

Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave  
Fakultet elektrotehnike i računarstva  
Sveučilište u Zagrebu

# **Prirodom inspirirani optimizacijski algoritmi**

**mr.sc. Marko Čupić**

Zagreb, 2009.-2010.



Autor:

mr.sc. Marko Čupić, Fakultet elektrotehnike i računarstva, Zagreb

Zaštićeno licencijom Creative Commons Imenovanje–Nekomercijalno–Bez prerada 3.0 Hrvatska.

<http://creativecommons.org/licenses/by-nc-nd/3.0/hr/>

Verzija: 1.0.7 (2010-05-26)

Skripta je namijenjena uporabi na kolegiju *Umjetna inteligencija*.



## Sadržaj

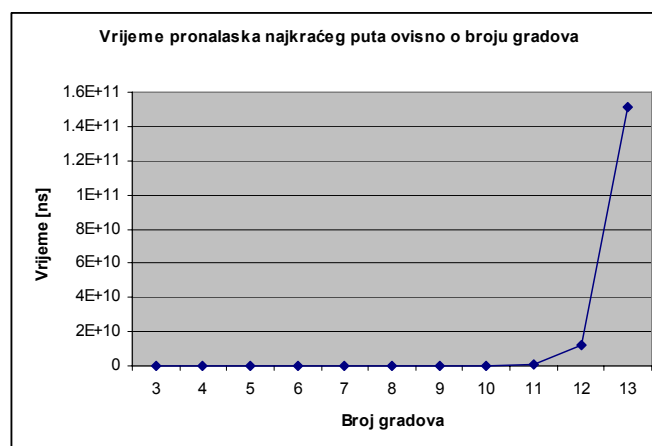
<b>1. Uvod</b>	<b>1</b>
<b>2. Genetski algoritam</b>	<b>7</b>
<b>2.1. Vrste genetskog algoritma</b>	<b>8</b>
2.1.1. Steady-state genetski algoritam	8
2.1.2. Generacijski genetski algoritam	9
<b>2.2. Prikaz rješenja</b>	<b>10</b>
<b>2.3. Operator križanja</b>	<b>12</b>
2.3.1. Križanje s jednom točkom prekida	13
2.3.2. Križanje s t-točaka prekida	13
2.3.3. Uniformno križanje	13
<b>2.4. Operator mutacije</b>	<b>14</b>
<b>2.5. Operator selekcije</b>	<b>14</b>
2.5.1. Proporcionalna selekcija	15
2.5.2. k-turnirska selekcija	17
<b>2.6. Detaljniji primjer</b>	<b>17</b>
<b>2.7. Drugi prikazi kromosoma</b>	<b>21</b>
<b>2.8. Genetski algoritam za izradu rasporeda međuispita</b>	<b>22</b>
2.8.1. Opis problema	22
2.8.2. Prikaz kromosoma	23
2.8.3. Genetski operatori	24
<b>2.9. Genetsko programiranje</b>	<b>24</b>
<b>3. Algoritam kolonije mrava</b>	<b>29</b>
<b>3.1. Pojednostavljeni matematički model</b>	<b>30</b>
<b>3.2. Algoritam Ant System</b>	<b>33</b>
<b>4. Algoritam roja čestica</b>	<b>39</b>
<b>4.1. Opis algoritma</b>	<b>39</b>
<b>4.2. Utjecaj parametara i modifikacije algoritma</b>	<b>42</b>
4.2.1. Dodavanje faktora inercije	42
4.2.2. Stabilnost algoritma	42
4.2.3. Lokalno susjedstvo	43
<b>4.3. Primjer rada algoritma</b>	<b>44</b>
<b>5. Umjetni imunološki sustav</b>	<b>47</b>
<b>5.1. Jednostavni imunološki algoritam</b>	<b>48</b>
<b>5.2. Algoritam CLONALG</b>	<b>50</b>
<b>5.3. Pregled korištenih operatora</b>	<b>53</b>
5.3.1. Operator kloniranja	53
5.3.2. Operatori mutacije	53
5.3.3. Operator starenja	54
<b>5.4. Druga područja</b>	<b>54</b>
<b>6. Programski zadatci</b>	<b>59</b>
<b>6.1. Poopćena Rastriginova funkcija</b>	<b>59</b>
6.1.1. Prikladni algoritmi za rješavanje problema	59

<b>6.2.</b>	<b>Normalizirana Schwefelova funkcija</b>	<b>59</b>
6.2.1.	Prikladni algoritmi za rješavanje problema	60
<b>6.3.</b>	<b>Problem popunjavanja kutija</b>	<b>60</b>
6.3.1.	Naputci	61
<b>6.4.</b>	<b>Izrada rasporeda timova studenata</b>	<b>62</b>
6.4.1.	Ograničenja	62
6.4.2.	Naputci	62
6.4.3.	Prikladni algoritmi za rješavanje problema	63
<b>6.5.</b>	<b>Izrada prezentacijskih grupa za seminare (1)</b>	<b>63</b>
6.5.1.	Zadatak	63
6.5.2.	Naputci	64
6.5.3.	Prikladni algoritmi za rješavanje problema	65
<b>6.6.</b>	<b>Izrada prezentacijskih grupa za seminare (2)</b>	<b>65</b>
6.6.1.	Zadatak	65
6.6.2.	Naputci	66
6.6.3.	Prikladni algoritmi za rješavanje problema	67
<b>7.</b>	<b>Implementacija evolucijskih algoritama u programskom jeziku Java</b>	<b>69</b>
<b>7.1.</b>	<b>Genetski algoritam</b>	<b>69</b>
7.1.1.	Razred GeneticAlgorithm	69
7.1.2.	Razred Kromosom	72
7.1.3.	Razred KromosomDekoder	73
<b>7.2.</b>	<b>Mravlji algoritmi</b>	<b>75</b>
7.2.1.	Razred SimpleACO	75
7.2.2.	Razred AntSystem	78
<b>7.3.</b>	<b>Algoritam roja čestica</b>	<b>82</b>
7.3.1.	Razred ParticleSwarmOptimization	82
7.3.2.	Razred Particle	85
7.3.3.	Sučelje Neighborhood	85
7.3.4.	Razred GlobalNeighborhood	86
7.3.5.	Razred LocalNeighborhood	87
<b>7.4.</b>	<b>Algoritmi umjetnog imunološkog sustava</b>	<b>89</b>
7.4.1.	Razred SimpleIA	89
7.4.2.	Razred ClonAlg	91
<b>7.5.</b>	<b>Pomoćni razredi</b>	<b>95</b>
7.5.1.	Sučelje IFunkcija	95
7.5.2.	Razred City	95
7.5.3.	Razred TSPSolution	96
7.5.4.	Razred TSPSolutionPool	96
7.5.5.	Razred TSPUtil	97
7.5.6.	Razred ArraysUtil	99
7.5.7.	Razred PrepareTSP	100

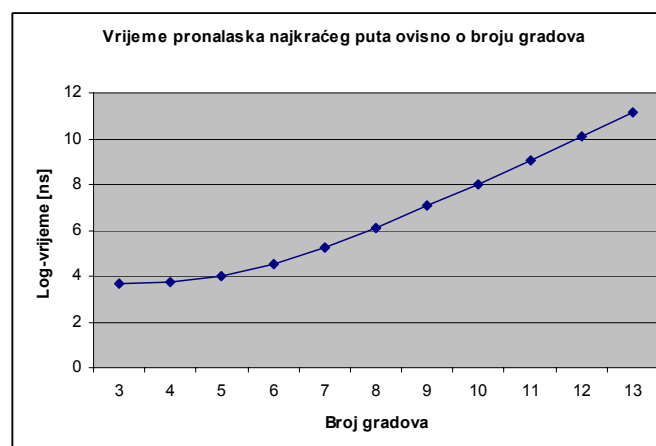
## 1. Uvod

Razvojem računarstva i povećanjem procesne moći računala, ljudi su počeli rješavati izuzetno kompleksne probleme koji su do tada bili nerješivi – i to naprosto tehnikom grube sile (engl. *brute-force*, koristi se još i termin *iscrpnom pretragom*), tj. pretraživanjem cjelokupnog prostora rješenja. S obzirom da su danas računala ekstremno brza, nije rijetka situacija da se njihovom uporabom u jednoj sekundi mogu istražiti milijuni ili čak milijarde potencijalnih rješenja, i time vrlo brzo pronaći ono najbolje. Ovo, dakako, vrijedi samo ako smo jako sretni, pa rješavamo problem koji se grubom silom daje riješiti. Nažalost, postoji čitav niz problema koji ne spadaju u ovu kategoriju, pa spomenimo samo neke od njih.

*Problem trgovačkog putnika* definiran je ovako: potrebno je pronaći redoslijed kojim trgovački putnik mora obići sve gradove, a da pri tome niti jedan ne posjeti više puta, te da ukupno prevali najmanje kilometara. Na kraju, trgovački se putnik mora vratiti u onaj grad iz kojeg je krenuo. Danas je poznato da ovaj naoko trivijalan problem pripada razredu NP-teških problema – problema za koje još uvijek nemamo efikasnih algoritama kojima bismo ih mogli riješiti. Kako bismo ilustrirali složenost ovog problema, napisan je jednostavan program u Javi koji ga rješava tehnikom grube sile, i potom je napravljeno mjerenje za probleme od 3 do 13 gradova. Rezultate prikazuje slika 1-1 i 1-2.



Slika 1-1 Vrijeme pronalaska najkraćeg puta kod problema trgovačkog putnika



Slika 1-2 Vrijeme pronalaska najkraćeg puta kod problema trgovačkog putnika (logaritamska skala)

Eksperiment je napravljen na računalu s AMD Turion 64 procesorom na 1.8 GHz i s 1 GB radne memorije. Vrijeme potrebno za rješavanje problema s 12 gradova iznosilo je približno 12.3 sekunde, dok je za 13 gradova bilo potrebno poprilično 2,5 minute. Očekivano vrijeme rješavanja problema s 14 gradova je oko pola sata, za 15 gradova oko 7,6 sati, dok bi za 16 gradova trebalo oko 4,7 dana.

Detaljnijom analizom ovog problema te samih podataka eksperimenta, lako je utvrditi da je složenost problema faktorijelna, pa je jasno da je problem koji se sastoji od primjerice 150 gradova primjenom tehnike grube sile praktički nerješiv. U teoriji grafova, problem trgovačkog putnika odgovara pronalasku Hamiltonovog ciklusa u grafu.

Probleme sličnog tipa u svakodnevnom životu neprestano susrećemo. Evo još nekoliko problema vezanih uz akademski život.

*Raspoređivanje neraspoređenih studenata u grupe za predavanja* je problem koji se javlja naknadnim upisom studenata na predmete, nakon što je napravljen raspored studenata po grupama. Veličina grupe za predavanje ograničena je dodijeljenom prostorijom u kojima se izvode predavanja. Neraspoređene studente potrebno je tako razmjestiti da grupe ne narastu iznad maksimalnog broja studenata (određenog prostorijom), te da se istovremeno osigura da student nema preklapanja s drugim dodijeljenim obavezama. Primjerice: neka imamo 50 studenata koji su ostali neraspoređeni na 5 kolegija, i neka u prosjeku svaki kolegij ima 4 grupa u kojima se održavaju predavanja. Potrebno je za prvog studenta i njegov prvi kolegij provjeriti 4 grupa, pa za njegov drugi kolegij 4 grupa, itd. Za sve studente to ukupno daje:

$$(4 \cdot 4 \cdot 4 \cdot 4 \cdot 4) \cdot (4 \cdot 4 \cdot 4 \cdot 4 \cdot 4) \cdot \dots \cdot (4 \cdot 4 \cdot 4 \cdot 4 \cdot 4) = 4^5 \cdot 4^5 \cdot \dots \cdot 4^5 = 4^{50} \approx 5 \cdot 10^{34}$$

Kada bismo imali računalo koje bi jednu kombinaciju moglo provjeriti u jednoj nanosekundi, za provjeru svih kombinacija trebali bismo  $1,59 \cdot 10^{18}$  godina! Kako bismo bolje shvatili koliko je ovo vremena, spomenimo samo da je svemir star oko  $1,37 \cdot 10^{10}$  godina.

*Izrada rasporeda međuispita* je problem u kojem je unaprijed definiran skup raspoloživih termina u kojima se mogu održati ispiti, kapaciteti termina, skup kolegija koji imaju ispite, te popis studenata po kolegijima. Jednostavnija varijanta problema zahtjeva pronalazak takvog rasporeda kolegija po terminima uz koji će svi kolegiji održati svoje ispite, i u kojem ne postoji student koji u istom terminu piše više od jednog ispita. Teža varijanta problema dodatno traži da svaki student između dva ispita ima što je moguće više slobodnog vremena. Uzmimo kao ilustraciju problem izrade rasporeda međuispita na FER-u u ljetnom semestru akademske godine 2008/2009. Broj termina je 40 a broj kolegija 130. Za prvi kolegij možemo uzeti jedan od 40 termina, za drugi kolegij jedan od 40 termina, ... Ukupno imamo  $40^{130} \approx 10^{208}$  kombinacija. Opisani problem je  $10^{174}$  puta složeniji od prethodno opisanog problema raspoređivanja neraspoređenih studenata.

*Izrada rasporeda laboratorijskih vježbi* je problem u kojem u kojem svaki kolegij definira broj potrebnih laboratorijskih vježbi, trajanje vježbi, prihvatljive dane za održavanje vježbi, prihvatljive prostorije, i još niz drugih ograničenja. Dodatno, za svakog studenta se zna koje je kolegije upisao, te kada je zauzet predavanjima i drugim obavezama. Zadatak je pronaći takav raspored koji će osigurati da svi studenti odrade sve tražene vježbe, i da pri tome nemaju konflikte s drugim vježbama ili s



vanjskim zauzećima. Kao ilustraciju složenosti uzmimo sljedeći pojednostavljeni primjer: postoji 200 događaja (događaj definiramo kao održavanje jedne vježbe jednog predmeta podskupu studenata tog predmeta), te imamo na raspolaganju 30 termina u koje događaj možemo smjestiti. Kako za svaki događaj možemo uzeti jedan od 30 termina, ukupan broj kombinacija koje treba ispitati je  $30^{200} \approx 10^{295}$ .

Na sreću, optimalno rješenje često nam nije nužno; obično smo zadovoljni i s rješenjem koje je *dovoljno* dobro. Algoritme koji pronalaze rješenja koja su zadovoljavajuće dobra, ali ne nude nikakve garancije da će uspjeti pronaći optimalno rješenje, te koji imaju relativno nisku računsku složenost (tipično govorimo o polinomijalnoj složenosti) nazivaju se približni algoritmi, heurističke metode ili jednostavno *heuristike* [1]. Dijelimo ih na konstrukcijske te one koji koriste lokalnu pretragu.

*Konstrukcijski algoritmi* rješenje problema grade dio po dio (bez povratka unatrag) sve dok ne izgrade kompletno rješenje. Tipičan primjer je algoritam najbližeg susjeda (engl. *nearest-neighbor procedure*). Na problemu trgovačkog putnika, ovaj algoritam započinje tako da nasumice odabere početni grad, i potom u turu uvijek odabire sljedeći najbliži grad.

*Algoritmi lokalne pretrage* rješavanje problema započinju od nekog početnog rješenja, koje potom pokušavaju inkrementalno poboljšati. Ideja je da se definira skup jednostavnih izmjena koje je moguće obaviti nad trenutnim rješenjem, čime se dobivaju susjedna rješenja. U najjednostavnijoj verziji, algoritam za trenutno rješenje pretražuje skup svih susjednih rješenja i bira najboljeg susjeda kao novo trenutno rješenje (tada govorimo o *metodi uspona na vrh*, ili engl. *hill-climbing method*). Ovo se ponavlja tako dugo dok kvaliteta rješenja raste.

U današnje doba posebno su nam zanimljive metaheuristike. *Metaheuristika* [2] je skup algoritamskih koncepata koji koristimo za definiranje heurističkih metoda primjenjivih na širok skup problema. Možemo reći da je metaheuristika heuristika opće namjene čiji je zadatak usmjeravanje problemski specifičnih heuristika prema području u prostoru rješenja u kojem se nalaze dobra rješenja [1].

Primjeri metaheuristika su simulirano kaljenje, tabu pretraživanje, evolucijsko računanje i slični.

*Simulirano kaljenje* [3,4,5] motivirano je procesom kaljenja metala. Ideja algoritma jest da se dobra rješenja  $f(s') > f(s)$  automatski prihvaćaju (pretpostavimo da se traži maksimum funkcije), a loša  $f(s') < f(s)$  prihvaćaju s vjerojatnošću:

$$p_{\text{prihvati}}(s, s', T) = e^{-\frac{f(s) - f(s')}{T}}$$

što bi algoritmu trebalo osigurati da se povremeno prihvate i lošija rješenja što će pomoći izbjegavanju lokalnih optimuma. pri tome je vjerojatnost prihvatanja to veća što je razlika između lošeg rješenja  $f(s')$  i do tada pronađenog najboljeg  $f(s)$  manja, te što je temperatura  $T$  veća. Algoritam potom kreće od visokih temperatura čime se osigurava grubo pretraživanje prostora stanja, i potom temperaturu postupno smanjuje čim se pokušava osigurati fina pretraga u okolini najboljeg rješenja, i u konačnici konvergencija.

*Tabu pretraživanje* [6,7,8] je tehnika kod koje se pamti lista zadnjih  $n$  posjećenih rješenja, i ta su rješenja zabranjena (tabu). Prilikom pretraživanja prostora, iz rješenja  $s$  bira se najbolji susjed  $s'$  koji pri tome nije u listi zabranjenih rješenja – čak ako je taj

lošiji od trenutno najboljeg rješenja. Ovom tehnikom algoritam nudi mogućnost izlaska iz lokalnih optimuma i napredak prema globalnom optimumu.

*Evolucijsko računanje* danas se dijeli na tri velika područja: genetske algoritme [9,10,11,12], evolucijske strategije [13,14] te evolucijsko programiranje [15,16]. Zajednička im je ideja da rade s populacijom rješenja nad kojima se primjenjuju evolucijski operatori (selekcija, križanje, mutacija, zamjena) čime populacija iz generacije u generaciju postaje sve bolja i bolja.

Velika skupina algoritama koji se dobro nose s opisanim teškim problemima nastala je proučavanjem procesa u prirodi. Priroda je oduvijek bila inspiracija čovjeku, u svemu što je radio. I to ne bez razloga – prirodni procesi optimiraju život u svakom njegovom segmentu već preko 4 milijarde godina. Proučavanjem ovih procesa i načina na koji živa bića danas rješavaju probleme znanost je došla do niza uspješnih tehnika kojima je moguće napasti prethodno opisane probleme. U nastavku ćemo opisati nekoliko takvih tehnika: genetski algoritam, optimizaciju kolonijom mrava, optimizaciju rojem čestica te optimizaciju umjetnim imunološkim sustavom.

### **Pitanja za ponavljanje**

- Opišite *pretraživanje grubom silom*.
- Definirajte *problem trgovačkog putnika*. Koja je njegova složenost?
- Što su to *heuristike*? Koja im je tipična složenost?
- Definirajte pojam *konstrukcijski algoritmi*.
- Definirajte pojam *algoritmi lokalne pretrage*.
- Što su to *metaheuristike*?
- Što podrazumjevamo pod pojmom *evolucijsko računanje*? Kako ga dijelimo?

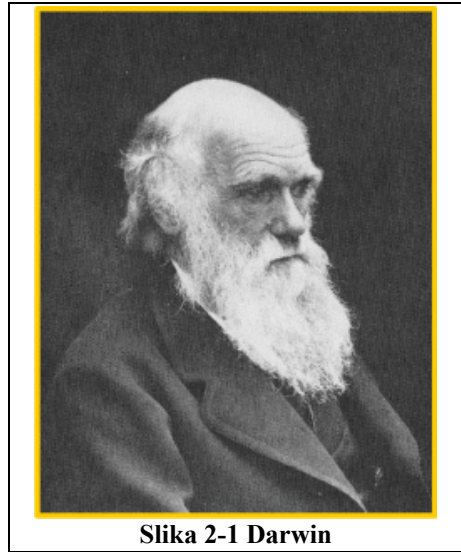
## Literatura

- [1] Dorigo, M., & Stützle, T. Ant Colony Optimization. MIT Press, Cambridge, MA, 2004.
- [2] Glover, F. G.A. Kochenberger. 2003. Handbook of Metaheuristics. Kluwer Academic Publishers, Boston, MA.
- [3] Kirkpatrick, S., C. D. Gelatt Jr., M. P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220: 671-680.
- [4] Eglese, R.W.1990. Simulated annealing: a tool for operation research. *European Journal of Operational Research* 46: 271-281.
- [5] Fleischer, M.A. 1995. Simulated annealing: past present and future. *Proceedings of the 1995 Winter Simulation Conference*, 155-161.
- [6] Glover, F. 1989. Tabu Search – Part I. *ORSA Journal on Computing* 1: 190-206.
- [7] Glover, F. 1990. Tabu Search –Part II. *ORSA Journal on Computing* 2: 4-32.
- [8] Glover, F., M. Laguna. 1997. *Tabu Search*. Kluwer Academic, Boston.
- [9] Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press, 1975.
- [10] Goldberg, D.E. 1989. *Genetic Algorithm in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA.
- [11] Liepins, G. E., M. R. Hilliard. 1989. Genetic algorithm: foundations and applications. *Annals of the Operations Research* 21: 31-58.
- [12] Muhlenbein, H. 1997. Genetic algorithms. In *Local Search in Combinatorial Optimization*. Eds. E. Aarts and J. K. Lenstra, 137-172.
- [13] Rechenberg, I. (1973) *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Information*. Freiburg, Germany, Fromman Verlag.
- [14] Schwefel, H.-P. (1981) *Numerical Optimization of Computer Models*. Chichester, UK, John Wiley & Sons.
- [15] Fogel, L. J., Owens, A. J. and Walsh, M. J.: *Artificial Intelligence through Simulated Evolution*. New York: John Wiley, 1966.



## 2. Genetski algoritam

Evolucijsko računanje u proteklih je pola stoljeća iznjedrilo nekoliko različitih smjerova: L. J. Fogel, A. J. Owens i M. J. Walsh stvorili su 1966. [1] *evolucijsko programiranje*, I. Rechenberg 1973. [2] i H.-P. Schwefel 1975. [3] *evolucijske strategije*, a H. J. Holland 1975. [4] *genetski algoritam*. Paralelno tome, tekao je i razvoj *genetskog programiranja* koji je 1992. popularizirao J. R. Koza [5]. Inspiracija za razvoj došla je proučavanjem Darwinove teorije (slika 2-1) o postanku vrsta [6].



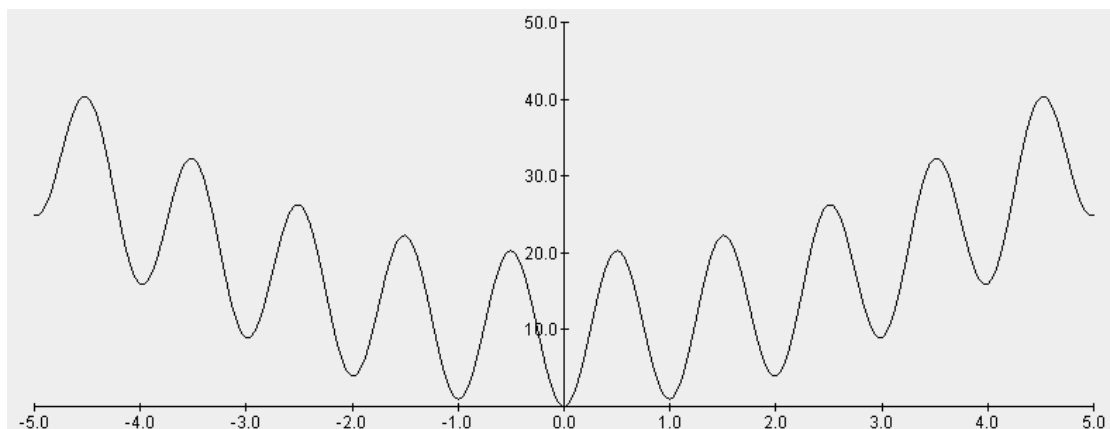
Slika 2-1 Darwin

Darwin teoriju razvoja vrsta temelji na 5 postavki: (i) plodnost vrsta – potomaka uvijek ima više no što je potrebno, (ii) veličina populacije je približno stalna, (iii) količina hrane je ograničena, (iv) kod vrsta koje se seksualno razmnožavaju, nema identičnih jedinki već postoje varijacije, te (v) najveći dio varijacija prenosi se nasljeđem. Iz navedenoga slijedi da će u svakoj populaciji postojati borba za preživljavanje: ako potomaka ima više no što je potrebno a količina hrane je ograničena, sve jedinke neće preživjeti. Pri tome će one bolje i jače imati veću šansu da prežive i dobiju priliku dalje se razmnažati. Prilikom razmnažanja, djeca su u velikoj mjeri određena genetskim materijalom svojih roditelja, no nisu ista roditeljima – postoji određeno odstupanje.

Ova razmatranja osnova su za razvoj genetskog algoritma. Problemi koje rješavamo genetskim algoritmom tipično su optimizacijski problemi, koje možemo opisati na sljedeći način. Zadana je funkcija  $f(\vec{x})$  gdje je  $\vec{x} = (x_1, x_2, \dots, x_n)$ . Potrebno je pronaći  $\vec{x}^*$  koji maksimizira (ili minimizira) funkciju  $f$ . Pogledajmo ovo na primjeru funkcije jedne varijable (slika 2-2) koja je definirana izrazom:

$$f(x) = 10 + x^2 - 10 \cdot \cos(2 \cdot \pi \cdot x).$$

Iz slike je jasno vidljivo da funkcija ima jedan globalni minimum za  $x=0$ ,  $f(x)=0$ .



Slika 2-2 Funkcija jedne varijable čiji tražimo minimum

Kako sam genetski algoritam radi? Postoji puno različitih izvedbi, no generalna ideja je sljedeća. Postoji populacija jedinki. Svaka jedinka je jedno moguće rješenje

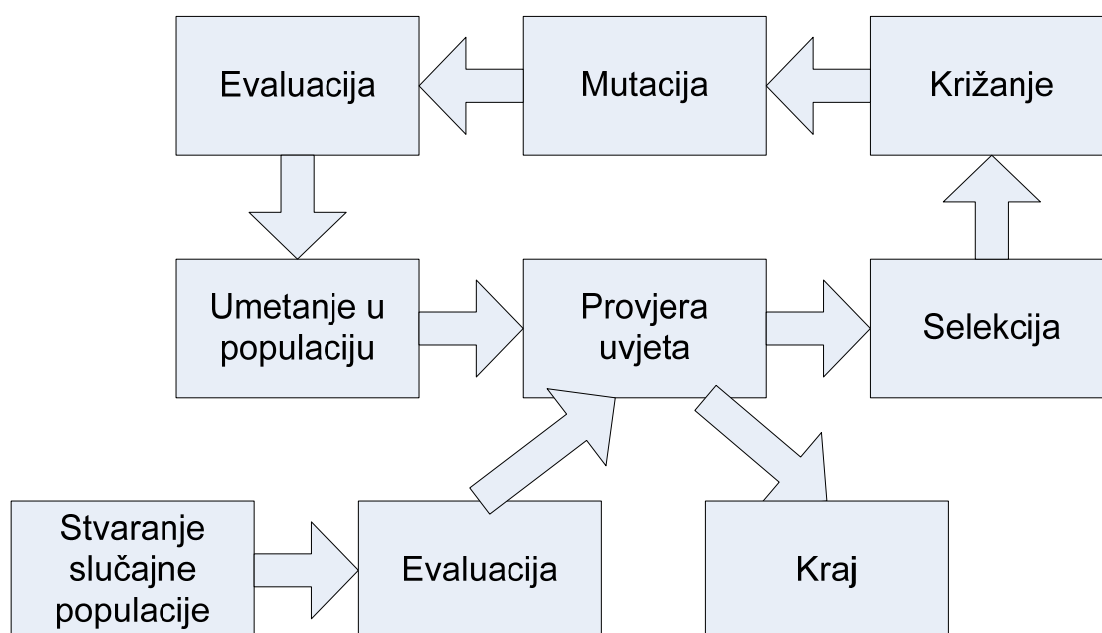
zadanog problema (u našem slučaju, to će biti vrijednost varijable  $x$ ). Jedinke još nazivamo i kromosomima. Svakoj jedinki možemo odrediti njezinu dobrotu (engl. *fitness*). Dobrotu jedinke odredit ćemo tako da izračunamo vrijednost funkcije u toj točki; što je vrijednost funkcije veća, to je jedinka lošija (prisjetimo se, rješavamo problem minimizacije funkcije). Operatorom *selekcije* (engl. *selection*) biraju se iz populacije jedinke koje postaju *roditelji*. Roditelji pomoću operatora *križanja* (engl. *crossover*) stvaraju djecu, čime se emulira izmjena genetskog materijala (engl. *recombination*). Nad djecom potom djeluje operator *mutacije* (engl. *mutation*). Konačno, operatorom *zamjene* (engl. *reinsertion*) djeca ulaze u populaciju rješenja, čime se zatvara ciklus rada algoritma.

## 2.1. Vrste genetskog algoritma

Genetski algoritmi dijele se na sekvencijske i paralelne. Paralelni genetski algoritmi omogućavaju nam rad u višedretvenim i raspodijeljenim okruženjima, i ovdje se neće razmatrati. Sekvencijski genetski algoritam možemo podijeliti na dvije tipične izvedbe [7]: steady-state genetski algoritam te generacijski genetski algoritam, koji su opisani u nastavku.

