Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andrej Čopar

Kombinatorična optimizacija s porazdeljeno implementacijo genetskih algoritmov

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Marko Robnik-Šikonja

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00020/2012
Datum: 10.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANDREJ ČOPAR**

Naslov: **KOMBINATORIČNA OPTIMIZACIJA S PORAZDELJENO IMPLEMENTACIJO GENETSKIH ALGORITMOV**

COMBINATORIAL OPTIMIZATION WITH DISTRIBUTED GENETIC ALGORITHMS

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Pri številnih računsko zahtevnih problemih skušamo najti dobre rešitve s kombinatorično optimizacijo. Pogosto se uporabljajo hevristične metode, denimo genetski algoritmi.
Zasnujte porazdeljeno implementacijo genetskega algoritma in jo preizkusite na zbirki problemov trgovskega potnika. Analizirajte različne parametre algoritma in porazdeljenosti ter poskušajte izboljšati delovanje algoritma z izmenjavo informacij med posameznimi računskimi enotami v predlagani arhitekturi.

Mentor:
prof. dr. Marko Robnik Šikonja

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Čopar, z vpisno številko **63090054**, sem avtor diplomskega dela z naslovom:

Kombinatorična optimizacija s porazdeljeno implementacijo genetskih algoritmov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Marka Robnik-Šikonje,
- 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 7. septembra 2012 Podpis avtorja:

Zahvala

Zahvaljujem se svojemu mentorju prof. dr. Marku Robnik-Šikonji za nasvete in pomoč pri izdelavi diplomskega dela. Iskreno se zahvaljujem tudi prijateljem in pa seveda družini za vso podporo.

Kazalo

Povzetek

S kombinatorično optimizacijo skušamo najti dobre rešitve pri številnih računsko zahtevnih problemih. Te probleme pogosto rešujemo s hevrističnimi metodami, denimo genetskimi algoritmi. V diplomskem delu opišemo razreda kompleksnosti NP in P ter povezave med njima. Opišemo genetske algoritme in različne modele paralelizacije. V osrednjem delu implementiramo porazdeljen genetski algoritem z uporabo sheme strežnik-odjemalec in modela z otoki. Razvijemo komunikacijski protokol in grafični uporabniški vmesnik. Analiziramo različne parametre algoritma in porazdeljenosti ter algoritem preizkusimo na zbirki problemov trgovskega potnika.

Ključne besede:

kombinatorična optimizacija, NP polni problemi, genetski algoritmi, problem trgovskega potnika, porazdeljeno računanje

Abstract

With combinatorial optimization we try to find good solutions for many computationaly difficult problems. For this type of problems we often use metaheuristics such as genetic algorithms. We describe complexity classes NP and P and relation between them. We present genetic algorithms with different parallelization models. Main part consists of distributed genetic algorithm implementation using client-server schema and island model. We develop communications protocol and graphical user interface. We analyze several algorithm and distribution parameters and test our implementation using traveling salesman problem collection.

Key words:

combinatorial optimization, NP complete problems, genetic algorithms, traveling salesman problem, distributed computing

Poglavje 1

Uvod

Z večanjem števila računalnikov se nam ponuja vse več procesne moči, ki je porazdeljena po celem svetu in jo je zato težko izkoristiti. Z mrežno implementacijo lahko realiziramo take algoritme, ki z uporabo več računalnikov razširi procesno moč za reševanje težjih problemov. Za reševanje optimizacijskih problemov se pogosto uporabljajo evolucijski algoritmi, ki iščejo rešitve problema z uporabo metod iz naravnega sveta. Genetski algoritmi so vrsta evolucijskih algoritmov, ki jih uporabljamo za optimizacijo težkih problemov, katerih rešitev se ne da najti v sprejemljivem času. Sestavljeni so iz korakov selekcije, križanja in mutacije.

V poglavju 2 podrobneje predstavimo razrede kompleksnosti P in NP ter njune lastnosti in razlike. Problem trgovskega potnika uvrstimo med NP polne probleme in opišemo, kako mu ocenimo kvaliteto obhoda.

V poglavju 3 razložimo delovanje genetskih algoritmov in opišemo glavne faze: selekcijo, križanje in mutacijo. Zatem opišemo nekaj načinov paralelizacije, med njimi tudi model z otoki. Ta način omogoča istočasno izvajanje več algoritmov, ki si rešitve delijo med seboj.

Cilj diplomskega dela je zasnova porazdeljenega genetskega algoritma, ki ga predstavimo v poglavju 4. Program je razdeljen na več podenot, med drugim mrežo strežniških aplikacij, ki na zahtevo poganjajo programe na poljubnem računalniku, če pred tem zaženemo strežnik. Sestavljen je tudi

iz programa odjemalca in grafičnega vmesnika. Sledi opis protokola, preko katerega strežniki in odjemalci med seboj komunicirajo, na koncu pa podrobneje opišemo, kako smo realizirali posamezne faze algoritma.

V poglavju 5 analiziramo parametre genetskega algoritma ter stopnjo porazdeljenosti. Primerjamo algoritme, ki delujejo neodvisno z algoritmi, ki med izvajanjem komunicirajo z drugimi delujočimi enotami. V zaključku zapišemo sklepne ugotovitve ter možne izboljšave.

Poglavje 2

Opis problema

V računalništvu se srečujemo s problemi različnih težavnosti. Probleme s podobnimi lastnostmi uvrstimo v razrede, za katere veljajo določene zakonitosti. Najpogosteje se srečujemo z razredoma P in NP. Razred P predstavlja probleme, ki jih lahko učinkovito rešimo, razred NP pa probleme, za katere lahko učinkovito potrdimo rešitev.

V teoriji kompleksnosti algoritmov se še vedno srečujemo z mnogimi nerešenimi vprašanji, eno izmed njih je vprašanje enakosti $P = NP$. Bistvo tega vprašanja je, ali obstaja hiter algoritem za iskanje rešitev NP polnih problemov. Reševanje problema od začetka se precej razlikuje od preverjanja, ali je dana rešitev pravilna. Pri P problemih je iskanje rešitev enostavno, za NP probleme pa velja, da je iskanje rešitev težje kot pa ugotavljanje njene pravilnosti. Pri danem problemu, za katerega lahko rešitev učinkovito preverimo, nas zanima ali obstaja metoda, ki hitro najde tako rešitev. V tem poglavju opišemo razred NP, razpravljamo o kompleksnosti problema trgovskega potnika in predstavimo nekatere izmed osnovnih lastnosti teh problemov.

2.1 NP polni problemi

NP polni problemi so družina kombinatoričnih problemov, ki so med seboj tesno povezani. Zbirka teh problemov vključuje veliko znanih problemov iz diskretne matematike, kot na primer problem trgovskega potnika, problem barvanja grafov, problem neodvisnih množic in problem pakiranja nahrbtnika. Ti težki problemi imajo lahko tudi zelo enostavne primere. Za graf s tremi vozlišči je npr. zelo enostavno najti najkrajši obhod. V nadaljevanju opišemo glavne lastnosti teh problemov ter osnovne pojme s tega področja.

2.1.1 Lastnosti

Zapišimo skupne značilnosti NP polnih problemov:

- problemi se vsi zdijo zelo težki in za nobenega še ni odkrit polinomski algoritem.
- ni dokazano, da polinomski čas reševanja za te probleme ne obstaja.
- če najdemo hiter algoritem za enega od NP polnih problemov, obstaja hiter algoritem za vse probleme iz te družine.
- če bi dokazali, da ne obstaja hiter algoritem za enega od NP polnih problemov, potem ne obstaja hiter algoritem za nobenega od teh problemov.

2.1.2 Odločitveni in optimizacijski problem

Odločitveni problem je vprašanje, ki zahteva pozitiven ali negativen odgovor. Večina problemov, ki jih preučujemo je optimizacijskih problemov. Ti imajo lahko množico različnih rešitev. Odločitveni in optimizacijski problemi so si med seboj podobni. Vprašanje "Ali lahko pobarvamo graf G s k barvami?" je odločitveni problem. "Kaj je najmanjše število barv, s katerimi lahko

pobarvamo graf G?" pa je optimizacijski problem. Če najdemo hiter algoritem za odločitveni problem, lahko ponavadi z malo dodatnega dela rešimo ustrezen optimizacijski problem.

2.1.3 Razred NP

NP je družina odločitvenih problemov, za katere se da v polinomskem času preveriti pravilnost rezultata. Za razliko od polinomskih problemov, za katere se da tudi rešitev najti hitro, pri NP problemih rešitve ne moremo enostavno najti, če pa jo že imamo, jo lahko hitro preverimo. Za problem trgovskega potnika dobimo rešitev, ki ima dolžino obhoda skozi vsa mesta enako k. Da preverimo, če obhod res obišče vsa mesta natanko enkrat in pride nazaj, potrebujemo algoritem, ki deluje v polinomskem času.

2.1.4 NP polnost

Odločitveni problem je NP poln, če spada med NP probleme in je vsak problem iz NP učinkovito prevedljiv na ta problem. Če lahko dokažemo, da je odločitveni problem Q NP poln, potem lahko usmerimo svoj trud v iskanje hevrističnega algoritma za ta problem. Hiter algoritem za en NP poln problem bi pomenil, da obstaja hiter algoritem za ostale NP polne probleme, ki jih lahko prevedemo na problem Q.

2.2 Problem trgovskega potnika

V grafu z danim naborom vozlišč in razdalij med pari vozlišč, je problem trgovskega potnika (TSP) najti najkrajšo pot, ki obišče vsa vozlišča natanko enkrat in se vrne v začetno vozlišče. Iščemo najkrajšo skupno razdaljo, ki jo mora potnik prepotovati. Pri osnovnem tipu problema, ki jo preučujemo, so razdalje med vozlišči simetrične, tako da je razdalija iz X v Y enaka razdaliji iz Y v X. Problem trgovskega potnika je eden najbolj preučevanih problemov v računski matematiki. Spada med NP polne probleme, za katere

ne poznamo hitrega algoritma. Da TSP spada med NP polne probleme, bomo dokazali s pomočjo problema Hamiltonovega cikla. Hamiltonova pot je pot v neusmerjenem grafu, ki obišče vsako vozlišče natanko enkrat. Če se Hamiltonova pot zaključi v isti točki, v kateri se je začela, to pot imenujemo Hamiltonov cikel. Ugotavljanje, ali v grafu obstaja taka pot, je NP poln problem. Dokažimo, da TSP pripada družini NP polnih problemov.

