenjuju svaku iteraciju.
Svakih k iteracija	Informacije se razmjenjuju svakih k iteracija.
Ovisno o kvaliteti rješenja	Npr. kad nađemo bolje rješenje od trenutno najboljeg, šaljemo ga svim ostalim kolonijama.

Tablica 3.4: Vrijeme komunikacije.

Poglavlje 4

Distribuirano simulirano kaljenje

4.1 Simulirano kaljenje

Simulirano kaljenje je stohastički algoritam koji pretražuje prostor rješenja.

Algorithm 4 Algoritam simuliranog kaljenja

```
Odaberemo inicijalno stanje S_0
S_old = S_0
while temperatura T nije dovoljno niska do
    Odaberemo susjedno stanje S_new
    ΔS = f(S_old) − f(S_new)
    if ΔS < 0 then
        S_old = S_new
    else if e^(−ΔS/T) > r then
        S_old = S_new
    end if
    Smanjimo temperaturu
end while
```

Na početku algoritma odaberemo neko stanje S_0 i od njega počinjemo pretragu. U petlji algoritma pretražujemo nova rješenja. Ako smo među susjednim rješenjima odabrali bolje rješenje od trenutnog, prihvaćamo ga i nastavljamo pretragu od njega. Ako smo odabrali lošije rješenje od trenutnog, prihvaćamo ga s određenom vjerojatnošću. Ako rješenje po kvaliteti nije puno lošije od trenutnog, prihvaćamo ga s većom vjerojatnošću, a ako je puno gore od trenutnog, prihvaćamo ga s malom vjerojatnošću. S f smo označili funkciju kvalitete rješenja, a r je neki slučajan broj. Petlja algoritma završava nakon što smo snizili temperaturu na odgovarajuću razinu.

4.2 Distribuiranje simuliranog kaljenja

Postoji nekoliko načina na koje možemo distribuirati simulirano kaljenje. Neki od načina distribuiranja, koje ćemo opisati kasnije, su:

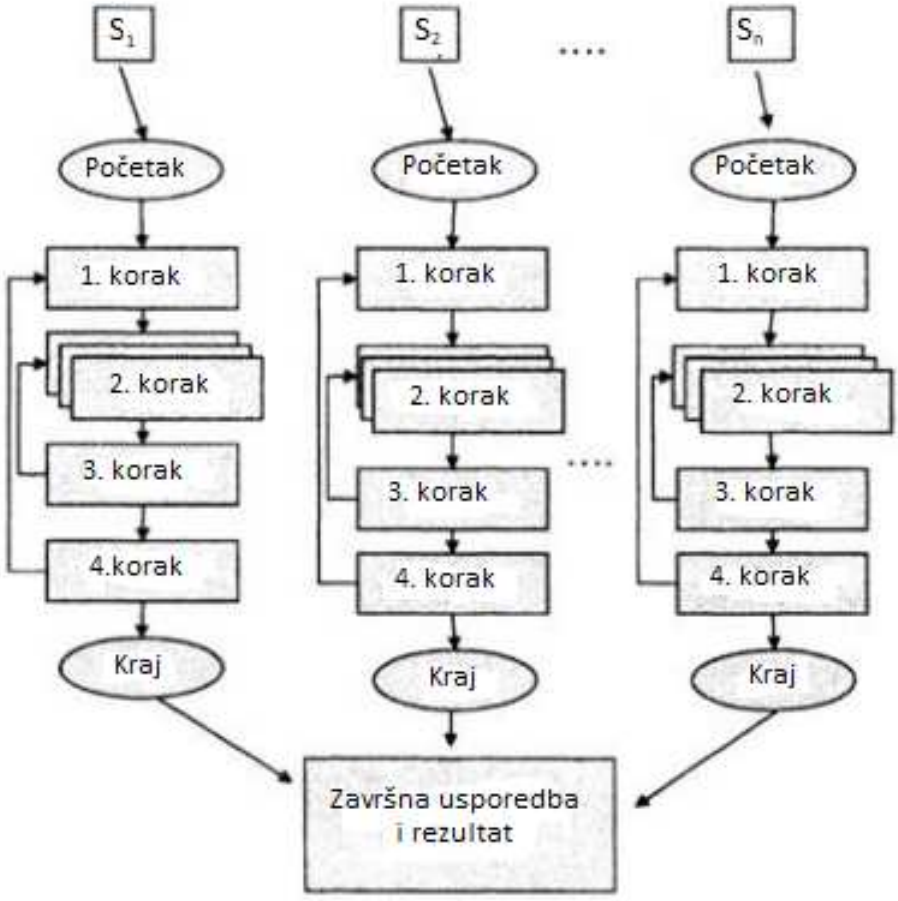
- distribuiranje po podatcima,
- više nazavisnih izvođenja i
- distribuirani potezi.

Distribuiranje po podatcima

Ideja ovog pristupa je da skup podataka, koji pretražujemo, podijelimo na nekoliko dijelova i svaki dio pridružimo nekom procesu. Ako nemamo skup podataka nego prostor rješenja, tada njega podijelimo na isti način. Ovakav pristup je ovisan o problemu kojeg želimo riješiti, jer ne mogu se prostori rješenja svih problema podijeliti na način da možemo primijeniti algoritam simuliranog kaljenja na njih.

Više nezavisnih izvođenja

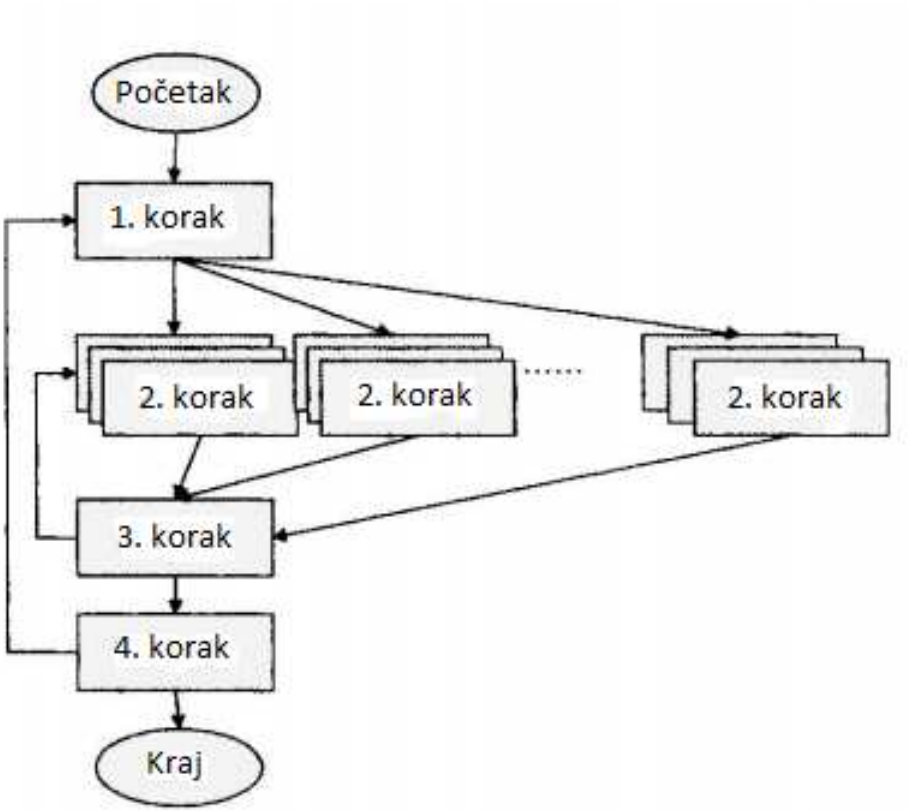
Ideja ovog pristupa je da pokrenemo algoritam simuliranog kaljenja na više procesa s različitim početnim parametrima. Nakon što svi procesi završe s radom, izabiremo najbolje rješenje. Skica ovog načina distribuiranja algoritma simuliranog kaljenja prikazana je na slici 4.1.