### 2.1.1. Steady-state genetski algoritam

Ova vrsta genetskog algoritma prikazana je na slici 2-3. U svakom koraku (možemo još koristiti i termin generaciji), iz čitave se populacije odabiru dva roditelja nad kojima izvodi križanje, čime nastaje novo dijete. Dijete se potom mutira i ubacuje u populaciju. Kako veličina populacije mora biti stalna, ovo ubacivanje izvodi se tako da dijete zamijeni neku jedinku iz populacije (primjerice, najgoru).



Slika 2-3 Steady-state genetski algoritam

Selekcija svakog od roditelja može se obaviti nekim od operatora selekcije; primjerice, proporcionalnom selekcijom ili pak turnirskom selekcijom (opis selekcija bit će dan u sljedećim poglavljima). Isto tako, odabir jedinke koja će biti u populaciji biti zamijenjena nastalim djetetom također se može obaviti nekim od operatora

```

P = stvori_početnu_populaciju(VEL_POP)
evaluira(P)
ponavljaj_dok_nije_kraj
  odaberi R1 i R2 iz P
  D = križaj(R1, R2)
  mutiraj D
  evaluiraj D
  umetni D u P operatorom zamjene
kraj

```

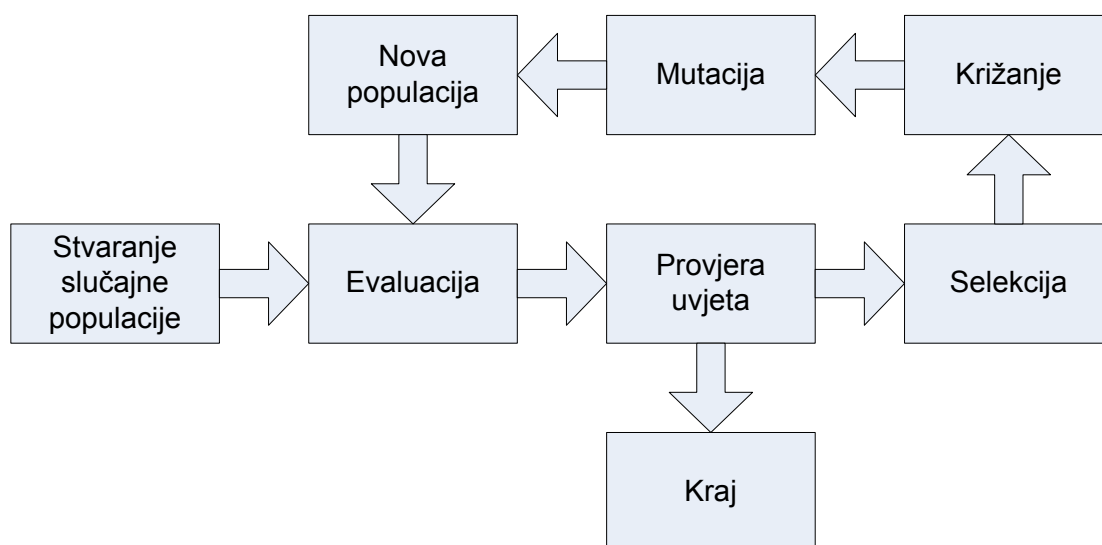
Okvir 2.1 Pseudokod steady-state genetskog algoritma

selekcije, pri čemu se operator tako prilagodi da veću šansu odabira daje lošijim jedinkama. Pseudokod ove vrste genetskog algoritma dan je u okviru 2.1.

Uočimo da kod ove vrste genetskog algoritma nastalo dijete već u sljedećem trenutku može postati roditelj koji će se križati sa svojim pretcima i dalje stvarati potomke.

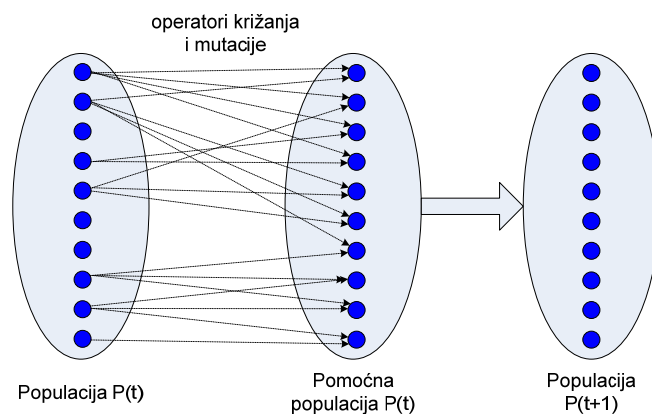
### 2.1.2. Generacijski genetski algoritam

Za razliku od prethodno opisane vrste kod koje ne postoji čista granica između roditelja i djece, generacijski genetski algoritam iz koraka u korak iz populacije roditelja stvara novu populaciju djece koja potom postaju roditelji – stara populacija roditelja time odmah izumire. Shematski prikaz ove vrste genetskog algoritma dan je na slici 2-4.



Slika 2-4 Shematski prikaz rada genetskog algoritma

Generacijski genetski algoritam iz trenutne populacije operatorima selekcije, križanja i mutacije gradi novu pomoćnu populaciju sastavljenu isključivo iz djece (sliku 2-5).



Slika 2-5 Korak generacijskog genetskog algoritma

Onog trenutka kada se izgradi nova populacija čija je veličina jednaka staroj, stara populacija roditelja se odbacuje, a novoizgrađena populacija djece postaje trenutna populacija.

Pseudokod ove vrste genetskog algoritma prikazan je u okviru 2.2.

```

P = stvori_početnu_populaciju(VEL_POP)
evaluiraj(P)
ponavljaj_dok_nije_kraj
  nova_populacija P' = ∅
  ponavljaj_dok_je_veličina(P') < VEL_POP
    odaberi R1 i R2 iz P
    {D1, D2} = krizaj(R1, R2)
    mutiraj D1, mutiraj D2
    dodaj D1 i D2 u P'
  kraj
P = P'
kraj

```

Okvir 2.2 Pseudokod generacijskog genetskog algoritma

Uočimo da se direktnom implementacijom prikazanog pseudokôda dobiva algoritam koji nema *elitizma* (dakle, najbolja se jedinka ne čuva). Naime, sasvim je moguće da sva nastala djeca budu lošija od najboljeg roditelja. Aktiviranjem populacije djece, tada dolazi do gubitka najboljeg pronađenog rješenja. Ovome se može doskočiti tako da se najprije u populaciju djece iskopira najbolji roditelj, a ostatak populacije potom izgradi na uobičajeni način.

Prilikom rješavanja problema genetskim algoritmom, potrebno je definirati način na koji kromosom kodira rješenje, način na koji se obavlja križanje, način na koji se obavlja mutacija te način na koji se odabiru jedinke za razmnožavanje. Krenimo redom.

## 2.2. Prikaz rješenja

Najjednostavniji način prikaza rješenja jest nizom bitova (engl. *bit-string*). Svaki kromosom (tj. jedinka) sastoji se od jednakog i fiksnog broja bitova. Neka za potrebe ovog razmatranja broj bitova  $n$  bude 10. Prostor koji pretražujemo također je ograničen na područje  $[x_{\min}, x_{\max}]$ . Uzmimo opet za potrebe primjera da je  $[x_{\min}, x_{\max}] = [-5, 5]$ . Potrebno je definirati način na koji se niz nula i jedinica iz kromosoma preslikava u rješenje, tj. način na koji se obavlja *dekodiranje*. Najjednostavniji način jest niz nula i jedinica u kromosomu promatrati kao binarno zapisan cijeli broj. Kako imamo kromosom od  $n$  bitova, cijeli brojevi koje on može predstavljati su brojevi od




0 do  $2^n-1$ , što u našem slučaju znači od 0 do 1023. Najmanji broj (0) pri tome predstavlja  $x_{\min}$ , najveći broj (1023) predstavlja  $x_{\max}$ , a bilo koji broj između najmanjeg i najvećeg predstavlja neki broj između  $x_{\min}$  i  $x_{\max}$ . Najčešće se uzima linearno skaliranje, pa ako vrijednost cijelog broja zapisanog u kromosomu označimo s  $k$ , pripadnu vrijednost rješenja dobit ćemo izrazom:

$$x = x_{\min} + \frac{k}{2^n - 1} \cdot (x_{\max} - x_{\min})$$



## Primjer 2.1


Genetski algoritam za prikaz rješenja koristi trobitni binarni kromosom. Prostor koji se pretražuje je ograničen na područje  $[-2, 2]$ . Koja sve rješenja genetski algoritam može istražiti? S kojom preciznošću algoritam obavlja pretraživanje? 

Uporabom 3 bita možemo zapisati 8 cijelih brojeva: od 0 do 7, što prema navedenom izrazu odgovara rješenjima: -2, -1,43, -0,87, -0,29, 0,29, 0,87, 1,43 te 2. Preciznost  $\tilde{x}$  (odnosno kvant) kojim se obavlja pretraživanje je razlika između bilo koja dva susjedna rješenja:

$$\begin{aligned}\tilde{x} &= x_{k+1} - x_k \\ &= x_{\min} + \frac{k+1}{2^n - 1} \cdot (x_{\max} - x_{\min}) - \left( x_{\min} + \frac{k}{2^n - 1} \cdot (x_{\max} - x_{\min}) \right) \\ &= \frac{1}{2^n - 1} \cdot (x_{\max} - x_{\min})\end{aligned}$$

što u našem slučaju iznosi 0,57.

## Primjer 2.2

Koliko bismo bitova trebali koristiti u prethodnom primjeru, da bismo pretraživanje obavili preciznošću  $\tilde{x} = 10^{-2}$ ? 

Preciznost od  $10^{-2}$  znači da želimo da kromosom bude u stanju zapisati redom: -2, -1,99, -1,98, ..., 1,99, 2. Očito da za svako ovo rješenje binarni kromosom treba po jedan cijeli broj, pa trebamo utvrditi koliko zapravo različitih rješenja želimo prikazati i potom utvrditi koliko nam za to minimalno treba bitova. Broj različitih rješenja uz zadanu preciznost određen je izrazom:

$$\frac{x_{\max} - x_{\min}}{\tilde{x}}$$

što u našem slučaju iznosi 400. Želimo pronaći prvi  $n$  za koji vrijedi:

$$2^n \geq \frac{x_{\max} - x_{\min}}{\tilde{x}}$$

Logaritmiranjem slijedi:

$$n \geq \log\left(\frac{x_{\max} - x_{\min}}{\tilde{x}}\right) / \log(2)$$

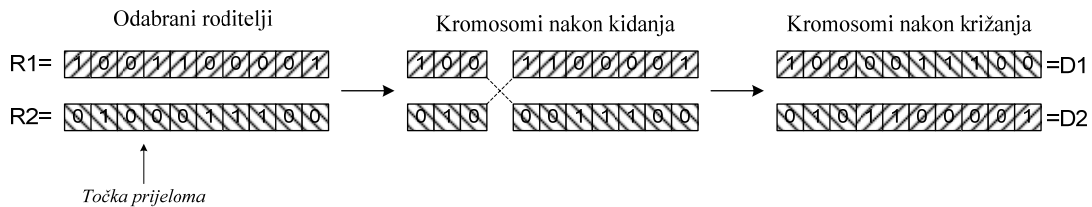
što u našem slučaju daje  $n=9$ .

### 2.3. Operator križanja

Nakon što su odabrana dva roditelja, operatorom križanja stvaraju se njihovi potomci. Operator križanja često se opisuje parametrom  $p_c$  – vjerojatnošću da se križanje doista dogodi (uobičajene vrijednosti su između 60% i 90%). Križanje je moguće napraviti na više načina. Pretpostavimo da smo odabrali dva roditelja: R1=1001100001, R2=0100011100.

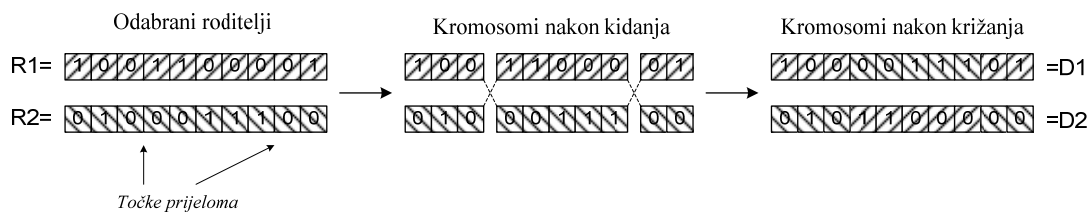
### 2.3.1. Križanje s jednom točkom prekida

Kromosomi-roditelji poslože se jedan ispod drugoga. Posredstvom slučajnog mehanizma odabere se točka kidanja oba kromosoma. Potom se stvaraju dva djeteta D1 i D2: prvo dobiva prvi dio kromosoma roditelja R1 i drugi dio kromosoma roditelja R2, a drugo dijete dobiva prvi dio kromosoma roditelja R2 i drugi dio kromosoma roditelja R1. Postupak je ilustriran na sljedećoj slici.



### 2.3.2. Križanje s t-točaka prekida

Križanje s  $t$ -točaka prekida općenitiji je slučaj prethodno opisanog križanja, gdje  $t$  može biti između 1 i  $k-1$  (gdje je  $k$  broj bitova kromosoma). Primjerice, slučaj za  $t=2$ , te s točkama prijeloma nakon 3. i 8. bita prikazan je u nastavku.



### 2.3.3. Uniformno križanje

Uniformno križanje može se promatrati kao križanje s  $k-1$  točkom prekida (gdje je  $k$  broj bitova kromosoma). Kromosomi oba roditelja raspadnu se nakon svakog bita, i potom svako dijete bira što će uzeti od kojeg roditelja. Ovo se u računalnim programima najčešće izvodi na sljedeći način:

$$D1 = R1 \cdot R2 + R \cdot (R1 \oplus R2)$$

$$D2 = R1 \cdot R2 + \bar{R} \cdot (R1 \oplus R2)$$

pri čemu  $\cdot$  predstavlja operator logičko-I nad pojedinim bitovima,  $\oplus$  operator logičko-ILI nad pojedinim bitovima,  $\bar{\phantom{x}}$  zaokruženi plus operator logičko-isključivo-ILI nad pojedinim bitovima, te  $\bar{\phantom{x}}$  povlaka operator komplementa nad pojedinim bitovima.  $R$  je pri tome slučajno generirani niz od  $k$  bitova. Lako je uočiti da će logičko-I na mjestima gdje su bitovi u oba roditelja isti tu vrijednost prekopirati u dijete, a operator logičko-isključivo-ILI na onim mjestima na kojima su bitovi roditelja različiti odabrati slučajno generirani iz  $R$  (ili komplement od  $R$  kod djeteta 2) i njega prekopirati u dijete.

Primjer 2.3

Za postupak križanja kod genetskog algoritma odabrana su dva binarna kromosoma:  $R1=1010100000$  i  $R2=0001101111$ . Prikažite rezultat uniformnog križanja tih roditelja. Prilikom križanja stvoren je slučajni broj  $R=0110111100$ .

$$R1 \cdot R2 = 0000100000$$

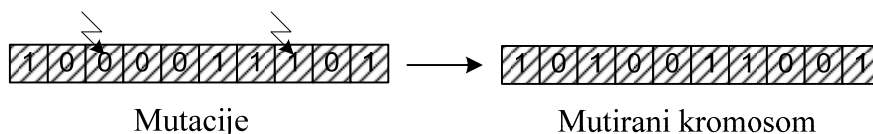
$$R1 \oplus R2 = 1011001111$$

$$D1 = R1 \cdot R2 + R \cdot (R1 \oplus R2) = 0000100000 + 0110111100 \cdot 1011001111 \\ = 0000100000 + 0010001100 = 0010101100$$

$$D2 = R1 \cdot R2 + \bar{R} \cdot (R1 \oplus R2) = 0000100000 + 1001000011 \cdot 1011001111 \\ = 0000100000 + 1001000011 = 1001100011$$

## 2.4. Operator mutacije

Operator mutacije nad binarnim kromosomima djeluje tako da posredstvom slučajnog mehanizma nekim bitovima promijeni vrijednost. Operator se najčešće opisuje vjerojatnošću promjene bita  $p_m$ , koja uobičajeno iznosi između 1% i 5% (a često je i manja). Djelovanje je prikazano na sljedećoj slici.



## 2.5. Operator selekcije

Selekcija je postupak kojim se odabiru jedinke iz populacije (najčešće u svrhu razmnožavanja). Najjednostavniji način selekcije – slučajni odabir jedinki iz populacije nije dobro rješenje; da bi genetski algoritam napredovao, veću šansu trebaju dobiti bolje jedinke. Stoga se za selekciju često koriste sljedeće tehnike: *proporcionalna selekcija* te *turnirska selekcija*.

Dobrota svake pojedine jedinke uobičajeno se procjenjuje "mjeranjem" funkcijom koju optimiramo. Primjerice, ako obavljamo maksimizaciju funkcije, česta je praksa dobrotu poistovjetiti s iznosom same funkcije:

$$fitness(i) = f(\bar{x}_i).$$

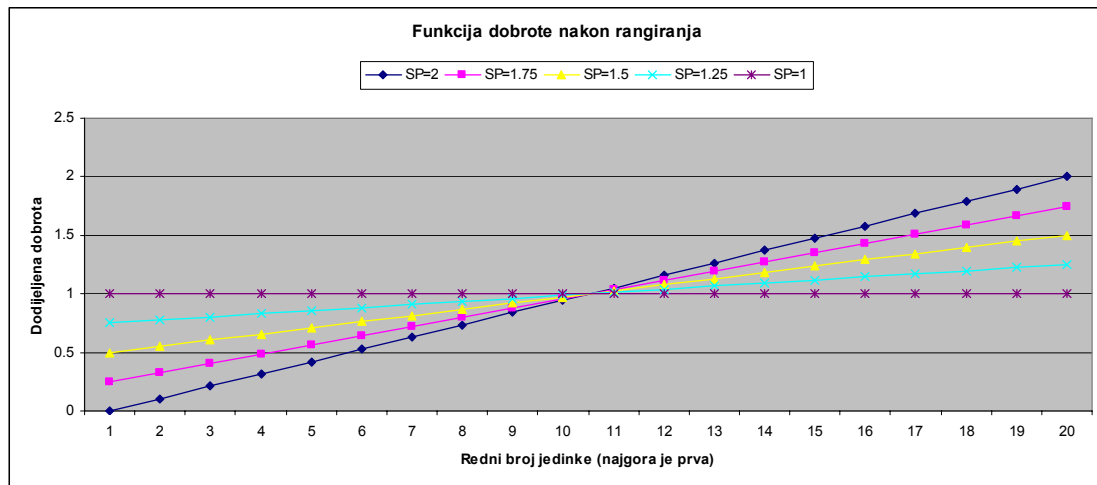
Međutim, često se pokazuje da je određivanje dobrote jedinke rangiranjem bolje rješenje. Evo o čemu se radi. Populacija se najprije sortira prema vrijednosti funkcije koja se optimira (dakle prema  $f(\bar{x}_i)$ ). Najlošija jedinka (u slučaju maksimizacije, to je ona koja ima najmanji iznos  $f(\bar{x}_i)$ ) pri tome dolazi na 1. mjesto, a najbolja na zadnje mjesto.  $i$ -toj se jedinki kao dobrota tada postavlja vrijednost definirana sljedećim izrazom:

$$fitness(i) = 2 - SP + 2 \cdot (SP - 1) \cdot \frac{i - 1}{n - 1}$$

gdje je  $SP$  parametar koji opisuje *seleksijski pritisak*, i može biti u intervalu  $[1, 2]$ , a  $n$  je veličina populacije (vidi sliku za  $n=20$  i različite vrijednosti  $SP$ -a).

Kako vidimo, uz  $SP=1$ , svim se jedinkama, neovisno o njihovoj stvarnoj dobroti pridjeljuje ista vrijednost dobrote. Ovo će rezultirati time da sve jedinke – i dobre i loše – imaju jednaku šansu biti odabrane, što neće rezultirati usmjerenim pretraživanjem prostora mogućih rješenja. Potpuno se suprotna situacija pojavljuje uz  $SP=2$ : najgoroj jedinki pridjeljena je vrijednost 0, čime se ta jedinka nikada neće birati kao potencijalni roditelj, dok najbolja jedinka ima najveći iznos pridijeljene dobrote. Za vrijednosti  $SP$ -a između 1 i 2 omjer dodijeljene dobrote najboljem i najgorem

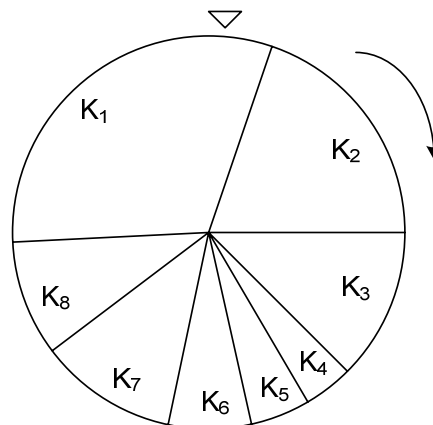
rješenju  $fitness(\bar{x}_{best}) / fitness(\bar{x}_{worst})$  mijenja se od 1 (potpuno slučajna pretraga) do  $\infty$  (jako usmjerena pretraga). Stoga se ovim parametrom može fino podešavati koliki naglasak algoritam stavlja na najbolje pronađeno rješenje.



Pojasnimo odmah i zašto je ovo bitno. Uz veliki selekcijski pritisak, algoritam će puno više vremena raditi s najboljom pronađenom jedinkom, čime će se u određenoj mjeri ponašati kao algoritam lokalne pretrage – radit će fino pretraživanje u okolici trenutno najboljeg rješenja. Problem koji se pri tome javlja jest sljedeći – što ako to rješenje nije globalni optimum? U tom slučaju postoji mogućnost da će algoritam "zaglaviti" u lokalnom optimumu, pa govorimo o preranoj odnosno prebrzoj konvergenciji. Smanjivanjem selekcijskog pritiska i manje dobra rješenja dobivaju veću šansu sudjelovanja u generiranju potomstva, i time osiguravaju da algoritam obavlja šire pretraživanje prostora, čime se povećava vjerojatnost pronalaska još boljih rješenja i smanjuje vjerojatnost prerane konvergencije. Posljedica smanjenja selekcijskog pritiska je veća robusnost algoritma; no istovremeno, povećava se i vrijeme potrebno da algoritam konvergira ka stvarnom globalnom optimumu – jednom kada ga pronađe.

### 2.5.1. Proporcionalna selekcija

Ova selekcija još je poznata pod nazivom Roulette-wheel selection. Ideja je sljedeća: postavimo sve jedinke na kolo, tako da bolja jedinka ima veću površinu na kolu (vidi sliku u nastavku). Potom zavrtimo kolo i pogledamo na kojoj se je jedinki kolo zaustavilo.



Programski, ovo možemo postići tako da sve jedinke preslikamo na kontinuirani dio pravca duljine 1. Pri tome svaka jedinka dobiva dio proporcionalan njezinoj dobnosti. Ako s  $fit(i)$  označimo dobnost  $i$ -te jedinke, a s  $len(i)$  označimo duljinu segmenta koji pripada toj jedinki, vrijedi:

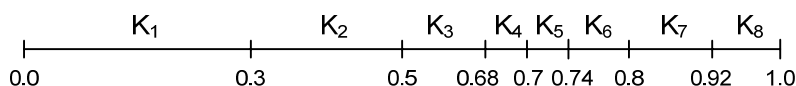
$$len(i) = \frac{fit(i)}{\sum_{j=1}^n fit(j)}$$



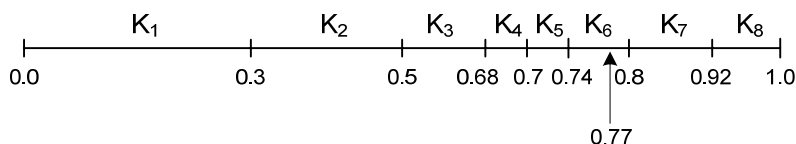
Pogledajmo to na primjeru s 8 jedinki (tablica ispod). Uočimo da na ovaj način normirana duljina segmenta direktno odgovara i vjerojatnosti odabira odnosno jedinke.

Jedinka	1	2	3	4	5	6	7	8
Dobrota jedinke	6	4	3.6	0.4	0.8	1.2	2.4	1.6
Vjerojatnost odabira	0.3	0.2	0.18	0.02	0.04	0.06	0.12	0.08

Stavimo li sve jedinke na segment pravca, dobit ćemo:



Odabir se radi tako da generiramo slučajni broj iz intervala  $[0, 1]$ , i pogledamo u čije je područje taj broj pao. Primjerice, ako generiramo broj 0.77, odabrat ćemo jedinku K6.



Spomenimo odmah i problem koji se javlja kod ovako izvedene proporcionalne selekcije, i ilustrirajmo ga na primjeru prikazanom sljedećom tablicom.

Jedinka	1	2	3	4	5	6	7	8
Dobrota jedinke	1007	1008	1005	1008	1003	1011	1017	1012
Vjerojatnost odabira	0.124768	0.124892	0.12452	0.124892	0.124272	0.125263	0.126007	0.125387

Radi se o primjeru maksimizacije funkcije pri čemu za potrebe ovog primjera radimo sa samo 8 jedinki, a funkcija je takva da joj je maksimum neki relativno veliki broj (primjerice 200000). U nekom trenutku populacija se sastoji od jedinki prikazanih u prethodnoj tablici. Uvidom u podatke vidimo da je trenutno najbolje rješenje ono pod brojem 7. Međutim, vjerojatnost njegovog odabira praktički je jednaka vjerojatnosti odabira najgoreg rješenja u populaciji, i iznosi nešto više od 12%. Problem koji ovdje vidimo jest osjetljivost prikazanog algoritma odabira na *skalu*. Ako su funkcije dobnosti relativno mali brojevi koji se pri tome dosta razlikuju, algoritam radi zadovoljavajuće. Međutim, kako vrijednosti dobnosti jedinki rastu, tako se smanjuje

relativna razlika između njih i vjerojatnosti odabira najgore i najbolje jedinke postaju ujednačene. Rješenje ovog problema leži ili u uporabi relativne dobrote (kao efektivna dobrota svake jedinke uzme se razlika između dobrote jedinke i dobrote najgore jedinke – što je bitno manje osjetljivo na skalu) odnosno u uporabi prethodno opisanog rangiranja, gdje se jedinkama temeljem njihovom ranga u populaciji dodijeli dobrota (što je potpuno neosjetljivo na skalu, ali zahtjeva sortiranje populacije), i potom se tako definirana dobrota koristi za proporcionalnu selekciju.

### 2.5.2. *k*-turnirska selekcija

*k*-turnirska selekcija iz populacije posredstvom slučajnog mehanizma odabire *k* jedinki, i potom uzima najbolju (ili najgoru – ovisno za što je korstimo). Ukoliko trebamo *n* roditelja, naprosto ćemo *n* puta ponoviti turnirsku selekciju. Čest je slučaj uporabe 3-turnirske selekcije, gdje se nasumice odabiru 3 jedinke, i potom uzima najbolja od njih. Da bismo dobili dva roditelja koja ćemo potom križati, potrebno je obaviti dva turnira.

U praksi se često koristi i dodatno pojednostavljeno 3-turnirske selekcije u kombinaciji sa *steady-state* genetskim algoritmom: da bismo dobili dva roditelja za križanje, nasumice iz populacije izvučemo 3 jedinke i uzmemo dvije bolje, a najlošiju jedinku odbacimo i zamijenimo nastalim djetetom. Time smo umjesto obavljanja tri turnira sve riješili samo jednim.

## 2.6. Detaljniji primjer

Pogledajmo sada malo detaljniji primjer generacijskog genetskog algoritma opisanog pseudokodom u nastavku.

```

generacijski_ga(VEL_POP, pm, pc, k, fja)
P = stvori_slucajnu_populaciju(VEL_POP, k)
evaluiraj_populaciju(P, fja)
ponavljaj
  sortiraj_populaciju(P)
  P' = {P(1), P(2)}
  ponavljaj za i iz {1,2,...,VEL_POP/2-1}
    R1 = odaberi_roditelja_iz(P)
    R2 = odaberi_roditelja_iz(P)
    {D1,D2}=kriزاز_1_tocka_prijeloma(R1, R2, pc)
    mutiraj(D1, pm)
    mutiraj(D2, pm)
    P' += {D1, D2}
  kraj_ponavljanja
  P = P'
  evaluiraj_populaciju(P, fja)
dok_nije_gotovo
  vrati_najbolju_jedinku
kraj

```

**Okvir 2.3 Detaljniji primjer generacijskog GA, pseudokod**

U prethodnom primjeru, *P* označava aktivnu populaciju, *P'* sljedeću generaciju jedinki koje će zamijeniti populaciju *P*, *k* je broj bitova kromosoma, *pm* vjerojatnost mutacije bita a *pc* vjerojatnost križanje jedinki.

Algoritam započinje stvaranjem inicijalne populacije i njenom evaluacijom. Potom se ulazi u petlju koja se ponavlja sve dok uvjet zaustavljanja nije zadovoljen (primjerice, dosegnut maksimalni broj generacija ili pronađen željeni minimum, ukoliko je on poznat).

Petlja započinje sortiranjem jedinki populacije, i to tako da se najbolje jedinke postave na početak. Potom se u novu populaciju kopiraju dvije najbolje jedinke. Ovo je učinjeno kako bi se osiguralo da jednom pronađeno najbolje rješenje nikada ne bude naknadno uništeno (govorimo o *elitizmu*). Time smo u novu populaciju već dodali 2 jedinke, pa je ostalo mjesta za još  $VEL\_POP-2$  jedinke. Kako nam svako križanje stvara dva djeteta, ulazimo u novu petlju koju ponovimo  $(VEL\_POP-2)/2$  puta: odaberemo dva roditelja i križamo ih. Nastalu djecu mutiramo i ubacimo u novu generaciju.