Dokaz. Sprva pokažemo da TSP spada med NP probleme. Kot možno rešitev uporabimo zaporedje n vozlišč, ki mu pravimo obhod. Potrebujemo algoritem, ki preveri, da to zaporedje vsebuje vsako vozlišče natanko enkrat, sešteje vse poti in preveri, če je vsota največ k. Ta proces lahko zagotovo naredimo v polinomskem času. Za dokaz, da je TSP NP-poln, potrebujemo še prevedbo iz znanega NP polnega problema. To lahko naredimo s prevedbo problema na problem Hamiltonovega cikla. Pokažemo, da je Hamiltonov cikel $\leq_p$ TSP. Naj bo $G = (V, E)$ primer Hamiltonovega cikla. Problem trgovskega potnika zgradimo tako, da oblikujemo popoln graf $G' = (V, E')$, kjer je $E' = \{(i, j) : i, j \in V \ in \ i \neq j\}$, in definiramo funkcijo cene c po naslednji formuli.

$$c(i,j) = \begin{cases} 0 & \text{če } (i,j) \in E, \\ 1 & \text{če } (i,j) \notin E. \end{cases} \tag{2.1}$$

Pokažemo, da ima graf G Hamiltonov cikel natanko tedaj, ko ima graf G' obhod dolžine največ 0. Predpostavimo, da ima graf G Hamiltonov cikel h. Vsaka povezava v h pripada E in ima v grafu G' ceno 0. Torej je h obhod v G' s ceno 0. Nato predpostavimo, da ima graf G' obhod h' s ceno največ 0. Ker v E' ni negativnih povezav, je edina možna cena obhoda h' točno 0, kjer imajo vse povezave ceno 0. Torej h' vsebuje samo povezave v E. Iz tega sledi, da je h' Hamiltonov cikel v grafu G. □

TSP smo prevedli na problem Hamiltonovega cikla in dokazali, da spada med NP težke probleme. Iz tega sledi, da TSP spada med NP-polne probleme. V nadaljevanju je predstavljenih več različic problema trgovskega potnika, kjer zahtevamo, da so določene povezave del rešitve problema.

2.2.1 Simetrični TSP

Pri dani množici n vozlišč in povezavami med vsakim parom vozlišč, moramo najti najkrajši obhod, ki obišče vsako vozlišče natanko enkrat in se vrne v začetno vozlišče. Razdalja od vozlišča i do vozlišča j je enaka razdalji od j do i.

2.2.2 Asimetrični TSP

Pri dani množici n vozlišč, moramo najdi najkrajši obhod, ki obišče vsako vozlišče natanko enkrat, a v tem primeru razdalja od i do j ni nujno enaka razdalji od j do i.

2.2.3 Problem urejanja zaporedja

Ta problem je asimetrični problem trgovskega potnika z dodatnimi omejitvami. Pri dani množici n vozlišč in razdaljami med njimi, moramo najti Hamiltonovo pot iz vozlišča 1 do vozlišča n z minimalno dolžino, pri čemer upoštevamo prioriteto omejitev. Vsaka prioritetna omejitev zahteva, da mora biti vozlišče i obiskano pred vozliščem j.

2.2.4 Problem usmerjanja vozil s kapacitetami

Podanih imamo $n-1$ vozlišč, eno skladišče ter razdalje od vozlišč do skladišča in med vozlišči. Vsa vozlišča imajo zahteve, ki morajo biti izpolnjene. Za dostavo do vozlišč imamo na razpolago tovornjake z identičnimi kapacitetami. Problem je najti obhode tovorjakov z minimalno skupno dolžino, ki izpolnijo zahteve vseh vozlišč, ne da bi kršili omejitve kapacitet tovorjakov. Število

tovorjakov ni navedeno. Vsak obhod obišče podmnožico vozlišč in začne ter konča v skladišču.

2.3 Računanje razdalje med mesti

Opišemo lahko različne načine računanje razdalj med mesti. Najbolj osnoven način merjenja razdalje je evklidska razdalja, poznamo pa tudi manhattansko, maksimalno in geografsko. Dobljena razdalja je ocena kakovosti osebkov, ki jih kasneje uporabimo v genetskih algoritmih. Manjša razdalja pomeni boljšo rešitev.

2.3.1 Evklidska razdalja

Evklidska razdalja med točkami p in q, je izračunana po formuli (2.2), kjer je n dimenzija prostora. Razdalijo med mesti v ravnini izračunamo z formulo (2.2).

$$d_{euc}(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} \tag{2.2}$$

2.3.2 Manhattanska razdalja

Pri manhattanski razdalji merimo razdaljo, kot da se sprehajamo po pravokotni mreži.

$$d_{man}(p,q) = \sum_{i=1}^{n} |(p_i - q_i)| \tag{2.3}$$

2.3.3 Maksimalna razdalja

Maksimalna razdalja je razdalja med dvema komponentama točk, ki sta najbolj oddaljeni med sabo.

$$d_{max}(p,q) = max(d(p_1,q_1), d(p_2,q_2), \cdots, d(p_n,q_n)) \tag{2.4}$$

2.3.4 Geografska razdalja

V primeru, da je TSP geografski problem, vozlišča ustrezajo točkam na Zemlji in razdalje med njimi ustrezajo razdalji na krogli s polmerom $R = 6378\ km$. Koordinate vozlišča so geografska širina in dolžina, razdaljo med mesti i in j pa dobimo po formuli (2.5)

$$d_{geo}(p, q) = R \cdot acos(0.5\ q_2(1 + q_1) - q_3(1 - q_1)) + 1) \tag{2.5}$$

Dolžine q_1, q_2 in q_3 izračunamo po naslednjih enačbah.

$$q_1 = cos(long_i - long_j) \tag{2.6}$$

$$q_2 = cos(lat_i - lat_j) \tag{2.7}$$

$$q_3 = cos(lat_i + lat_j) \tag{2.8}$$

Poglavje 3

Genetski algoritmi

Genetski algoritmi so vrsta optimizacijskih algoritmov, ki s pomočjo mehanizmov naravne evolucije iščejo rešitve problemov, ki jih je nepraktično reševati s tradicionalnimi metodami. Ne zagotavljajo najboljših rešitev, večinoma pa dajejo dobre približke v sprejemljivem času. Delujejo tako, da na populaciji osebkov, ki predstavljajo možne rešitve, izvajajo tehnike selekcije, križanja, mutacije in nadomeščanja osebkov z novimi. V tem poglavju opišemo posamezne faze genetskih algoritmov ter načine kako lahko algoritem paraleliziramo.

3.1 Predstavitev osebkov

Genetski algoritmi za predstavitev osebkov uporabljajo več vrst zapisov:

- bitno - osebke zapišemo z zaporedjem bitov: 010011
- permutacije - osebke zapišemo kot permutacijo: (1,2,5,3)
- vektorsko - osebke predstavimo z vektorjem števil: (5.2, 14.1, 3,2)
- nizovno - osebke predstavimo kot zaporedje znakov: ATCGA
- drevesno - osebke predstavimo z drevesno strukturo: (1+3)*(1-5)

Za problem trgovskega potnika je smiselna predstavitev s permutacijami, ker se pri premutacijah vsako mesto pojavi samo enkrat in križanje zato ne proizvede neveljavnih rešitev. Prednost permutacij je podobna predstavitev podobnih osebkov in omogoča enostavno računanje raznolikosti.

3.2 Selekcija

Selekcija je eden izmed najpomembnejših procesov v genetskem algoritmu. Ideja selekcije je ohranjanje boljših osebkov, ki v populaciji nadomestijo slabše, pri tem pa se poskušati izogniti prezgodnji konvergenci populacije v lokalni minimum. Izbiranje osebkov samo glede na funkcijo kakovosti preveč spodbuja določene osebke, ki pa niso nujno najboljši za napredovanje populacije. Z uporabo nekaterih mehanizmov, na primer stohastičnega vzorčenja, zagotavljamo stopnjo diverzitete in damo možnost tudi nekoliko slabšim osebkom, ki pa vsebujejo dober genetski material. Izbiro osebkov s proporcionalno, rangovno, turnirsko in enoturnirsko izbiro ter stohastičnim univerzalnim vzorčenjem bomo predstavili v naslednjih podpoglavjih.

3.2.1 Proporcionalna izbira

Proporcionalna izbira je preprost način izbire, pri katerem je verjetnost, da bo osebek izbran za razmnoževanje proporcionalna njegovi funkciji kakovosti (Φ). Verjetnosti izbire normaliziramo, z vsoto kvalitete vseh osebkov v populaciji (μ) in dobimo:

$$Pr_{prop}(i) = \frac{\Phi(i)}{\sum_{i=1}^{\mu} \Phi(i)} \tag{3.1}$$

Boljši osebki imajo večjo verjetnost, da bodo izbrani, še vedno pa dopuščamo možnost, da izberemo slabše osebke. Slabost te izbire je, da osebki z višjo kvaliteto prevladajo, in selekcija postane podobna izbiri najboljših osebkov.