Slika 4.1: Više nezavisnih izvođenja

Paralelni potezi

Algoritam simuliranog kaljenja možemo distribuirati tako što pretraživanju susjedstva pridružimo više procesa. Budući da izbor susjeda nije deterministički, svaki proces će provjeriti drugog susjeda. Također, možemo birati koliki broj poteza će procesi izvršavati petlju simuliranog kaljenja, prije nego međusobno razmijene rješenja i izaberu najbolje od njih za nastavak algoritma. Skica ovog načina distribuiranja algoritma simuliranog kaljenja prikazana je na slici 4.2.



Slika 4.2: Paralelni potezi

Poglavlje 5

Distribuirano tabu-traženje

5.1 Tabu-traženje

Tabu-traženje je proširenje algoritma lokalnog traženja. Kao kod algoritma lokalnog traženja, tabu-traženje pretražuje prostor pretrage ispitujući susjede trenutnog rješenja (koristi se operator susjedstva). Sljedeće rješenje od kojeg se nastavlja pretraga najčešće se bira po pravilima:

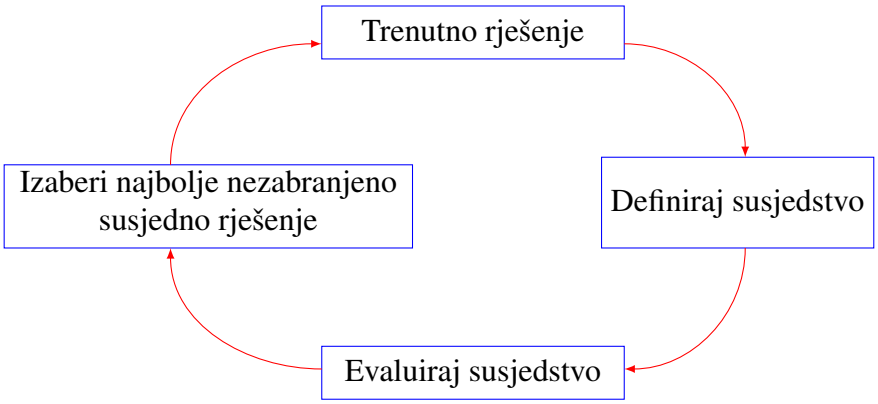
- najbolje rješenje u susjedstvu
- ili prvo bolje rješenje u susjedstvu na koje smo naišli.

Metode koje koriste lokalnu pretragu često zapnu u lokalnom optimumu. U slučaju da tabu-traženje dođe do lokalnog optimuma, ono prihvaća i rješenje koje nije bolje od trenutno najboljeg rješenja kako bi pobjeglo iz lokalnog optimuma. Da bi se spriječilo cikličko traženje, uvodi se tabu lista u koju se sprema zadnjih nekoliko već posjećenih rješenja. Najčešće korišteni kriteriji zaustavljanja su:

- procesorsko vrijeme,
- broj iteracija algoritma
- i broj iteracija algoritma bez poboljšanja trenutno najboljeg rješenja.

Algorithm 5 Algoritam tabu-traženja

Inicijaliziraj početno rješenje i tabu listu
while kriterij zaustavljanja nije postignut **do**
    Pretraži susjedna rješenja
    Evaluiraj susjedna rješenja
    Uzmi novo najbolje rješenje za nastavak pretrage (nije u tabu-listi)
    Osvježi tabu-listu
end while



Slika 5.1: Iteracija tabu-traženja.

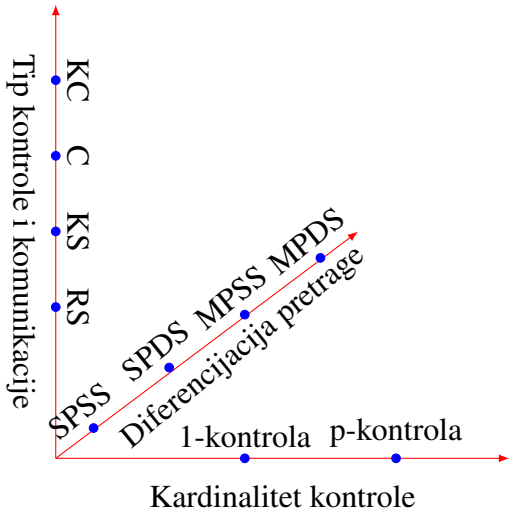
Prethodno opisani algoritam se smatra osnovnim tabu-traženjem. Mnoge implementacije tabu-traženja su kompleksnije od osnovnog algoritma tabu-traženja. One se zasnivaju na kombinaciji lokalnog pretraživanja i iskorištavanju raznih informacija spremljenih u memoriji.

5.2 Distribuiranje tabu-traženja

U nastavku opisujemo taksonomiju distribuiranih algoritama tabu-traženja. Taksonomija je bazirana na trodimenzionalnoj klasifikaciji svojstava algoritma. Prva dimenzija se zove Kardinalitet kontrole. Kardinalitet kontrole određuje je li se pretraga odvija pod kontrolom samo jednog procesa (master-slave model) ili se pretraga odvija na način da svaki proces upravlja svojim dijelom pretrage i surađuje s ostalim procesima ukoliko mu se za to ukaže potreba. Ova dva načina nazivaju se redom 1-kontrola (1C) i p-kontrola (pC). Druga dimenzija se zove Tip kontrole i komunikacije. Ova dimenzija uzima u obzir organizaciju komunikacije, sinkronizaciju, hijerarhiju i način na koji procesi obrađuju i dijele informacije. Ova dimenzija sadrži četiri stupnja. Prvi stupanj se naziva Rigidna sinkro-

nizacija (RS). U Rigidnoj sinkronizaciji se malo informacija razmjenjuje među procesima na istoj razini hijerarhije (ili se uopće ne razmjenjuju informacije). Primjer Rigidne sinkronizacije je 1-kontrola master-slave model, u kojem master proces zadaje zadatke slave procesima, prima rezultate i određuje daljnji tijek pretrage. Primjer Rigidne sinkronizacije koja spada u p-kontrola algoritme je Direktno distribuiranje nezavisnih procesa, u kojoj svaki od procesa pretražuje prostor pretrage i kad svi procesi završe pretraživanje određuje se najbolje rješenje. Drugi stupanj se naziva Sinkronizacija znanja (KS). Sinkronizacija znanja u 1-kontrola algoritmima odgovara slučaju gdje slave procesi izvode kompleksnije zadatke (npr. nekoliko koraka tabu-traženja). Kod p-kontrola algoritama, Sinkronizacija znanja odgovara slučaju kad procesi izvode nezavisne pretrage neko vrijeme, nakon kojeg se upuštaju u intenzivnu komunikaciju da bi podijelili informacije koje su otkrili. Treći stupanj naziva se Kolegijalnost (C). Kolegijalnost pokriva slučajeve gdje se koristi asinkrona komunikacija. Kolegijalnost ima smisla samo kod p-kontrola algoritama. Ona se zasniva na tome da svaki pojedini proces obavlja pretragu prema svojoj logici, te razmjenjuje informacije s ostalim procesima ukoliko on to želi (npr. ako nađe globalno najbolje rješenje pa ga pošalje svim ostalim procesima). Četvrti stupanj se naziva Kolegijalnost znanja (KC). Ovaj stupanj se odnosi na algoritme u kojima se koristi asinkrona komunikacija i u kojima se analiziraju informacije koje se razmjenjuju između procesa. Treća dimenzija taksonomije je Diferencijacija pretrage. Ona odgovara na pitanja: ”Je li procesi započinju pretragu od istog početnog rješenja?” i ”Je li procesi koriste iste strategije pretrage?”. Ova dva pitanja vode k četiri načina kreiranja algoritma:

- isti početak, iste strategije (SPSS),
- isti početak, različite strategije (SPDS),
- različiti početci, iste strategije (MPSS) i
- različiti početci, različite strategije (MPDS).