Odabir roditelja prikazan je pseudokodom u okviru 2.4. Budući da rješavamo minimizacijski problem, vrijednost funkcije u točki koju predstavlja kromosom ne možemo uzeti kao mjeru dobrote, budući da što je ta vrijednost veća, dobrota je manja. Stoga se pribjegava jednostavnoj transformaciji: u čitavoj populaciji pronađe se ona jedinka za koju je vrijednost funkcije najveća (u pseudokodu ta je vrijednost označena sa  $NV$ ). Potom se dobrota pojedine jedinke definira kao razlika između  $NV$  i vrijednosti funkcije u točki koju jedinka predstavlja. Ovime automatski slijedi da će dobrota jedinke za koju je vrijednost funkcije najveća biti upravo nula, a dobrota jedinke za koju je funkcija minimalna bit će najveća.

```

odaberi_roditelja_iz(P)
  NV = najveca_vrijednost_funkcije(P)
  za svaku jedinku Pi iz P
    dobrota(Pi) = NV - funkcija(Pi)
  kraj
  SD = sumiraj_dobrote_svih_jedinki()
  za svaku jedinku Pi iz P
    len(Pi) = dobrota(Pi) / SD
  kraj
  B = generiraj_slucajni_broj_iz_intervala(0, 1)
  akumulirana_suma = 0
  za i od 1 do VEL_POP
    akumulirana_suma += len(Pi)
    ako je B < akumulirana_suma vrati Pi
  kraj
  vrati zadnju jedinku iz populacije
kraj

```

**Okvir 2.4 Detaljniji primjer generacijskog GA, odabir jedinke**

Potom se u kodu implementira preslikavanje na segmente pravca duljine 1, odabire slučajni broj iz intervala  $[0, 1]$  te traži kojoj jedinki na pravcu pripada odabrana točka.

Procedura započinje stvaranjem slučajnog broja iz intervala  $[0, 1]$ , i potom provjerom treba li uopće obaviti križanje. Ako je stvoreni broj  $B$  veći od vjerojatnosti križanja, tada se kao djeca direktno vraćaju roditelji. Ako se pak križanje treba obaviti, onda se posredstvom slučajnog mehanizma odabire točka prijeloma  $T$ , i potom stvaraju djeca. Svako dijete prvih  $T$  bitova preuzme od jednog roditelja, a preostalih  $k-T$  od drugog roditelja.

Operator križanja koji se koristi je križanje s jednom točkom prijeloma, i opisan je pseudokodom u okviru 2.6.



```

krizaj_1_tocka_prijeloma(R1, R2)
  B = generiraj_slučajni_broj_iz_intervala(0, 1)
  ako je B > pc tada
    D1 = R1
    D2 = R2
    vrati {D1, D2}
  inace
    T = slucajno_izaberi_tocku_prijeloma_iz(1, k-1)
    za i iz 1 do T
      D1(i) = R1(i)
      D2(i) = R2(i)
    kraj
    za i iz T+1 do k
      D1(i) = R2(i)
      D2(i) = R1(i)
    kraj
    vrati {D1, D2}
  kraj
kraj

```

Okvir 2.6 Detaljniji primjer generacijskog GA, križanje

Algoritam mutacije prikazan je u okviru 2.5. Procedura je izuzetno jednostavna: za svaki bit kromosoma stvara se jedan slučajni broj iz intervala  $[0, 1]$ , i provjerava je li taj broj manji ili jednak vjerojatnosti obavljanja mutacije. Ako je, dotični se bit invertira.

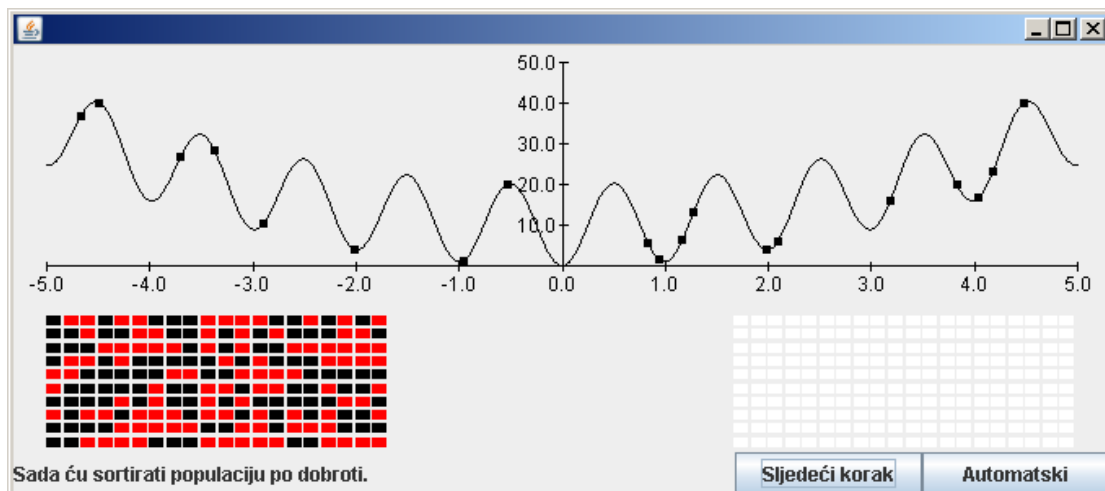
```

mutiraj(D)
  za i iz 1 do k
    B = generiraj_slučajni_broj_iz_intervala(0, 1)
    ako je B <= pm tada
      okreni_bit(D, i)
    kraj
  kraj
kraj

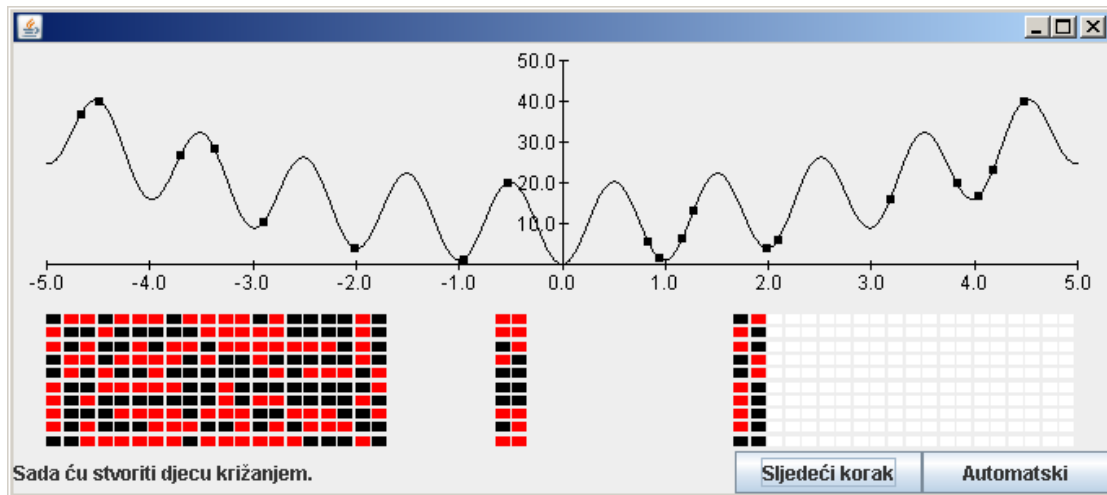
```

Okvir 2.5 Detaljniji primjer generacijskog GA, mutacija

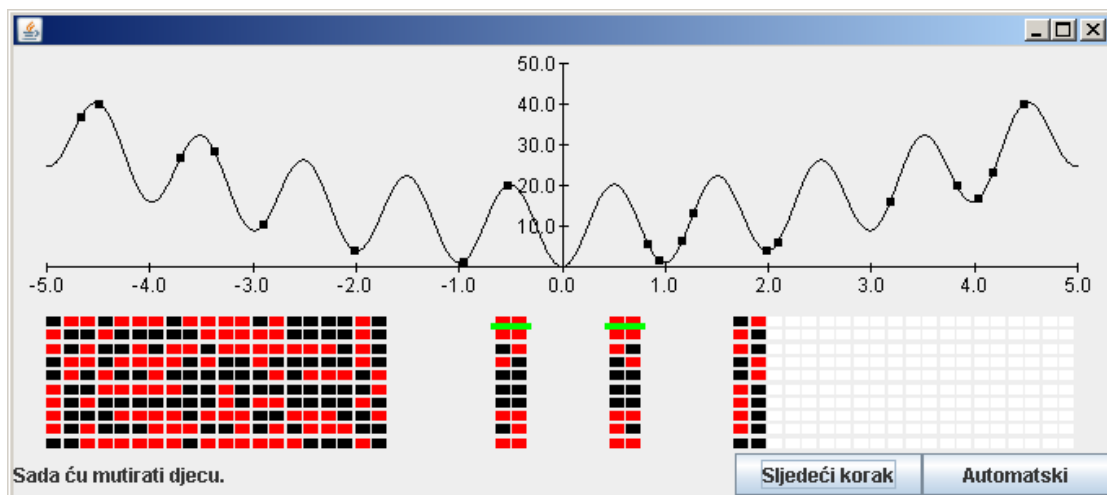
Rad programa koji implementira opisani kod prikazan je u nastavku (slike 2-6, 2-7, 2-8, 2-9, 2-10, 2-11 i 2-12). Pri tome se populacija sastoji od 20 jedinki (kromosoma), gdje je svaki kromosom sastavljen od 10 bitova (na slici dolje, lijevo). Odgovarajuća rješenja prikazana su i na samoj funkciji čiji tražimo minimum.



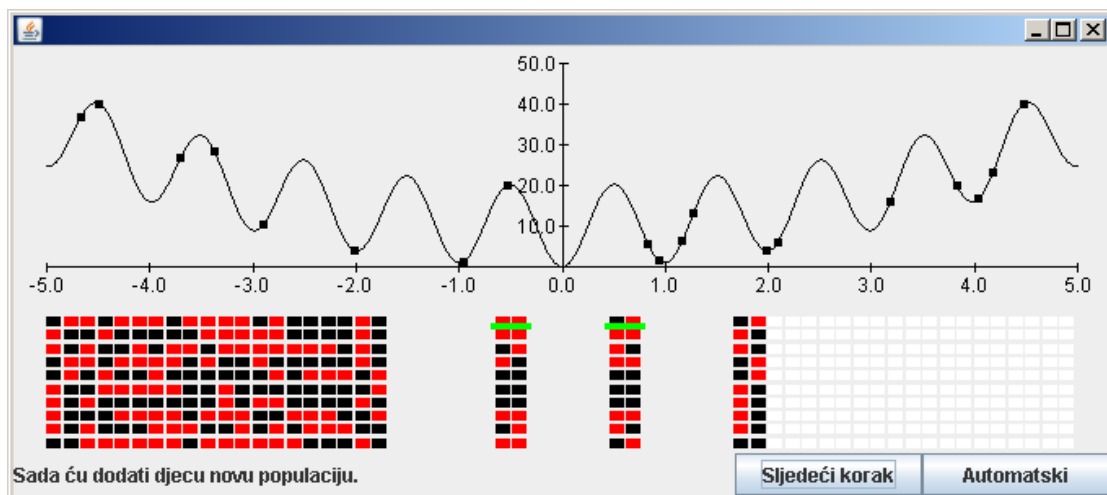
Slika 2-6 Stvorena početna populacija



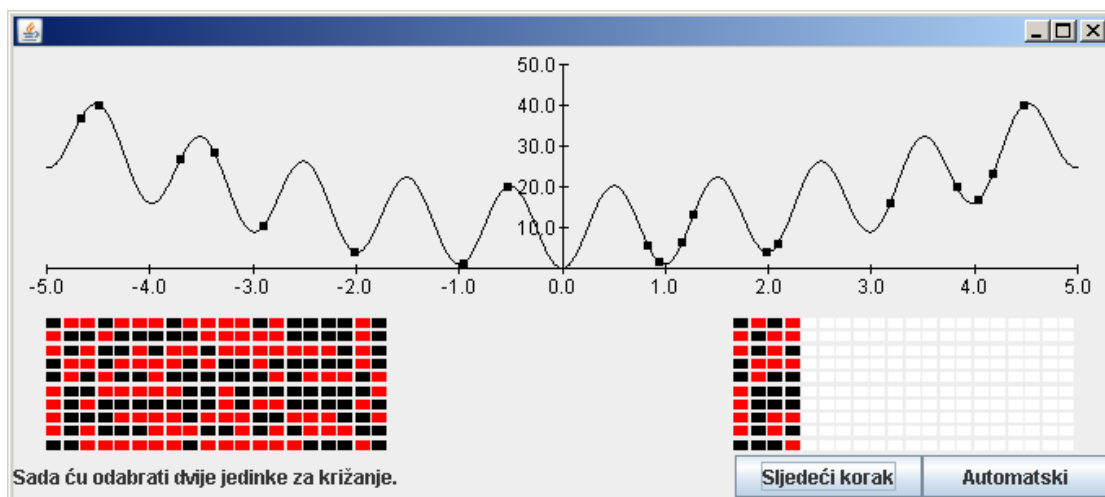
Slika 2-7 Najbolji roditelji dodani u novu populaciju, odabrani roditelji za križanje



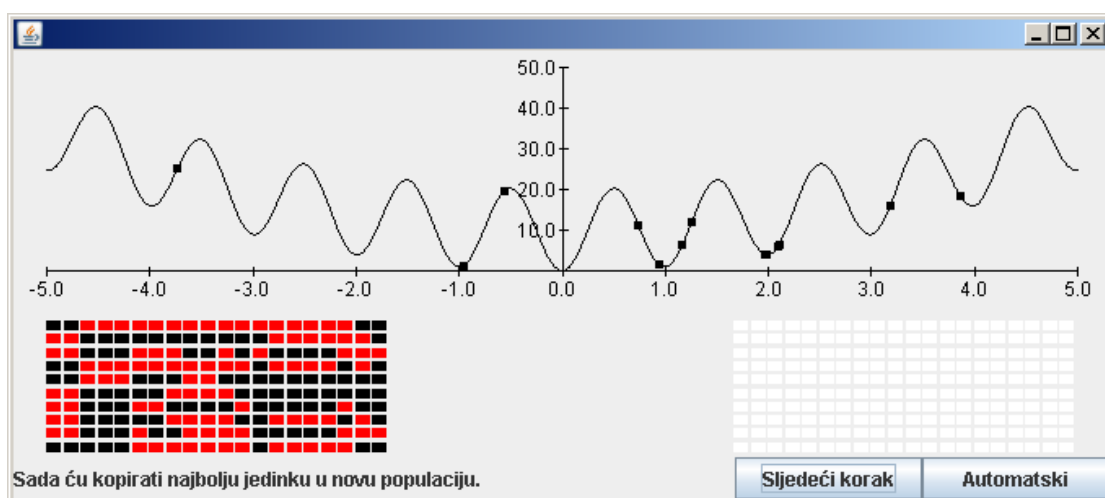
Slika 2-8 Definirana točka prijeloma i obavljeno križanje



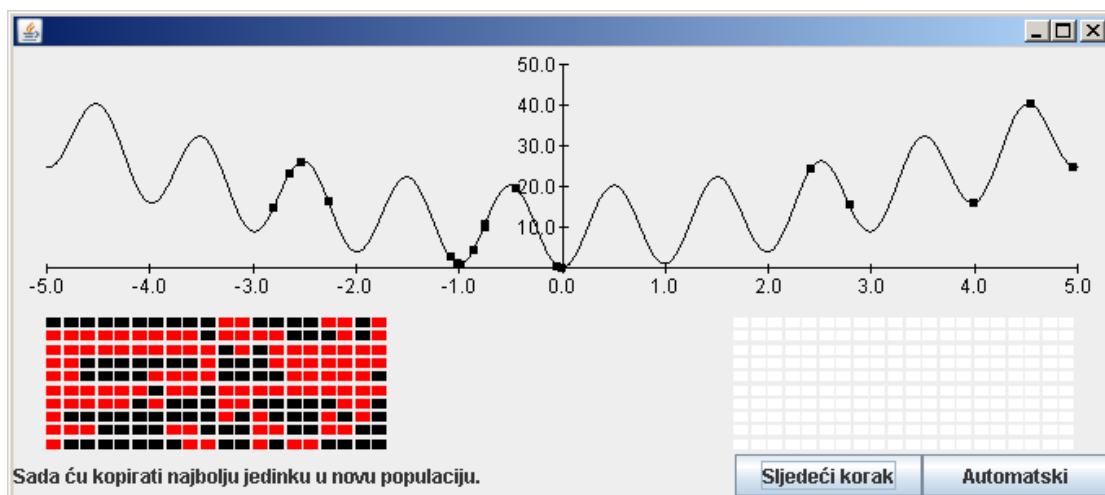
Slika 2-9 Djeca su mutirana



Slika 2-10 Djeca su dodana u novu populaciju



Slika 2-11 Početak nove generacije



Slika 2-12 I nekoliko generacija kasnije – rješenje je pronađeno

## 2.7. Drugi prikazi kromosoma

Binarni kromosomi prikladan su prikaz kada se radi o jednostavnijim problemima. Vrlo često, međutim, radimo s problemima gdje bi cijena kodiranja/dekodiranja rješenja u/iz binarnog prikaza bila potpuno neprihvatljiva. Primjerice, rješavamo li

problem trgovačkog putnika genetskim algoritmom, kao kromosom često se koristi polje cijelih brojeva čija veličina odgovara broju gradova, a na  $i$ -tom mjestu se nalazi redni broj grada koji u  $i$ -tom koraku treba posjetiti. Dakako, u tom slučaju treba razviti i odgovarajuće operatore križanja i mutacije. Primjerice, mutacija bi mogla odabrati dva elementa polja i zamijeniti im redoslijed. S križanjem ovdje također moramo biti posebno oprezni, jer ne smijemo dopustiti da jednostavnim prijelomom kromosoma roditelja dobijemo dijete koje neke gradove posjećuje dva puta a neke nikada.

Ako je rješenje problema koji rješavamo više decimalnih brojeva, umjesto binarnog prikaza možemo koristiti polje decimalnih brojeva. U tom slučaju operator križanja na  $i$ -to mjesto u dijete može staviti aritmetičku sredinu vrijednosti koje su bile na  $i$ -tom mjestu u oba roditelja. Mutaciju pak možemo implementirati tako da svakom elementu polja dodamo neki slučajno generirani broj izvučen iz normalne distribucije.

## 2.8. Genetski algoritam za izradu rasporeda međuispita

U nastavku slijedi još jedan primjer iz stvarnog života: opis genetskog algoritma koji rješava problem izrade rasporeda međuispita na Fakultetu elektrotehnike i računarstva.

### 2.8.1. Opis problema

Problem izrade rasporeda međuispita definiran je na sljedeći način.

Definiran je skup disjunktih termina  $\mathbf{T}=\{t_1, t_2, \dots, t_n\}$ . Za svaki termin poznat je dan održavanja te kapacitet termina (koliko studenata termin može prihvatiti). Definirana je funkcija  $dayDistance(t_i, t_j)$  koja za dva zadana termina vraća razmak između tih termina. Funkcija vraća 0 ako su termini u istom danu, 1 ako su u sujednim danima, itd.

Definiran je skup kolegija  $\mathbf{C}=\{c_1, c_2, \dots, c_k\}$  koji trebaju obaviti ispit. Svaki kolegij ispit obavlja samo jednom terminu (pretpostavka je da su termini dovoljno veliki za prihvati barem jednog kolegija).

Definirana je relacija *zajedno*:  $\mathbf{C} \times \mathbf{C} \rightarrow \{0,1\}$ , na sljedeći način: kolegiji  $c_i$  i  $c_j$  su u relaciji *zajedno* ako oba moraju biti smještene u isti termin (nije bitno koji).

Definiran je skup studenata  $\mathbf{S}=\{s_0, s_1, \dots, s_l\}$ .

Definirana je relacija *sluša*  $\mathbf{S} \times \mathbf{C} \rightarrow \{0,1\}$ , na sljedeći način: kolegij  $c_i$  i student  $s_j$  su u relaciji *sluša*, ako student  $s_j$  sluša kolegij  $c_i$ , i stoga treba pisati ispit iz tog kolegija.

Definirana je relacija *prihvatljivo*  $\mathbf{C} \times \mathbf{T} \rightarrow \{0,1\}$ , na sljedeći način: kolegij  $c_i$  i termin  $t_j$  su u relaciji *prihvatljivo* ako je termin  $t_j$  prihvatljiv za održavanje ispita kolegija  $c_i$ .

Zadatak je napraviti mogući raspored izvođenja međuispita. Mogući označava da niti u jednom terminu  $t_i$  ne postoje dva kolegija  $c_j$  i  $c_k$  pridjeljena tom terminu koji imaju zajedničkog studenta, te da ne postoji termin u kojem je broj smještenih studenata veći od kapaciteta termina.

Kvaliteta mogućih rješenja potom se definira sljedećim željama:

- jako je loše ako postoje studenti koji u istom danu imaju više ispita (ovo je moguće jer uobičajeno u danu postoji više termina kada se mogu pisati ispiti); raspored je to lošiji što ovakvih studenata ima više, te

- loše je ako postoje studenti koji imaju ispite u susjednim danima (međutim, ne toliko loše kao u prethodnom slučaju).

Programski, kvalitetu je suprotna funkciji kazne, koju računamo na sljedeći način: za svakog studenta koji u istom danu ima dva predmeta kazni nadodaj faktor P1. To napravi onoliko puta koliko se to dogodi u tom danu, i to za svaki dan kada se dogodi, i za svakog studenta. P1 je obično neki relativno veliki broj; primjerice: 50. Potom za svakog studenta koji u susjednim danima ima dva predmeta kazni nadodaj faktor P2. To napravi onoliko puta koliko se to dogodi i za svakog studenta kojemu se to dogodi. P2 je obično neki manji broj; primjerice: 5 (argument: 10 je puta lošije pisati dva ispita u jednom danu nego ispite u dva susjedna dana).

Rasporedi napravljeni genetskim algoritmom moraju biti mogući, a zadatak samog genetskog algoritma je dodatno između svih mogućih rasporeda pronaći onaj čija je kvaliteta maksimalna (odnosno koji ima minimalnu kaznu).

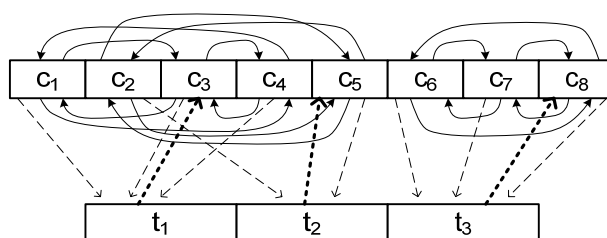
Pokazuje se da je prvu uvjet relativno lagano ispuniti – do mogućih rasporeda dolazi se relativno brzo. Optimizacija kvalitete tih rasporeda potom je puno teži problem.

### 2.8.2. Prikaz kromosoma

Za prikaz kromosoma u ovom se slučaju ne koristi binarni kromosom – to bi zahtijevalo enormnu količinu posla oko kodiranja i dekodiranja (dakle, da bismo shvatili kakav je to raspored). Umjesto toga, kromosom je podatkovna struktura koja se sastoji od dva polja:

- polje termini je polje svih termina, gdje se za svaki termin pamti popis kolegija smještenih u taj termin, te
- polje kolegiji je polje svih kolegija i tu se dodatno pamti termin u koji kolegij raspoređen.

Dodatno, svi kolegiji smješteni u isti termin međusobno su povezani u cirkularnu dvostruku listu, pa svaki termin pamti samo referencu na jedan od kolegija (svi ostali dostupni su praćenjem veza u listi). Slika 2-13 prikazuje ovaj način povezivanja.



Slika 2-13 Struktura kromosoma za problem rasporeda međuispita

U primjeru imamo tri termina i osam kolegija. U termin  $t_1$  raspoređeni su kolegiji  $c_1$ ,  $c_3$  i  $c_4$ , u termin  $t_2$  kolegiji  $c_2$ , i  $c_5$  te u termin  $t_3$  kolegiji  $c_6$ ,  $c_7$  i  $c_8$ . Iako se ovakva struktura čini pomalo redundantna, omogućava izuzetno efikasan dohvat termina za poznati kolegij, te listanje kolegija koji pripadaju nekom terminu. Također, vrlo se efikasno mogu implementirati operacije micanja kolegija iz termina i dodavanja kolegija u termin (a time i operacija seljenja kolegija iz jednog termina u drugi).

### 2.8.3. Genetski operatori

*Operator križanja* izveden je po principu uniformnog križanja. Stvara se samo jedno dijete, pri čemu se svaki kolegij stavlja ili u termin određen prvim roditeljem, ili u termin određen drugim roditeljem (izbor se donosi slučajno).

*Operator mutacije* izveden je tako da slučajno odabire kolegije koje će mutirati, i potom im slučajno odabire nove termine. Iako se čini jednostavno, prilikom seljenja kolegija treba pripaziti ima li još koji kolegij koji mora biti u istom terminu kao i ovaj kojeg selimo, pa ako ima, sve ih treba preseliti.

Izbor roditelja obavlja se troturnirskom selekcijom; točnije, slučajno se izabiru tri jedinke; dvije bolje postaju roditelji, stvaraju jedno dijete i to dijete zamjenjuje u populaciji treću (najlošiju) jedinku.

## 2.9. Genetsko programiranje

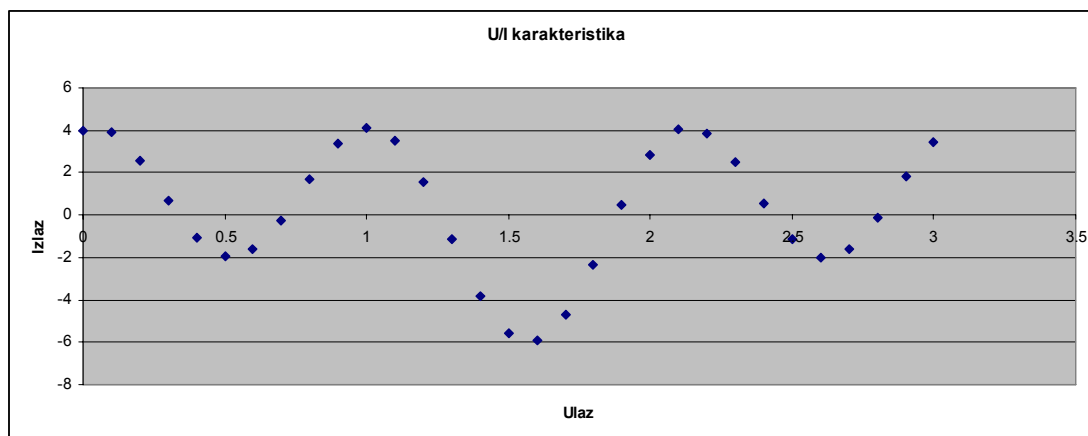
Genetsko programiranje tehnika je izuzetno srodna genetskom algoritmu. Razlika je u tome što kromosom predstavlja *program* koji je rješenje zadanog problema. Tipična struktura podataka koja se kod genetskog programiranja koristi za prikaz kromosoma jest stablo.

Pogledajmo to na jednostavnom primjeru. Mjerenjem izlaza nekog procesa snimljena je njegova ulazno-izlazna karakteristika.

Tablica 2-1 Ulazno-izlazna karakteristika promatranog sustava

x	f(x)	x	f(x)
0	4	1.6	-5.93108
0.1	3.892383	1.7	-4.70869
0.2	2.578716	1.8	-2.32285
0.3	0.657845	1.9	0.472592
0.4	-1.0855	2	2.816585
0.5	-1.96498	2.1	4.031366
0.6	-1.63934	2.2	3.846619
0.7	-0.23462	2.3	2.480139
0.8	1.700922	2.4	0.548066
0.9	3.393531	2.5	-1.16275
1	4.122921	2.6	-1.97962
1.1	3.485439	2.7	-1.58571
1.2	1.548364	2.8	-0.13352
1.3	-1.15971	2.9	1.809713
1.4	-3.82031	3	3.465504
1.5	-5.59958		

Zanima nas algebarski izraz koji najbolje opisuje tabeliranu funkciju (grafički prikaz dat je na slici 2-14).

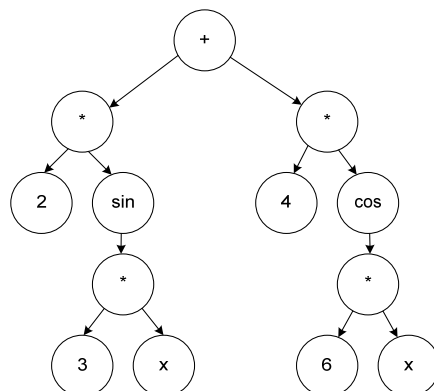


Slika 2-14 Ulazno-izlazna karakteristika sustava

Zadatak genetskog programiranja jest pronaći funkciju. Sjetimo se da se svaka funkcija može prikazati operatorskim stablom. Primjerice, funkciju:

$$f(x) = 2 \cdot \sin(3 \cdot x) + 4 \cdot \cos(6 \cdot x)$$

je moguće prikazati stablom sa slike 2-15.