3.2.2 Rangovna izbira

Pri rangovni izbiri je verjetnost, da bo osebek izbran sorazmerna z njegovo pozicijo v seznamu osebkov populacije, urejenem glede na funkcijo kakovosti. Osebkom v urejenem seznamu priredimo rang r_i, glede na njegovo pozicijo v seznamu. Uporaba pozicije namesto kakovosti osebka izravna velike razlike med osebki. Verjetnost izbire izračunamo po formuli (3.2).

$$Pr_{rang}(i) = \frac{r_i}{\sum_{i=j}^{\mu} r_j} \tag{3.2}$$

3.2.3 Stohastično univerzalno vzorčenje

Stohastično univerzalno vzorčenje je metoda podobna proporcionalni izbiri, ki mogoča manjšo varianco glede na funkcijo kvalitete. Poteka tako, da osebke razvrstimo na številski trak med 0 in 1, sorazmerno njihovi kvaliteti po enačbi:

$$p_i = \frac{f_i}{\sum_{i=j}^{n} f_j} \tag{3.3}$$

Določimo N osebkov, ki jih želimo generirati in naključno število r, ki ga izberemo iz intervala $[0, 1/N]$. Osebke izberemo na mestih $r+i/N, i \in [0..N-1]$.

3.2.4 Turnirska izbira

Pri turnirski izbiri opravimo več turnirjev. Za posamezen turnir izberemo iz populacije g elementov in jih med seboj primerjamo. Tisti, ki ima v primerjavi boljšo rešitev zmaga. Za vsak osebek štejemo, kolikokrat je zmagal v dvoboju. Tisti z največjim številom zmag je najboljši osebek turnirja. Da dodamo nedeterminizem, je zmagovalec izbran z verjetnostjo p, drugi najboljši z verjetnostjo $p(1-p)$, tretji z $p(1-p)^2$ in tako naprej. Turnir je lokaliziran na g elementov, zato imajo možnost tudi osebki, ki niso absolutno najboljši. Prednost turnirske izbire je, da populacije ni treba predhodno urediti glede na funkcijo kakovosti. Ker turnirji potekajo neodvisno, je možna tudi paralelizacija.

3.2.5 Enoturnirska izbira

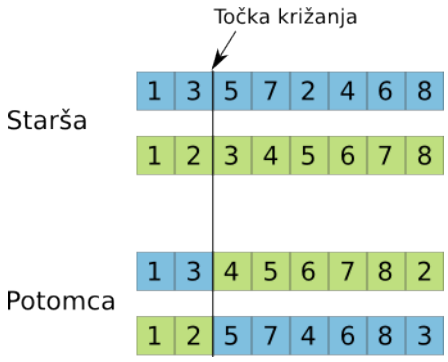
Enoturnirska izbira je preprost način hkratne izbire osebkov za razmnoževanje in istočasno nadomeščanje osebkov. Najprej razbijemo populacijo na skupine velikosti g, nato pa znotraj skupine potomca najboljših osebkov nadomestita najslabša osebka v skupini. Prednost enoturnirske izbire je, da preživi v naslednjo generacijo $(g - 2)$ najboljših osebkov in zato ne izgubljamo osebkov z najboljšo kvaliteto. Po drugi strani strategija zagotavlja, da še tako dober osebek nima več kot dveh potomcev, in tako ohranja raznolikost populacije.

3.3 Križanje

Namen križanja je pridobivanje novih osebkov s kombiniranjem dednega zapisa izbranih staršev. Idealno bi potomca križanih staršev podedovala dobre lastnosti obeh staršev in imela boljšo oceno kakovosti. Če otroci dobijo kombinacijo slabših lastnosti, jih kmalu nadomestimo z novimi, boljšimi osebki. Poznamo več načinov križanja, najbolj se uporablja eno- ali dvomestno križanje. Nekateri osebki imajo del dednega materiala dober, del pa slab. Če uspemo dobri del obdržati, slabšega pa zamenjati z boljšim, najdemo boljše rešitve. Treba je paziti, da osebki po križanju ohranijo veljaven dedni zapis. Za problem trgovskega potnika to zagotovimo z uporabo permutacij, s katerimi v nadaljevanju predstavimo eno- in dvomestno križanje.

3.3.1 Enomestno križanje

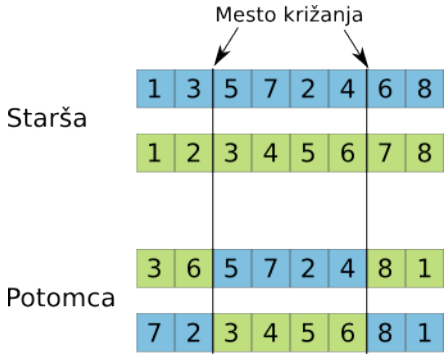
Pri enomestnem križanju najprej naključno izberemo točko križanja. Do te točke ostane dedni material enak, preostanek prevzamemo od drugega starša, pri tem pa izpustimo mesta, ki jih permutacija že vsebuje. Primer enomestnega križanja na problemu TSP je predstavljen na sliki 3.1.



Slika 3.1: Enomestno križanje permutacij.

3.3.2 Dvomestno križanje

Dvomestno križanje poteka tako, da naključno izberemo dve točki križanja, med katerima se genetski material ne spremeni. Za prvega potomca prevzamemo dedni material drugega starša od druge točke križanja naprej, pri tem pa izpustimo mesta, ki že obstajajo v permutaciji prvega starša. Če pridemo do konca permutacije, se pomaknemo na začetek in prevzem dednega materiala nadaljujemo. Za drugega potomca je postopek enak, prevzemamo pa dedni material prvega starša pred spremembami. Primer postopka za križanje dveh osebkov TSP z 8 mesti je prikazan na sliki 3.2.



Slika 3.2: Dvomestno križanje permutacij.

3.4 Mutacija

Evolucijski algoritmi potrebujejo mehanizme za ohranjanje raznolikosti populacije. Eden takih mehanizmov je mutacija, kjer z naključnim spreminjanjem genetskega zapisa izbranih osebkov preprečujemo, da bi tekom generacij raznolikost populacije padla. Z mutacijami pridobivamo osebke z dednim zapisom, ki ni omejen na variacije osebkov iz začetnega nabora in zato zmanjša verjetnost, da ostanemo v lokalnem minimumu. Prevelika verjetnost mutacij ni koristna, ker generiramo naključne rešitve in zmanjšamo učinkovitost preiskovanja. V bitnem zapisu so mutacije spreminjanje vrednosti bita nič v ena in obratno. Drugačne so mutacije na osebkih predstavljenih s permutacijami, kjer spreminjamo položaj dveh ali več mest, saj moramo paziti, da ne ustvarimo neveljavnih permutacij. Pri nekaterih algoritmih v istem koraku izvedemo več mutacij in izberemo najboljšo.

3.5 Nadomeščanje osebkov

Skozi generacije pridobivamo nove osebke, zato je treba skrbeti da se populacija ne razširi preveč, saj to upočasni delovanje algoritma. Poznamo več tehnik, kako nadomestimo osebke v populaciji. Pri nekaterih načinih selekcije, na primer enoturnirski izbiri, ta korak ni potreben, ker imajo že vključen postopek za nadomeščanje osebkov. V naslednjih podpoglavjih opišemo najbolj pogosta načina nadomeščanja osebkov.

3.5.1 Nadomesti vse

Pri tem načinu zamenjamo vse osebke z njihovimi potomci, pri tem pa lahko uporabimo iste tehnike kot pri selekciji, npr. proporcionalno ali rangovno izbiro, stohastično univerzalno vzorčenje, itd.

3.5.2 Elitizem

Elitizem je strategija, kjer se del najboljših osebkov prenese v naslednjo generacijo. Dobra stran je, da imamo v vsaki generaciji na voljo najboljše osebke in jih ni potrebno posebej hraniti.

3.6 Paralelizacija

Evolucijske algoritme lahko paraleliziramo, kar poveča hitrost, včasih pa tudi kvaliteto rešitev za isti problem. Prednosti paralelizacije so odvisne od števila istočasno izvajajočih se enot in začasne neodvisnosti osebkov. Prevladujoča motivacija za paralelizacijo je pohitritev in je odvisna od podrobnosti arhitekture, ki izvaja paralelen algoritem. Paralelne arhitekture ločimo glede na stopnjo zrnatosti. Groba zrnatost predstavlja večje, šibko povezane enote, na primer več računalnikov povezanih prek mreže. Fina zrnatost predstavlja manjše, močneje povezane enote, na primer večjedrni procesorji. V nadaljevanju bomo opisali več modelov paralelizacije, vsak je primeren za drugo vrsto zrnatosti. Za fino zrnate paralelne sisteme je bolj primeren model z več procesorji, za srednje zrnate model soseske, za grobo zrnate pa model z otoki.

3.6.1 Več procesorjev

Ta način paralelizacije ne zahteva spremembe delovanja algoritma. Poteka tako, da dele kode, ki se lahko izvajajo vzporedno, izvajamo na različnih procesorjih. Uporablja se pri računalnikih z več jedri, nitmi ali procesorji, kjer prenos podatkov med posameznimi enotami ni preveč zamuden. Imamo eno skupno populacijo, ocena kvalitete pa poteka vzporedno.

3.6.2 Model z otoki

Model z otoki je vrsta paralelnega algoritma z ločenimi subpopulacijami, ki jih imenujemo otoki. Selekcija, križanje in mutacije potekajo v vsakem otoku

izolirano. Z določeno verjetnostjo ali po poteku vnaprej določenega časovnega intervala se posamezniki selijo med otoki. Tako se informacija med različnimi pojavitvami genetskega algoritma izmenjuje, hkrati pa se ohranja določena stopnja izoliranosti. Ideja tega modela je zmanjšati čas računanja in povečati kvaliteto rešitve za določen problem z distribucijo problema na več pojavitev algoritma, ki delujejo vzporedno.