Slika 5.2: Taksonomija distribuiranih algoritama tabu-traženja.

Osnovne razlike između stupnjeva dimenzija su dane u sljedećim tablicama.

	Broj procesa koji upravljaju pretragom
1-kontrola	jedan proces
p-kontrola	više procesa

Tablica 5.1: Razlike stupnjeva dimenzije Kardinalitet kontrole.

	Sinkorona/asinkrona komunikacija	Količina razmjenjenih informacija na istoj razini hijerarhije
Rigidna sinkronizacija (RS)	sinkrona	mala količina informacija
Sinkronizacija znanja (KS)	sinkrona	veća količina informacija
Kolegijalnost (C)	asinkrona	mala količina informacija
Kolegijalnost znanja (KC)	asinkrona	veća količina informacija

Tablica 5.2: Razlike stupnjeva dimenzije Tip kontrole i komunikacije.

Primijetimo da je ova taksonomija, iako je uvedena za distribuirano tabu traženje, primjenjiva na mnoge druge distribuirane metaheuristike.

Poglavlje 6

Implementacija distribuiranog genetskog algoritma i primjena na MAXSAT problem

6.1 Opis MAXSAT problema i podataka

Definicija 5. Propozicionalnu varijablu i njenu negaciju nazivamo *literal*. Formulu oblika $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ nazivamo *konjunkcija*, a formulu oblika $A_1 \vee A_2 \vee \cdots \vee A_n$ *disjunkcija*, gdje su A_i, $i \in \{1, \ldots, n\}$ proizvoljne formule.
Elementarna konjunkcija je konjunkcija literala, a *elementarna disjunkcija* je disjunkcija literala. *Konjunktivna normalna forma (knf)* je konjunkcija elementarnih disjunkcija. *Disjunktivna normalna forma (dnf)* je disjunkcija elementarnih konjunkcija.

Definicija 6. Svako preslikavanje sa skupa svih propozicionalnih varijabli u skup $\{0, 1\}$, tj. $I : \{P_0, P_1, \ldots\} \to \{0, 1\}$, nazivamo *totalna interpretacija* ili kratko *interpretacija*. Ako je preslikavanje definirano na podskupu skupa propozicionalnih varijabli tada kažemo da je to *parcijalna interpretacija*.

Definicija 7. Neka je I interpretacija (totalna ili parcijalna). Tada *vrijednost interpretacije I na proizvoljnoj formuli* definiramo rekurzivno ovako:

$$
\begin{aligned}
I(\neg A) &= 1 \text{ ako i samo ako } I(A) = 0,\\
I(A \wedge B) &= 1 \text{ ako i samo ako } I(A) = 1 \text{ i } I(B) = 1,\\
I(A \vee B) &= 1 \text{ ako i samo ako } I(A) = 1 \text{ ili } I(B) = 1,\\
I(A \to B) &= 1 \text{ ako i samo ako } I(A) = 0 \text{ ili } I(B) = 1,\\
I(A \leftrightarrow B) &= 1 \text{ ako i samo ako } I(A) = I(B).
\end{aligned}
$$

Definicija 8. Za formulu logike sudova kažemo da je *ispunjiva* ako postoji interpretacija I tako da vrijedi $I(F) = 1$.

Definicija 9. Konjunktivna normalna forma koja u svakoj elementarnoj disjunkciji sadrži točno k literala (za neki $k \in \mathbb{N}\backslash\{0\}$), nazivamo *k-knf*. Problem *k-SAT* se sastoji od ispitivanja ispunjivosti k-knf, odnosno od prepoznavanja elemenata (formula) iz skupa:

$$k\text{-SAT} = \{F : F \text{ je ispunjiva } k\text{-knf}\}. \tag{6.1}$$

Definicija 10. Neka je $k \in \mathbb{N}$ i F konjunktivna normalna forma. Problem *MAXSAT* definiramo:

$$\text{MAXSAT} = \{F : \text{za } F \text{ postoji interpretacija } I \text{ takva da za barem } k \text{ elementarnih disjunkcija } A_i \text{ iz } F \text{ vrijedi } I(A_i) = 1\}. \tag{6.2}$$

Primjer 1:

Neka je $A = \{P_1, P_2, P_3, P_4\}$ podskup skupa propozicionalnih varijabli. Definirajmo 3-knf F sljedećom formulom:

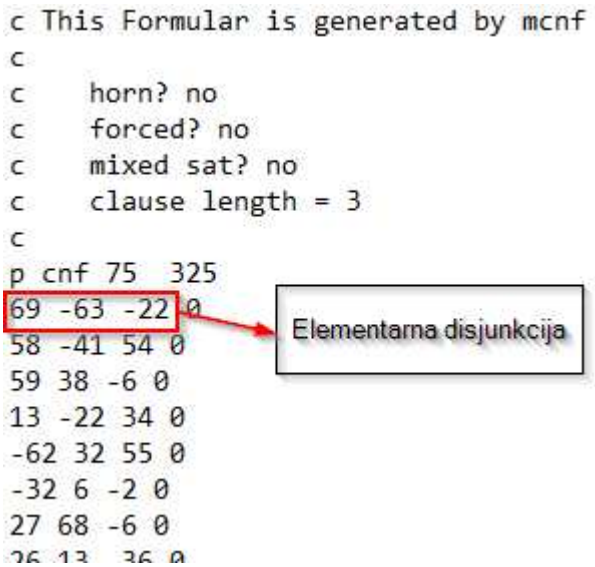
$$F \equiv (P_1 \vee \neg P_2 \vee P_3) \wedge (P_1 \vee \neg P_1 \vee P_4) \wedge (P_2 \vee \neg P_3 \vee \neg P_4).$$

Neka je I parcijalna interpretacija, $I : A \to \{0, 1\}$, za koju vrijedi:

- $I(P_1) = 1$,
- $I(P_2) = 1$,
- $I(P_3) = 0$
- i $I(P_4) = 1$.

Tada vrijedi $I(F) = 1$, odnosno postoji parcijalna interpretacija I za koju je $I(F) = 1$. Dakle, F je ispunjiva 3-knf, tj. $F \in$ 3-SAT. Također, primijetimo da vrijedi $F \in$ MAXSAT za parametar $k = 2$.

Distribuirani genetski algoritam ćemo primijeniti na MAXSAT problem. Konjunktivna normalna forma na koju primjenjujemo algoritam je 3-knf sa 75 propozicionalnih varijabli i 325 elementarnih disjunkcija. Na slici 6.1 prikazana je tekstualna datoteka iz koje ćemo učitavati *knf*. Elementarne disjunkcije su zapisane u redcima na način da su indeksi propozicionalnih varijabli odvojeni razmacima. Ukoliko se radi o negativnom literalu, ispred indeksa propozicionalne varijable je stavljen znak $-$. Nula na kraju retka označava kraj elementarne disjunkcije.



Slika 6.1: Konjunktivna normalna forma

6.2 Implementacija distribuiranog genetskog algoritma za rješavanje MAXSAT problema

Programsko rješenje se sastoji od 5 klasa i infrastrukture za razmjenu poruka između procesa s kolegija Distribuirani procesi. Više o infrastrukturi za razmjenu poruka se može pročitati u [3]. Program je napisan u programskom jeziku Java. Klase od kojih se sastoji program su:

- Clause,
- CNF,
- Individual,
- Population i
- DistributedGA.