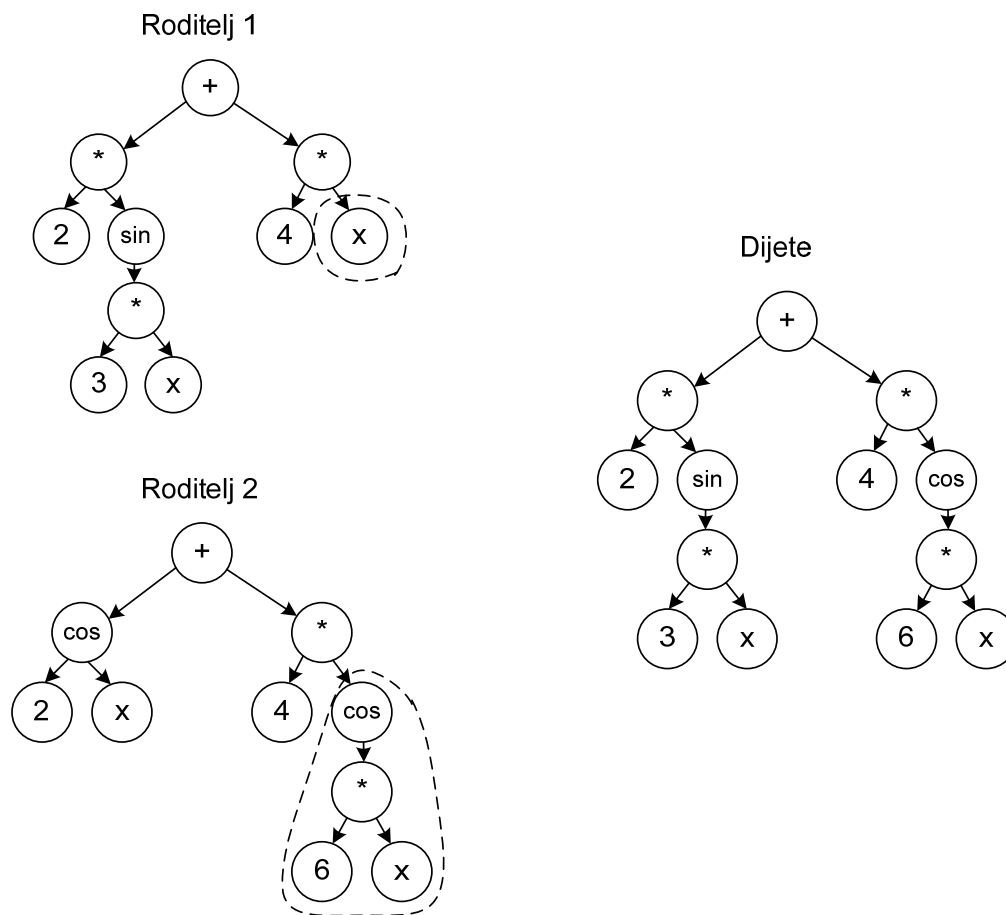


Slika 2-15 Prikaz kromosoma kod genetskog programiranja

Kromosomi kod genetskog programiranja upravo su operatorska stabla, čiji čvorovi mogu biti, primjerice, aritmetičke operacije (+, -, \*, /), ugrađene funkcije (sin, cos), konstante (bilo koji cijeli ili decimalni broj) te varijable. Što se više vrsta čvorova dozvoli, veća je šansa da se pronađe adekvatno stablo koje odgovara zadanoj funkciji, ali raste i vrijeme potrebno za njegov pronalazak.

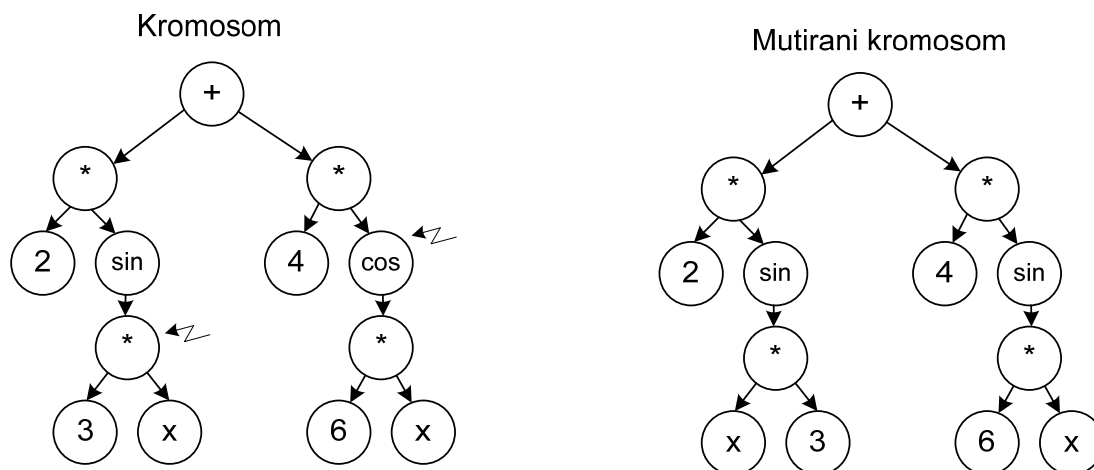
Evaluacija kromosoma u našem slučaju može se izvesti kao ukupna suma razlike izlaza funkcije u promatranoj točki i izmjerenoj podatka (ili možemo računati srednje kvadratno odstupanje).

Križanje dvaju roditelja tipično se radi tako da se neko podstablo jednog roditelja zamijeni nekim podstablom drugog roditelja (slika 2-16).



Slika 2-16 Prikaz djelovanja operatora križanja kod genetskog programiranja

Operator mutacije može se izvesti na više načina – primjerice, slučajnom zamjenom čvora, brisanjem podstabla i zamjenom s novim slučajno stvorenim podstablom, zamjenom redosljeda djece i sl. Jedan primjer prikazan je u nastavku.



Slika 2-17 Djelovanje operatora mutacije kod genetskog programiranja

Jednom kada smo definirali sve potrebno (način prikaza kromosoma, način evaluacije, djelovanje operatora križanja i mutacije) postupak je dalje standardan (jednak kao kod klasičnog genetskog algoritma).

Uočimo također da problem koji rješavamo genetskim programiranjem ne mora biti ovako jednostavan. Zamislimo sljedeći primjer: imamo parove podataka (*slučajno*



*stvoreno polje brojeva, sortirano polje brojeva*). Sjetimo li se da se svaki program može prikazati sintaksnim stablom – mogli bismo definirati dozvoljene vrste čvorova, i potom tražiti genetskim programiranjem program koji nesortirana polja prevodi u sortirana. Kromosom bi tada doslovno bio program koji bismo uporabom jezičnog procesora mogli prevesti, pokrenuti i evaluirati njegov rad.

### **Pitanja za ponavljanje**

- Opišite *steady-state* GA (pseudokod).
- Opišite *generacijski* GA (pseudokod).
- Opišite binarni prikaz rješenja. Preciznost.
- Navedite i opišite vrste križanja binarno prikazanih kromosoma.
- Opišite mutaciju kod binarno prikazanih kromosoma.
- Opišite mehanizam proporcionalne selekcije. Koji su problemi i kako ih rješavamo?
- Opišite mehanizam turnirske selekcije.
- Opišite druge načine prikaza rješenja te navedite kako tada izvodimo genetske operatore (križanje i mutaciju).
- Opišite ideju genetskog programiranja.

## Literatura

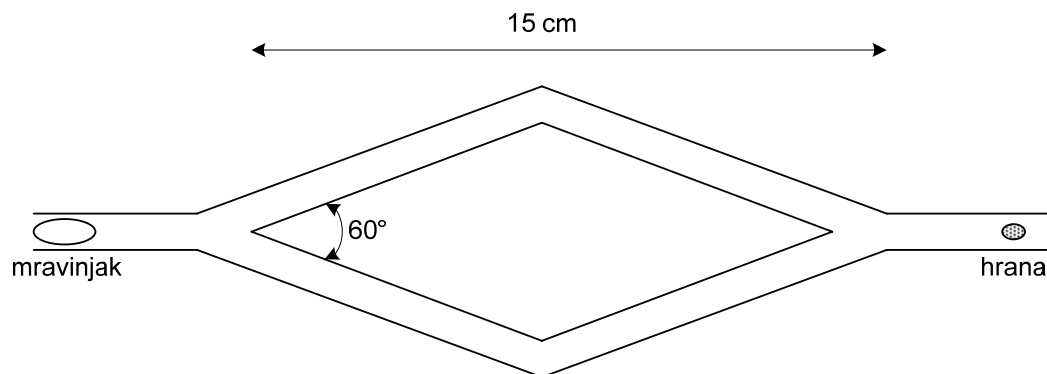
- [1] Fogel, L. J., Owens, A. J. and Walsh, M. J.: Artificial Intelligence through Simulated Evolution. New York: John Wiley, 1966.
- [2] Rechenberg, I., Evolutionsstrategie: Optimierung technischer Systeme und Prinzipien der biologischen Evolution, Frommann-Holzboog, Stuttgart, 1973
- [3] Schwefel, H.-P.: Evolutionsstrategie und numerische Optimierung. Dissertation, Technische Universität Berlin, 1975.
- [4] Holland, J. H. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Ann Arbor, MI: University of Michigan Press, 1975.
- [5] Koza, J.R., Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.
- [6] Darwin, C., On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life (1st ed.), London: John Murray, 1859.
- [7] Alba, E., Dorronsoro, B., *Cellular Genetic Algorithms*, Series: Operations Research/Computer Science Interfaces Series, Vol.42, Springer, 2008

### 3. Algoritam kolonije mrava

Mravi su izuzetno jednostavna bića – usporedimo li ih, primjerice, s čovjekom. No i tako jednostavna bića zahvaljujući socijalnim interakcijama postižu zadivljujuće rezultate. Sigurno ste puno puta u životu naišli na mravlju autocestu – niz mrava koji u koloni idu jedan za drugim prenoseći hranu do mravinjaka. O ponašanju ovih stvorenja snimljen je čitav niz dokumentarnih filmova: o njihovoj suradnji, o organizaciji mravinjaka, o nevjerojatnoj kompleksnosti i veličini njihovih kolonija. No kako jedno jednostavno biće poput mrava može ispoljavati takvo ponašanje?

Čitav dijapazon različitih znanstvenih disciplina iskazao je interes za mrave. A nama (znanstvenicima s područja računalne znanosti) mravi su posebno zanimljivi iz vrlo praktičnog razloga – mravci rješavaju optimizacijske probleme! Naime, uočeno je da će mravci uvijek pronaći najkraći put između izvora hrane i njihove kolonije, što će im omogućiti da hranu dopremaju maksimalno brzo. Da bi misterij bio još veći, spomenimo samo da mravlje vrste ili uopće nemaju razvijen vidni sustav, ili je on ekstremno loš.

Kako bi istražili ponašanje mrava, Deneubourg i suradnici [1,2] napravili su niz eksperimenata koristeći dvokraki most postavljen između mravinjaka i hrane (slika 3-1).



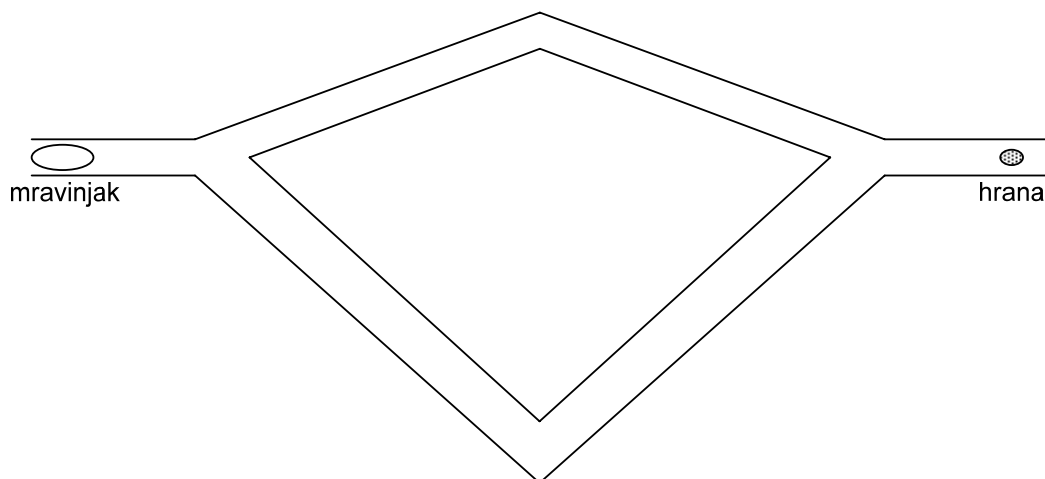
Slika 3-1 Eksperiment dvokrakog mosta, prvi

Mravi prilikom kretanja ne koriste osjet vida, već je njihovo kretanje određeno socijalnim interakcijama s drugim mravima [3,4]. Na putu od mravinjaka do hrane, te na putu od hrane do mravinjaka, svaki mrav ostavlja kemijski trag – feromone. Mravi imaju razvijen osjet feromona, te prilikom odlučivanja kojim putem krenuti tu odluku donose obzirom na jakost feromonskog traga koji osjećaju. Tipično, mrav će se kretati smjerom jačeg feromonskog traga.

Deneubourg i suradnici prilikom eksperimenata varirali su duljine krakova mosta. U prvom eksperimentu oba su kraka jednako duga. Mravi su na početku u podjednakom broju krenuli preko oba kraka. Nakon nekog vremena, međutim, dominantni dio mrava kretao se je samo jednim krakom (slučajno odabranim). Objašnjenje je sljedeće. Na početku, mravi slučajno odabiru jedan ili drugi krak, i krećući se njima ostavljaju feromonski trag. U nekom trenutku dogodi se da nekoliko mrava više krene jednim od krakova (uslijed slučajnosti) i tu nastane veća koncentracija feromona. Privučeni ovom većom količinom feromona, još više mrava krene tim krakom što dodatno poveća količinu feromona. S druge strane, kako drugim krakom krene manje mrava, količina feromona koja se osvježava je manja, a feromoni s vremenom i isparavaju. Ovo u konačnici dovodi do situacije da se stvori jaki feromonski trag na

jednom kraku, i taj feromonski trag privuče najveći broj mrava. Eksperiment je ponovljen više puta, i uočeno je da u prosjeku u 50% slučajeva mravi biraju jedan krak, a u 50% slučajeva biraju drugi krak.

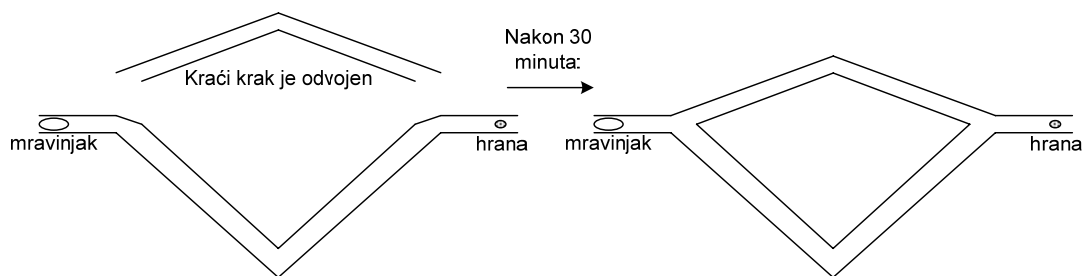
Sljedeći eksperiment napravljen je s dvokrakim mostom kod kojeg je jedan krak dvostruko dulji od drugoga (vidi sliku 3-2). U svim pokušajima eksperimenta pokazalo se je da najveći broj mrava nakon nekog vremena bira kraći krak.



Slika 3-2 Eksperiment dvokrakog mosta, drugi

Ovakvo ponašanje na makroskopskoj razini – pronalazak najkraće staze između hrane i mravinjaka rezultat je interakcija na mikroskopskoj razini – interakcije između pojedinih mrava koji zapravo nisu svjesni "šire slike" [5,6,7]. Takvo ponašanje temelj je onoga što danas znamo pod nazivom *izranjajuća inteligencija* (engl. *emerging intelligence*). ★

Konačno, kako bi se provjerila dinamika odnosno sposobnost prilagodbe mrava na promjene, napravljen je treći eksperiment (vidi sliku 3-3).



Slika 3-3 Eksperiment dvokrakog mosta, treći

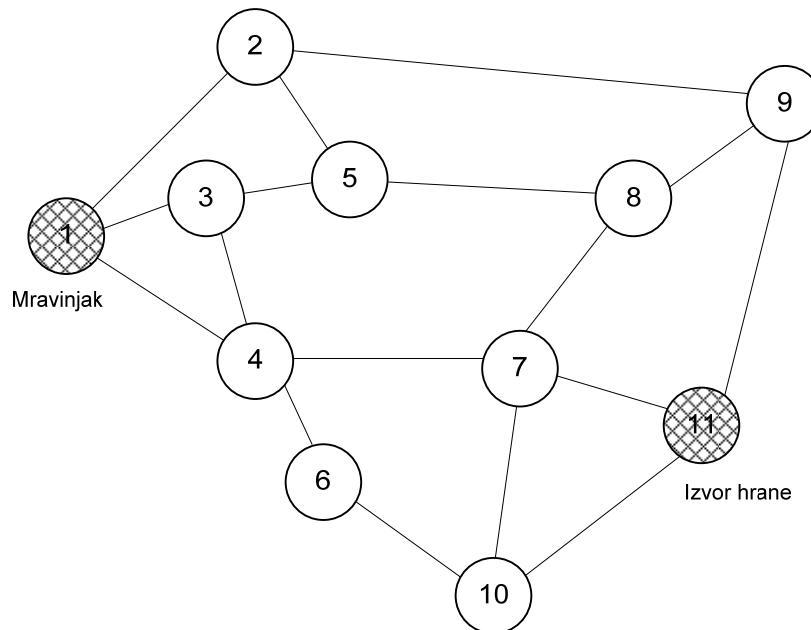
Kod ovog eksperimenta mravinjak i izvor hrane najprije su spojeni jednokrakim mostom (čiji je krak dugačak). Mravi su krenuli tim krakom do hrane i natrag. Nakon 30 minuta kada se je situacija stabilizirala, mostu je dodan drugi, dvostruko kraći krak. Međutim, eksperiment je pokazao da se je najveći dio mrava i dalje nastavio kretati duljim krakom, zahvaljujući stvorenom jakom feromonskom tragu.

### 3.1. Pojednostavljeni matematički model

Matematički model koji opisuje kretanje mrava dan je u [8]. Mravi prilikom kretanja u oba smjera (od mravinjaka pa do izvora hrane, te od izvora hrane pa do mravinjaka) neprestano ostavljaju feromonski trag. Štoviše, neke vrste mrava na povratku prema mravinjaku ostavljaju to jači feromonski trag što je izvor pronađene hrane bogatiji. Simulacije ovakvih sustava, posebice uzmemo li u obzir i dinamiku isparavanja

feromona izuzetno su kompleksne. Međutim, ideje i zakonitosti koje su ovdje uočene našle su primjenu u nizu mravljih algoritama – u donekle pojednostavljenom obliku.

Pogledajmo to na primjeru. Između mravinjaka i izvora hrane nalazi se niz tunela (slika 3-4, tuneli su modelirani bridovima grafa).



Slika 3-4 Primjer pronalaska puta od mravinjaka do izvora hrane

Ideja algoritma je jednostavna. U fazi inicijalizacije, na sve se bridove postavi ista (fiksna) količina feromona. U prvom koraku, mrav iz mravinjaka (čvor 1) mora odlučiti u koji čvor krenuti. Ovu odluku donosi na temelju vjerojatnosti odabira brida  $p_{ij}^k$ :

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha}{\sum_{l \in N_i^k} \tau_{il}^\alpha}, & \text{ako } j \in N_i^k \\ 0, & \text{ako } j \notin N_i^k \end{cases}$$

pri čemu  $\tau_{ij}$  predstavlja vrijednost feromonskog traga na bridu između čvorova  $i$  i  $j$ , a  $\alpha$  predstavlja konstantu. Skup  $N_i^k$  predstavlja skup svih indeksa svih čvorova u koje je u koraku  $k$  moguće prijeći iz čvora  $i$ . Konkretno, u našem slučaju  $N_1^1 = \{2, 3, 4\}$ . Ako iz čvora  $i$  nije moguće prijeći u čvor  $j$ , vjerojatnost će biti 0. Suma u nazivniku prethodnog izraza ide, dakle, po svim bridovima koji vode do čvorova u koje se može stići iz čvora  $i$ .

Nakon što odabere sljedeći čvor, mrav ponavlja proceduru sve dok ne stigne do izvora hrane (čvor 11). Uočimo kako se rješenje problema ovom tehnikom gradi dio po dio – mravlji algoritmi pripadaju porodici *konstrukcijskih algoritama*.

Jednom kada stigne do izvora hrane, mrav zna koliki je put prešao. Naši umjetni mravi feromone tipično ostavljaju na povratku, i to na način da je količina feromona proporcionalna dobroti rješenja (odnosno obrnuto proporcionalna duljini puta). Ažuriranje radimo za sve bridove kojima je mrav prošao, i to prema izrazu:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta \tau^k$$

gdje je:

$\Delta\tau^k = \frac{1}{L}$ , pri čemu je  $L$  duljina pronađenog puta.

Isparavanje feromona modelirano je izrazom:

$$\tau_{ij} \leftarrow \tau_{ij} \cdot (1 - \rho)$$

gdje je  $\rho$  brzina isparavanja (iz intervala 0 do 1). Isparavanje se primjenjuje na sve bridove grafa.

Ovaj pristup, dakako, ima svojih problema. Primjerice, kada mrav dinamički gradi put, ako se u svakom čvoru dopusti odabir bilo kojeg povezanog čvora, moguće je dobiti cikluse, što je nepoželjno. Također, ažuriranje feromonskog traga nakon svakog mrava pokazalo se je kao loše.

Temeljeći se na opisanim principima moguće je napisati jednostavan optimizacijski algoritam (vidi okvir 3.1).

```
ponavljaj dok nije kraj
  ponovi za svakog mrava
    stvori rješenje
    vrednuj rješenje
  kraj ponovi
  odaberi podskup mrava
  ponovi za odabrane mrave
    azuriraj feromonske tragove
  kraj ponovi
  ispari feromonske tragove
kraj ponavljanja
```

**Okvir 3.1 Jednostavan mravlji algoritam**

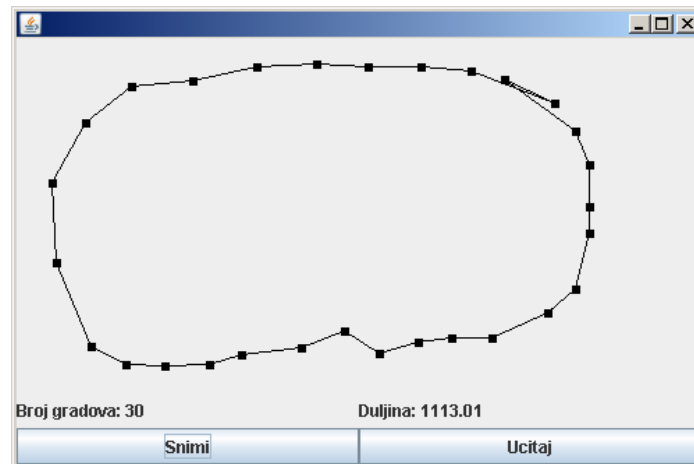


Algoritam radi s populacijom od  $m$  mrava, pri čemu u petlji ponavlja sljedeće. Svih  $m$  mrava pusti se da stvore rješenja, i ta se rješenja vrednuju (izračunaju se duljine puteva). Pri tome, svaki mrav prilikom izgradnje pamti do tada pređeni put, i kada treba birati u koji sljedeći čvor krenuti, automatski odbacuje čvorove kroz koje je već prošao. Tek kada je svih  $m$  mrava stvorilo prijedloge rješenja, odabire se  $n$  mrava koji će obaviti ažuriranje feromonskih tragova (pri tome  $n$  može biti čak i jednak  $m$ , što znači da svi mravi ažuriraju feromonske tragove, iako se je takav pristup pokazao kao loša praksa, pa je bolje pustiti samo bolji podskup ili čak samo najboljeg mrava da obavi ažuriranje). Potom se primjenjuje procedura isparavanja feromona, i postupak se ciklički ponavlja.

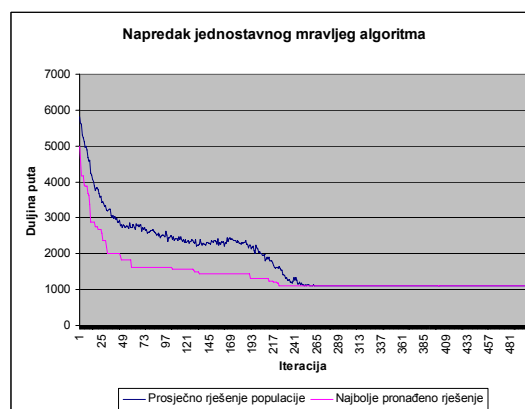
Kako bi se ilustrirao rad ovog algoritma, napisana je implementacija u Javi, i to na problemu trgovačkog putnika s 30 gradova (podsjetimo se – radi se o NP teškom problemu).

Rezultati su dobiveni uz sljedeće parametre:  $m=40$ ,  $\rho=0.2$ ,  $\alpha=1$ , maksimalni broj iteracija iznosi 500. Feromonski tragovi svih bridova izvorno su postavljeni na  $1/5000$ .

Sa slike se može uočiti da pronađeno rješenje nije idealno, ali je vrlo blizu optimalnog.



Slika 3-5 Rješenje TSP-a dobiveno jednostavnim mravljim algoritmom



Slika 3-6 Napredak jednostavnog mravljeg algoritma

Sada nakon što smo opisali kako primijeniti prikazane ideje, pogledajmo stvarne algoritme koji se danas koriste.

### 3.2. Algoritam Ant System

Algoritam Ant System predložili su Dorigo i suradnici [8,9,10]. Prilikom inicijalizacije grafa, na sve bridove deponira se količina feromona koja je nešto veća od očekivane količine koju će u jednoj iteraciji algoritma deponirati mravi. Koristi se izraz:

$$\tau_0 = \frac{m}{C^{mn}}$$



gdje je  $C^{mn}$  najkraća duljina puta pronađena nekim jednostavnim algoritmom (poput algoritma najbližeg susjeda). Ideja je zapravo dobiti kakvu-takvu procjenu duljine puta od koje će mravi dalje tražiti bolja rješenja, te ponuditi optimalnu početnu točku za rad algoritma. Prema [8], uz premali  $\tau_0$  pretraga će brzo biti usmjerena prema području koje su mravi slučajno odabrali u prvoj iteraciji, a uz preveliki  $\tau_0$  količine feromona koje mravi ostavljaju u svakoj iteraciji bit će premale da bi mogle usmjeravati pretragu, pa će se morati potrošiti puno iteracija kako bi mehanizam isparavanja uklonio višak feromona.

Rad algoritma započinje tako što  $m$  mrava stvara rješenja problema. U slučaju TSP-a, mravi se slučajno raspoređuju po gradovima iz kojih tada započinju konstrukciju

rješenja. Prilikom odlučivanja u koji grad krenuti, umjesto prethodno prikazanog pravila, mravi koriste slučajno proporcionalno pravilo (engl. *random proportional rule*) koje uključuje dvije komponente: jakost prethodno deponiranog feromonskog traga te vrijednost heurističke funkcije. Vjerojatnost prelaska iz grada  $i$  u grad  $j$  određena je izrazom:

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_{l \in N_i^k} (\tau_{il}^\alpha \cdot \eta_{il}^\beta)}, & \text{ako } j \in N_i^k \\ 0, & \text{ako } j \notin N_i^k \end{cases}$$



$\eta_{ij}$  je pri tome heuristička informacija koja govori koliko se čini da je dobro iz grada  $i$  otići u grad  $j$ . Ovo je tipično informacija koja je poznata unaprijed, i u slučaju problema trgovačkog putnika računa se kao  $\eta_{ij}=1/d_{ij}$ , gdje je  $d_{ij}$  udaljenost grada  $i$  od grada  $j$ . Parametri  $\alpha$  i  $\beta$  pri tome određuju ponašanje samog algoritma. Ako je  $\alpha=0$ , utjecaj feromonskog traga se poništava, i pretraživanje se vodi samo heurističkom informacijom. Ako je pak  $\beta=0$ , utjecaj heurističke informacije se poništava, i ostaje utjecaj isključivo feromonskog traga, što često dovodi do prebrze konvergencije suboptimalnom rješenju (gdje mravi slijede jedan drugoga po relativno lošoj stazi). Dobri rezultati postižu se uz  $\alpha \approx 1$  i  $\beta$  između 2 i 5.

Nakon što su svi mravi napravili rješenja, rješenja se vrednuju. Potom se pristupa isparavanju feromona sa svih bridova, prema izrazu:

$$\tau_{ij} \leftarrow \tau_{ij} \cdot (1 - \rho)$$



Nakon isparavanja, svi mravi deponiraju feromonski trag na bridove kojima su prošli, i to proporcionalno dobroti rješenja koje su pronašli:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^m \Delta \tau_{ij}^k$$



pri čemu je:

$$\Delta \tau_{ij}^k = \begin{cases} 1/C^k, & \text{ako je brid } i - j \text{ na stazi } k - \text{tog mrava} \\ 0, & \text{inačn} \end{cases}$$

Pseudokôd ovog algoritma prikazan je u okviru 3.2.