Selitveni vzorec je pomemben za potek evolucije in je določen s stopnjo povezanosti, stopnjo komunikacije in njeno frekvenco. Ti parametri določajo stopnjo izolacije in interakcije med otoki. Ko povečujemo stopnjo povezanosti, večamo frekvenco komunikacije. Čas izolirane evolucije je skrajšan, kar je podobno eni večji populaciji. Velike populacije kmalu dosežejo stopnjo stabilnosti in proces se ustavi. Če se stopnja povezanosti manjša, model z otoki čedalje bolj deluje kot več samostojnih poskusov zaporednih algoritmov z manjšimi populacijami. Vmesne stopnje povezanosti in pogostosti interakcij omogočajo dinamiko, ki je dovolj velika, da zadostuje za izkoriščanje in uporabo. Naslednja koda predstavlja potek algoritma modela z otoki:

```
// parameter E pomeni večjih iteracij, ki vsebujejo selitve
// parameter N pomeni število otokov
void function island_model(E,N){
  // vsakemu otoku populacijo priredimo vzporedno
  concurently for (i=1; i<N; i++){
    initialize subpopulation P;
  }
  for(i=0; i<E; i++) {
    // na vsakem otoku izvajamo genetski algoritem G generacij
    concurrently for (i=1; i<N; i++) {
      GA(P,G);
    }
    // za vsak otok sprožimo selitev
    for(i=0; i<N; i++) {
      for each neighbor j of i {
        migrate (P_i,P_j);      // selitev osebkov med populacijami
        assimilate (P_i);       // osebke vključimo v populacijo
      }
    }
  }
  problem solution = best individual of all subpopulations
}
```

Število otokov, ki so vključeni v algoritem označuje parameter N. Celotna struktura modela z otoki tvori E večjih iteracij, med posameznimi iteracijami vsaka subpopulacija G generacij neodvisno izvaja evolucijski algoritem GA. Po vsaki iteraciji je komunikacijska faza, med katero se osebki selijo med sosednjimi otoki. Po vsaki selitvi vsaka subpopulacija vključi priseljence. Ta vključitveni korak je odvisen od podrobnosti selitvenega procesa. Če velikost subpopulacije ostaja enaka tudi po selitvi, je asimilacija samo ocena kakovosti, sicer mora vključevati redukcijo. Ta model vsebuje popolnoma ločene populacije, zato je primeren tudi za grobo zrnate arhitekture.

3.6.3 Model soseske

Model soseske vsebuje populacije, ki se med seboj prekrivajo. Vsakemu posamezniku je omejena lokalna regija v kateri se ne sme križati. Na primer, če je populacija razvrščena kot sferična struktura, posameznikom ni dovoljeno križanje s sosedi znotraj nekega polmera. Za take sisteme učinek selitve pridobimo z izbiro. Paralelni genetski algoritmi z prekrivajočimi subpopulacijami so najbolj primerni za srednje zrnate paralelne arhitekture, pri katerih je pomnilnik deljen.

Poglavje 4

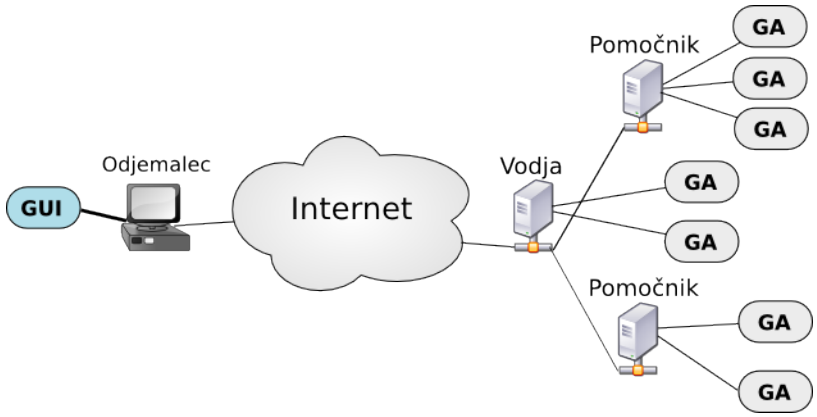
Realizacija porazdeljenega algoritma

Genetske algoritme lahko razdelimo na več med seboj neodvisnih enot, ki z različnimi naključno izbranimi populacijami najdejo boljšo rešitev kot ena sama enota. Algoritem izboljšamo tako, da omogočimo enotam sočasno izvajanje in medsebojno komunikacijo, kar vpliva na hitrost in kvaliteto rešitve. Model odjemalec-strežnik omogoča komuniciranje prek interneta, zato je dober za sisteme, kjer se programi izvajajo na različnih računalnikih. Strežniška komponenta omogoča storitev, odjemalec pa odpira zahtevke. V tem poglavju bomo predstavili arhitekturo, s katero je aplikacija realizirana, njene komponente in način komunikacije med njimi.

4.1 Arhitektura porazdeljenega genetskega algoritma

Pri implementaciji je uporabljen model odjemalec/strežnik, kjer je strežnik porazdeljena aplikacija, ki deluje na več računalnikih (slika 4.1). Strežniška enota je sestavljena iz več programov, izmed katerih eden prevzame vlogo vodje, drugi pa vlogo pomočnikov. Programi so med seboj šibko povezani prek internetnih vtičnic (ang. *socket*). Na vsakem strežniku lahko teče

več enot genetskega algoritma. Enote se med seboj razlikujejo po različnih parametrih, na primer velikosti populacije, maksimalnem številu generacij, verjetnosti mutacije in verjetnosti selitev. V tem podpoglavju bomo opisali vse enote, iz katerih je sestavljen naš algoritem. Algoritem vsebuje enega strežnik-vodjo, ki daje ukaze več strežnikom-pomočnikom. Vsak strežnik-pomočnik lahko gosti poljubno enot genetskega algoritma. Odjemalec pošilja zahtevke oddaljenemu strežniku-vodji, grafični vmesnik pa obdela podatke, ki jih odjemalec sprejme prek interneta.



Slika 4.1: Shema arhitekture aplikacije.

4.1.1 Strežnik-vodja

Strežnik-vodja je osrednja enota, preko katere poteka komunikacija z ostalimi enotami. Preden začnemo z iskanjem rešitev je treba najprej prebrati koordinate mest iz datoteke v podatkovno strukturo. Strežnik-vodja hrani:

- seznam mest in njihovih koordinat,
- seznam vseh enot, ki sodelujejo pri problemu,
- selitveno populacijo, ki jo lahko pobrišemo z ukazi `stop`, `reset` ali `open`,

- najboljšo najdeno rešitev in
- zgodovino rešitev posameznih enot.

Strežnik-vodja je zmožen kreirati nove podprocese algoritma. Ker zasede vrata za komunikacijo, lahko na enem računalniku sočasno deluje največ en strežnik. Strežnik-vodja je napisan v programskem jeziku C.

4.1.2 Strežnik-pomočnik

Na vsakem računalniku, na katerem želimo izvajati algoritem, poženemo strežnik. Lokacije strežnikov niso ustaljene, zato moramo ob zagonu strežnika v ukazni vrstici dodati IP številko strežnika-vodje. Tako podprogrami vedo kam pošiljati pakete. Naloga strežnika-pomočnika je upravljanje s procesi, ki predstavljajo posamezno enoto genetskega algoritma. Ukazi, ki jih strežnik-pomočnik lahko prevzame od strežnika vodje:

- nastavljanje parametrov - shrani parametre skupaj z številko enote algoritma,
- kreiranje instance algoritma - ustvari nov proces s prej nastavljenimi parametri,
- ustavljanje algoritma - ustavi proces, ki ustreza določeni številki enote.

Najprej strežniku pomočniku pošljemo identifikacijsko številko procesa in parametre, s katerimi bomo kasneje zagnali proces. Tako imamo možnost, da nastavimo parametre vseh procesov, preden se začnejo izvajati. Računsko moč računalnikov z več procesorji izkoristimo tako, da omogočimo več enot algoritma na enem računalniku. Strežnik to prepozna, če več enotam dodelimo isto IP številko. Procese lahko zaženemo posamično ali vse naenkrat. Za prenos informacij nazaj do strežnika vodje skrbijo podprocesi, strežnik-pomočnik čaka, če želimo predčasno ustaviti katerega od algoritmov, ali če želimo dodati nove enote.

4.1.3 Enota algoritma

Enota genetskega algoritma predstavlja en proces na enem od računalnikov. Vsaka enota ima svojo identifikacijsko številko zato, da lahko vsako posebej naslavljamo. Takoj, ko strežnik dobi zahtevo za zagon algoritma, ustvari podproces in si zapomni njegovo številko procesa. Podproces nato izvaja naslednjo kodo:

```
void function GA {
   i = 0;
   init_population P;          // naključno generiramo populacijo P
   evaluate P;                 // ocenimo osebke

   while i < max_iter {        // izvajamo korake max_iter generacij
      i = i + 1;
      select C from P;         // izberemo starša C
      crossover C to C';       // starša križamo v potomca C'
      mutate C';               // potocma lahko mutirata
      evaluate C';             // oceni potomca
      replace P from C';       // potomca nadomestita 2 osebka iz populacije
      if C' is new best {
        send C' to server;     // obvestimo strežnik če sta potomca najboljša
      }
      select M from P;         // izberemo osebke za selitev
      migrate M from P to server;  // izbrane osebke pošljemo strežniku
      migrate M' from server to P; // nove osebke dobimo od strežnika
   }
}
```

Posamezni ukazi v kodi so:

- init_population - algoritem prevzame seznam mest in njihovih koordinat od strežnika vodje, nato pa kreira n naključnih permutacij teh mest, kjer n predstavlja velikost populacije,
- select - z enoturnirsko izbiro izberemo osebke,
- crossover - križamo osebke z dvotočkovnim križanjem,
- mutate - mutacija se zgodi z verjetnostjo p_{mut},
- migrate - selitev se zgodi z verjetnostjo p_{mig},

- evaluate - ocena kvalitete se izračuna glede na tip podatkov, najpogosteje je to evklidska razdalja,
- test new best - če se izkaže, da je kateri izmed osebkov nov najboljši osebek, se rešitev pošlje strežniku-vodji,
- replace - zamenjaj osebke z novimi osebki.

4.1.4 Odjemalec

Odjemalec je program, ki pošlje strežniku zahtevek, strežnik pa mu odgovori z ustreznimi podatki. Naloga odjemalca je dekodiranje binarnih paketov dobljenih preko interneta v tekstovni zapis ter kontrola izvajanja algoritmov. Odjemalec deluje kot samostojna enota, zato ga lahko enostavno avtomatiziramo s skriptnim jezikom. Naša implementacija odjemalca je napisana v jeziku C in uporablja iste funkcije za serializacijo kot strežniki. Z odjemalcem lahko do podatkov dostopamo iz poljubne lokacije, če vemo IP naslov strežnika-vodje. Podatki, ki jih odjemalec sprejme so:

- seznam mest in koordinat,
- najboljšo do sedaj najdeno rešitev,
- zgodovino rešitev,
- seznam enot algoritma.