U nastavku redom opisujemo navedene klase.

```
1  public class Clause {
2      int variables[];
3      int clauseLength;
4
5      Clause(int numberOfLiterals) {
6          variables = new int[numberOfLiterals];
7          clauseLength = numberOfLiterals;
8      }
9      Clause(int numberOfLiterals, int[] valuesOfLiterals) {
10         variables = new int[numberOfLiterals];
11         variables = valuesOfLiterals;
12         clauseLength = numberOfLiterals;
13     }
14 }
```

Slika 6.2: Program `Clause.java`

Klasa `Clause` predstavlja jednu elementarnu disjunkciju. Slijedi detaljnije objašnjenje te klase:

- Varijabla `clauseLength` sadrži broj literala u elementarnoj disjunkciji.
- Polje `variables` sadrži literale.
- Klasa `Clause` sadrži dva konstruktora. Prvi za ulazni parametar prima duljinu elementarne disjunkcije, a drugi za ulazne parametre prima duljinu elementarne disjunkcije i literale.

```
15 public class CNF {
16     int[][] variables;
17     int numberOfClauses;
18     int clauseLength;
19
20     CNF(int numberOfClauses, int clauseLength) {
21         variables = new int[numberOfClauses][clauseLength];
22         this.numberOfClauses = numberOfClauses;
23         this.clauseLength = clauseLength;
24     }
```

```
25      CNF(String fileName, int numberOfClauses, int clauseLength) {
26
27          variables = new int[numberOfClauses][clauseLength];
28          this.numberOfClauses = numberOfClauses;
29          this.clauseLength = clauseLength;
30          int numberOfClausesCounter = 0;
31
32          try {
33              File CNFfile = new File(fileName) ;
34              Scanner scan = new Scanner(CNFfile);
35
36              while(scan.hasNextLine()) {
37                  String line = scan.nextLine();
38                  char firstSign = line.charAt(0);
39                  if(firstSign == 'c' || firstSign == 'p' ||
                       firstSign == '0' || firstSign == '%')
40                      continue;
41                  String[] literals = line.split(" ");
42                  for(int z = 0; z < clauseLength; ++z) {
43                      variables[numberOfClausesCounter][z] =
                           Integer.parseInt(literals[z]);
44                  }
45                  numberOfClausesCounter++;
46                  if(numberOfClausesCounter == numberOfClauses)
47                      break;
48              }
49              scan.close();
50          } catch(FileNotFoundException e) {
51              System.out.println("Datoteka nije pronadena.");
52              e.printStackTrace();
53          }
54      }
55      public Clause getClauseByIndex(int index) {
56          return new Clause(this.clauseLength, this.variables[index
               ]);
57      }
58  }
```

Slika 6.3: Program `CNF.java`

Klasa `CNF` predstavlja jednu *k*-knf. Slijedi detaljniji opis te klase:

- Varijabla `clauseLength` sadrži broj literala u elementarnim disjunkcijama konjunktivne normalne forme.
- Varijabla `numberOfClauses` sadrži broj elementarnih disjunkcija u konjunktivnoj normalnoj formi.
- Dvodimenzionalno polje `variables` sadrži literale konjunktivne normalne forme.
- Klasa `CNF` sadrži dva konstruktora. Prvi za ulazne parametre prima broj elementarnih disjunkcija i broj literala u elementarnim disjunkcijama. Drugi za ulazne parametre prima ime datoteke iz koje učitavamo *k*-knf (izgled datoteke je prikazan na slici 6.1), broj elementarnih disjunkcija i broj literala u elementarnim disjunkcijama.
- Funkcija `getClauseByIndex` kao ulazni parametar prima redni broj elementarne disjunkcije u *k*-knf koju želimo dohvatiti i vraća tu elementarnu disjunkciju.

```
59 public class Individual {
60     boolean[] values;
61     int size;
62     static Random random = new Random();
63 
64     Individual() {
65         values = new boolean[0];
66         size = 0;
67     }
68     Individual(int size) {
69         values = new boolean[size];
70         this.size = size;
71     }
72     Individual(boolean[] values, int size) {
73         this.values = values;
74         this.size = size;
75     }
76     public static Individual generateRandomIndividual(int size) {
77         Individual newIndividual = new Individual(size);
78         for(int i = 0; i < size; ++i) {
79             newIndividual.values[i] = random.nextBoolean();
80         }
81         return newIndividual;
82     }
```

```
83      public int evaluateIndividual(CNF formula) {
84          Clause nClause;
85          int result = 0;
86
87          for(int i = 0; i < formula.numberOfClauses; ++i) {
88              nClause = formula.getClauseByIndex(i);
89              for(int j = 0; j < nClause.clauseLength; ++j) {
90                  if((values[ Math.abs(nClause.variables[j]) -1] &&
                        nClause.variables[j] > 0) ||
91                          (!values[ Math.abs(nClause.variables[j])
                                -1] && nClause.variables[j] < 0)) {
92                      result++;
93                      break;
94                  }
95              }
96          }
97          return result;
98      }
99      public static Individual recombine(Individual firstIndividual
            , Individual secondIndividual) {
100         Individual newIndividual = new Individual(firstIndividual
                .size);
101         for(int i = 0; i < newIndividual.size; ++i) {
102             if(random.nextBoolean() == true) {
103                 newIndividual.values[i] = firstIndividual.values[
                        i];
104             } else {
105                 newIndividual.values[i] = secondIndividual.values
                        [i];
106             }
107         }
108         return newIndividual;
109     }
110     public void mutate(float probabilityOfChange) {
111         if(random.nextFloat() < probabilityOfChange) {
112             int index = ThreadLocalRandom.current().nextInt(0,
                    this.size);
113             this.values[index] = !(this.values[index]);
114         }
115     }
116 }
```

Slika 6.4: Program `Individual.java`

Klasa `Individual` predstavlja jednu parcijalnu interpretaciju $I : \{P_1, \ldots, P_n\} \rightarrow \{0, 1\}$, čija je domena skup svih propozicionalnih varijabli koje se pojavljuju u k-knf. Slijedi detaljniji opis te klase:

- Polje `values` sadrži parcijalnu interpretaciju.
- Varijabla `size` sadrži kardinalitet domene parcijalne interpretacije.
- Klasa `Individual` sadrži tri konstruktora. Prvi ne prima nikakve ulazne parametre. Drugi konstruktor za ulazni parametar prima kardinalitet domene parcijalne interpretacije. Treći konstruktor za ulazne parametre prima polje vrijednosti parcijalne interpretacije i kardinalitet domene parcijalne interpretacije.
- Funkcija `generateRandomIndividual` generira parcijalnu interpretaciju. $I(P_i)$, $i \in \{1, \ldots, n\}$ (n je kardinalitet domene parcijalne interpretacije), je pridružena neka slučajna vrijednost iz skupa $\{0, 1\}$.
- Funkcija `evaluateIndividual` za ulazni parametar prima k-knf. Za danu k-knf i parcijalnu interpretaciju I, ova funkcija određuje broj elementarnih disjunkcija iz k-knf, A_i, za koje vrijedi $I(A_i) = 1$.
- Funkcija `recombine` za ulazne parametre prima dva objekta tipa `Individual` i vraća objekt tipa `Individual`, koji je kombinacija primljena dva objekta. Dakle, ova funkcija prima dvije parcijalne interpretacije, I_1 i I_2, i vraća parcijalnu interpretaciju I_3. $I_3(P_i)$ je slučajno odabrana vrijednost iz skupa $\{I_1(P_i), I_2(P_i)\}$.
- Funkcija `mutate` za ulazni parametar prima vjerojatnost promjene vrijednosti interpretacije u nekoj propozicionalnoj varijabli. Propozicionalna varijabla u kojoj će se promijeniti interpretacija se izabire slučajno.

```
117 public class Population {
118     Individual[] population;
119     int size;
120     int[] grades;
121     int maxGrade;
122     int maxGradeIndex;
123     static Random random = new Random();
```

```
125     Population(int size) {
126         population = new Individual[size];
127         grades = new int[size];
128         this.size = size;
129         maxGrade = -1;
130         maxGradeIndex = -1;
131     }
132     Population(int size, Individual[] individuals, int[] grades,
            int maxGradeIndex, int maxGrade) {
133         this.size = size;
134         this.population = individuals;
135         this.grades = grades;
136         this.maxGradeIndex = maxGradeIndex;
137         this.maxGrade = maxGrade;
138     }
139     Population(Individual[] individuals, int size) {
140         this.population = individuals;
141         this.maxGradeIndex = -1;
142         this.maxGrade = -1;
143         this.size = size;
144         grades = new int[size];
145     }
146     public static Population generatePopulation(int
            populationSize, int individualSize) {
147         Population newPopulation = new Population(populationSize)
                ;
148         for(int i = 0; i< populationSize; ++i) {
149             newPopulation.population[i] = Individual.
                    generateRandomIndividual(individualSize);
150         }
151         return newPopulation;
152     }
153     public void evaluate(CNF formula) {
154
155         for(int i = 0; i < this.size; ++i) {
156             grades[i] = this.population[i].evaluateIndividual(
                        formula);
157             if(grades[i] > maxGrade) {
158                 maxGrade = grades[i];
159                 maxGradeIndex = i;
160             }
161         }
162     }
```

```
163     public Population selection(int newPopulationSize, int
            numberOfClauses) {
164         Individual[] individuals = new Individual[
                newPopulationSize];
165         int[] newGrades = new int[newPopulationSize];
166         int newMaxGradeIndex = 0;
167         int currentNumberOfIndividuals = 0;
168 
169         while(currentNumberOfIndividuals < newPopulationSize) {
170             for(int i = 0; i < this.size; ++i) {
171                 if(this.grades[i] == numberOfClauses) {
172                     individuals[currentNumberOfIndividuals] =
                            this.population[i];
173                     newGrades[currentNumberOfIndividuals] = this.
                            grades[i];
174                     currentNumberOfIndividuals++;
175                     if(currentNumberOfIndividuals ==
                            newPopulationSize)
176                         break;
177                 }
178             }
179             numberOfClauses--;
180         }
181         return new Population(newPopulationSize, individuals,
                newGrades, newMaxGradeIndex, this.maxGrade);
182     }
183     public Population recombination(int newPopulationSize, int
            numberOfNewIndividuals) {
184         Individual[] individuals = new Individual[
                newPopulationSize];
185 
186         for(int i = 0; i < this.size; ++i) {
187             individuals[i] = this.population[i];
188         }
189         for(int j = newPopulationSize - numberOfNewIndividuals; j
                < newPopulationSize; j++) {
190             Individual firstIndividual = individuals[
                    ThreadLocalRandom.current()
191                         .nextInt(0, newPopulationSize -
                            numberOfNewIndividuals)];
192             Individual secondIndividual = individuals[
                    ThreadLocalRandom.current()
193                         .nextInt(0, newPopulationSize -
                            numberOfNewIndividuals)];
```

```
194             Individual newIndividual = Individual.recombine(
                    firstIndividual, secondIndividual);
195             individuals[j] = newIndividual;
196         }
197         return new Population(individuals, newPopulationSize);
198     }
199     public void mutation(float probabilityOfChange) {
200         for(int i = 0; i < this.size; ++i) {
201             this.population[i].mutate(probabilityOfChange);
202         }
203     }
204     public String best20Individuals() {
205         String best20Individuals = "";
206         int bestIndividualsCounter = 0;
207         int currentGrade = maxGrade;
208 
209         while( currentGrade >= 0) {
210             for( int i = 0; i < size; ++i ) {
211                 if( grades[i] == currentGrade ) {
212                     for(int j = 0; j < populatoion[i].size; ++j)
                            {
213                         int individualValue = population[i].
                                values[j] ? 1 : 0;
214                         best20Individuals += individualValue + "
                                " ;
215                     }
216                     best20Individuals += "% ";
217                     bestIndividualsCounter++;
218                     if(bestIndividualsCounter == 20) {
219                         return best20Individuals;
220                     }
221                 }
222             }
223             currentGrade--;
224         }
225         return best20Individuals;
226     }
```

```
227     public void involveNewIndividuals(String newIndividuals, int
           numberOfNewIndividuals) {
228         StringTokenizer tokenizer = new StringTokenizer(
               newIndividuals, " ");
229
230         while(tokenizer.hasMoreTokens()){
231             int randomInt = random.nextInt(size);
232             boolean[] values = new boolean[population[0].size];
233
234             while(true) {
235                 String token = tokenizer.nextToken();
236                 int j = 0;
237                 if(token.equals(%))
238                     break;
239                 if(token.equals("1")){
240                     values[j] = true;
241                     ++j;
242                 }
243                 if(token.equals("0")) {
244                     values[j] = false;
245                     ++j;
246                 }
247             }
248             Individual newIndividual = new Individual(values,
                   population[0].size);
249             population[randomInt] = newIndividual;
250         }
251     }
252 }
```

Slika 6.5: Program `Population.java`

Klasa `Population` predstavlja populaciju (skup) parcijalnih interpretacija. Slijedi detaljniji opis te klase:

- `population` je polje objekata tipa `Individual`, dakle `population` predstavlja populaciju parcijalnih interpretacija.
- Varijabla `size` sadrži broj elemenata populacije.
- `grades` je polje koje za svaku parcijalnu interpretaciju iz populacije sprema vrijednost koju daje funkcija `evaluateIndividual`.

- Varijabla `maxGrade` sadrži najveću vrijednost iz polja `grades`.
- Varijabla `maxGradeIndex` sadrži indeks elementa populacije kojem pripada `maxGrade`.
- Klasa `population` ima tri konstruktora. Prvi konstruktor za ulazni parametar prima broj elemenata populacije. Drugi konstruktor za ulazne parametre prima broj elemenata populacije, populaciju parcijalnih interpretacija, polje `grades`, varijable `maxGradeIndex` i `maxGrade`. Treći konstruktor za ulazne parametre prima populaciju parcijalnih interpretacija i broj elemenata populacije.
- Funkcija `generatePopulation` prima dva ulazna parametra `populationSize` i `individualSize`. Funkcija `generatePopulation` kreira populaciju, veličine `populationSize`, parcijalnih interpretacija čiji kardinalitet domene je `individualSize`.
- Funkcija `evaluate` za ulazni parametar prima *k*-knf i primjenjuje funkciju `evaluateIndividual`, s tom *k*-knf kao ulaznim parametrom, na svaku parcijalnu interpretaciju populacije. Na prethodno naveden način, funkcija `evaluate` izračunava elemente polja `grades`. Također, funkcija `evaluate` određuje varijable `maxGrade` i `maxGradeIndex`.
- Funkcija `selection` vraća populaciju, čiji je broj elemenata `newPopulationSize`, parcijalnih interpretacija koje imaju najveći pripadni element `grades[i]`. Dakle, funkcija `selection` vraća populaciju najboljih parcijalnih interpretacija za prethodno zadanu *k*-knf.
- Funkcija `recombination` za ulazne parametre prima `newPopulationSize` i `numberOfNewIndividuals`. Funkcija `recombination` iz populacije stvara još `numberOfNewIndividuals` novih parcijalnih interpretacija pomoću funkcije `recombine` klase `Individual`.
- Funkcija `mutation` za ulazni parametar prima `probabilityOfChange`, te poziva funkciju `mutate`, s tim parametrom, na elementima populacije.
- Funkcija `best20Individuals` pronalazi dvadeset najboljih (prema varijabli `grades[i]`) parcijalnih interpretacija i vraća njihovu reprezentaciju u obliku `String`-a (vrijednosti parcijalne interpretacije $I(P_i)$ su odvojene razmacima, a same parcijalne interpretacije su odvojene znakom "%" i razmakom).
- Funkcija `involveNewIndividuals` kao ulazni parametar prima `String` reprezentaciju parcijalnih interpretacija kakvu vraća funkcija `best20Individuals`