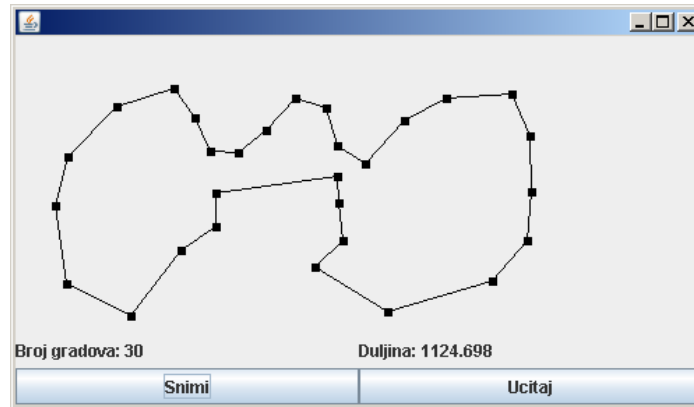
```
ponavljaj dok nije kraj
  ponovi za svakog mravca
    stvori rješenje
    vrednuj rješenje
  kraj ponovi
  ispari feromonske tragove
  ponovi za sve mrave
    azuriraj feromonske tragove
  kraj ponovi
kraj ponavljanja
```



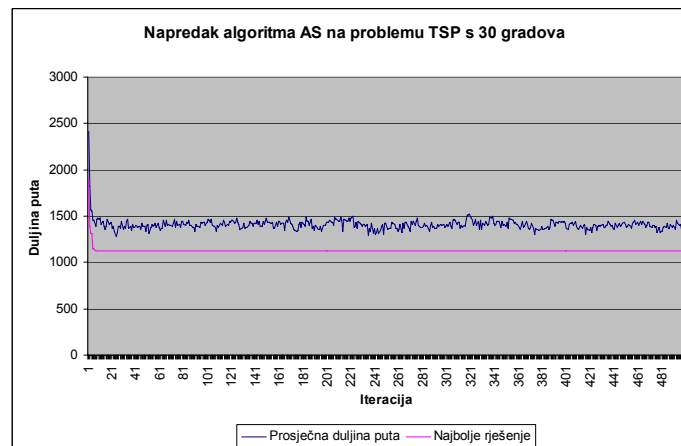
**Okvir 3.2 Pseudokod algoritma Ant System**

U svrhu ilustracije rada algoritma napravljen je program u programskom jeziku Java, i pokrenut nad problemom trgovačkog putnika s 30 gradova. Parametri algoritma su redom:  $m=30$ ,  $\alpha=1$ ,  $\beta=2$ ,  $\rho=0.5$ . Algoritam je zaustavljen nakon 500 iteracija. Rezultati su prikazani na slikama 3-7 i 3-8.





Slika 3-7 Rješenje problema TSP algoritmom AS



Slika 3-8 Napredak algoritma AS na problemu TSP s 30 gradova

Elitistička verzija algoritma (engl. *Elitist Ant System* – EAS) predložena je kao poboljšanje algoritma u [9,11]. Kod ove verzije dodatno se pamti globalno najbolje pronađeno rješenje. U svakoj iteraciji to se rješenje također koristi za ažuriranje feromonskih tragova s određenom težinom  $e$ . Ovo si možemo predočiti kao da postoji još jedan mrav (označimo ga s  $bs$ ) koji u svakoj iteraciji prođe upravo najboljim zapamćenim putem. Pravilo ažuriranja feromona tada dobiva još jedan član, pa glasi:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^m \Delta \tau_{ij}^k + e \cdot \Delta \tau_{ij}^{bs}$$

U [8] se kao vrijednost konstante  $e$  navodi upravo broj mrava  $m$ .

Prema [12], dodatno poboljšanje donosi algoritam rangirajućeg sustava mrava (engl. *Rank-based Ant System*). Kod ovog algoritma, nakon što svi mravi stvore rješenja, mravi se sortiraju prema kvaliteti rješenja. Ažuriranje feromona obavlja samo najboljih  $w-1$  mrava ( $i$  to proporcionalno svojem rangu), te mrav koji čuva globalno najbolje rješenje (koje ulazi s najvećim utjecajem):

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{w-1} (w-k) \Delta \tau_{ij}^k + w \cdot \Delta \tau_{ij}^{bs} .$$

Svaki mrav pri tome ažurira samo bridove koji su dio njegovog puta (za sve ostale bridove  $\Delta \tau_{ij}^k=0$ ).

*Max-Min Ant System (MMAS)* [13,14,15] još je jedno unaprjeđenje algoritma Ant System, koje uvodi 4 modifikacije:

1. Samo najbolji mrav smije obavljati ažuriranje feromona. Postoje varijante algoritma koje ažuriranje dopuštaju samo globalno najboljem mravu, varijante algoritma koje ažuriranje dopuštaju samo najboljem mravu u trenutnoj iteraciji, te varijante algoritma koje ažuriranje dopuštaju i globalno najboljem i najboljem u svakoj iteraciji, i to različitim intenzitetom.
2. Kako bi se spriječila prerana stagnacija algoritma zbog prethodne modifikacije, uvodi se gornja i donja granica na jakost feromonskog traga:  
$$\tau_{ij} \in [\tau_{\min}, \tau_{\max}]$$
.
3. Inicijalizacija algoritma feromonski trag na svim bridovima postavlja na njihovu maksimalnu vrijednost. Ovo zajedno s niskom stopom isparavanja (predlaže se  $\rho=0.02$ ) osigurava da mravi na početku obavljaju široko pretraživanje prostora rješenja.
4. Svaki puta kada algoritam dođe u stagnaciju, odnosno kada nema poboljšanja u kvaliteti rješenja unutar zadanog broja iteracija, obavlja se reinicijalizacija feromonskih tragova na njihove maksimalne vrijednosti.

Gornja granica pri tome se postavlja na vrijednost:

$$\tau_{\max} = \frac{1}{\rho \cdot C^{bs}}$$

i ažurira svaki puta kada se pronađe novo najbolje rješenje  $C^{bs}$ . Donja granica određena je parametrom  $a$  prema izrazu:

$$\tau_{\min} = \frac{\tau_{\max}}{a}$$

Pri svakoj promjeni gornje granice, algoritam automatski mijenja i donju granicu.

Danas postoji još niz drugih varijanti mravljih algoritama, a zainteresirani čitatelj se upućuje na [8].

### Pitanja za ponavljanje

- Opišite eksperiment dvokrakog mosta.
- Opišite jednostavni mravlji algoritam:
  - pseudokod
  - definirajte vjerojatnost odabira sljedećeg čvora
  - navedite i objasnite formulu za ažuriranje feromonskih tragova
  - navedite formulu koja opisuje isparavanje feromonskih tragova
- Opišite mravlji algoritam Ant System:
  - pseudokod
  - inicijalizacija
  - definirajte vjerojatnost odabira sljedećeg čvora
  - navedite formulu koja opisuje isparavanje feromonskih tragova
  - navedite i objasnite formulu za ažuriranje feromonskih tragova

- Opišite elitističku verziju mravljeg algoritma.
- Opišite rangirajuću verziju mravljeg algoritma.
- Opišite algoritam Max-Min Ant System. Koje su razlike u odnosu na mravlji algoritam Ant System?

Literatura

- [1] Goss, S., Aron, S., Deneubourg, J. L., & Pasteels, J. M. (1989). Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76, 579-581.
- [2] Deneubourg, J. L., Aron, S., Goss, S., & Pasteels, J. M. (1990). The self-organizing exploratory pattern of the Argentine ant. *Journal of Insect Behaviour*, 3, 159-168.
- [3] Bonabeau, E., Sobkowski, A., Theraulaz, G., & Deneubourg, J. L. (1997). Adaptive task allocation inspired by a model of division of labor in social insects. In D. Lundha, B. Olsson & A. Narayanan (Eds.), *Bio-Computation and Emergent Computing* (pp. 36-45). Singapore, World Scientific Publishing.
- [4] Bonabeau, E., Theraulaz, G., Deneubourg, J. L., Aron, S., & Camazine, S. (1997). Self-organization in social insects. *Tree*, 12(5), 188-193.
- [5] Camazine, S., Deneubourg, J. L., Franks, N. R., Sneyd, J., Theraulaz, G., & Bonabeau, E. (Eds.). (2001). *Self-organization in Biological Systems*, Princeton, NJ, Princeton University Press.
- [6] Haken, H. (1983). *Synergetics*. Berlin. Springer-Verlag.
- [7] Nicolis, G., & Prigogine, I. (1977). *Self-Organisation in Non-Equilibrium Systems*. New York, John Wiley & Sons.
- [8] Dorigo, M., & Stützle, T. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.
- [9] Dorigo, M., Maniezzo, V., & Colorni, A. (1991). Positive feedback as a search strategy. Technical report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan.
- [10] Colorni, A., Dorigo, M., & Maniezzo, V. (1992). Distributed optimization by ant colonies. In F. J. Varela & P. Bourguin (Eds.), *Proceedings of the First European Conference on Artificial Life* (pp. 134-142), Cambridge, MA, MIT Press.
- [11] Dorigo, M., Maniezzo, V., Colorni, A. (1996). Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 26(1), 29-41.
- [12] Bullnheimer, B., Hartl, R. F., & Strauss, C. (1999). A new rank-based version of the Ant System: A computational study. *Central European Journal for Operations Research and Economics.*, 7(1), 25-38.
- [13] Stützle, T., & Hoos, H.H. (1997). The Max-Min Ant System and local search for the traveling salesman problem. In T. Bäck, Z. Michalewicz, & X. Yao (Eds.), *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)* (pp. 309-314). Piscataway, NJ, IEEE Press.
- [14] Stützle, T., & Hoos, H.H. (2000). Max-Min Ant System. *Future Generation Computer Systems*, 16(8), 889-914.
- [15] Stützle, T., & Hoos, H.H. (1999). Max-Min Ant System and local search for combinatorial optimization problem. In S. Voss, S. Martello, I. Osmann, & C. Roucairol (Eds.), *Meta-Heuristics: Advances and Trend sin Local Search Paradigms for optimization* (pp. 137-154). Dordrecht, Netherlands, Kluwer Academic Publishers.

## 4. Algoritam roja čestica

Algoritam roja čestica (engl. Particle Swarm Optimization) otkriven je sasvim slučajno, pri pokušaju da se na računalu simulira kretanje jata ptica. C. W. Reynolds u svom radu [1] promatra jato ptica kao sustav čestica, gdje svaka čestica (tj. ptica) svoj let ravna prema sljedećim pravilima: (i) izbjegavanje kolizija s bliskim pticama, (ii) usklađivanje brzine leta s bliskim pticama te (iii) pokušaj ostanka u blizini drugih ptica. Inspiriran ovim i sličnim radovima, R. C. Eberhart i J. Kennedy shvaćaju da se takav sustav može koristiti kao optimizator, te svoje ideje objavljuju 1995. godine u dva temeljna rada [2, 3]. Eberhart, Simpson i Dobbins 1996. godine izdaju knjigu [4] o uporabi algoritma roja čestica kao univerzalnog optimizacijskog alata. Tako je, primjerice, u knjizi opisana vrlo uspješna primjena algoritma roja česta za treniranje umjetne neuronske (točnije, unaprijedne umjetne neuronske mreže ili višeslojnog perceptrona), gdje se algoritam koristi kao zamjena algoritma Backpropagation. Konačno, 1997. godine Eberhart i Kennedy objavljuju rad o prilagodbi algoritma za rad nad diskretnim domenama [5]. Sam algoritam u određenoj mjeri inspiriran je i sociološkim interakcijama između pojedinaca u populaciji, gdje svaki pojedinac pamti svoje do tada pronađeno najbolje rješenje problema, te ima uvid u najbolje pronađeno rješenje svojih susjeda, te pretraživanje usmjerava uzimajući u obzir obje komponente.



Slika 4-1 R. C. Eberhart

### 4.1. Opis algoritma

Algoritam roja čestica je populacijski algoritam. Populacija se sastoji od niza jedinki (čestica) koje lete kroz višedimenzijски prostor koji pretražuju, i pri tome svoj položaj mijenjaju temeljem vlastitog iskustva, te iskustva bliskih susjeda (čime se modeliraju socijalne interakcije između jedinki). Prilikom određivanja smjera kretanja, svaka jedinka u određenoj mjeri uzima u obzir svoje do tada pronađeno najbolje rješenje (individualni faktor), te najbolje pronađeno rješenje svoje bliske okoline (socijalni faktor). Utjecaj koji svaka od ovih komponenti ima uvelike određuje ponašanje same jedinke: radi li istraživanje prostora stanja (ukoliko je dominantni individualni faktor) ili fino podešavanje pronađenog rješenja (ukoliko je dominantni socijalni faktor). Na ovaj način sam algoritam kombinira globalno pretraživanje prostora stanja te lokalnu pretragu kojom se obavlja fino podešavanje rješenja.

Pseudokod algoritma prikazan je u okviru 4.1. Algoritam koristi populaciju veličine  $VEL\_POP$ , te pretražuje  $DIM$ -dimenzijski prostor rješenja. Pri tome  $x$  sadrži trenutne pozicije svih čestica a  $v$  njihove brzine. Polja  $x$  i  $v$  su dvodimenzijska: imaju onoliko redaka koliko ima čestica, te onoliko stupaca koliko rješenje ima dimenzija. Čestice pretražuju ograničeni prostor rješenja, a granice se čuvaju u poljima  $xmin$  i  $xmax$ . Brzina svake čestice (te u svakoj dimenziji) ograničena je svojom minimalnom i maksimalnom vrijednosti (primjerice, od -5 do +5), što čuvaju polja  $vmin$  i  $vmax$ .

Algoritam započinje inicijalizacijom populacije. Svaka se čestica smješta na neku slučajno odabranu poziciju, i dodjeljuje joj se neka slučajno odabrana brzina.

```

// inicijaliziraj_populaciju:
za i = 1 do VEL_POP
    za d iz 1 do DIM
        x[i][d] = random(xmin[d], xmax[d])
        v[i][d] = random(vmin[d], vmax[d])
    kraj
kraj

ponavljaj dok nije kraj

// evaluiraj_populaciju:
za i = 1 do VEL_POP
    f[i] = funkcija(x[i]);
kraj

// ima li čestica svoje bolje rješenje?
za i = 1 do VEL_POP
    ako je f[i] bolji od pbest_f[i] tada
        pbest_f[i] = f[i]
        pbest[i] = x[i]
    kraj
kraj

// ima li čestica globano najbolje rješenje?
za i = 1 do VEL_POP
    ako je f[i] bolji od gbest_f[i] tada
        gbest_f[i] = f[i]
        gbest[i] = x[i]
    kraj
kraj

// ažuriraj brzinu i poziciju čestice
za i = 1 do VEL_POP
    za d iz 1 do DIM
        v[i][d] = v[i][d] + c1*rand()*(pbest[i][d]-x[i][d])
        + c2*rand()*(gbest[d]-x[i][d])
        v[i][d] = iz_opsega(v[i][d], vmin[d], vmax[d])
        x[i][d] = x[i][d] + v[i][d]
    kraj
kraj
kraj
    
```

**Okvir 4.1 Pseudokod algoritma roja čestica**

Slijedi glavni dio algoritma koji se ponavlja tako dugo dok se ne ispuni uvjet zaustavljanja (pronalazak dovoljno dobrog rješenja ili dostizanje maksimalnog broja iteracija).

- Za svaku se česticu izračuna vrijednost funkcije u točki koju čestica predstavlja.
- Za svaku se česticu provjeri njeno do tada zapamćeno najbolje rješenje i njeno novo pronađeno rješenje. Ako je novo bolje, pamti se kao novo najbolje rješenje te čestice. U pseudokodu ovo se pamti u dvodimenzijском polju *pbest* (engl. *particles best solution*). Potrebno je pamtit i rješenje i vrijednost funkcije u tom rješenju.
- U čitavoj populaciji se pronađe najbolje rješenje, i ako je prethodno zapamćeno globalno rješenje lošije, ažurira se na novo pronađeno. U pseudokodu ovo se pamti u polju *gbest* (engl. *global best solution*). Potrebno je pamtit i rješenje i vrijednost funkcije u tom rješenju.

- Za svaku česticu se obavlja ažuriranje trenutne brzine a potom i položaja. Najprije se obavi ažuriranje brzine tako da se na trenutnu brzinu doda individualna komponenta modulirana faktorom individualnosti ( $c_1$ ) i slučajnim brojem iz intervala  $[0, 1]$  te socijalna komponenta modulirana faktorom socijalnosti ( $c_2$ ) i slučajnim brojem iz intervala  $[0, 1]$ . Potom se provjeri je li brzina izvan dozvoljenog raspona, i ako je, korigira se. Konačno, u skladu s novom brzinom ažurira se položaj čestice.

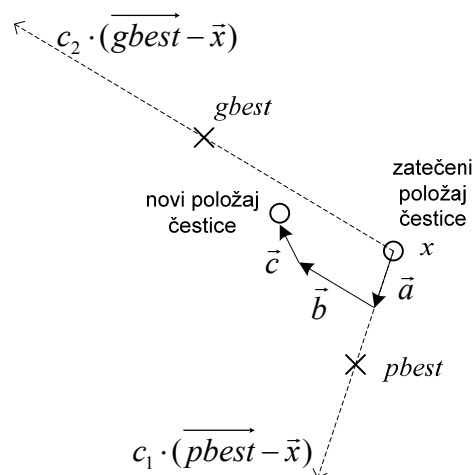
Kao što vidimo iz opisanoga, ažuriranje se obavlja prema sljedećim izrazima:

$$v_{i,d} = v_{i,d} + c_1 \cdot \text{rand}() \cdot (pbest_{i,d} - x) + c_2 \cdot \text{rand}() \cdot (gbest_d - x)$$



$$x_{i,d} = x_{i,d} + v_{i,d}$$

Faktori  $c_1$  i  $c_2$  uobičajeno se postavljaju na 2,0. Veća vrijednost faktora  $c_1$  omogućit će veći stupanj individualnosti jedinke, i time poticati istraživanje prostora, dok veća vrijednost faktora  $c_2$  jači naglasak stavlja na najbolje rješenje koje je čitav kolektiv do tada pronašao, i time osigurava detaljnije istraživanje okoline tog rješenja. Slika 4-2 jednostavnim vektorskim prikazom ilustrira "sile" koje djeluju na kretanje čestice, i pretpostavlja da se sve dimenzije vektora razlike  $pbest$  i  $x$  množe istim slučajno generiranim brojem (ista je pretpostavka i za vektor razlike  $gbest$  i  $x$ ). Također, slika je nacrtana uz  $c_1=2,0$  i  $c_2=2,0$ .



Slika 4-2 Grafički prikaz pomaka čestice

Prema formuli za ažuriranje brzine, nova brzina rezultat je triju komponenti: vektora  $a$ ,  $b$  i  $c$ :

$$\vec{a} = c_1 \cdot \text{rand}() \cdot (\overrightarrow{pbest - x})$$

$$\vec{b} = c_2 \cdot \text{rand}() \cdot (\overrightarrow{gbest - x})$$

$$\vec{c} = \vec{v}$$

Vektor  $a$  predstavlja pomak prema najboljem rješenju koje je pronašla sama jedinka, odgovarajuće skaliranom. Vektor  $b$  predstavlja pomak prema najboljem rješenju kolektiva, također odgovarajuće skaliranom. Vektor  $c$  poprima staru trenutnu vrijednost brzine, i zapravo predstavlja inerciju same čestice. Rezultantna brzina suma je sva tri djelovanja: čestica se po inerciji dalje nastavlja gibati, i pri tome brzinu djelomično modificira uslijed privlačenja svojeg i kolektivnog najboljeg rješenja.

## 4.2. Utjecaj parametara i modifikacije algoritma

Da bismo pokrenuli opisani algoritam, nužno je odrediti vrijednosti svih parametara. Pogledajmo kakav je njihov utjecaj na rad algoritma.

Ograničenje brzine nužno je jer bez njega algoritam može doći u divergentno stanje. Stavimo li ograničenje brzine preveliko, čestica može preletiti preko područja dobrih rješenja. Stavimo li pak ograničenje brzine na premalu vrijednost, može se dogoditi situacija da čestica ostane zatočena u lokalnim optimumima – kako je brzina premala, čestica se više ne može osloboditi utjecaju lokalnog optimuma. Prema [7],  $v_{max}$  se tipično stavlja na 10% do 20% raspona prostora koji se pretražuje.

Konstante  $c_1$  i  $c_2$  modeliraju jakost privlačne sile između najboljih rješenja i čestice – što veći broj, to je veća privlačna sila pa će čestica moći manje istraživati.

Veličina populacije tipično se kreće od 20 do 50. Naravno, postoje i problemi kod kojeg se do zadovoljavajućeg rješenja dolazi tek s većim populacijama.

### 4.2.1. Dodavanje faktora inercije

Proces pretraživanja prostora uobičajeno se podijeliti u dva koraka. U prvom korak obavlja se grubo pretraživanje kako bi se locirala "zanimljiva" područja, a potom se u drugom koraku obavlja fino pretraživanje unutar lociranih kandidata. Kako bi se osiguralo ovakvo ponašanje, izraz za ažuriranje brzine modificira se na sljedeći način:

$$v_{i,d} = w(t) \cdot v_{i,d} + c_1 \cdot rand() \cdot (pbest_{i,d} - x) + c_2 \cdot rand() \cdot (gbest_d - x)$$

Inercijska komponenta brzine množi se vremenski promjenjivim faktorom  $w$ . Inicijalno,  $w$  se postavlja na vrijednost blizu 1 (primjerice, 0.9), a s povećanjem broja iteracija  $t$  vrijednost se smanjuje prema nekoj minimalnoj vrijednosti. Ako s  $w_{max}$  i  $w_{min}$  označimo željenu maksimalnu i minimalnu vrijednost, te s  $T$  označimo iteraciju u kojoj težinski faktor treba pasti na  $w_{min}$ , za ažuriranje možemo koristiti sljedeći izraz:

$$w(t) = \begin{cases} \frac{t}{T} \cdot (w_{min} - w_{max}) + w_{max}, & t \leq T \\ w_{min}, & t > T \end{cases}$$

Ažuriranje pozicije čestice radi se na uobičajeni način.

### 4.2.2. Stabilnost algoritma

Već smo spomenuli da je jedan od načina da se pokuša osigurati stabilnost algoritma (u smislu da se spriječi divergencija) ograničavanje iznosa brzine. Matematička analiza samog algoritma (vidi [8]) pokazuje da se stabilnost može postići ako se izraz za ažuriranje brzine modificira dodavanjem faktora ograničenja (engl. *constriction factor*)  $K$ :

$$v_{i,d} = K \cdot [v_{i,d} + c_1 \cdot rand() \cdot (pbest_{i,d} - x) + c_2 \cdot rand() \cdot (gbest_d - x)]$$

gdje je  $K$  funkcija parametara  $c_1$  i  $c_2$  i definiran je prema izrazu:

$$K = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}$$

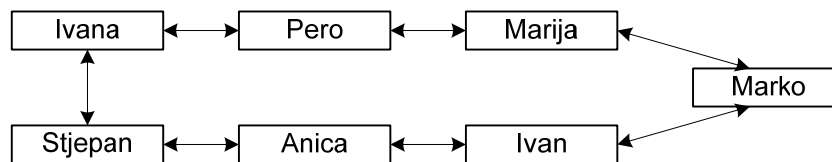
Pri tome je  $\varphi = c_1 + c_2$ ,  $\varphi > 4$ . Prema [7],  $\varphi$  se tipično postavlja na 4,1 ( $c_1=2,05$ ,  $c_2=2,05$ ), što daje za  $K$  vrijednost 0,729.



### 4.2.3. Lokalno susjedstvo

Kod algoritma roja čestica opisanog u okviru 4.1 svaka čestica osjeća utjecaj dviju sila. Prva sila je vlastito iskustvo, odnosno najbolje rješenje koje je sama čestica pronašla. Druga sila je iskustvo kolektiva, odnosno najbolje rješenje koje je pronašao čitav kolektiv. Ovakva vrsta algoritma roja čestica još se naziva potpuno-informirani algoritam roja čestica (engl. Fully informed PSO). Nezgodno svojstvo ove inačice algoritma jest mogućnost prerane konvergencije – čim jedna jedinka nađe potencijalno dobro rješenje, sve su jedinke automatski privučene k tom rješenju, što može spriječiti temeljito istraživanje prostora stanja i pronalazak eventualno još boljih rješenja.

Kako bi se tomu doskočilo, razvijena je inačica algoritma kod koje jedinkama nije dostupna informacija o globalno najboljem rješenju. Umjesto toga, uveden je topološki uređaj u populaciju. Za svaku se jedinku zna tko su joj susjedi. Poslužimo se za trenutak ilustrativnim primjerom. Neka se naša populacija sastoji od sljedećih jedinki: Ivana, Pero, Marija, Marko, Ivan, Anica i Stjepan. Susjedstvo se definira preko poznanstava: Ivana zna Peru i Stjepana, pa su Pero i Stjepan njezini susjedi, i Ivana će komunicirati samo s njima. Pero zna Ivanu i Mariju, pa su Ivana i Marija njegovi susjedi, i Pero će komunicirati samo s njima. Slično možemo napraviti i za ostale jedinke (vidi sliku 4-3; susjedi su osobe direktno povezane strelicama).



Slika 4-3 Primjer definiranog susjedstva

Prilikom pretraživanja prostora stanja, kada jedinki zatreba kolektivno najbolje rješenje, jedinka će to rješenje izračunati tako da pogleda svoje najbolje rješenje i najbolja rješenja svojih susjeda – globalna informacija jedinki nije dostupna. Primjerice, kada Ivana treba utvrditi kamo dalje, za utvrđivanje najboljeg rješenja kolektiva pogledat će svoje najbolje rješenje, najbolje rješenje Pere i najbolje rješenje Stjepana. Ovdje je važno uočiti da se susjedstvo ne definira prema blizini u prostoru rješenja. Prilikom pretraživanja, Ivana i Pero mogu se udaljiti, i Ivani Marija može doći i puno bliža no što je Pero; međutim, Marija time neće postati susjeda Ivani.

Kada pišemo konkretnu implementaciju ove vrste algoritma roja čestica, moramo se odlučiti na koji ćemo način definirati susjedstvo, jer to nije jednoznačno. Jedan od čestih načina jest da se definira parametar  $ns$  (veličina susjedstva), a čestice slože u niz. Ako je  $ns=1$ , svaka je čestica isključivo susjed sama sebi. Ako je  $ns=2$ , susjedi od čestica( $i$ ) su čestica( $i-1$ ), čestica( $i$ ) te čestica( $i+1$ ). Ako je  $ns=3$ , susjedi od čestica( $i$ ) su sve čestice od čestica( $i-2$ ) do čestica( $i+2$ ), itd. U tom slučaju, kolektivno najbolje rješenje označavamo s  $lbest$  (engl. local best). Izraz za ažuriranje brzine tada umjesto  $gbest$  koristi  $lbest(i)$ , gdje je  $lbest(i)$  najbolje rješenje susjedstva  $i$ -te čestice:

$$v_{i,d} = w(t) \cdot v_{i,d} + c_1 \cdot rand() \cdot (pbest_{i,d} - x) + c_2 \cdot rand() \cdot (lbest_{i,d} - x)$$



Prema [7], prikladne veličine susjedstva su oko 15% ukupne veličine populacije. Ako s  $VEL\_SUS$  označimo ukupan broj čestica koje čine susjedstvo, vrijedi:

$$VEL\_SUS = 1 + 2 \cdot ns.$$

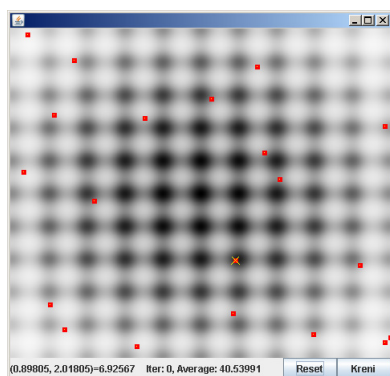
Za više informacija na ovu temu čitatelj se upućuje na nedavno objavljeni rad [9].

### 4.3. Primjer rada algoritma

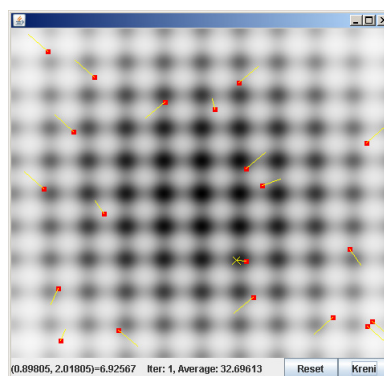
Primjer u nastavku prikazuje optimizaciju funkcije:

$$f(x_1, x_2, \dots, x_n) = 10 \cdot n + \sum_{i=1}^n (x_i^2 - 10 \cdot \cos(2\pi \cdot x_i)).$$

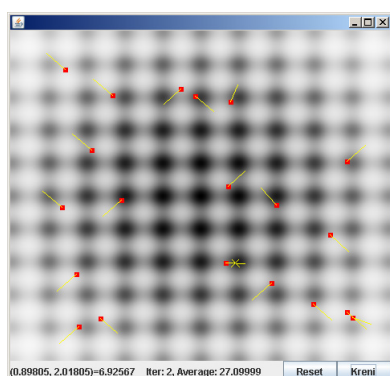
Konkretno, tražimo vektor  $x$  za koji funkcija poprima minimalnu vrijednost. Analitički, rješenje već znamo: to je ishodište, u kojem je vrijednost funkcije 0. Ova funkcija uzeta je stoga što ima niz lokalnih optimuma koji postupak pretraživanja globalnog optimuma bitno otežavaju.