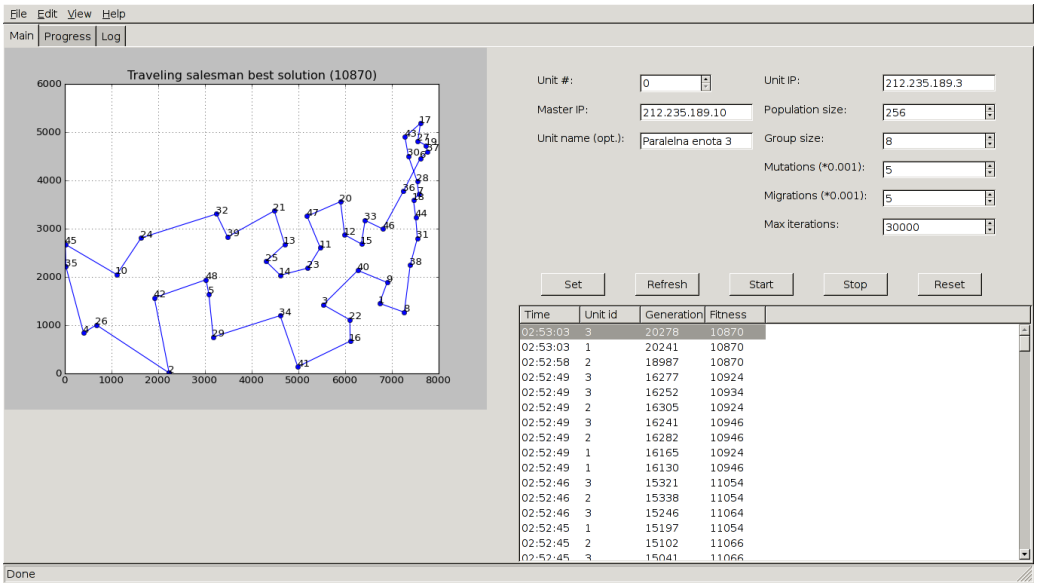
4.1.5 Grafični vmesnik

Aplikacijo lahko uporabljamo preko tudi preko grafičnega vmesnika, ki smo ga implementirali v programskem jeziku Python s knjižnico wxWidgets. Grafični vmesnik razširi funkcionalnost odjemalca z vizualno predstavitvijo rešitev in uporabniku prijazno kontrolno enoto. Vmesnik je sestavljen iz treh zavihkov. Na prvem je vizualizacija najboljše rešitev skupaj z vnosnimi polji za določanje parametrov in zgodovino rešitev, na drugem je graf kvalitete rešitve

v odvisnosti od časa, na tretjem pa dnevnik dogodkov in napak. Glavne funkcionalnosti vmesnika so:

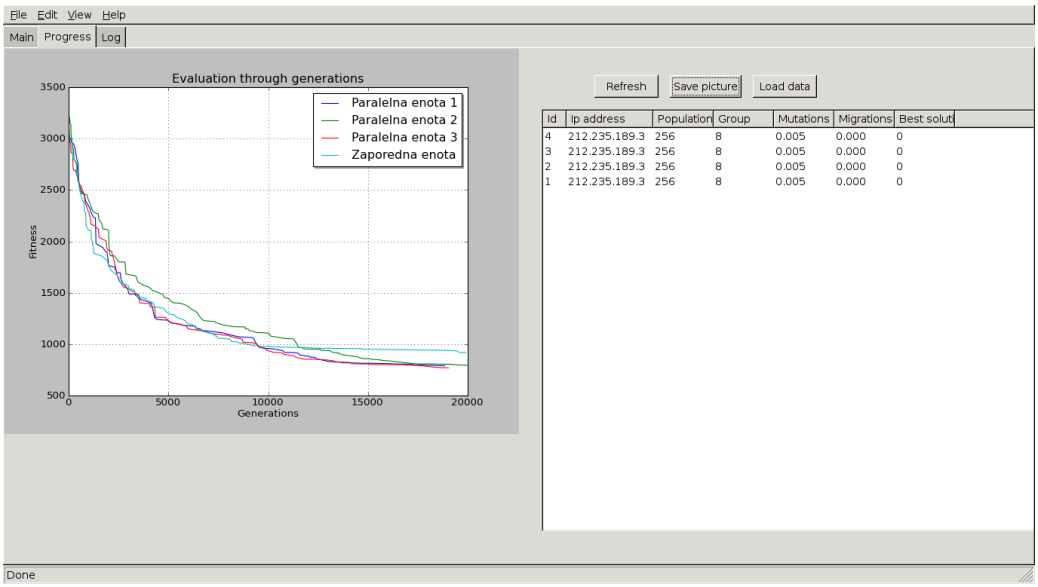
- dvo dimenzionalna predstavitev najboljše rešitve,
- nastavitev parametrov za posamezne enote,
- graf kvalitete rešitev v odvisnosti od časa,
- odpiranje datotek in
- zgodovina dobrih rešitev.

Pri osnovnem progledu imamo na levi strani grafično predstavitev trenutno najboljše rešitve. Njena vrednost je zapisana v naslovu grafa. Na desni imamo na vrhu vnosna polja za nastavitev parametrov ter dodatno polje, ki predstavlja ime enote vidne v legendi grafa napredka. Desno spodaj imamo izpis zgodovine dobrih rešitev glede na čas. Za posodobitev grafa ter tabele zgodovine moramo klikniti gumb 'Refresh'.



Slika 4.2: Osnovni pogled grafičnega vmesnika.

V drugem zavihku imamo pregled napredka. Na levi je graf kakovosti rešitve v odvisnosti od generacije populacije, na desni pa pregled parametrov enot. S klikom na gumb 'Refresh' izrišemo graf za trenutne podatke, z gumbom 'Load data' pa jih preberemo iz datoteke. Graf shranimo s klikom na gumb 'Save picture'. V tretjem zavihku je dnevnik klicev odjemalca skupaj z argumenti.



Slika 4.3: Pogled napredka algoritma.

4.2 Komunikacija med pojavitvami programa

Enote genetskih algoritmov med seboj ne komunicirajo neposredno. Informacije se prenašajo posredno prek strežnika vodje, ki hrani seznam vseh vozlišč prisotnih pri računanju. Ukazi, ki so namenjeni upravljanju strežnika, ne pa pridobivanju podatkov, odgovora strežnika ne potrebujejo, pri drugih strežnik odgovori z izpisom zahtevanih podatkov. V nadaljevanju bomo opisali, kako so podatki zakodirani v pakete, in tipe ukazov, na katere se strežnik odzove.

4.2.1 Protokol

Z besedo protokol opisujemo pravila, ki jih morajo upoštevati tako odjemalci kot strežniki, da je komunikacija uspešna:

- bajt 0 je rezerviran za operacijsko kodo ukaza,
- bajti 1-3 predstavljajo dolžino paketa, istočasno določajo maksimalno dolžino paketa $2^{24} - 16$ bajtov,
- bajti 4-7 predstavljajo prvi argument,
- bajt 8-11 predstavlja drugi argument,
- bajti 12-15 predstavljajo tretji argument,
- bajti od 16 do konca predstavljajo podatkovni del paketa.

Vsi trije argumenti so splošnonamenski in se uporabljajo po potrebi. Glava paketa je dolga 16 bajtov, kar je tudi najkrajša dolžina paketa. Podatkovni del je opcijski in se uporablja za pošiljanje koordinat, najboljših vozlišč, osebkov za migracijo, imen datotek ter seznamov za statistično obdelavo.

4.2.2 Enosmerna komunikacija

Ukazi, ki so namenjeni upravljanju strežnika, ali obveščanju o stanju, ne potrebujejo povratne informacije. Ko kličemo odjemalca, vrsto ukaza predstavlja prvi argument v ukazni vrstici. Drugi argument je IP številka strežnika-vodje, ostali argumenti sledijo po vrsti od tretjega naprej:

- quit - ustavi vse enote in se ugasne,
- stop i - ustavi podproces algoritma i ali vse podprocese, če je argument 0,
- done - podproces sporoči strežniku, da se je zaključil,

- start i - ustvari novo enoto algoritma i; z argumentom 0 ustvarimo vse enote istočasno,
- set i - nastavi parametre za podproces i,
- reset - ustavi algoritem, in pobriše podatke o vozliščih,
- open $file$ - prebere novo datoteko z imenom $file$.

4.2.3 Dvosmerna komunikacija

Pri zahtevkih za pridobivanje podatkov odjemalec pričakuje povratno informacijo. Posebnost je ukaz migrate, kjer zahtevka ne pošilja odjemalec, pač pa enota algoritma, ko se sproži postopek selitve. Pri selitvi se nekaj dobrih osebkov pošlje strežniku, ki vrne druge osebke iz selitvene populacije. Ukazi so:

- cities - vrne seznam mest s koordinatami,
- best - vrne trenutno najboljšo rešitev problema skupaj z njeno oceno,
- migrate - v zahtevku pošljemo kandidate za selitev, strežnik vrne osebke, ki so jih našle druge pojavitve algoritma,
- units - vrne seznam enot algoritma, njihove IP številke in parametre,
- history - vrne zgodovino dobrih rešitev.

4.3 Faze algoritma

Algoritem je sestavljen iz naslednjih faz: selekcija, križanje, mutacija in selitev. V nadaljevanju opišemo, kako delujejo in kako smo jih realizirali.