i te parcijalne interpretacije uključuje u populaciju umjesto slučajno odabranih parcijalnih interpretacija zadane populacije.

```
253 public class DistributedGA {
254     public static void main(String[] args) {
255 
256         final int numberOfPropositionalVariables = 75;
257         final int numberOfClauses = 325;
258         final int numberOfLiterals = 3;
259         final int populationSize = 400;
260         final int newPopulationSize = 200;
261         final int runTime = 5000;
262         final float probabilityOfChange = 0.9f;
263 
264         Linker comm = null;
265         Msg m;
266 
267         try {
268             // Povezivanje procesa
269             String baseName = args[0];
270             int myId = Integer.parseInt( args[1] );
271             int numProc = Integer.parseInt( args[2] );
272             comm = new Linker( baseName, myId, numProc );
273             // Ucitavanje formule
274             CNF formula = new CNF("** putanja do knf ** ",
                    numberOfClauses, numberOfLiterals);
275 
276             // **Jedan otok distribuiranog genetskog algoritma**
277 
278             long startTime = System.currentTimeMillis();
279             // Generiranje slucajne populacije i evaluacija te
                    populacije
280             Population population = Population.generatePopulation
                    (populationSize, numberOfPropositionalVariables);
281             population.evaluate(formula);
282             int numberOfEvaluations = 1;
283 
284             long endTime = System.currentTimeMillis();
285             long timeElapsed = endTime - startTime;
```

```
287             // Petlja genetskog algoritma u kojoj se redom
                   izvrsavaju:
288             // selekcija, rekombinacija, mutacija i migracija.
289             while(population.maxGrade < formula.numberOfClauses
                   && timeElapsed < runTime) {
290
291                 population = population.selection(
                       newPopulationSize, numberOfClauses);
292                 population = population.recombination(
                       populationSize, newPopulationSize);
293                 population.mutation(probabilityOfChange);
294                 population.evaluate(formula);
295
296                 numberOfEvaluations++;
297
298                 if( numberOfEvaluations % 100 == 0 ) {
299                     String best20Individuals = population.
                           best20Individuals();
300
301                     for( int i = 0; i < numProc; i++ ) {
302                         if( i != myId ) {
303                             comm.sendMsg(i, "tag",
                                   best20Individuals);
304                         }
305                     }
306                     for( int i = 0; i < numProc; i++ ) {
307                         if( i != myId ) {
308                             m = comm.receiveMsg(i);
309                             String newIndividuals = m.getMessage
                                   ();
310                             population.involveNewIndividuals(
                                   newIndividuals,
                                   numberOfPropositionalVariables);
311                             population.evaluate(formula);
312                         }
313                     }
314                 }
315                 endTime = System.currentTimeMillis();
316                 timeElapsed = endTime - startTime;
317             }
```

```
318                 System.out.println(population.maxGrade);
319                 System.out.println("Broj evaluacija: " +
                        numberOfEvaluations);
320             }
321             catch( Exception e ) {
322                 System.err.println(e);
323             }
324         }
325 }
```

Slika 6.6: Program `DistributedGA.java`

Klasa `DistributedGA` sadrži `main` funkciju u kojoj se izvršava distribuirani genetski algoritam za rješavanje MAXSAT problema. Slijedi detaljniji opis `main` funkcije. Na početku `main` funkcije definirane su konstante koje koristimo kasnije u programu:

- `numberOfPropositionalVariables`... kardinalitet domene parcijalne interpretacije.
- `numberOfClauses`... broj elementarnih disjunkcija u k-knf za koju rješavamo MAXSAT problem.
- `numberOfLiterals`... broj literala u elementarnim disjunkcijama k-knf.
- `populationSize`... broj parcijalnih interpretacija u populaciji.
- `newPopulationSize`... broj parcijalnih interpretacija u populaciji nakon primjene funkcije `selection`. Dakle, to je broj najboljih parcijalnih interpretacija koje će ostati u populaciji nakon selekcije.
- `runTime`... vrijeme izvršavanja petlje genetskog algoritma (selekcije, rekombinacije, mutacije i migracije) u milisekundama.
- `probabilityOfChange`... vjerojatnost mutacije parcijalne interpretacije.

Objekt klase `Linker` se koristi za povezivanje procesa (otoka) distribuiranog genetskog algoritma i slanje (primanje) poruka između procesa. Objekt klase `Msg` se koristi za spremanje poruka koje se razmjenjuju između procesa (otoka). Nakon toga se odvija glavni dio distribuiranog genetskog algoritma koji je opisan u 6 (s $P(i)$ je označena populacija u i-toj iteraciji algoritma).

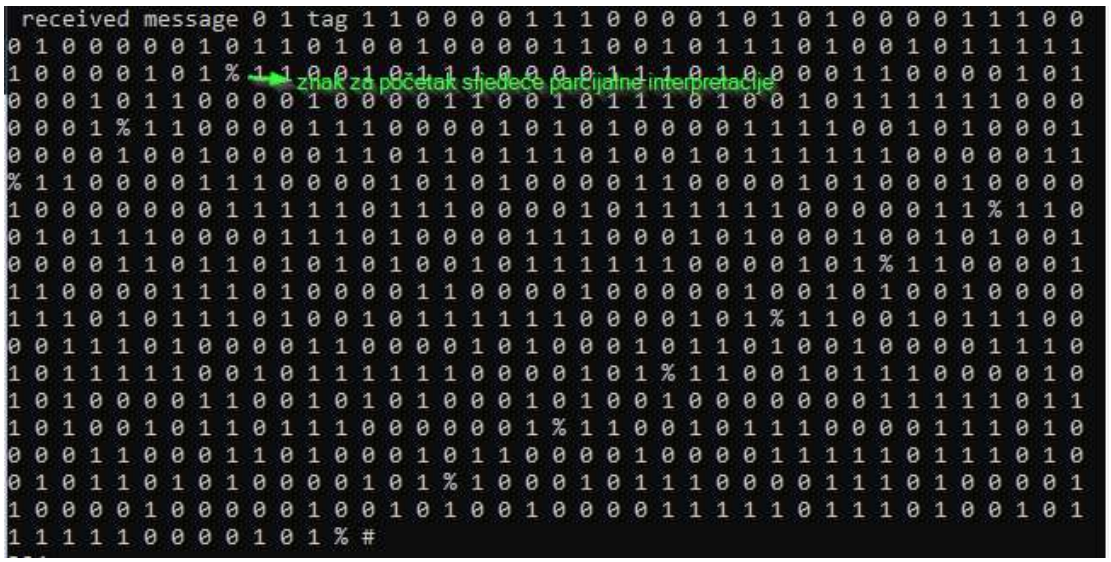
Algorithm 6 Distribuirani genetski algoritam

```
Generiraj(P(0))
Evaluiraj(P(0))
t = 0
while not KriterijZaustavljanja(P(t)) do
    P'(t) =Selekcija(P(t))
    P''(t) =Rekombinacija(P'(t))
    P'''(t) =Mutacija(P''(t))
    if Kriterij migracije je zadovoljen then
        P''''(t) =Migracija(P'''(t))
        Evaluiraj(P''''(t))
        P(t + 1) = P''''(t)
    else
        Evaluiraj(P'''(t))
        P(t + 1) = P'''(t)
    end if
    t = t + 1
end while
```

i-ti proces se pokreće u naredbenom redku oblika:

```
> java DistributedGA <bazno ime> <i> <N>,
```