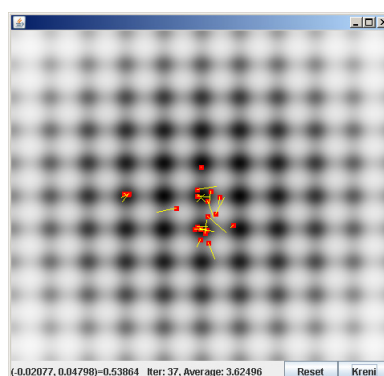
a) Početno stanje



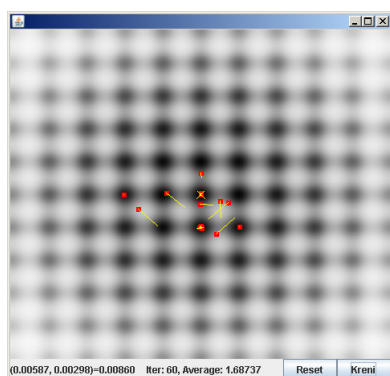
b) Nakon prvog koraka



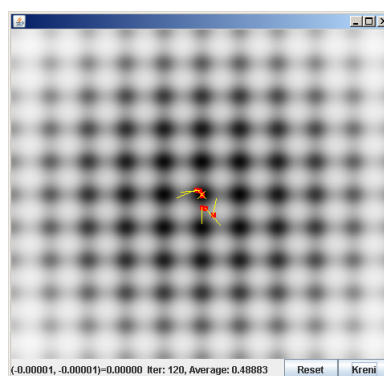
c) Nakon drugog koraka



d) Nakon 37. koraka



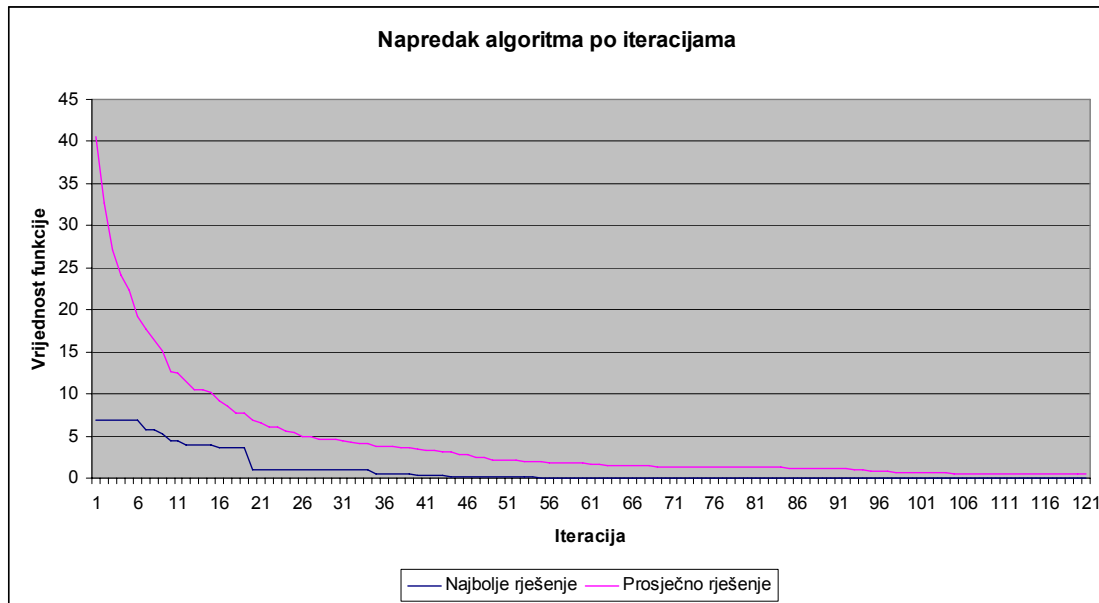
e) Nakon 60. koraka



f) Nakon 120 koraka

Slika 4-4 Prikaz rada algoritma roja čestica

U nastavku je prikazano i najbolje te prosječno rješenje ovisno o iteraciji.



Slika 4-5 Ovisnost najboljeg i prosječnog rješenja o iteraciji kod algoritma roja čestica

### Pitanja za ponavljanje

- Napišite pseudokod algoritma roja čestica.
- Navedite i objasnite formulu za ažuriranje brzine čestice.
- Kako ažuriramo poziciju čestice?
- Objasnite faktor inercije kod algoritma roja čestica.
- Objasnite kako se radi kontrola stabilnosti algoritma roja čestica.
- Kako kod algoritma roja čestica možemo definirati susjedstvo, i zašto to radimo?

## Literatura

- [1] Reynolds, C. W. (1987). Flocks, herds and schools: a distributed behavioral model. *Computer Graphics*, 21(4):25-34.
- [2] Kennedy, J. and Eberhart, R. C. (1995) Particle swarm optimization. In *Proceedings of the IEEE 1995 International Conference on Neural Networks*, pp. 1942-1948. (<http://www.engr.iupui.edu/~shi/Coference/psopap4.html>)
- [3] Eberhart, R. C. and Kennedy, J. (1995) A new optimizer using particle swarm theory. *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, pp. 39-43. Piscataway, NJ: IEEE Service Center.
- [4] Eberhart, R. C., Simpson, P. K., and Dobbins, R. W. (1996). *Computational Intelligence PC Tools*. Boston, MA: Academic Press Professional.
- [5] Kennedy, J. and Eberhart, R. C. (1997) A discrete binary version of the particle swarm algorithm. In *Proceedings of the IEEE 1997 International Conference on Systems, Man and Cybernetics*, pp. 4104-4109.
- [6] Swarm Intelligence, početna web stranica algoritma PSO <http://www.swarmintelligence.org/>
- [7] Eberhart, R. C., and Shi, Y. (2001)(b). Particle swarm optimization: developments, applications and resources. *Proc. Congress on Evolutionary Computation 2001*, Seoul, Korea. Piscataway, NJ: IEEE Service Center.
- [8] Clerc, M. (1999). The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. *Proc. 1999 Congress on Evolutionary Computation*, Washington, DC, pp. 1951-1957. Piscataway, NJ: IEEE Service Center.
- [9] Montes de Oca, M. A., Stützle, T., Birattari, M. and Dorigo M. (2009). Frankenstein's PSO: A Composite Particle Swarm Optimization Algorithm, *IEEE Trans. on Evolutionary Computation*. 13(5):pp. 1120-1132.

## 5. Umjetni imunološki sustav

Darwinova teorija o razvoju vrsta poslužila je kao inspiracija za jednu veliku porodicu algoritama; spomenimo, primjerice, samo genetske algoritme s kojima smo se već upoznali u poglavlju 2. Temeljna premisa ove teorije jest seksualno razmnožavanje kod kojega nove jedinke dobivaju genetski materijal od oba roditelja i dodatno bivaju mutirane. Pod utjecajem okoline, bolje i prilagođenije jedinke imaju veću šansu za daljnje razmnožavanje, što u konačnici dovodi do izumiranja lošijih jedinki, i postupnom prilagođavanju vrste okolini u kojoj živi. Ovi principi iskorišteni su za izgradnju genetskog algoritma koji proces evolucije simulira uporabom dva genetska operatora: križanjem koje temeljem genetskog materijala dvaju roditelja stvara novog potomka, te mutacijom koja u genetski kôd djeteta unosi slučajne izmjene.


Međutim, pogledamo li sada malo dublje u jednog pojedinca, otkrit ćemo također nevjerojatan mehanizam koji, primjerice, čovjeku omogućava preživljavanje – njegov imunološki sustav! Prilikom rođenja, čovjekov imunološki sustav već raspolaže određenim mehanizmima za borbu protiv poznatih antigena (napadača). Primjerice, ukoliko u tijelo uđu antigeni, tijelo raspolaže posebnom vrstom stanica koje su okružene antigenom i potom ga unište. Ovaj mehanizam štiti nas od velikog broja bolesti, no nažalost, ne svih.

Posebnost imunološkog sustava jest sposobnost prilagodbe i sposobnost učenja, što nam omogućava obranu od bolesti kojima prethodno nismo bili izloženi, te stjecanje imuniteta. Ukoliko urođeni imunološki sustav ne prepozna i ne uništi antigene, aktivirat će se drugi dio imunološkog sustava koji je zadužen za reakciju na specifične antigene (ovo je proces koji može potrajati i nekoliko dana). Što se tu zapravo događa? Za obranu od antigena u tijelu su zadužene B-stanice koje luče antitijela (B-stanice proizvode se u koštanoj srži). Ulaskom antigena u tijelo dio B-stanica započet će lučenje antitijela. Antitijela imaju receptore kojima se mogu povezati s antigenom, a do ovog povezivanja će doći ukoliko su antitijelo i receptor kompatibilni. Mjeru ove kompatibilnosti zvat ćemo afinitetom. Povezivanje antigena s antitijelom stimulira B-stanicu koja je proizvela antitijelo i B-stanica započinje proces dijeljenja (mitoza), čime nastaje velika količina klonova B-stanice koji sazrijevaju i luče nove količine antitijela kao odgovor na nastalu infekciju. Prilikom procesa kloniranja B-stanice dolazi do nasumičnih mutacija u genetskom kodu B-stanice. Uslijed ovih mutacija dio nastalih klonova proizvodit će antitijela koja će imati još veći afinitet prema antigenu, što će pak uzrokovati novi ciklus kloniranja tih stanica. Nakon nekog vremena, na ovaj način u tijelu će se razviti B-stanice koje mogu vrlo efikasno odgovoriti na specifični antigen koji je uzrokovao infekciju. Uništenjem antigena, međutim, dio B-stanica ostat će u tijelu, i ukoliko se nakon nekog vremena opet nađe na isti antigen (ponovni razvoj infekcije), odgovor imunološkog sustava sada će biti daleko jači i efikasniji no što je to bilo prvog puta. Opetovanim izlaganjem istim antigenima, tijelo će u konačnici razviti brz i efikasan odgovor na taj antigen, i mi time postajemo imuni na tu bolest. Zanimljivo je uočiti da je čitav postupak usavršavanja B-stanica vođen isključivo slijepim nasumičnim mutacijama receptora.


Važno je napomenuti da prethodni opis predstavlja samo grupu pojednostavljenije složenih interakcija koje se odvijaju u imunološkom sustavu – no i to je dovoljno kako bismo opisali osnovne algoritme. Teoriju koja objašnjava što kako radi imunološki sustav razvio je Burnet [1, 2, 3], počev još davne 1957. godine. Područje umjetnih imunoloških sustava (engl. AIS – Artificial Immune Systems) bavi se

razvojem modela i apstrakcija imunološkog sustava i njihovom primjenom u algoritmima za rješavanje problema u znanosti i inženjerstvu [4]. Mi ćemo se ovdje fokusirati na *algoritme klonske selekcije* (engl. *CSA – Clonal Selection Algorithms*). To su algoritmi koji pripadaju razredu *imunoloških algoritama* (engl. *IA – Immunological Algorithms*), za čiji je razvoj iskorišten prethodno opisani *princip klonske selekcije* (engl. *Clonal Selection Principle*) [5,6]. Važno je napomenuti da zbog jednostavnosti, većina algoritama ne uvodi distinkciju između pojmova B-stanica i antitijelo, te ta dva pojma tretira kao jedan.

### 5.1. Jednostavni imunološki algoritam

Jednostavni imunološki algoritam (engl. *SIA – Simple Immunological Algorithm*) autora Cutello i Nicosia razvijen je 2002. godine, i opisan u [7,8,5]. Algoritam može služiti za izradu klasifikatorskih sustava te za optimizaciju. Ovdje ćemo ga opisati s aspekta optimizacijskog algoritma. 

SIA je populacijski algoritam koji radi s populacijom antitijela. Pri tome je svako antitijelo zapravo jedno rješenje problema koji se optimira. Antigen u ovom kontekstu predstavlja samu funkciju čiji se optimum traži. Afinitet pojedinog antitijela (rješenja) prema antigenu (funkciji) tada je predstavljen kvalitetom (tj. dobrotom; engl. *fitness*) samog rješenja. Ukoliko se radi o maksimiziranju funkcije, tada je afinitet najčešće jednak upravo iznosu same funkcije u promatranom rješenju.

Izvorni opis algoritma za kodiranje rješenja koristi binarne nizove duljine  $l$  bitova (direktna analogija je binarno kodiranje kromosoma kod genetskog algoritma); međutim, postupak je proširiv na bilo kakvu reprezentaciju rješenja. Pseudokôd samog algoritma prikazan je u okviru 5.1. 

```
SIA(Ag, l, d, dup)
t = 0
inicijaliziraj  $P^{(0)} = \{x_1, x_2, \dots, x_d\}$ 
evaluiraaj ( $P^{(0)}$ , Ag)
ponavljaj dok nije zaustavi ( $P^{(t)}$ )
   $P^{cl}$  = kloniraj ( $P^{(t)}$ , dup)
   $P^{hyp}$  = hipermutiraj ( $P^{cl}$ )
  evaluiraaj ( $P^{hyp}$ , Ag)
   $P^{(t+1)}$  = odaberi ( $P^{hyp} \cup P^{(t)}$ , d)
  t = t+1
kraj ponavljanja
kraj
```

**Okvir 5.1 Pseudokod jednostavnog imunološkog algoritma**

Parametri algoritma su:

- $Ag$  – funkcija koju se optimira,
- $l$  – broj bitova rješenja
- $d$  – veličina populacije rješenja
- $dup$  – broj klonova svakog rješenja

Algoritam započinje stvaranjem inicijalne populacije antitijela (rješenja)  $P^{(0)}$  veličine  $d$ . U slučaju da se rješenja prikazuju binarno, ovo znači slučajno stvaranje  $d$  sljedova nula i jedinica, svaki duljine  $l$ . Potom se računa afinitet svakog antitijela (tj. rješenja se vrednuju).

Potom se ulazi u petlju koja se ponavlja tako dugo dok uvjet zaustavljanja nije zadovoljen. To, primjerice, može biti pronalazak dovoljno dobrog rješenja ili prekoračenje maksimalnog broja iteracija.

U petlji se radi sljedeće. Svako antitijelo klonira se  $dup$  puta. Ovime iz populacije  $P^{(t)}$ , gdje je  $t$  oznaka iteracije, nastaje populacija klonova  $P^{clo}$  veličine  $d \cdot dup$ . Potom svako antitijelo iz populacije  $P^{clo}$  prolazi kroz proces hipermutacije, čime nastaje nova populacija  $P^{hyp}$  iste veličine. Operator hipermutacije predstavlja nasumičnu promjenu receptora antitijela u pokušaju da se antitijelo bolje prilagodi povezivanju s antigenom. Kod ovog algoritma hipermutacija nasumično mijenja jedan bit na nasumično odabranoj poziciji (ili u slučaju nekog drugog načina predstavljanja rješenja obavlja jednu jednostavnu promjenu). Potom se za sve jedinice iz populacije  $P^{hyp}$  računaju afiniteti (vrednuju se nastala rješenja). Na kraju se iz unije populacija  $P^{(t)}$  i  $P^{hyp}$  izabire  $d$  antitijela najvećeg afiniteta koji postaju nova populacija  $P^{(t+1)}$  za sljedeći prolaz kroz petlju. Uočite kako je zbog ovoga algoritam automatski elitistički: ako su hipermutacijom nastala samo gora rješenja od trenutno najboljeg iz  $P^{(t)}$ , to najbolje će biti preneseno u novu populaciju i time očuvano.

Analiza složenosti samog algoritma može se pogledati u [5].

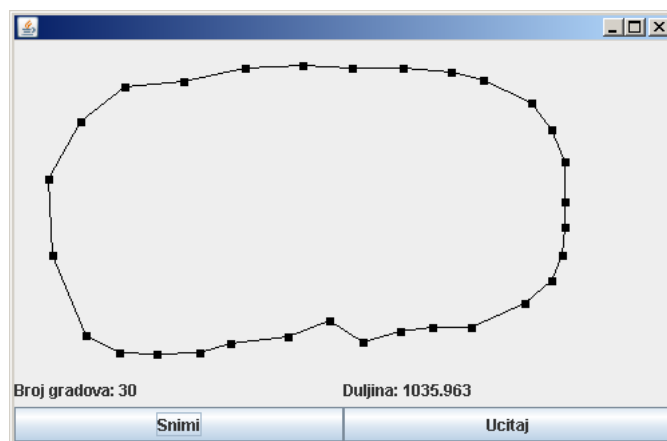
Kako bismo demonstrirali rad algoritma na nekom malo složenijem problemu, napravljena je implementacija u programskom jeziku Java koja uporabom ovog algoritma rješava problem trgovačkog putnika s 30 gradova.

Antitijelo je pri tome predstavljeno kao vektor od 30 elemenata. Element na  $i$ -tom mjestu je cijeli broj koji predstavlja indeks grada koji trgovački putnik treba posjetiti u  $i$ -tom koraku. Operator hipermutacije izveden je tako da nasumično odabere dva koraka i zamijeni gradove koje trgovački putnik posjećuje u tim koracima.

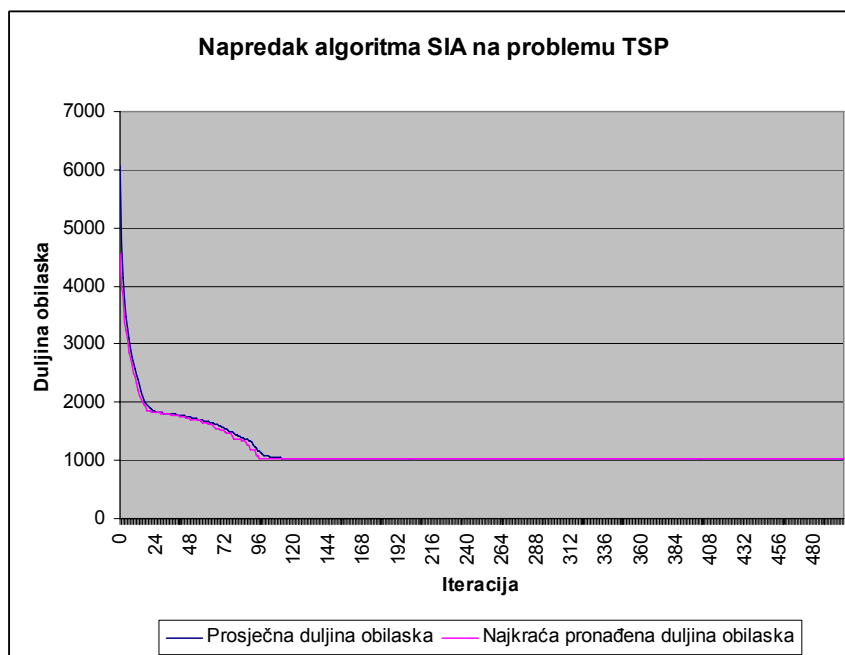
Program je pokrenut uz sljedeće parametre:

- broj antitijela u populaciji: 50
- broj klonova za svaku stanicu: 10
- maksimalni broj iteracija: 500

Na testnom primjeru (vidi sliku 5-1) teorijski najkraći put iznosi 1035,963 km, što je algoritam i pronašao. Kretanje prosječnog rješenja populacije kao najboljeg rješenja ovisno o broju iteracija algoritma prikazuje slika 5-2.



Slika 5-1 Problem obilaska 30 gradova riješen algoritmom SIA



Slika 5-2 Napredak algoritma SIA na problemu TSP s 30 gradova

U nastavku slijedi opis algoritma CLONALG.

## 5.2. Algoritam CLONALG

CLONALG je kratica od *Clonal Selection Algorithm*. Ovaj algoritam predložili su 1999. i kasnije razradili (2000., 2002.) De Castro i Von Zuben [9,10,11].

Pseudokôd algoritma prikazan je u okviru 5.2.

```

CLONALG (Ag, n, d, β)
t = 0
inicijaliziraj P(0) = {x1, x2, ..., xd}
ponavljaj dok nije zaustavi(P(t))
  evaluiraj (P(t), Ag)
  P(t) = odaberi (P(t), n)
  Pclo = kloniraj (P(t), β)
  Phyp = hipermutiraj (Pclo)
  evaluiraj (Phyp, Ag)
  P' = odaberi (Phyp, n)
  Pbirth = stvoriNove (d)
  P(t+1) = zamijeni (P', Pbirth)
  t = t+1
kraj ponavljanja
evaluiraj (P(t), Ag)
kraj

```

Okvir 5.2 Pseudokod algoritma CLONALG

Ideja algoritma je slična kao i kod SIA, pri čemu se antitijela podvrgavaju postupku kloniranja proporcionalno njihovom afinitetu, te postupku hipermutacije obrnuto proporcionalno njihovom afinitetu. Antitijela su sada nizovi od  $l$  elemenata, pri čemu su elementi realni brojevi pa se svako antitijelo može promatrati kao točka u  $l$ -dimenzijском prostoru.

Parametri algoritma su sljedeći:

- $Ag$  – antigen, funkcija koju optimiramo



- $n$  – broj antitijela u populaciji,
- $d$  – broj novih jedinki koje ćemo u svakom koraku slučajno stvoriti i dodati u populaciju, te
- $\beta$  – parametar koji određuje veličinu populacije klonova.

Algoritam započinje stvaranjem početne populacije antitijela  $P^{(0)}$ . Antitijela se stvaraju nekim slučajnim mehanizmom. Potom se ulazi u petlju koja se ponavlja dok uvjet zaustavljanja nije zadovoljen (autori izvorno kao uvjet zaustavljanja jedino navode fiksni broj iteracija).

U petlji se najprije računa afinitet svih antitijela obzirom na antigen (vrednuju se rješenja obzirom na funkciju koja se optimira). Potom se operatorom *odaberi* odabiru antitijela koja će se klonirati. Izvorno, algoritam odabire svih  $n$  antitijela (pa u tom slučaju ovo naprosto možemo preskočiti), no dozvoljava se i odabir manjeg broja.

Pristupa se procesu kloniranja odabranih antitijela (stvara se populacija  $P^{clo}$ ). Pri tome je broj klonova pojedinog antitijela direktno proporcionalan afinitetu tog antitijela: što je afinitet veći, to je broj klonova veći. Ukupni broj klonova koji će ući u populaciju klonova  $P^{clo}$  označen je s  $N_C$  i određen parametrom  $\beta$  prema izrazu:

$$N_C = \sum_{i=1}^n \lfloor (\beta \cdot n) / i \rfloor$$



Umjesto funkcije poda  $\lfloor x \rfloor$  može se koristiti i klasično zaokruživanje na najbliži cijeli broj.

Nakon što je završen proces kloniranja, pristupa se hipermutacijama klonova. Pri tome je intenzitet hipermutacije obrnuto proporcionalan afinitetu antitijela. Izvorno, autori predlažu da se koristi vjerojatnosna distribucija određena izrazom:

$$p = e^{-\rho \cdot f}$$

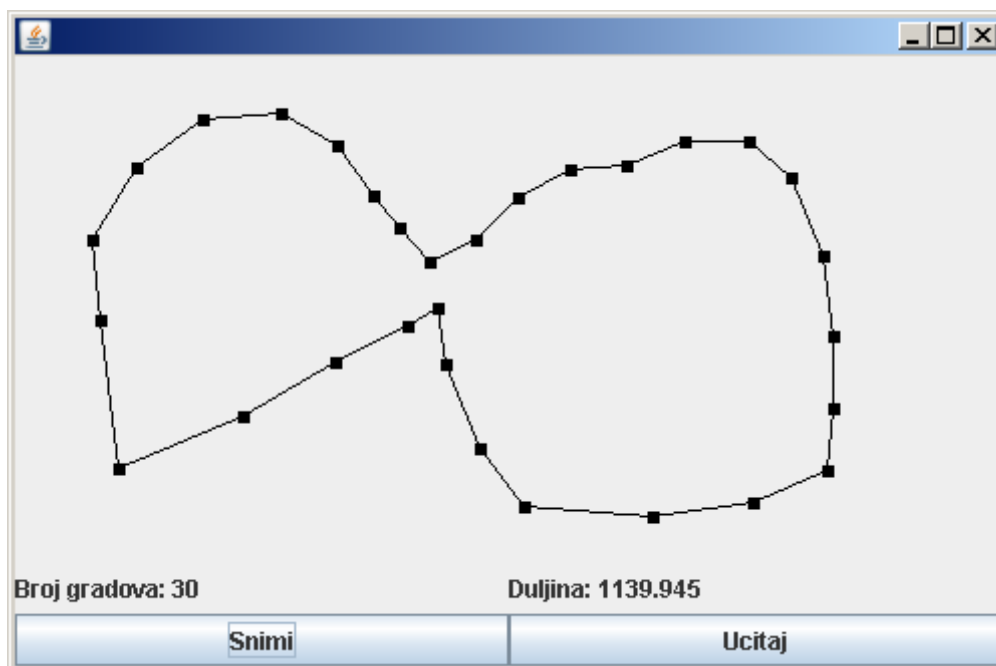
gdje je  $\rho$  korisnički definiran parametar, a  $f$  normalizirana vrijednost afiniteta. Izraz koji se također često koristi je:

$$p = \frac{1}{\rho} e^{-f}$$

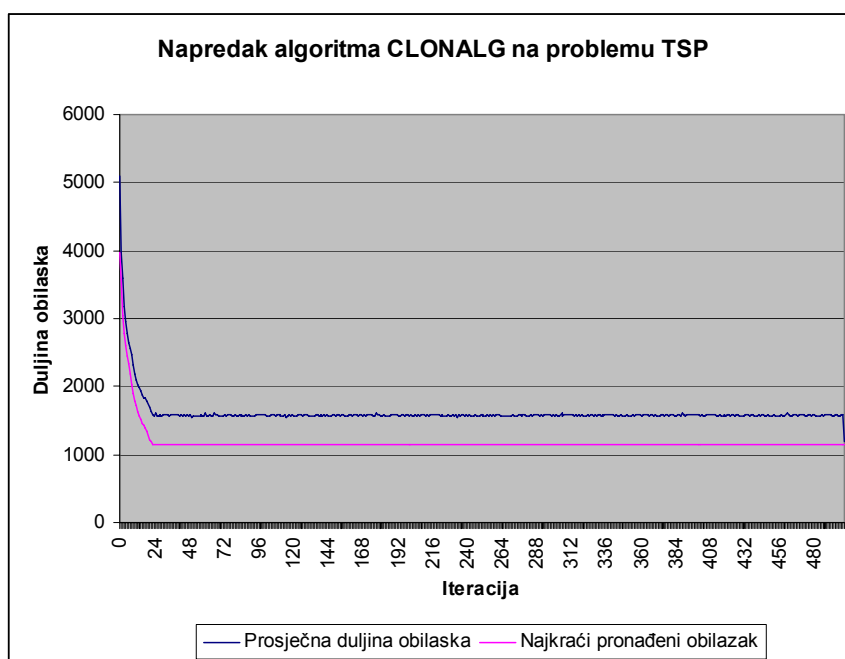
Nakon što je operatorom hipermutacije stvorena populacija  $P^{hyp}$ , računaju se afiniteti svih stvorenih antitijela obzirom na antigen. Potom se novu populaciju odabire  $n$  najboljih antitijela iz populacije  $P^{hyp}$ . Dodatno, kako bi se diverzificirala populacija (unio novi genetski materijal), stvara se  $d$  novih antitijela (slučajnim mehanizmom), i tih  $d$  antitijela zamjenjuje  $d$  antitijela s najmanjim afinitetom.

Analiza složenosti samog algoritma može se pogledati u [5]. Spomenimo samo da je ona veća od prethodno opisanog algoritma jer uključuje i sortiranje kompletnih populacija.

Kako bi se provjerio rad algoritma, napravljena je implementacija u programskom jeziku Java. Rezultat izvođenja algoritma prikazan je na slici 5-3, dok je napredak algoritma po iteracijama prikazan na slici 5-4.



Slika 5-3 Primjer rješenja problema TSP algoritmom CLONALG



Slika 5-4 Napredak algoritma CLONALG na problemu TSP s 30 gradova

Rezultati su dobiveni uz  $n=200$ ,  $d=20$  i  $\beta=5$ .

Prokomentirajmo još način na koji je u tom konkretnom primjeru izveden operator hipermutacije. Kako je svaki antigen predstavljen kao niz indeksa gradova, mutacija se izvodi zamjenom gradova koje treba posjetiti u dva slučajno odabrana koraka. Ovih se zamjena pri tome obavlja to više, što je afinitet antitijela manji. Odabrano je da se broj zamjena kreće od 1 za najbolje antitijelo pa sve do  $1+l \cdot \rho$  za najlošije antitijelo (gdje je  $l$  broj gradova,  $\rho$  pozitivna konstanta manja od 1). Postavljen je zahtjev da točan broj ( $k$ ) bude određen izrazom:

$$k = 1 + l \cdot (1 - e^{-r/\tau})$$

pri čemu je  $r$  rang antitijela (0 za najbolje antitijelo, 1 za sljedeće, itd). Promatramo li  $k$  kao funkciju od  $r$ , iz zahtjeva:

$$k(0) = 1 \text{ i } k(n-1) = 1 + l \cdot \rho$$

slijedi

$$\tau = -\frac{n-1}{\ln(1-\rho)}.$$

Za potrebe eksperimenta odabrano je  $\rho=0.25$ , što uz  $n=200$  daje  $\tau \approx 691.736$  (uočite da je maksimalni rang jednak  $n-1$ ).

Kako bi se očuvalo najbolje rješenje, jedan klon antitijela ranga 0 direktno je propušten u populaciju  $P^{hyp}$  (na tom jednom klonu najboljeg primjerka zabranjena je mutacija).

### 5.3. Pregled korištenih operatora

U nastavku ćemo ukratko nabrojati i opisati vrste operatora koji se danas uobičajeno koriste kod imunoloških algoritama. Detaljna razmatranja ovih operatora kao i dokaz konvergencije imunoloških algoritama nalazi se u [12].

#### 5.3.1. Operator kloniranja

Operator kloniranja zadužen je za stvaranje populacije klonova iz postojeće populacije antitijela. Operator statičkog kloniranja (engl. *static cloning operator*) [7] svako antitijelo iz izvorne populacije klonira  $dup$  puta, čime stvara populaciju klonova  $P^{clo}$  veličine  $d \cdot dup$ . Operator proporcionalnog kloniranja (engl. *proportional cloning operator*) [11] za svako antitijelo stvara broj klonova koji je proporcionalan afinitetu antitijela (tako da bolja antitijela dobiju više klonova). Vjerojatnosno kloniranje (engl. *probabilistic cloning*) [13] definira parametar  $p_c$  (engl. *clonal selection rate*) temeljem kojeg iz izvorne populacije bira antitijela koja će biti klonirana.