4.3.1 Selekcija

Uporabili smo enoturnirski način selekcije, ki zagotavlja ohranjanje dobrih osebkov. Potomci pri tej selekciji ne nadomestijo svojih staršev, saj bi v primeru slabih potomcev izgubili dobre rešitve. Skupina mora vsebovati najmanj dva starša ter dva potomca, zato je njena velikost navzdol omejena s štirimi osebki, navzgor pa z velikostjo populacije.

```
void function select {
  // izberemo osebke iz populacije in jih dodamo v skupino
  group g = random k individuals from P;
  sort g ascending; // sortiramo osebke po kvaliteti
  parent1 = g[0]; // najboljši osebek
  parent2 = g[1]; // drugi najboljši osebek
  child1 = g[k-1]; // najslabši osebek
  child2 = g[k-2]; // drugi najslabši osebek
}
```

4.3.2 Križanje

Uporabili smo dvomestno križanje. Najprej naključno izberemo dve poziciji v osebku. Podatke med točkama skopiramo iz staršev v potomca. Mesta, ki se nahajajo izven vmesnega območja skopiramo od drugega starša tako, da izpustimo elemente, ki jih potomec že vsebuje. V spodnji pseudokodi je zaradi lažje ponazoritve tabela prikazana krožno. Zapis `parent1+p2` predstavlja elemente, ki se začnejo pri indeksu p2 do konca, ter od začetka do p2.

```
void function crossover {
  // naključno izberemo dve mesti križanja
  p1 = random()%size;
  p2 = random()%size;

  // dedni material prvega starša
  child1[p1:p2] = parent1[p1:p2]
  // dedni material drugega starša brez vsebovanih mest
  child1[p2:p1] = parent2+p2 not in child1[p1:p2]
  // dedni material drugega starša
  child2[p1:p2] = parent2[p1:p2]
  // dedni material prvega starša brez vsebovanih mest
  child2[p2:p1] = parent1+p2 not in child2[p1:p2]
}
```

4.3.3 Mutacija

Mutacije se pojavljajo v dveh oblikah. Prva se zgodi z verjetnostjo p_{mut}, ki jo podamo kot argument. Pri tej obliki se zamenjata poziciji dveh naključno določenih mest. Da preprečimo izgubo dobrih rešitev se mutacija ohrani le, če je mutiran osebek kvalitetnejši od osebka pred mutacijo.

Druga oblika mutacij nadomesti podvojene osebke. Ko algoritem napreduje, so si osebki vse bolj podobni, zato so pojavitve duplikatov vse bolj verjetne. Križanje identičnih osebkov je nesmiselno, ker ustvari potomca, ki sta identična staršema. Tako križanje populacijo spreminja v monotono, kar povzroči, da se algoritem ustavi. Algoritem zato detektira podvojene osebke in jim naključno premeša ves dedni material. S tem dobivamo nove osebke tudi v poznejših fazah algoritma in upočasnimo konvergenco.

4.3.4 Selitev

Selitev se izvaja med različnimi enotami algoritma, ki jim pravimo otoki. Vsak otok ima svojo populacijo, na kateri izvaja selekcijo, križanje in mutacijo, včasih pa se z verjetnostjo p_{mig} izvede postopek selitve, ki poteka po naslednjih korakih:

- izberi starša P in otroka C iz populacije,
- pošlji starša P strežniku,
- strežnik pošlje nov osebek M nazaj in nadomesti M s staršem P,
- algoritem prejme M,
- nadomesti otroka C z novim osebkom M.

Za izbiro osebkov uporabljamo enoturnirsko izbiro, tako kot za križanje. Tako strežniku vedno pošljemo dobre osebke in jih istočasno obdržimo, slabe osebke pa nadomestimo z dobrimi priseljenci. Izbor osebka M na strežniku poteka naključno, da podenote dobijo čim bolj raznolike osebke. Osebki

v migracijski populaciji so vedno dobri, ker ne nastanejo z mutacijami ali križanjem, ampak so izbrani glede na kvaliteto, še preden jih podenote pošljejo nazaj. Selitvena populacija na strežniku predstavlja skupno elito vseh otokov udeleženih v selitvah.

Poglavje 5

Rezultati

V tem poglavju predstavimo, kako vpeljava modela z otoki vpliva na učinkovitost algoritma ter kakovost rešitev. S pomočjo grafičnih prikazov analiziramo, kako spreminjanje parametrov vpliva na potek algoritma. Meritve smo izvajali na dveh virtualnih strojih z operacijskim sistemom Linux, ki imata različna IP naslova, in tako preverili pravilnost delovanja.

5.1 Analiza modela z otoki

Algoritem smo izvajali na 2 računalnikih istočasno in primerjali kvaliteto rešitev pri različnem številu otokov. V tabeli 5.1 so podani parametri za posamezne konfiguracije. Za testno množico smo uporabili algoritem s 16, 8, 6 in 4 vzporedno delujočimi enotami, za kontrolno množico smo uporabili al-

Parametri konfiguracij						
konfiguracija	1	2	3	4	5	6
velikost populacije	128	128	128	128	128	128
velikost grupe	4	4	4	4	4	4
verjetnost mutacij	0 %	0 %	0 %	0 %	0 %	0 %
verjetnost migracij	5 %	5 %	5 %	5 %	0 %	0 %
število generacij	20000	20000	20000	20000	20000	20000
število vzporednih algoritmov	16	8	6	4	6	1

Tabela 5.1: Prikaz parametrov za posamezno konfiguracijo.

goritma brez migracij z 1 in 6 enotami. Mutacij nismo uporabili, da izboljšave lažje pripišemo vzporednemu računanju ne pa večjemu številu mutacij.

Za upravljanje algoritma smo uporabili odjemalec `gatspc`, ki sprejema argumente iz ukazne vrstice. Uporaba odjemalca olajša avtomatizacijo reševanja problemov. Skripta, ki izračuna podatke za 6 otokov v tabeli 5.3, vsebuje naslednjo kodo:

```
# nastavimo skupne argumente
args = $pop_size $group_size $max_iter $mutations
for problem in $library # iteracija po imenih datotek problemov
do
        gatspc "open $master $problem" # prebere podatke
        # nastavljanje parametrov za vsakega od 6 otokov
        gatspc "set $master 1 $slave1 $args 0.05"
        gatspc "set $master 2 $slave1 $args 0.05"
        gatspc "set $master 3 $slave1 $args 0.05"
        gatspc "set $master 4 $master $args 0.05"
        gatspc "set $master 5 $master $args 0.05"
        gatspc "set $master 6 $master $args 0.05"
        # začnemo izvajanje vseh otokov
        gatspc "start $server 0"

        # čakamo dokler se algoritem izvaja
        while [ $(gatspc node $master |
                  grep "running" | wc -l) -gt 0 ];
        do
                sleep 1
        done
        # izpišemo zgodovino rešitev v datoteko
        gatspc "history $master" >> $problem.log
done
```

5.1.1 Kvaliteta rešitev

Algoritem smo testirali na problemih iz zbirke TSPLIB [11]. Uporabili smo probleme, ki imajo manj kot 500 mest. Iz tabele 5.3 je razvidno, da uporaba modela z otoki praviloma izboljša rešitev problema. Konfiguraciji 5 in 6 predstavljata reševanje problemov brez selitev in ustrezata več zaporednim izvajanjem algoritma. Konfiguracije 1 do 4 predstavljajo reševanje problemov

s selitvami. Za vsako konfiguracijo smo izpisali najboljšo rešitev in izračunali razdaljo d_i do optimalne rešitve po formuli (5.1), kjer je f_i kvaliteta najboljše rešitve konfiguracije i, f_{opt} pa vrednost optimalne rešitve. Optimalne rešitve za probleme v tabeli 5.3 so navedene v zbirki problemov [11].

$$d_i = \frac{f_i - f_{opt}}{f_{opt}} \tag{5.1}$$

Konfiguracija 5 daje boljše rezultate od konfiguracije 6, ker lahko z več poskusi preiščemo večji prostor. V konfiguracijah z več enotami izberemo najboljšo rešitev izmed vseh enot. Tabela 5.2 prikazuje povprečno razdaljo d_{avg} ter mediano razdalj d_{med} do optimalne rešitve.

Razdalja do optimalne rešitve		
konfiguracija	d_{avg}	d_{med}
1	0.80	0.25
2	1.00	0.45
3	1.12	0.49
4	1.39	0.67
5	1.61	0.93
6	1.80	1.12

Tabela 5.2: Razdalja rešitve do optimuma.

Iz tabel 5.2 in 5.3 razberemo, da uporaba modela z otoki izboljša rezultate. Pri konfiguraciji s 16 otoki se polovica rešitev približa na 25% optimalne rešitve. Rešitve dobljene s tem modelom primerjamo z rešitvami z enakim številom enot (konfiguraciji 3 in 5). Konfiguracija modela z otoki najde rešitve, ki so 49 procentov bližje optimalni kot pa rešitve genetskega algoritma brez selitev z istimi parametri. Za težje primere je razlika večja, ker je algoritem še v zgodnji fazi in še ne konvergira. Model z otoki v zgodnji fazi daje boljše rezultate zaradi hitrejšega preiskovanja. Ko algoritma konvergirata so razlike manj opazne, algoritem s selitvami pa na koncu pride zaradi kompleksne strukture do boljših rešitev.