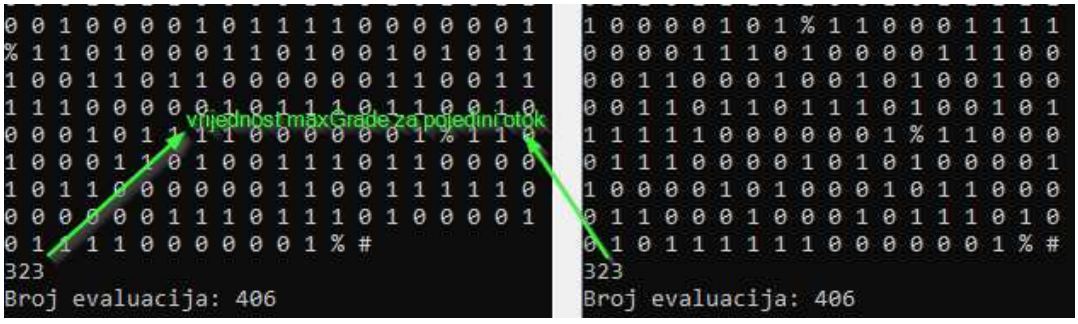
gdje je `bazno ime` jednako za sve procese, `i` redni broj procesa, a `N` ukupan broj procesa. Također, napomenimo da za vrijeme rada programa u drugom prozoru mora raditi i `NameServer`.



Slika 6.7: Primjer poruke između procesa (otoka)

Na slici 6.7 prikazan je primjer poruke koju je proces s rednim brojem `1` primio od procesa s rednim brojem `0`. U poruci se nalazi dvadeset parcijalnih interpretacija koje su zapisane na način koji je opisan prilikom objašnjenja funkcije `best20Individuals` klase `Population`.

Na slici 6.8 je prikazan ispis distribuiranog genetskog algoritma s 2 otoka.



Slika 6.8: Ispis distribuiranog genetskog algoritma s 2 otoka

6.3 Eksperimentalni rezultati

Sada ćemo usporediti distribuirani genetski algoritam sa sekvencijalnim genetskim algoritmom za rješavanje MAXSAT problema. Vrijednosti parametara s kojima pokrećemo oba

algoritma su:

- `populationSize` = 400,
- `newPopulationSize` = 200
- i `probabilityOfChange` = 0.9.

Algoritme izvršavamo na 3-knf koja ima 325 elementarnih disjunkcija i 75 propozicionalnih varijabli. Algoritme pokrećemo za nekoliko različitih vremena izvršavanja, a za svako vrijeme izvršavanja eksperiment ponavljamo 30 puta. Vrijeme izvršavanja se odnosi na vrijeme proteklo od početka rada svakog od procesa do njegovog završetka rada. U nastavku tablično prikazujemo rezultate eksperimenta. Broj ispred ”dGA” označava broj procesa od kojih se sastoji distribuirani genetski algoritam. Sa ”sGA” je označen sekvencijalni genetski algoritam. Stupac ”Prosječan broj evaluacija” sadrži zbroj broja evaluacija u svim izvršavanjima algoritma podijeljen s brojem izvršavanja. Stupac ”Prosječan `maxGrade`” sadrži zbroj varijabli `maxGrade` na kraju izvršavanja svih ponavljanja algoritma podijeljen s brojem izvršavanja.

	Prosječan broj evaluacija	Prosječan `maxGrade`
sGA	226.16	322.67
2 dGA	210.17	322.90
3 dGA	170.30	323.17
4 dGA	139.83	323.20
5 dGA	116.53	323.10
6 dGA	105.07	322.97

Tablica 6.1: Rezultati izvršavanja algoritma za vrijeme izvršavanja od 1 sekunde

	Prosječan broj evaluacija	Prosječan `maxGrade`
sGA	483.53	322.77
2 dGA	401.33	323.10
3 dGA	333.94	323.37
4 dGA	287.87	323.30
5 dGA	238.90	323.30
6 dGA	200.20	323.30

Tablica 6.2: Rezultati izvršavanja algoritma za vrijeme izvršavanja od 2 sekunde

	Prosječan broj evaluacija	Prosječan `maxGrade`
sGA	1179.13	322.87
2 dGA	951.42	323.40
3 dGA	836.70	323.54
4 dGA	715.19	323.77
5 dGA	560.75	323.40
6 dGA	445.36	323.50

Tablica 6.3: Rezultati izvršavanja algoritma za vrijeme izvršavanja od 5 sekundi

	Prosječan broj evaluacija	Prosječan `maxGrade`
sGA	1793.17	323.00
2 dGA	1614.97	323.47
3 dGA	1347.40	323.60
4 dGA	1164.54	323.63
5 dGA	870.27	323.13
6 dGA	734.27	323.33

Tablica 6.4: Rezultati izvršavanja algoritma za vrijeme izvršavanja od 8 sekundi

Iz rezultata vidimo da distribuirani genetski algoritam pronalazi bolja rješenja nego sekvencijalni genetski algoritam. Dakle, raznolikija pretraga prostora rješenja distribuiranog genetskog algoritma rezultira pronalaskom boljih rješenja od sekvencijalnog genetskog algoritma. Također, primijetimo da se porastom broja procesa smanjuje broj evaluacija distribuiranog genetskog algoritma. To se događa zato što se više vremena troši na komunikaciju između procesa. Primijetimo još da dodavanje većeg broja procesa ne rezultira pronalaskom boljih rezultata.

Pokažimo kako promjena parametra `probabilityOfChange` utječe na kvalitetu rezultata. Eksperiment provodimo s istim parametrima kao i ranije i s istim brojem ponavljanja. Vrijeme izvršavanja iznosi 5 sekundi i koristimo distribuirani genetski algoritam s 3 procesa. Tablica 6.5 prikazuje rezultate eksperimenta.

`probabilityOfChange`	Prosječan broj evaluacija	Prosječan `maxGrade`
0.10	1055.73	323.26
0.30	1029.03	323.36
0.50	1029.33	323.46
0.70	948.93	323.30
0.90	859.30	323.60
0.95	824.97	323.50

Tablica 6.5: Rezultati izvršavanja algoritma za različite vrijednosti parametra `probabilityOfChange`

Povećavanjem vrijednosti parametra `probabilityOfChange` postižemo veću raznolikost pretrage prostora rješenja. Ta raznolikost do određene vrijednosti parametra (kod nas 0.9) pozitivno utječe na kvalitetu pronađenih rješenja. Također, primijetimo da povećanjem vrijednosti parametra `probabilityOfChange` broj evaluacija se smanjuje. To se događa zato što program troši više vremena na mutacije.

U sljedećem eksperimentu pokrećemo distribuirani genetski algoritam u kojem je kriterij zaustavljanja prolazak 500 iteracija petlje genetskog algoritma bez poboljšanja trenutno najboljeg rješenja. Eksperiment provodimo s istim parametrima kao i ranije i s istim brojem ponavljanja. Tablica 6.6 prikazuje rezultate eksperimenta. Stupac "Vrijeme izvršavanja" označava prosječno vrijeme izvršavanja distribuiranog genetskog algoritma u sekundama.