#### 5.3.2. Operatori mutacije

Operator mutacije djeluje nakon kloniranja, i tipičnu u populaciju unosi slijepe promjene nad genima. Broj mutacija koje će se napraviti nad jednim antitijelom određen je *potencijalom slučajne mutacije* (engl. *random mutation potential*) [14].

Kod *statičke hipermutacije* (engl. *static hypermutation*) broj mutacija ne ovisi o funkciji dobrote  $f(x)$ , već je ograničen nekom konstantom  $c$ .

Kod *proporcionalne hipermutacije* (engl. *proportional hypermutation*) broj mutacija nad jednim antitijelom proporcionalan je funkciji dobrote  $f(x)$ , te je određen izrazom  $(f(x)-f^*) \cdot (c \cdot l)$ . Pri tome je  $f^*$  minimalna funkcija dobrote antitijela iz trenutne populacije.

Kod *inverzno proporcionalne hipermutacije* (engl. *inversely proportional hypermutation*) broj mutacija nad jednim antitijelom obrnuto je proporcionalan funkciji dobrote  $f(x)$ , te je određen izrazom  $(1 - \frac{f^*}{f(x)}) \cdot (c \cdot l) + (c \cdot l)$  gdje je  $f^*$  minimalna funkcija dobrote antitijela iz trenutne populacije.

Kod *hipermakromutacije* (engl. *hypermakromutation*) broj mutacija ne ovisi niti o funkciji dobrote, niti o konstanti  $c$ . Mutacija se radi tako da se slučajno odaberu dva

indeksa  $i$  i  $j$  tako da vrijedi  $1 \leq i < j \leq l$  (gdje je  $l$  broj gena) i potom se s određenom vjerojatnošću mutiraju svi geni u nizu počev od  $i$ -tog pa do  $j$ -tog.

### 5.3.3. Operator starenja

Starenje je operator koji osigurava da antitijelo ne ostane predugo u populaciji.

Kod *statičkog operatora starenja* (engl. *static pure aging operator*) definira se vrijeme  $\tau_B$  kao broj iteracija koje antitijelo može preživjeti u populaciji. Po isteku tog vremena, antitijelo se briše iz populacije, neovisno o njegovoj dobroti. Kod ove vrste algoritama, operator kloniranja klonovima prepisuje starost roditelja. Potom, nakon faze vrednovanja klonovima koji su bolji od svojih roditelja starost se resetira na 0. Elitistička verzija algoritma postiže se tako da se u svakoj iteraciji starost najboljeg antitijela također resetira na 0 (čime se osigurava da najbolje antitijelo nikada ne bude uništeno).

Kod *stohastičkog operatora starenja* (engl. *stochastic aging operator*) antitijelo iz populacije može biti izbrisano i prije isteka  $\tau_B$ , što je definirano vjerojatnošću preživljavanja koja se smanjuje povećanjem starosti antitijela. Elitistička verzija algoritma tada se dobiva tako da se vjerojatnost preživljavanja za najbolje antitijelo uvijek postavi na 1.

## 5.4. Druga područja

U okviru ovog poglavlja od imunoloških algoritama prikazano je samo jedno usko područje koje se temelji na primjeni principa klonske selekcije. Radi potpunosti pregleda, nužno je spomenuti da danas postoje četiri glavna područja istraživanja: *algoritmi negativne selekcije* (engl. *negative selection algorithms – NSA*), *algoritmi imunoloških mreža* (engl. *immune network algorithms – INA*), *algoritmi zasnovani za teoriji opasnosti* (engl. *danger theory algorithms – DTA*) te *algoritmi klonske selekcije* (engl. *clonal selection algorithms*).

### Pitanja za vježbu

- U kontekstu algoritama umjetnih imunoloških sustava objasnite sljedeće pojmove:
  - antigen
  - antitijelo
  - klon
  - afinitet
- Opišite jednostavan umjetni imunološki sustav (SIA). Navedite pseudokod.
- Opišite imunološki algoritam ClonAlg. Navedite pseudokod. Objasnite kako je definirana veličina populacije klonova kod tog algoritma?
- Opišite razliku između operatora statičkog kloniranja i operatora proporcionalnog kloniranja.
- Opišite razliku između operatora statičke hipermutacije i operatora inverzne proporcionalne hipermutacije.

- Opišite kako djeluje operator hipermakromutacije.
- Opišite izvedbe operatora starenja. Koja je njihova uloga?

Literatura

- [1] Frank Macfarlane Burnet, A modification of Jerne's theory of antibody production using the concept of clonal selection *Australian Journal of Science*, vol. 20, pp. 67-69, 1957.
- [2] Frank Macfarlane Burnet. *The clonal selection theory of acquired immunity*, Nashville, Tennessee, U.S.A.: Vanderbilt University Press, 1959.
- [3] Frank Macfarlane Burnet, Clonal selection and after *Theoretical Immunology*, vol. pp. 63-85, 1978.
- [4] Leandro N. de Castro and Jon Timmis. *Artificial Immune Systems: A new computational intelligence approach*, Great Britain: Springer-Verlag, 2002.
- [5] Vincenzo Cutello and Giuseppe Nicosia. Chapter VI. The Clonal Selection Principle for In Silico and In Vitro Computing. In: *Recent Developments in Biologically Inspired Computing*, eds. Leandro Nunes de Castro and Fernando J. Von Zuben. Hershey, London, Melbourne, Singapore: Idea Group Publishing, 2005. pp. 104-146.
- [6] Leandro N. de Castro and Jon Timmis. *Artificial Immune Systems: A new computational intelligence approach*, Great Britain: Springer-Verlag, 2002.
- [7] Vincenzo Cutello and Giuseppe Nicosia, "An Immunological Approach to Combinatorial Optimization Problems," *Proceedings of the 8th Ibero-American Conference on AI: Advances in Artificial Intelligence*, Seville, Spain, pp. 361-370, 2002.
- [8] Vincenzo Cutello and Giuseppe Nicosia, "Multiple learning using immune algorithms," *Proceedings of 4th International Conference on Recent Advances in Soft Computing*, RASC 2002, Nottingham, UK, pp. 102-107, 2002.
- [9] Leandro N. de Castro and Fernando Jos Von Zuben , "Artificial Immune Systems - Part I: Basic Theory and Applications," Department of Computer Engineering and Industrial Automation, School of Electrical and Computer Engineering, State University of Campinas, Brazil, TR DCA 01/99, Dec 1999
- [10] Leandro N. de Castro and Fernando J. Von Zuben, "The Clonal Selection Algorithm with Engineering Applications," *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00), Workshop on Artificial Immune Systems and Their Applications*, Las Vegas, Nevada, USA, pp. 36-37, 2000.
- [11] Leandro N. de Castro and Fernando J. Von Zuben, Learning and optimization using the clonal selection principle *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 239-251, Jun, 2002.
- [12] V. Cutello, G. Nicosia, P. S. Oliveto, M. Romeo, " On the Convergence of Immune Algorithms", *The First IEEE Symp. on Foundations of Computational Intelligence*, FOCI 2007, 1-5 April 2007, Honolulu, Hawaii, USA. IEEE Press, pp. 409-415, 2007.
- [13] V. Cutello and G. Nicosia and M. Pavone, A Hybrid Immune Algorithm with Information Gain for the Graph Coloring Problem, *GECCO '03*, Chicago, IL, USA, LNCS, Springer, vol. 2723, pp.171-182, 2003.

- [14] V. Cutello and G. Narzisi and G. Nicosia and M. Pavone, Clonal Selection Algorithms: A Comparative Case Study using Effective Mutation Potentials, ICARIS 2005, LNCS, Springer, vol. 3627, pp.13-28, 2005.





## 6. Programski zadatci

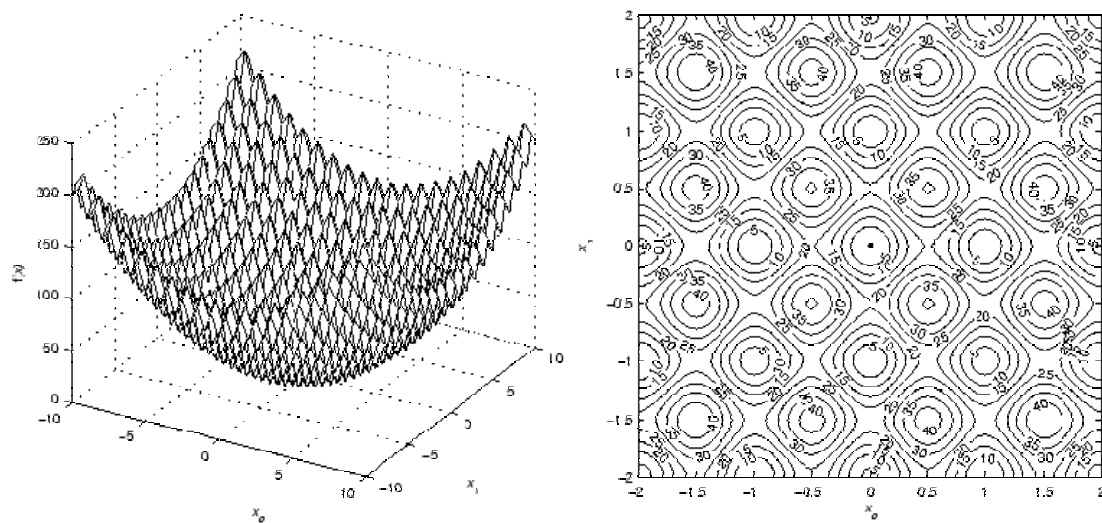
U ovom poglavlju opisani su konkretni programski zadatci čijim će se rješavanjem steći puno bolji dojam o ponašanju pojedinih evolucijskih algoritama.

### 6.1. Poopćena Rastriginova funkcija

Ovo je primjer minimizacije funkcije nad kontinuiranim varijablama. Funkcija  $f$  je funkcija definirana nad  $D$ -dimenzijskim vektorom realnih brojeva, na sljedeći način.

$$f(\vec{x}) = 10 \cdot D + \sum_{i=1}^D (x_i^2 - 10 \cdot \cos(2\pi x_i))$$

Ova funkcija prikazana je na sljedećoj slici.



Globalni optimum ove funkcije je poznat. To je  $\vec{x}^* = (0, 0, \dots, 0)$  u kojem je  $f(\vec{x}^*) = 0$ . Napišite program koji će kao argumente komandne linije dobiti 3 parametra: dimenzionalnost vektora ( $D$ ), te minimalnu i maksimalnu vrijednost unutar koje se pretražuje prostor rješenja. Ako se za rješavanje koriste algoritmi koji pretražuju diskretan prostor, tada se treba raditi minimalno s preciznošću od  $10^{-4}$ . Zadana minimalna i maksimalna vrijednost vrijedi za sve komponente vektora  $\vec{x}$ .

Provjerite ponašanje Vaše implementacije za  $D=1$ ,  $D=4$  te  $D=10$  i prostor pretraživanja  $[-10, 10]$  po svakoj komponenti.

#### 6.1.1. Prikladni algoritmi za rješavanje problema

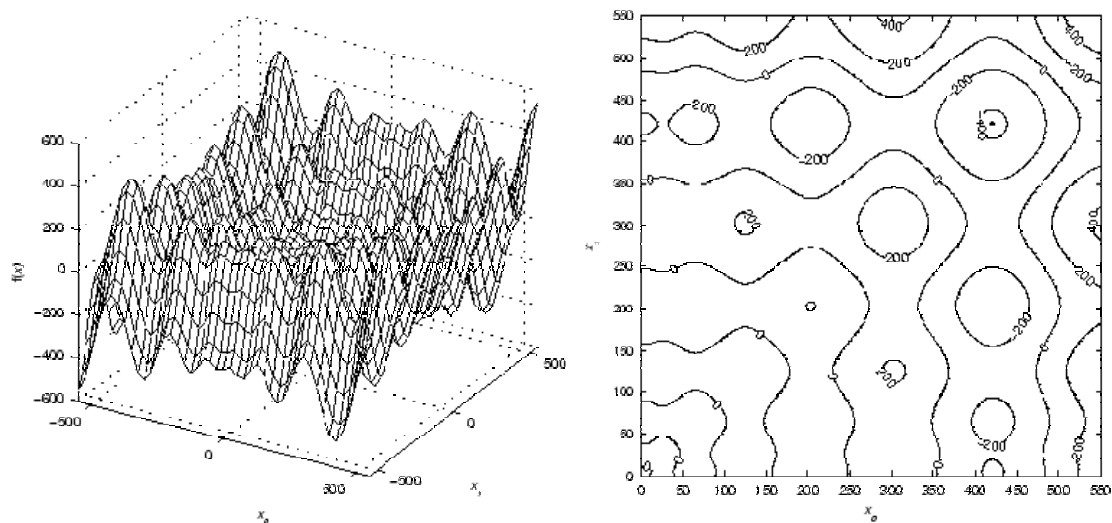
Ovaj zadatak možete rješavati genetskim algoritmom, algoritmom roja čestica te umjetnim imunološkim algoritmom.

### 6.2. Normalizirana Schwefelova funkcija

Ovo je primjer minimizacije funkcije nad kontinuiranim varijablama. Funkcija  $f$  je funkcija definirana nad  $D$ -dimenzijskim vektorom realnih brojeva, na sljedeći način.

$$f(\vec{x}) = \frac{\sum_{i=1}^D (-x_i \sin(\sqrt{|x_i|}))}{D}$$

Ova funkcija prikazana je na sljedećoj slici.



Optimum ove funkcije ovisi o intervalu vrijednosti koje varijable mogu poprimiti. Ako se ispituje interval  $\pm 512$ , to je  $x_i^* \approx 420,968746$ ,  $i=1,2,\dots,D$  u kojem je  $f(\bar{x}^*) \approx -418,982887$ . Napišite program koji će kao argumente komandne linije dobiti 3 parametra: dimenzionalnost vektora  $D$ , te minimalnu i maksimalnu vrijednost unutar koje se pretražuje prostor rješenja. Ako se za rješavanje koriste algoritmi koji pretražuju diskretan prostor, tada se treba raditi minimalno s preciznošću od  $10^{-4}$ . Zadana minimalna i maksimalna vrijednost vrijedi za sve komponente vektora  $\bar{x}$ .

Provjerite ponašanje Vaše implementacije za  $D=1$ ,  $D=4$  te  $D=10$  i prostor pretraživanja  $[-10, 10]$  po svakoj komponenti.

### 6.2.1. Prikladni algoritmi za rješavanje problema

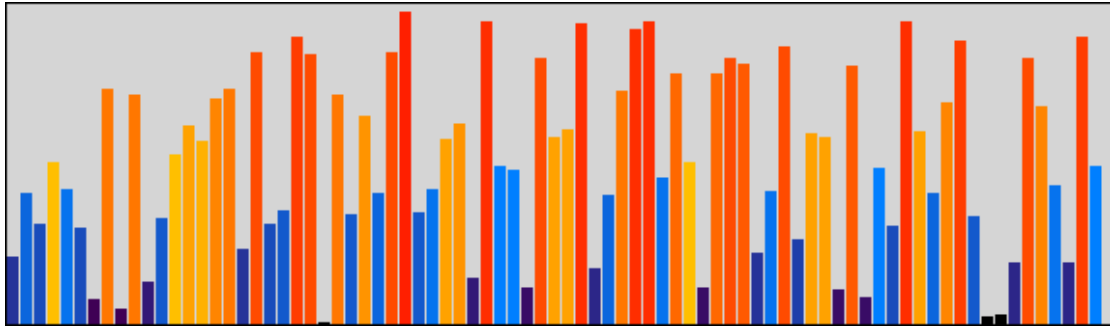
Ovaj zadatak možete rješavati genetskim algoritmom, algoritmom roja čestica te umjetnim imunološkim algoritmom.

### 6.3. Problem popunjavanja kutija

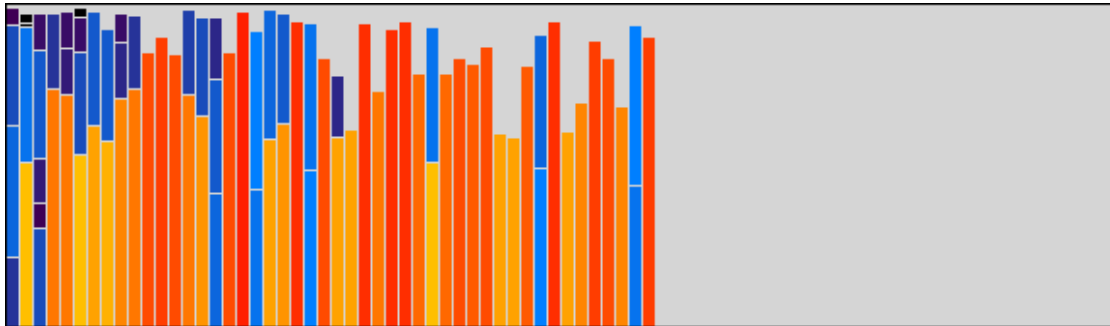
Ovo je jedan od problema koji se često javljaju u privredi. Ilustrirajmo ga na primjeru jednodimenzijskih objekata.

Neki proizvođač proizvodi štapove različitih duljina. Iz tvornice, štapove je potrebno transportirati do distribucijskog centra u spremnicima fiksne visine (radi jednostavnosti pretpostavimo da su spremnici tanki baš kao i štapovi). Štapovi se u spremnik mogu slagati i jedan na drugi (svi su jednakog promjera), tako dugo dok ukupna visina ne premaši visinu spremnika. Potrebno je pronaći takvo pakiranje štapova koje će zahtijevati spremnik minimalne duljine.

Pogledajmo to na sljedećem primjeru. Slika u nastavku prikazuje niz štapova koje treba pakirati; pri tome su svi štapovi poslagani jedan do drugoga. Ovisno o visini, štapovi su prikazani različitim bojama.



Primjer jednog mogućeg pakiranja prikazan je u nastavku.



U ovom slučaju duljina potrebnog spremnika očito je manja.

Vaš je zadatak problem riješiti nekim od prirodom inspiriranih optimizacijskih algoritama koji su prikazani u ovoj skripti. S obzirom da je ovo inherentno kombinatorički problem, najjednostavnije je to pokušati riješiti uporabom genetskog algoritma ili umjetnih imunoloških algoritama.

### 6.3.1. Naputci

Evo i nekoliko naputaka o kojima je dobro razmisliti prilikom izrade programske implementacije rješenja problema.

- *Optimalnost rješenja* – uočimo da rješenje uvijek postoji; pitanje je samo kvalitete tog rješenja. U ovom primjeru najjednostavnija mjera kvalitete uzimat će u obzir samo duljinu potrebnog spremnika – što manje, to bolje.
- *Tvrda ograničenja* – tvrda ograničenja ovdje su jasna: ukupna duljina naslaganih štapova ne smije biti veća od visine spremnika.
- *Vrijednost evaluacijske funkcije* – razmislite kako ćete vrednovati svako rješenje. Hoće li to biti samo jedan broj (duljina potrebnog spremnika) ili možda vektor brojeva (uz duljinu, možda uzimati u obzir još neke karakteristike "punjenja"), te kako ćete, posebice u ovom posljednjem slučaju, raditi usporedbu takvih rješenja (često će Vam trebati odgovor na pitanje "koje je rješenje bolje").
- *Kriterij zaustavljanja* – što će biti prikladan kriterij zaustavljanja u opisanom problemu?
- *Način prikaza rješenja* – način na koji ćete interno prikazati vaše rješenje imat će drastični efekt na izvedbu operatora optimizacijskog algoritma. Primjerice, iako je najjednostavniji, binarni prikaz rješenja (nizom bitova) u ovom slučaju nije baš najsretnije rješenje. Razmislite o tome da rješenje prikazete odgovarajućom strukturom podataka.

## 6.4. Izrada rasporeda timova studenata

Na kolegiju *Umjetnost, znanost i filozofija* studenti su podijeljeni u studentske timove. Na kolegiju ima nekoliko asistenata. Svakom timu jedan je asistent dodijeljen kao voditelj. Međutim, kako je broj timova veći od broja asistenata, jedan asistent vodi više timova. Raspodjela studenata po timovima te dodjela voditelja timovima napravljena je na početku semestra i ne može se mijenjati.

Za ovaj kolegij potrebno je organizirati predaju projekata. U terminu predaje projekta *svi članovi tima* trebaju doći istovremeno u određenu prostoriju. Potrebno je napraviti raspored predaje timskih projekata, tako da svi timovi mogu predati svoje projekte. Pri tome treba paziti na ograničenja opisana u nastavku.

### 6.4.1. Ograničenja

Prilikom izrade rasporeda, potrebno je voditi računa o sljedećim ograničenjima.

- Zauzeća studenata predavanjima i drugim obavezama – raspored treba napraviti tako da se uklopi u postojeći studentski raspored, odnosno da se ne generiraju konflikti. Sva studentska zauzeća dostupna su u datoteci `zauzetost.txt`. Svaki redak predstavlja jedno zauzeće, pri čemu su elementi retka šifra studenta, datum zauzeća, početak zauzeća, kraj zauzeća.
- Zauzeća dvorana – raspored se treba uklopiti u postojeća zauzeća dvorana. Kako bi se ovo olakšalo, u drugom dijelu datoteke `projekt.txt` nalazi se popis termina u kojima su dvorane slobodne (svaki termin vremenski odgovara točno jednoj predaji projekta). Uz svaki termin navedene su slobodne dvorane u tom terminu, te uz svaku dvoranu broj studenata koji ta dvorana može primiti u tom terminu.
- Ograničenja na voditelje – kako jedan asistent može biti voditelj u više timova, potrebno je osigurati da se ne napravi raspored u kojem projekt u istom terminu predaju dva tima istog voditelja, jer ih voditelj neće moći ispitati. U jednom terminu voditelj može ispitati samo jedan (svoj) tim. Također, voditelj ne može ispitivati druge timove. Nazivi timova, voditelji timova te šifre studenata koji pripadaju timu navedeni su u prvom dijelu datoteke `projekt.txt`.

Format datoteke koja predstavlja rješenje problema, kao i konkretni primjeri problema mogu se pogledati direktno u pripremljenoj arhivi dostupnoj na webu:

<http://java.zemris.fer.hr/nastava/ui/programskiZadatci/timovi.zip>

### 6.4.2. Naputci

Evo i nekoliko naputaka o kojima je dobro razmisliti prilikom izrade programske implementacije rješenja problema.

- *Optimalnost rješenja* – moguće je da rješenje problema bez konflikata ne postoji. Stoga postavite problem kao optimizacijski problem: za svako generirano rješenje izbrojite koliko minuta konflikata u tom rješenju imaju studenti. Zadatak optimizacije je pronaći rješenje s minimalnim brojem konflikata.
- *Tvrda vs. meka ograničenja* – razmislite koja će Vam od ograničenja biti tvrda a koja meka, i zašto. Prisjetimo se, tvrda ograničenja su ona čije

nezadovoljavanje povlači neprihvatljivost rješenja. Meka ograničenja imaju pak utjecaj na kvalitetu rješenja.

- *Vrijednost evaluacijske funkcije* – razmislite kako ćete vrednovati svako rješenje. Hoće li to biti jedan broj ili možda vektor brojeva, te kako ćete, posebice u ovom posljednjem slučaju, raditi usporedbu takvih rješenja (često će Vam trebati odgovor na pitanje "koje je rješenje bolje").
- *Kriterij zaustavljanja* – što će biti prikladan kriterij zaustavljanja u opisanom problemu?
- *Način prikaza rješenja* – način na koji ćete interno prikazati vaše rješenje imat će drastični efekt na izvedbu operatora optimizacijskog algoritma. Primjerice, iako je najjednostavniji, binarni prikaz rješenja (nizom bitova) u ovom slučaju nije baš najsretnije rješenje. Razmislite o tome da rješenje prikazete odgovarajućom strukturom podataka.

### **6.4.3. Prikladni algoritmi za rješavanje problema**

Ovaj problem možete rješavati svim algoritmima opisanim u ovoj skripti.

## **6.5. Izrada prezentacijskih grupa za seminare (1)**

Na preddiplomskom predmetu Seminar studente vode voditelji koji tipično imaju dodijeljena do četiri studenta. Jednog voditelja s njegovim studentima nazvat ćemo jednom mikro-grupom. Prezentacije seminarskih radova odvijaju se u većim prezentacijskim grupama, koje tipično broje do 30 studenata. Kako bi se ovo ostvarilo, više mikro-grupa potrebno je spojiti u jednu prezentacijsku grupu, što je moguće obaviti uzimajući u obzir različite kriterije. Obratite pažnju da se prilikom izrade prezentacijskih grupa ne može raditi na razini pojedinog studenta – grupe se formiraju iz mikro-grupa: uključivanjem jednog voditelja u grupu uključili ste sve njegove studente u tu grupu.

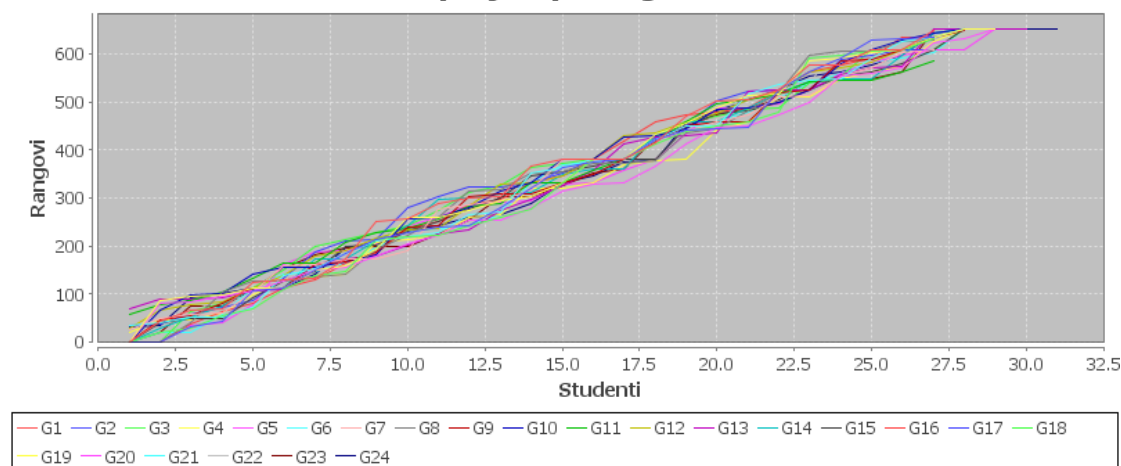
### **6.5.1. Zadatak**

U okviru ovog zadatka potrebno je uporabom evolucijskog računanja sve mikro-grupe podijeliti u 24 prezentacijske grupe koje su sve podjednake veličine ( $28 \pm 1$ ), s time da je potrebno minimizirati ukupno odstupanje veličine grupa (npr. bolje je rješenje s 22 grupom od 28 studenata i 2 grupe od 29 studenata, od rješenja koje ima 20 grupa od 28 studenata, 2 grupe od 27 studenata i 2 grupa od 29 studenata).

Unutar svih takvih mogućih raspodjela grupa, potrebno je pronaći takvu razdiobu koja u svakoj grupi ima što je moguće više ujednačenu distribuciju rangova studenata. Što to točno znači? Za svakog studenta dostupan je njegov rang studija. Rangovi se kreću od 1 do poprilično 800, i postoje jednako rangirani studenti. Prilikom spajanja mikro-grupa u jednu prezentacijsku grupu treba paziti da se ne pojave rješenja koja primjerice u jednu grupu smjeste puno studenata s rangom do 100, a malo studenata s rangovima od 101 do 800, ili pak da se ne pojavi grupa u koju su smješteni pretežno studenti s rangovima iznad 400, ili bilo kakva slična nepravilnost. Idealno bi bilo kada bi se unutar svake prezentacijske grupe našao podjednak broj studenata svih rangova. Ovo je ilustrirano na dijagramu u nastavku. Dijagram je nastao tako što su studenti unutar svake prezentacijske grupe sortirani prema rangovima (u tom primjeru grupe

su imale do 31 studenta). Pri tome je na prvom mjestu student s najmanjim rangom studija a na zadnjem mjestu student s najvećim rangom studija. Iz dijagrama je vidljivo da su u svakoj od 24 prezentacijske grupe prva tri studenta s rangom studija od 1 do 100, 10. student u svakoj je grupi po rangu između 200 i 300, itd. Ovakav prikaz je zgodan kako bismo vizualno mogli ocijeniti kvalitetu dobivenog rješenja, i nacrtan je uporabom biblioteke jfreechart (za programski jezik Java).

**Raspodjela po rangovima**



S obzirom da svaka prezentacijska grupa u ovom zadatku smije imati  $28 \pm 1$  studenta, očito je da u nju nije moguće smjestiti po jednog studenta ranga 1, jednog studenta ranga 2, jednog studenta ranga 3 i tako do 800. Stoga je vaš zadatak sljedeći:

- osmislite kako ćete vrednovati kvalitetu razdiobe rangova unutar jedne prezentacijske grupe
- osmislite kako ćete vrednovati kvalitetu razdiobe rangova unutar jednog rješenja (dakle temeljem svih 24 grupa)

### 6.5.2. Naputci

Prilikom rješavanja ovog zadatka trebat ćete odgovoriti i na pitanje kako usporediti dva rješenja, odnosno koje je rješenje bolje. Uočite da u ovom slučaju imate dvije vrste ograničenja s kojima radite:

- tvrdo ograničenje jest veličina svake od grupa na  $28 \pm 1$ ,
- meko ograničenje je da želimo minimizirati razliku u veličini pojedinih grupa, te
- meko ograničenje jest da želimo što ravnomjerniju razdiobu rangova unutar svake od grupa.

Također, razmislite kako ćete kombinirati dva meka ograničenja u konačnu odluku kojom ćete reći što je bolje.