Tabela rešitev								
			Konfiguracija					
problem	št. mest	opt.	1	2	3	4	5	6
att48	48	10628	0.02	0.02	0.04	0.05	0.04	0.08
eil51	51	426	0.05	0.03	0.05	0.07	0.07	0.12
berlin52	52	7542	0.09	0.07	0.08	0.04	0.07	0.11
st70	70	675	0.06	0.05	0.07	0.11	0.23	0.35
eil76	76	538	0.05	0.08	0.13	0.11	0.15	0.28
pr76	76	108159	0.12	0.10	0.11	0.10	0.13	0.34
rat99	99	1211	0.20	0.15	0.24	0.27	0.36	0.44
kroA100	100	21282	0.10	0.20	0.11	0.27	0.44	0.55
kroB100	100	22141	0.11	0.10	0.23	0.24	0.42	0.62
kroC100	100	20749	0.12	0.25	0.14	0.36	0.51	0.65
kroD100	100	21294	0.19	0.23	0.25	0.28	0.47	0.59
kroE100	100	22068	0.03	0.17	0.28	0.31	0.49	0.60
rd100	100	7910	0.16	0.12	0.13	0.31	0.42	0.40
eil101	101	629	0.13	0.23	0.17	0.31	0.30	0.48
lin105	105	14379	0.13	0.17	0.20	0.32	0.50	0.72
pr107	107	44303	0.13	0.14	0.22	0.25	0.45	0.48
pr124	124	59030	0.29	0.28	0.47	0.50	0.89	1.09
bier127	127	118282	0.18	0.28	0.22	0.32	0.47	0.62
ch130	130	6110	0.19	0.39	0.33	0.38	0.62	0.79
pr136	136	96772	0.22	0.25	0.29	0.50	0.72	0.90
pr144	144	58537	0.12	0.71	0.72	0.90	1.18	1.66
ch150	150	6528	0.16	0.49	0.50	0.90	0.93	1.14
kroA150	150	26524	0.31	0.45	0.49	0.68	1.08	1.16
kroB150	150	26130	0.36	0.36	0.47	0.73	1.01	1.05
pr152	152	73682	0.25	0.57	0.55	0.68	0.92	1.49
u159	159	42080	0.47	0.55	0.72	1.11	1.31	1.34
rat195	195	2323	0.62	0.83	0.96	1.35	1.52	1.51
d198	198	15780	0.36	0.56	0.57	0.73	1.03	1.12
kroA200	200	29368	0.70	0.85	0.95	1.58	1.71	1.96
kroB200	200	29437	0.78	1.11	1.15	1.46	1.74	1.63
ts225	225	126643	1.20	1.59	1.88	2.16	2.49	2.81
tsp225	225	3916	0.71	1.04	1.22	1.71	1.75	1.98
pr226	226	80369	1.06	1.30	1.84	1.99	3.03	3.06
gil262	262	2378	1.32	1.51	1.75	2.08	2.30	2.78
pr264	264	49135	1.09	1.20	1.57	2.34	2.45	2.79
a280	280	2579	1.85	2.06	2.05	2.72	2.75	3.15
pr299	299	48191	1.82	2.45	2.54	2.71	3.22	3.43
lin318	318	42029	1.96	2.19	2.44	3.39	3.36	3.60
linhp318	318	41345	1.88	2.32	2.65	3.24	3.45	3.73
rd400	400	15281	2.60	2.99	3.39	3.76	3.96	4.51
fl417	417	11861	3.41	4.83	5.54	7.47	8.00	8.64
pr439	439	107217	3.21	4.04	4.25	4.64	5.03	5.34
pcb442	442	50778	3.38	3.56	3.91	4.65	4.67	5.00
d493	493	35002	2.98	2.89	3.31	3.99	4.01	4.31

Tabela 5.3: Najboljše rešitve konfiguracij urejene po številu mest.

5.1.2 Hitrost algoritma

Algoritem smo izvajali na dveh štirijedrnih računalnikih, zato za štiri otoke ne potrebujemo štirikrat več časa, ker jedra lahko delujejo vzporedno. Za več kot 3 otoke smo razdelili obremenitev med dvema računalnikoma na polovico. Tabela 5.4 prikazuje potreben čas v odvisnosti od števila otokov ter upočasnitev glede na enoto algoritma brez vzporednega računanja. Dokler povečujemo število otokov do števila razpoložljivih jeder to malo vpliva na realno hitrost algoritma. Ko to število prekoračimo, se trajanje algoritma začne povečevati, vendar to ne vpliva na kvaliteto rešitev.

Izmerili smo, da za 6 enot brez selitev povprečno potrebujemo 1.6 krat toliko časa kot za eno enoto. Za 6 enot s selitvami potrebujemo 1.8 krat toliko časa kot za eno enoto, kar je 13% več kot pa brez selitev. Razlog je v tem, da pri selitvah izvedemo dodatno selekcijo ter dva prenosa podatkov. Pri istem številu generacij je potrebno več časa za računanje problemov z več mesti, ker za računanje ocene kvalitete osebkov porabimo več časa.

Čas trajanja algoritma		
število otokov	potreben čas [s]	upočasnitev
1	33.83	1.00
2	35.67	1.05
3	38.44	1.13
4	43.53	1.28
6	60.77	1.79
8	77.76	2.29
16	152.76	4.41

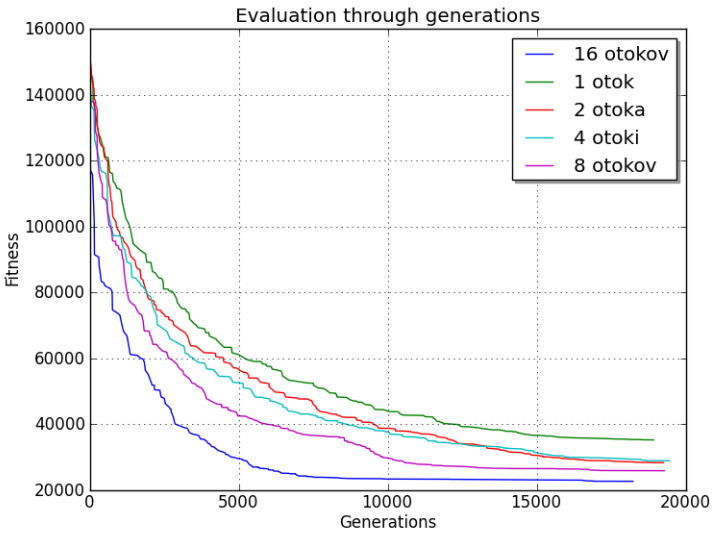
Tabela 5.4: Čas računanja glede na število otokov.

5.1.3 Število otokov

Število otokov je število vzporedno delujočih enot, med katerimi poteka selitev. Na sliki 5.1 so za vsako skupino otokov prikazane najboljše rešitve

v generaciji. Povečevanje števila otokov pozitivno vpliva na kvaliteto rešitev ne glede na generacijo. Vpliv različnega števa otokov je viden na sliki 5.1, rezultati na zbirki problemov pa v tabeli 5.3.

Upoštevati moramo, da z n vzporednimi algoritmi porabimo n-krat več računske moči v istem časovnem intervalu. Algoritmu, ki ne uporablja selitev, smo zato dodelili več možnih iteracij in ugotovili, da to le malo izboljša rešitev. Vzrok je, da se raznolikost populacije zmanjša in s križanjem redkeje dobivamo nove osebke. Napredek v poznejših fazah algoritma brez selitev je zato odvisen predvsem od mutacij. Model z otoki izkoristi selitveno populacijo, iz katere tudi v poznejših fazah jemlje nove osebke.



Slika 5.1: Kvaliteta rešitve z selitvami in brez.

Na sliki 5.1 je prikazana primerjava algoritma s paralelnimi enotami modela z otoki ter genetskega algoritma brez selitev. Zgolj povečanje populacije ne izboljšuje kvalitete rešitev, kar smo pojasnili v poglavju 5.2.1. Pri modelu z otoki na kvaliteto rešitev vpliva izoliranost in sočasno delovanje. Osebki si dobre rešitve med seboj izmenjajo in hitreje napredujejo, zato manj verjetno pristanejo v lokalnem ekstremu, ki je posledica neraznolike populacije.

5.2 Analiza parametrov

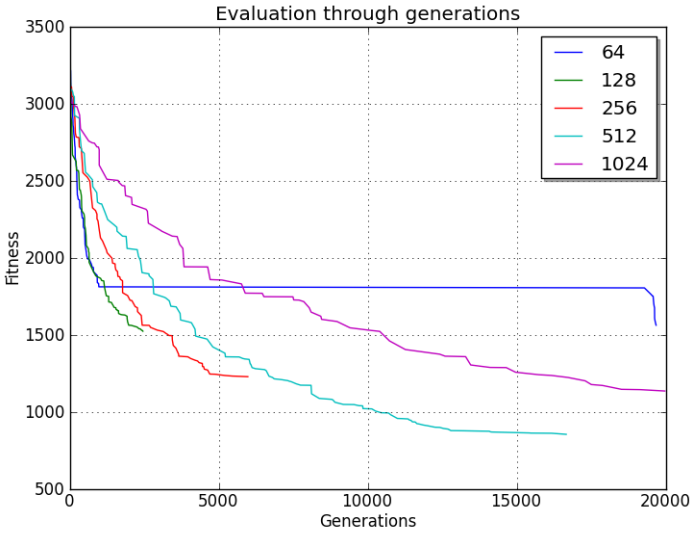
Vrednost parametrov, pri katerih je kvaliteta rešitve problema najvišja, je odvisna od problema in njegove zahtevnosti. Prilagajanje parametrov vsakemu problemu posebej, zato izboljša kvaliteto rešitve. Ugotovimo, da parametri na probleme vplivajo podobno, le točka optimuma se spreminja. Ko povečujemo vrednost parametra, se kvaliteta rešitve veča le do te točke, potem pa začne padati. Parametre lahko nastavljamo vsakič sproti, lahko pa testiramo parametre na podobnih problemih in nato uporabimo najbolj uspešne. V naslednjih podpoglavjih prikažemo vpliv posameznih parametrov na delovanje genetskih algoritmov.