	Prosječan maxGrade	Prosječan broj evaluacija	Vrijeme izvršavanja
2 dGA	323.20	622.10	3.22
3 dGA	323.20	604.60	3.61
4 dGA	323.37	609.90	4.33
5 dGA	323.40	625.20	5.71
6 dGA	323.37	607.05	7.00

Tablica 6.6: Rezultati izvršavanja algoritma, koji se zaustavlja nakon što u 500 uzastopnih iteracija ne dođe do poboljšanja, za različiti broj procesa.

Iz rezultata eksperimenta vidimo da algoritam pronalazi najbolje rješenje već nakon oko 110 iteracija, bez obzira na broj procesa. Također, možemo primijetiti da više procesa daju bolja rješenja, ali cijena za ta bolja rješenja je veći utrošak vremena. Bolja rješenja dobivamo zbog diverzifikacije pretrage, a u daljnjem radu mogli bi ispitati je li ovo ponašanje vrijedi za još veći broj procesa od 6.

U prethodnim eksperimentima smo uzimali uvijek istu veličinu populacije za svaki proces. Dakle, povećavanjem broja procesa, povećavala se ukupna populacija rješenja koja

razmatramo. Provjerimo sad je li veličina ukupne populacije utjecala na kvalitetu rješenja. Programe izvršavamo, kao i ranije, 30 puta za svaki broj procesa. Vrijednosti parametara s kojima pokrećemo program su:

- `populationSize` = $\frac{1000}{\text{broj procesa}}$,
- `runTime` = 5000,
- `probabilityOfChange` = 0.9
- i `newPopulationSize` = $\frac{\texttt{populationSize}}{2}$.

U tablici 6.7 i tablici 6.8 su prikazani rezultati eksperimenta.

	`populationSize`	Prosječan broj evaluacija	Prosječan `maxGrade`
sGA	1000	446.47	323.07
2 dGA	500	841.53	323.43
3 dGA	333	1045.20	323.77
4 dGA	250	1182.43	323.50
5 dGA	200	910.33	323.47
6 dGA	167	1035.83	323.40

Tablica 6.7: Rezultati izvršavanja algoritma za ukupnu populaciju veličine 1000

	`populationSize`	Prosječan broj evaluacija	Prosječan `maxGrade`
sGA	2000	226.50	323.07
2 dGA	1000	386.07	323.40
3 dGA	667	527.50	323.57
4 dGA	500	553.53	323.73
5 dGA	400	561.53	323.46
6 dGA	333	554.76	323.63

Tablica 6.8: Rezultati izvršavanja algoritma za ukupnu populaciju veličine 2000

Iz rezultata eksperimenta zaključujemo da, iako sekvencijalni i distribuirani genetski algoritam imaju istu veličinu ukupne populacije, distribuirani genetski algoritam daje bolja rješenja. Dakle, diverzifikacija pretrage i migracije pozitivno utječu na pronalazak boljih rješenja.

Poglavlje 7

Zaključak

U ovom radu opisali smo četiri distribuirane metaheuristike i implementirali distribuirani genetski algoritam za rješavanje MAXSAT problema. Sve opisane metaheuristike primjenjive su na mnoge probleme, te kod rješavanja problema pomoću distribuiranih metaheuristika potrebno je testirati razne mogućnosti kako bismo pronašli najbolju. Čak i kad izaberemo jednu distribuiranu metaheuristiku da bismo riješili neki problem, otvaraju se razne mogućnosti kako implementirati tu distribuiranu metaheuristiku i razne varijacije parametara s kojima ćemo rješavati zadani problem. Dakle, teorija distribuiranih metaheuristika nam daje jednu robusnu podlogu za rješavanje mnogih problema. U podpoglavlju 6.3 vidjeli smo, na primjeru distribuiranog genetskog algoritma za rješavanje MAXSAT problema, da distribuirane metaheuristike daju bolja rješenja od sekvencijalnih metaheuristika. Dakle, samo distribuiranje algoritama nam daje razne načine kako poboljšati taj algoritam. Distribuirane metaheuristike ne daju egzaktna rješenja pa ih ne možemo primjenjivati na probleme koji zahtijevaju egzaktno rješavanje.

Kao daljnji prostor za poboljšanje implementacije distribuiranog genetskog algoritma iz poglavlja 6.2 moglo bi se pokušati s raznim varijantama funkcija rekombinacije i migracije. Također, moglo bi se probati izvršiti algoritam s više procesa, te na više primjeraka problema kako bismo dobili značajniju procjenu kvalitete algoritama. Još jedan mogući prostor za daljnji rad mogao bi biti implementacija ostalih distribuiranih metaheuristika i primjena na razne probleme. Mogli bismo probati implementirati i razne kombinacije distribuiranih metaheuristika, tzv. hibridne distribuirane metaheurisitike, te ih primijeniti na iste probleme. Hibridne distribuirane metaheuristike koje prvo koriste distribuiranu metaheuristiku koja diverzificira potragu u prostoru rješenja, pa zatim, kad pronađemo obećavajuće područje za pretragu, koriste distribuiranu metaheuristiku koja se temelji na lokalnoj pretrazi su obećavajuće područje za daljnje istraživanje.

Bibliografija

[1] E. Alba, *Parallel Metaheuristics - A New Class of Algorithms*, Wiley - Interscience, Hoboken NJ, 2005.

[2] M. Gendreau i J-Y. Potvin, *Handbook of Metaheuristics, Third Edition*, Springer, Cham CH, 2018.

[3] R. Manger, *Distribuirani procesi*, PMF-Matematički odsjek, 2017, `http://web.studenti.math.pmf.unizg.hr/~manger/protect/DP-Skripta.pdf`.

[4] P. Siarry, *Metaheuristics*, Springer, Cham CH, 2016.

[5] El-Ghahazali Talbi, *Metaheuristics - From Design to Implementation*, Wiley, Hoboken NJ, 2009.

[6] M. Vuković, *Složenost algoritama*, PMF-Matematički odsjek, 2019, `https://www.pmf.unizg.hr/_download/repository/SA-skripta-2019%5B1%5D.pdf`.

Sažetak

U ovom diplomskom radu opisane su četiri distribuirane metaheuristike: distribuirani genetski algoritam, distribuirana mravlja optimizacija, distribuirano simulirano kaljenje i distribuirano tabu-traženje. Također, opisano je kako se mogu analizirati performanse distribuiranih metaheuristika. Predstavljena je implementacija distribuiranog genetskog algoritma koja je primijenjena na MAXSAT problem. Nakon predstavljanja implementacije distribuiranog genetskog algoritma analizirane su njezine performanse.

Summary

In this master thesis, we described four distributed metaheuristics: distributed genetic algorithms, distributed ant colony optimization, distributed simulated annealing and distributed tabu search. Also, the principle of analyzing the performance of distributed metaheuristics is described and the implementation of a distributed genetic algorithm applied to the MAXSAT problem is presented. After the presentation of the implementation of the distributed genetic algorithm, its performance was analyzed.

Životopis

Rođen sam u Šibeniku 20. 10. 1997. godine. Završio sam Osnovnu školu Vidici te prirodoslovno-matematički smjer Gimnazije Antuna Vrančića u Šibeniku. U Zagreb sam se preselio 2016. godine te sam upisao preddiplomski sveučilišni studij Matematika na Prirodoslovno-matematičkom fakultetu. Nakon što sam 2019. godine završio preddiplomski studij, upisao sam diplomski sveučilišni studij Računarstvo i matematika.

Tijekom cijelog razdoblja osnovne škole trenirao sam košarku u šibenskom klubu ŠK Dražen Petrović. Nakon upisa u srednju školu i prestanka igranja u klubu nastavio sam se aktivno baviti rekreacijskim sportom.