5.2.1 Velikost populacije

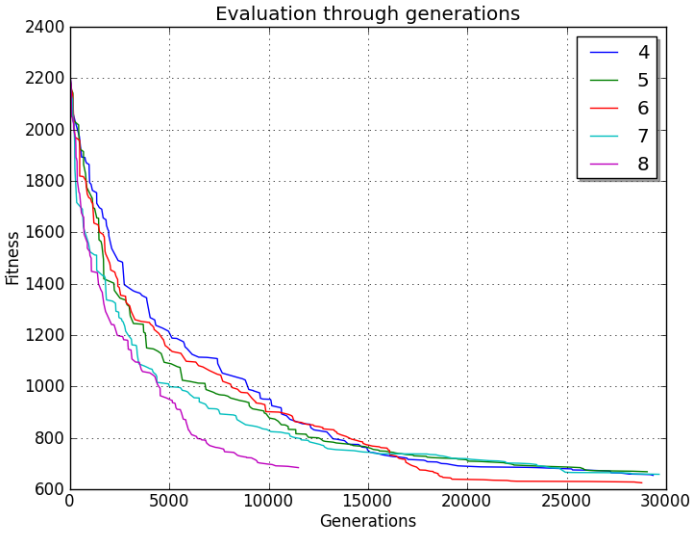
Velikost populacije lahko določimo na več načinov. Iščemo optimalno razmerje med težavnostjo in kvalitete rešitve tako, da je izkoriščenost računske moči največja. Druga možnost je, da populacijo vseskozi povečujemo in tako ne izgubljamo rešitev. Za težje probleme obstaja več dobrih rešitev, ki so med seboj neodvisne, zato je dobro, da za težje probleme povečamo velikost populacije. Vpliv velikosti populacije je viden iz slike 5.2. Večja populacija se zaradi počasnega napredovanja hitreje napolni s podobnimi osebki. Pri manjših populacijah algoritem hitro napreduje k boljšim rešitvam. Ker ima manjša populacija malo različnih osebkov, se hitreje ustavi v lokalnem ekstremu.

5.2.2 Velikost skupine

Velikost skupine je parameter, ki ga uporabimo pri enoturnirski izbiri osebkov, njegov vpliv pa je prikazan na sliki 5.3. Skupina z velikostjo celotne populacije je skrajni primer, kjer na podlagi enoturnirske izbire vedno križamo najboljša dva, in z njunima potomcema nadomestimo najslabša dva. Ta način preveč favorizira dobre osebke in algoritem se ustavi v lokalnem minimumu. Manjša kot je skupina, bolj je izbira podobna naključni in manj je



Slika 5.2: Kvaliteta rešitve v odvisnosti od velikosti populacije.



Slika 5.3: Kvaliteta rešitve v odvisnosti od velikosti skupine.

odvisna od kvalitete osebkov. Pri manjših skupinah populacija ostane zelo raznolika, zato algoritem še dolgo napreduje. Tak algoritem je počasnejši, vendar najde kvalitetnejše rešitve. Zaradi tega se ponavadi uporabljajo manjše skupine velikosti od 4 do 8.

5.2.3 Verjetnost mutacije

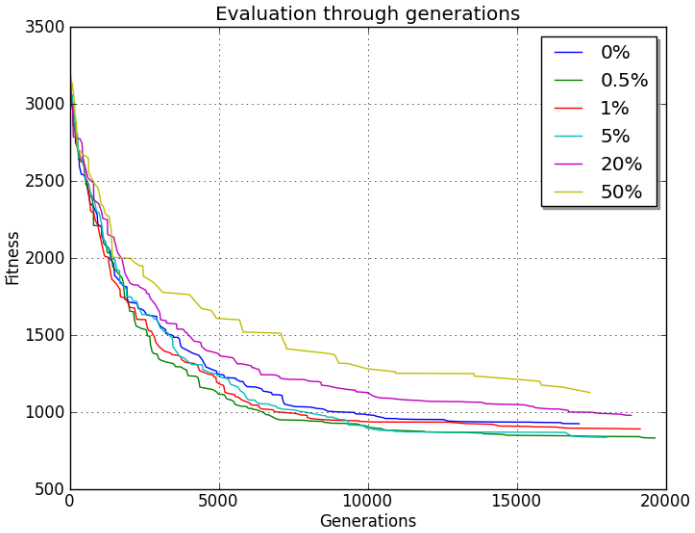
Mutacijo uporabljamo za preprečevanje prehitre konvergence. Iz grafa na sliki 5.4 je razvidno, da nižja verjetnost mutacije ne vpliva na zgodnjo fazo algoritma. Šele v pozni fazi vidimo, da z nizkimi verjetnostmi mutacije ne moremo zapustiti lokalnega minimuma, visoka stopnja mutacije pa z uvajanjem velikega števila slabih rešitev v populacijo negativno vpliva na kvaliteto rešitev. Temu se izognemo tako, da slabših mutacij ne vključujemo v populacijo. To dejansko zniža verjetnost mutacije in skrije vpliv velike verjetnosti mutacij na genetski algoritem, obenem pa poveča verjetnost novih rešitev v poznejših fazah.

5.2.4 Verjetnost selitve

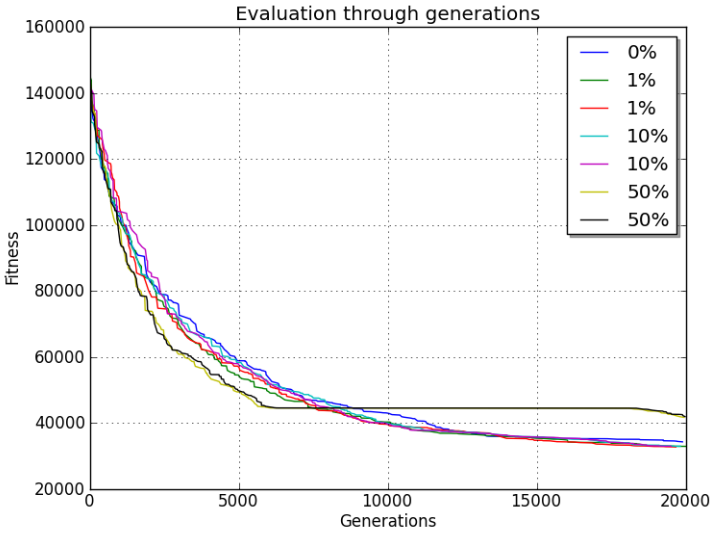
Selitve so koristne ker v populacijo dobivamo nove osebke. Če so osebki, ki se priselijo v populacijo, boljši od osebkov v populaciji, potem vsako križanje z domačini verjetno ustvari boljše osebke od že obstoječih. Posledica tega je manjša raznolikost populacije in hitrejša konvergenca. Z večanjem verjetnosti selitev med otoki se hitrost iskanja rešitev veča, vendar se zaradi večje povezanosti zmanjša raznolikost in algoritem hitreje zaide v lokalni minimum. Unikatne rešitve zahtevajo določeno stopnjo izoliranosti, večja verjetnost selitev pomeni bolj povezane rešitve. To pa lahko hitreje pripelje do konvergence, kar je razvidno iz grafa na sliki 5.5.

5.2.5 Število generacij

Maksimalno število generacij oziroma število iteracij genetskega algoritma je eden izmed ustavitvenih pogojev. Z njim določimo največje število iteracij, ki



Slika 5.4: Kvaliteta rešitve v odvisnosti od verjetnosti mutacije.



Slika 5.5: Kvaliteta rešitve v odvisnosti od verjetnosti selitev.

se lahko izvedejo. V eni iteraciji se razvrstijo koraki izbire, križanja, mutacije in selitve. Večanje števila generacij pomeni, da se bo algoritem dlje časa izvajal, zato je bolj verjetno, da bomo našli boljšo rešitev. Število generacij lahko določimo vnaprej, lahko pa se algoritem ustavi, če se sproži nek drug ustavitveni pogoj, npr. če več generacij ni bilo spremembe.

Poglavje 6

Zaključek

V diplomskem delu smo predstavili težavnostne razrede P in NP ter problem trgovskega potnika. Razložili smo delovanje genetskih algoritmov in opisali nekaj možnosti paralelizacije. V praktičnem delu diplomskega dela smo implementirali porazdeljen evolucijski algoritem, ki rešuje problem trgovskega potnika, ter dodali grafični vmesnik za predstavitev rešitev. Program smo testirali na zbirki problemov TSP ter analizirali vpliv parametrov na kakovost rešitev.

Ideja porazdeljenega algoritma se je izkazala za uspešno, saj z razporejanjem računskih opravil na več računalnikov pohitrimo iskanje rešitev ter izboljšamo njeno kvaliteto, hkrati pa omogočamo realizacijo razširjenih algoritmov, ki izkoriščajo vzporednost v svojem računskem modelu.

Algoritem bi lahko izboljšali tako, da bi ga kombinirali z lokalnim preiskovanjem v poznejših fazah preiskovanja. Pri trenutni implementaciji se na vseh enotah izvaja isti genetski algoritem. Na posameznih enotah lahko izvajali druge evolucijske algoritme, na primer roj delcev ali kolonija mravelj. Ti algoritmi bi med seboj tekmovali, ali pa bi si pomagali z izmenjavo rešitev.

Literatura

[1] T. Bäck, D. B. Fogel, Z. Michalewicz. *Evolutionary computation 1: Basic algorithms and operators*. Institute of Physics Publishing, 2000.

[2] T. Bäck, D. B. Fogel, Z. Michalewicz. *Evolutionary computation 2: Advanced algorithms and operators*. Institute of Physics Publishing, 2000.

[3] T. Bäck, D. B. Fogel, Z. Michalewicz. *Handbook of evolutionary computation*. Oxford University Press, 1997.

[4] D. Ashlock. *Evolutionary computation for modeling and optimization*. Springer-Verlag, 2006.

[5] I. Kononenko, M. R. Šikonja. *Inteligentni sistemi*. Založba FE in FRI, 2010.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms: third edition*. The MIT Press, 2009.

[7] O. Goldreich, *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[8] S. Arora, B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[9] M. Obitko (1998) Introduction to Genetic Algorithms. Dostopno na: http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php

[10] T. Weise (2009), Global Optimization Algorithms: Theory and Application. Dostopno na: http://www.it-weise.de/projects/book.pdf

[11] TSPLIB. Dostopno na:
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[12] Data for the Traveling Salesperson Problem. Dostopno na:
http://people.sc.fsu.edu/ jburkardt/datasets/tsp/tsp.html

[13] Beej's Guide to Network Programming. Dostopno na:
http://beej.us/guide/bgnet/

[14] Client-server model. Dostopno na:
http://en.wikipedia.org/wiki/Server-client_architecture

[15] WxPython wiki. Dostopno na:
http://wiki.wxpython.org/