Potrebne podatke s izmišljenim studentima, voditeljima i rangovima možete dohvatiti s adrese:

[http://java.zemris.fer.hr/nastava/ui/programskiZadatci/seminari\\_1.zip](http://java.zemris.fer.hr/nastava/ui/programskiZadatci/seminari_1.zip)

U toj ZIP arhivi nalazi se datoteka koja u svakom retku navodi JMBAG studenta, oznaku dodijeljenog voditelja te rang studija studenta.

Kao rješenje problema trebali biste ponuditi datoteku sličnu ulaznoj koja u svakom retku ima još i podatak o dodijeljenoj prezentacijskoj grupi (označite ih s G1 do G24), grafički prikaz raspodjele rangova te stupičasti prikaz veličina grupa (engl. *bar-chart*).

Kako biste lakše pratili napredak vašeg algoritma, preporuka je ove dijagrame crtati uživo za najbolje pronađeno rješenje (tijekom samog postupka optimizacije) i prikazivati ih na ekranu. U tom slučaju slike ne trebate pohranjivati na disk kao dio rješenja.

### **6.5.3. Prikladni algoritmi za rješavanje problema**

Ovaj zadatak možete rješavati genetskim algoritmom, algoritmom kolonije mrava te umjetnim imunološkim algoritmom.

## **6.6. Izrada prezentacijskih grupa za seminare (2)**

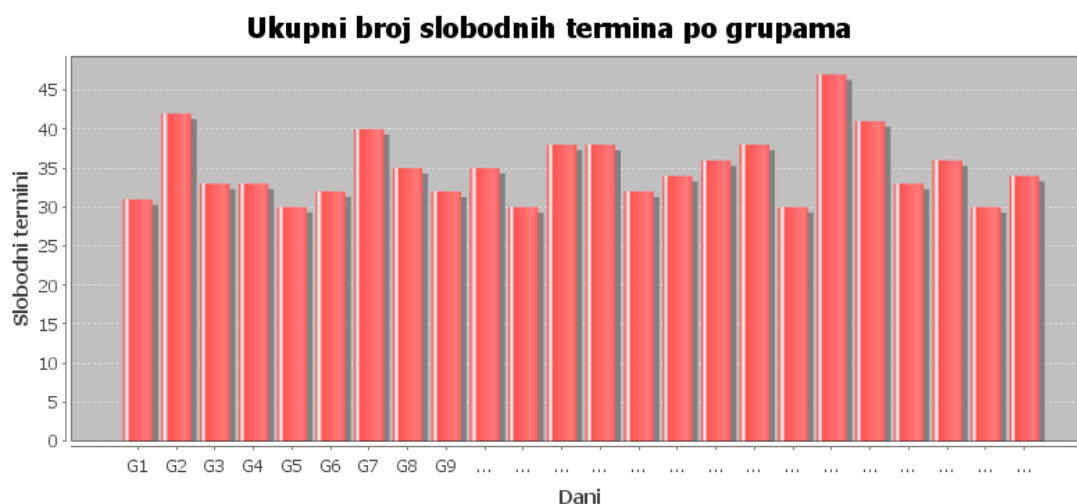
Na preddiplomskom predmetu Seminar studente vode voditelji koji tipično imaju dodijeljena do četiri studenta. Jednog voditelja s njegovim studentima nazvat ćemo jednom mikro-grupom. Prezentacije seminarskih radova odvijaju se u većim prezentacijskim grupama, koje tipično broje do 30 studenata. Kako bi se ovo ostvarilo, više mikro-grupa potrebno je spojiti u jednu prezentacijsku grupu, što je moguće obaviti uzimajući u obzir različite kriterije. Obratite pažnju da se prilikom izrade prezentacijskih grupa ne može raditi na razini pojedinog studenta – grupe se formiraju iz mikro-grupa: uključivanjem jednog voditelja u grupu uključili ste sve njegove studente u tu grupu.

### **6.6.1. Zadatak**

U okviru ovog zadatka potrebno je uporabom evolucijskog računanja sve mikro-grupe podijeliti u 24 prezentacijske grupe koje su sve podjednake veličine ( $28 \pm 1$ ), s time da je potrebno minimizirati ukupno odstupanje veličine grupa (npr. bolje je rješenje s 22 grupom od 28 studenata i 2 grupe od 29 studenata, od rješenja koje ima 20 grupa od 28 studenata, 2 grupe od 27 studenata i 2 grupa od 29 studenata).

Unutar svih takvih mogućih raspodjela grupa, potrebno je pronaći takvu razdiobu koja u svakoj grupi ostavlja što je moguće više vremena u terminima od 8h ujutro do 20h navečer kada bi se mogle održati prezentacije, a da to ne kolidira s postojećim obavezama svih studenata koji tada trebaju prisustvovati prezentaciji. Uočite da zadatak nije napraviti i raspored prezentacija – termine će naknadno odabrati voditelji u dogovoru sa studentima. Ideja je samo pronaći takvo grupiranje studenata uz koje će se maksimizirati broj potencijalnih termina kada su svi studenti grupe slobodni.

"Kvaliteta" jednog rješenja ovakvog problema prikazana je na slici u nastavku, i to u obliku stupičastog dijagrama (engl. *bar-chart diagram*). Ovakav prikaz je zgodan kako bismo vizualno mogli ocijeniti kvalitetu dobivenog rješenja, i nacrtan je uporabom biblioteke *jfreechart* (za programski jezik Java).



Termini za prezentacije traju po 45 minuta. Ako čitava grupa ima primjerice jedan slobodan sat (od 12h do 13h), to ćete brojati kao jedan termin. Da su imali slobodno od 12h do 13:30h, to biste brojali kao 2 slobodna termina.

### 6.6.2. Naputci

Prilikom rješavanja ovog zadatka trebat ćete odgovoriti i na pitanje kako usporediti dva rješenja, odnosno koje je rješenje bolje. Uočite da u ovom slučaju imate dvije vrste ograničenja s kojima radite:

- tvrdo ograničenje jest veličina svake od grupa na  $28 \pm 1$ ,
- meko ograničenje je da želimo za svaku grupu osigurati što je moguće više slobodnih termina.

Također, razmislite kako ćete vrednovati količinu slobodnog vremena. Naime, s jedne strane moći ćete izračunati koliko svaka od grupa ima slobodnih termina. Pitanje je samo kako te podatke iskoristiti da biste dobili ukupnu dobrotu rješenja. Ovo je posebice važno ako uzmemo u obzir i činjenicu da povećanje količine slobodnog vremena jedne grupe može dovesti do smanjenja količine slobodnog vremena neke druge grupe.

Potrebne podatke s izmišljenim studentima, voditeljima i rangovima, te zauzećima studenata možete dohvatiti s adrese:

[http://java.zemris.fer.hr/nastava/ui/programskiZadatci/seminari\\_2.zip](http://java.zemris.fer.hr/nastava/ui/programskiZadatci/seminari_2.zip)

U toj ZIP arhivi postoje tri datoteke.

- Datoteka `studenti-nastavnici.txt` u svakom retku navodi JMBAG studenta te oznaku dodijeljenog voditelja.
- Datoteka `zauzetost.txt` u svakom retku ima jedan zapis zauzeća nekog studenta: JMBAG, dan, početak zauzeća te kraj zauzeća.
- Datoteka `dani.txt` u svakom retku navodi po jedan dan u kojem se održavaju prezentacije. Datoteka je potrebna jer je moguće da postoji dan u kojem niti jedan student nema nikakvih obaveza, pa se taj dan neće niti spomenuti u datoteci s zauzećima. Ako u datoteci `zauzetost.txt` ima dana koji se ovdje ne spominju, zanemarite ih.



Kao rješenje problema trebali biste ponuditi datoteku sličnu ulaznoj (`studenti-nastavnici.txt`) koja u svakom retku ima još i podatak o dodijeljenoj prezentacijskoj grupi (označite ih s G1 do G24), grafički prikaz količine slobodnog vremena te stupičasti prikaz veličina grupa.

Kako biste lakše pratili napredak vašeg algoritma, preporuka je ove dijagrame crtati uživo za najbolje pronađeno rješenje (tijekom samog postupka optimizacije) i prikazivati ih na ekranu. U tom slučaju slike ne trebate pohranjivati na disk kao dio rješenja.

### **6.6.3. *Prikladni algoritmi za rješavanje problema***

Ovaj zadatak možete rješavati genetskim algoritmom, algoritmom kolonije mrava te umjetnim imunološkim algoritmom.



## 7. Implementacija evolucijskih algoritama u programskom jeziku Java

U nastavku slijede primjeri implementacija opisanih algoritama u programskom jeziku Java. Kompletan Eclipse projekt s algoritmima također je dostupan na sljedećoj adresi:

<http://java.zemris.fer.hr/nastava/ui/evoAlg.zip>

### 7.1. Genetski algoritam

Genetski algoritam smješten je u paket `hr.fer.zemris.ga`. Ostvaren je kroz tri razreda. Razred `GeneticAlgorithm` čini jezgru samog algoritma. Razred `Kromosom` predstavlja jedan kromosom. Razred `KromosomDekoder` sadrži funkcionalnost dekodiranja binarnog kromosoma u realne varijable.

#### 7.1.1. Razred `GeneticAlgorithm`

```
package hr.fer.zemris.ga;

import hr.fer.zemris.numeric.IFunkcija;

import java.util.Arrays;
import java.util.Random;

/**
 * Primjer implementacije genetskog algoritma za optimizaciju funkcije jedne
 * varijable.
 *
 * Program koristi binarni prikaz kromosoma.
 *
 * @author marcupic
 */
public class GeneticAlgorithm {

    /**
     * Glavni program.
     *
     * @param args argumenti komandne linije - ne koriste se.
     */
    public static void main(String[] args) {

        // Definiranje osnovnih parametara algoritma
        int VEL_POP = 50;
        double VJER_KRIZ = 0.7;
        double VJER_MUT = 0.005;

        // Generator slučajnih brojeva
        Random rand = new Random();

        // Stvaranje dekodera binarnog kromosoma:
        // jedna varijabla, 10 bitova, raspon pretraživanja [-5, 5]
        KromosomDekoder dekodeer = new KromosomDekoder(1, 10, -5, 5);

        // Stvaranje dviju populacija n-ta, i (n+1)-va
        // Zbog optimizacije, kasnije ćemo samo mijenjati ta polja
        Kromosom[] populacija = stvoriPopulaciju(VEL_POP, dekodeer, rand);
        Kromosom[] novaGeneracija = stvoriPopulaciju(VEL_POP, dekodeer, null);

        // Definiranje funkcije koju optimiramo
        IFunkcija funkcija = new IFunkcija() {
            @Override
            public double izracunaj(double[] varijable) {
                int n = varijable.length;
                double vrijednost = 10*n;
                for(int i = 0; i < n; i++) {
                    vrijednost += varijable[i]*varijable[i]
                        - 10*Math.cos(2*Math.PI*varijable[i]);
                }
            }
        };
    }
}
```

```
    }
    return vrijednost;
}
};

// Početna evaluacija populacije
evaluirajPopulaciju(populacija, funkcija);

// Ponovi kroz 1000 generacija
for(int generacija = 0; generacija < 1000; generacija++) {

    // Sortiraj populaciju po dobroti; najbolja jedinka bit će prva
    Arrays.sort(populacija);

    // U novu populaciju prekopiraj dvije najbolje (elitizam!)
    kopiraj(populacija[0], novaGeneracija[0]);
    kopiraj(populacija[1], novaGeneracija[1]);

    // Stvori preostale jedinke nove generacije
    for(int i=1; i < VEL_POP/2; i++) {
        Kromosom roditelj1 = odaberiRoditelja(populacija, rand);
        Kromosom roditelj2 = odaberiRoditelja(populacija, rand);
        Kromosom dijete1 = novaGeneracija[2*i];
        Kromosom dijete2 = novaGeneracija[2*i+1];
        krizajITockaPrijeloma(
            VJER_KRIZ, roditelj1, roditelj2, dijete1, dijete2, rand);
        mutiraj(VJER_MUT, dijete1, rand);
        mutiraj(VJER_MUT, dijete2, rand);
    }

    // Zamijeni staru i novu populaciju
    Kromosom[] pomocni = populacija;
    populacija = novaGeneracija;
    novaGeneracija = pomocni;

    // Vrednuj populaciju
    evaluirajPopulaciju(populacija, funkcija);

    // pronadi najbolje rjesenje
    Kromosom najbolji = null;
    for(int i = 0; i < populacija.length; i++) {
        if(i==0 || najbolji.fitnes > populacija[i].fitnes) {
            najbolji = populacija[i];
        }
    }
    // I ispiši ga...
    System.out.println("Trenutno rjesenje: f("
        + Arrays.toString(najbolji.varijable)
        + ") = "+funkcija.izracunaj(najbolji.varijable));
}

}

/**
 * Pomoćna funkcija koja obavlja kopiranje jednog kromosoma u drugi.
 * @param original
 * @param kopija
 */
private static void kopiraj(Kromosom original, Kromosom kopija) {
    for(int i = 0; i < original.bitovi.length; i++) {
        kopija.bitovi[i] = original.bitovi[i];
    }
}

/**
 * Stvaranje nove populacije.
 *
 * @param brojJedinki broj jedinki koji treba stvoriti
 * @param dekoder koji se dekoder koristi
 * @param rand generator slučajnih brojeva
 * @return novu populaciju
 */
public static Kromosom[] stvoriPopulaciju(int brojJedinki, KromosomDekoder dekoder,
    Random rand) {
    Kromosom[] populacija = new Kromosom[brojJedinki];
    for(int i = 0; i < populacija.length; i++) {
        if(rand==null) {
            populacija[i] = new Kromosom(dekoder);
        }
    }
}
```

```

        } else {
            populacija[i] = new Kromosom(dekoder, rand);
        }
    }
    return populacija;
}

/**
 * Metoda vrednuje predanu populaciju.
 *
 * @param populacija populacija
 * @param funkcija funkcija koja se optimira
 */
private static void evaluirajPopulaciju(Kromosom[] populacija, IFunkcija funkcija)
{
    for(int i = 0; i < populacija.length; i++) {
        evaluirajJedinku(populacija[i], funkcija);
    }
}

/**
 * Metoda vrednuje predani kromosom.
 *
 * @param kromosom kromosom
 * @param funkcija funkcija koja se optimira
 */
private static void evaluirajJedinku(Kromosom kromosom, IFunkcija funkcija) {
    kromosom.dekoder.dekodirajKromosom(kromosom);
    kromosom.fitness = funkcija.izracunaj(kromosom.varijable);
}

/**
 * Metoda za odabir jednog roditelja, gdje je vjerojatnost odabira
 * proporcionalna dobroti.
 *
 * @param populacija populacija iz koje se bira
 * @param rand generator slučajnih brojeva
 * @return odabranog roditelja
 */
private static Kromosom odaberiRoditelja(Kromosom[] populacija, Random rand) {
    double sumaDobrota = 0;
    double najvecaVrijednost = 0;
    for(int i = 0; i < populacija.length; i++) {
        sumaDobrota += populacija[i].fitness;
        if(i==0 || najvecaVrijednost<populacija[i].fitness) {
            najvecaVrijednost = populacija[i].fitness;
        }
    }
    sumaDobrota = populacija.length * najvecaVrijednost - sumaDobrota;
    double slucajniBroj = rand.nextDouble() * sumaDobrota;
    double akumuliranaSuma = 0;
    for(int i = 0; i < populacija.length; i++) {
        akumuliranaSuma += najvecaVrijednost - populacija[i].fitness;
        if(slucajniBroj<akumuliranaSuma) return populacija[i];
    }
    return populacija[populacija.length-1];
}

/**
 * Metoda obavlja križanje s jednom točkom prijeloma. Križanje se obavlja s
 * zadanom vjerojatnošću. Ako se ne dogodi križanje, kao djeca se vraćaju
 * roditelji.
 *
 * @param vjerKriz vjerojatnost križanja (decimalni broj između 0 i 1)
 * @param roditelj1 prvi roditelj
 * @param roditelj2 drugi roditelj
 * @param dijete1 prvo dijete
 * @param dijete2 drugo dijete
 * @param rand generator slučajnih brojeva
 */
private static void kriزاز1TockaPrijeloma(double vjerKriz, Kromosom roditelj1,
    Kromosom roditelj2, Kromosom dijete1, Kromosom dijete2, Random rand) {
    if(rand.nextFloat() <= vjerKriz) {
        int tockaPrijeloma = rand.nextInt(roditelj1.dekoder.ukupnoBitova-1)+1;
        for(int i = 0; i < tockaPrijeloma; i++) {
            dijete1.bitovi[i] = roditelj1.bitovi[i];

```

```
        dijete2.bitovi[i] = roditelj2.bitovi[i];
    }
    for(int i = tockaPrijeloma; i < roditelj1.dekoder.ukupnoBitova; i++) {
        dijete1.bitovi[i] = roditelj2.bitovi[i];
        dijete2.bitovi[i] = roditelj1.bitovi[i];
    }
} else {
    for(int i = 0; i < roditelj1.dekoder.ukupnoBitova; i++) {
        dijete1.bitovi[i] = roditelj1.bitovi[i];
        dijete2.bitovi[i] = roditelj2.bitovi[i];
    }
}
}
}

/**
 * Operator mutacije. Bitovi se okreću zadanom vjerojatnošću.
 *
 * @param vjerMut vjerojatnost mutacije bita (broj od 0 do 1)
 * @param dijete dijete koje se mutira
 * @param rand generator slučajnih brojeva
 */
private static void mutiraj(double vjerMut, Kromosom dijete, Random rand) {
    for(int i = 0; i < dijete.dekoder.ukupnoBitova; i++) {
        if(rand.nextFloat() <= vjerMut) {
            dijete.bitovi[i] = (byte) (1-dijete.bitovi[i]);
        }
    }
}
}
```

### 7.1.2. Razred Kromosom

```
package hr.fer.zemris.ga;

import java.util.Random;

/**
 * Binarni kromosom - jedno rješenje genetskog algoritma. Pretpostavlja
 * se da je posrijedi rješavanje problema koji se sastoji od više realnih
 * varijabli što se reflektira u građi samog kromosoma.
 *
 * Važno: ugrađena funkcija za usporedbu kromosoma pretpostavlja
 * da se radi o minimizacijskom problemu.
 *
 * @author marcupic
 */
public class Kromosom implements Comparable<Kromosom> {
    // Bitovi kromosoma
    byte[] bitovi;
    // Vrijednost funkcije dobrote kromosoma (zapravo, to je vrijednost
    // funkcije u promatranoj točki)
    double fitnes;
    // Vrijednosti realnih varijabli koje kromosom predstavlja
    double[] varijable;
    // Dekoder koji zna konvertirati binarni prikaz u realne varijable
    KromosomDekoder dekoder;

    /**
     * Konstruktor koji stvara novi kromosom ali ga ne inicijalizira.
     *
     * @param dekoder dekoder kromosoma
     */
    public Kromosom(KromosomDekoder dekoder) {
        this.dekoder = dekoder;
        this.bitovi = new byte[dekoder.ukupnoBitova];
        this.fitnes = 0;
        this.varijable = new double[dekoder.brojVarijabli];
    }

    /**
     * Konstruktor koji stvara novi kromosom i inicijalizira
     * ga na slučajni uzorak bitova.
     *
     * @param dekoder dekoder kromosoma
     * @param rand generator slučajnih brojeva
     */
}
```

```

*/
public Kromosom(KromosomDekoder dekodler, Random rand) {
    this.dekodler = dekodler;
    this.bitovi = new byte[dekoder.ukupnoBitova];
    this.fitnes = 0;
    this.varijable = new double[dekoder.brojVarijabli];
    for(int i = 0; i < dekodler.ukupnoBitova; i++) {
        this.bitovi[i] = rand.nextBoolean() ? (byte)1 : (byte)0;
    }
}

/**
 * Funkcija za definiranje prirodnog poretka kromosoma. Pretpostavka
 * je da se radi minimizacijski problem pa je manji (bolji) onaj kromosom
 * koji ima manju vrijednost {@linkplain #fitnes} koja zapravo čuva vrijednost
 * funkcije u promatranoj točki.
 *
 * @see java.lang.Comparable#compareTo(java.lang.Object)
 */
@Override
public int compareTo(Kromosom o) {
    if(this.fitnes < o.fitnes) {
        return -1;
    }
    if(this.fitnes > o.fitnes) {
        return 1;
    }
    return 0;
}
}

```

### 7.1.3. Razred KromosomDekoder

```

package hr.fer.zemris.ga;

/**
 * Razred koji predstavlja dekodler binarnog kromosoma. Temeljem informacija
 * o minimalnim i maksimalnim vrijednostima varijabli te broju varijabli konvertira
 * niz bitova kromosoma u vrijednosti varijabli.
 *
 * @author marcupic
 */
public class KromosomDekoder {
    // Donje granice varijabli
    double[] xMin;
    // Gornje granice varijabli
    double[] xMax;
    // Broj bitova koji se troši na svaku varijablu
    int[] bitova;
    // Koji je najveći binarni broj pridijeljen svakoj varijabli
    int[] najvećiBinarniBroj;
    // Koliko ukupno bitova ima kromosom
    int ukupnoBitova;
    // Koliko varijabli predstavlja kromosom
    int brojVarijabli;

    /**
     * Konstruktor dekodera kromosoma.
     *
     * @param brojVarijabli broj varijabli
     * @param brojBitovaPoVarijabli broj bitova koji će biti korišten za svaku
     varijablu
     * @param xMin donja granica (pretpostavka je da sve varijable imaju istu granicu)
     * @param xMax gornja granica (pretpostavka je da sve varijable imaju istu granicu)
     */
    public KromosomDekoder(int brojVarijabli, int brojBitovaPoVarijabli, double xMin,
    double xMax) {
        this.brojVarijabli = brojVarijabli;
        this.xMin = new double[brojVarijabli];
        this.xMax = new double[brojVarijabli];
        this.bitova = new int[brojVarijabli];
        this.najvećiBinarniBroj = new int[brojVarijabli];
        for(int i = 0; i < brojVarijabli; i++) {
            this.xMin[i] = xMin;
            this.xMax[i] = xMax;
            this.bitova[i] = brojBitovaPoVarijabli;
        }
    }
}

```

```
        this.najveciBinarniBroj[i] = (1 << brojBitovaPoVarijabli) - 1;
    }
    this.ukupnoBitova = brojBitovaPoVarijabli * brojVarijabli;
}

/**
 * Funkcija obavlja dekodiranje predanog kromosoma. Temeljem bitova u kromosomu
 * obavlja izračun stvarnih vrijednosti koje ti bitovi predstavljaju, i u kromosomu
 * popunjava polje {@linkplain Kromosom#varijable}.<br>
 * Napomena: ova funkcija ne poziva automatski i izračun dobrote kromosoma u
 * zadanoj točki; to treba obaviti naknadno.
 *
 * @param k kromosom koji treba dekodirati
 */
public void dekodirajKromosom(Kromosom k) {
    int indeksBita = 0;
    for(int brojVarijable = 0; brojVarijable < brojVarijabli; brojVarijable++) {
        int prviBit = indeksBita;
        int zadnjiBit = prviBit + bitova[brojVarijable] - 1;
        indeksBita += bitova[brojVarijable];
        int binarniBroj = 0;
        for(int i = prviBit; i <= zadnjiBit; i++) {
            binarniBroj = binarniBroj * 2;
            if(k.bitovi[i]==1) {
                binarniBroj = binarniBroj + 1;
            }
        }
        double vrijednostVarijable = (double)binarniBroj /
            ((double)najveciBinarniBroj[brojVarijable] *
            (xMax[brojVarijable]-xMin[brojVarijable]) + xMin[brojVarijable]);
        k.varijable[brojVarijable] = vrijednostVarijable;
    }
}
}
```



## 7.2. Mravlji algoritmi

Algoritmi su smješteni u razred `hr.fer.zemris.aco`.

### 7.2.1. Razred *SimpleACO*

```

package hr.fer.zemris.aco;

import hr.fer.zemris.graphics.tsp.PrepareTSP;
import hr.fer.zemris.tsp.City;
import hr.fer.zemris.tsp.TSPUtil;
import hr.fer.zemris.tsp.TSPSolution;
import hr.fer.zemris.util.ArraysUtil;

import java.io.IOException;
import java.util.List;
import java.util.Random;

/**
 * Jednostavni mravlji algoritam koji ne koristi
 * heurističku informaciju.
 *
 * @author marcupic
 */
public class SimpleACO {

    // Polje gradova
    private City[] cities;

    // Generator slučajnih brojeva
    private Random rand;

    // Polje indeksa radova (uvijek oblika 0, 1, 2, 3, ...).
    private int[] indexes;

    // Feromonski tragovi - simetrična matrica
    private double[][] trails;

    // Udaljenosti između gradova - simetrična matrica
    private double[][] distances;

    // Populacija mrava koji rješavaju problem
    private TSPSolution[] ants;

    // Pomoćno polje indeksa mravu dostupnih gradova
    private int[] reachable;

    // Pomoćno polje vjerojatnosti odabira grada
    private double[] probabilities;

    // Konstanta isparavanja
    private double ro;

    // Pomoćno rješenje koje pamti najbolju pronađenu turu - ikada.
    private TSPSolution best;
    private boolean haveBest = false;

    /**
     * Konstruktor.
     *
     * @param cities lista gradova
     */
    public SimpleACO(List<City> cities) {
        this.cities = new City[cities.size()];
        cities.toArray(this.cities);
        ro = 0.2;
        rand = new Random();
        indexes = new int[this.cities.length];
        ArraysUtil.linearFillArray(indexes);
        probabilities = new double[this.cities.length];
        reachable = new int[this.cities.length];
        distances = new double[this.cities.length][this.cities.length];
        trails = new double[this.cities.length][this.cities.length];
    }

```

```

double initTrail = 1.0/5000.0;
int m = 30;
for(int i = 0; i < this.cities.length; i++) {
    City a = this.cities[i];
    distances[i][i] = 0;
    trails[i][i] = initTrail;
    for(int j = i+1; j < this.cities.length; j++) {
        City b = this.cities[j];
        double dist = Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
        distances[i][j] = dist;
        distances[j][i] = dist;
        trails[i][j] = initTrail;
        trails[j][i] = initTrail;
    }
}
ants = new TSPSolution[m];
for(int i = 0; i < ants.length; i++) {
    ants[i] = new TSPSolution();
    ants[i].cityIndexes = new int[this.cities.length];
}
best = new TSPSolution();
best.cityIndexes = new int[this.cities.length];
}

/**
 * Glavna metoda algoritma.
 */
public void go() {
    int iter = 0;
    int iterLimit = 500;

    // ponavlja dozvoljeni broj puta
    while(iter < iterLimit) {
        iter++;
        // Za svakog mrava iz populacije
        for(int antIndex = 0; antIndex < ants.length; antIndex++) {
            // S kojim mravom radim?
            TSPSolution ant = ants[antIndex];
            doWalk(ant);
        }
        updateTrails();
        evaporateTrails();
        checkBestSolution();
    }
    System.out.println("Best length: "+best.tourLength);
    System.out.println(best);
    PrepareTSP.visualize(TSPUtil.reorderCities(cities, best.cityIndexes));
}

/**
 * Metoda koja obavlja hod jednog mrava.
 *
 * @param ant mrav
 */
private void doWalk(TSPSolution ant) {
    // Svi su gradovi dostupni
    System.arraycopy(indexes, 0, reachable, 0, indexes.length);
    // Permutirajmo redosljed gradova tako da krenemo iz slučajnog
    ArraysUtil.shuffleArray(reachable, rand);
    // Neka je prvi grad fiksiran
    ant.cityIndexes[0] = reachable[0];
    // trebamo utvrditi kamo iz drugoga pa na dalje:
    for(int step = 1; step < cities.length-1; step++) {
        int previousCityIndex = ant.cityIndexes[step-1];
        // Koji grad biram u koraku "step"?
        // Mogu ici u sve gradove od step do cities.length-1
        double probSum = 0.0;
        for(int candidate = step; candidate < cities.length; candidate++) {
            int cityIndex = reachable[candidate];
            probabilities[cityIndex] = trails[previousCityIndex][cityIndex];
            probSum += probabilities[cityIndex];
        }
        // Normalizacija vjerojatnosti:
        for(int candidate = step; candidate < cities.length; candidate++) {
            int cityIndex = reachable[candidate];
            probabilities[cityIndex] = probabilities[cityIndex] / probSum;
        }
    }
}

```

```

// Odluka kuda dalje?
double number = rand.nextDouble();
probSum = 0.0;
int selectedCandidate = -1;
for(int candidate = step; candidate < cities.length; candidate++) {
    int cityIndex = reachable[candidate];
    probSum += probabilities[cityIndex];
    if(number <= probSum) {
        selectedCandidate = candidate;
        break;
    }
}
if(selectedCandidate===-1) {
    selectedCandidate = cities.length-1;
}
int tmp = reachable[step];
reachable[step] = reachable[selectedCandidate];
reachable[selectedCandidate] = tmp;
ant.cityIndexes[step] = reachable[step];
}
ant.cityIndexes[ant.cityIndexes.length-1] = reachable[ant.cityIndexes.length-1];
TSPUtil.evaluate(ant, distances);
}

/**
 * Metoda koja obavlja ažuriranje feromonskih tragova
 */
private void updateTrails() {
    // Koliko mravaca radi ažuriranje feromona?
    int updates = ants.length;
    // Ako zelim samo da najbolji rade update...
    if(true) {
        updates = 5;
        //ili updates = ants.length / 10;
        TSPUtil.partialSort(ants, updates);
    }
    // Azuriranje feromonskog traga:
    for(int antIndex = 0; antIndex < updates; antIndex++) {
        // S kojim mravom radim?
        TSPSolution ant = ants[antIndex];
        double delta = 1.0 / ant.tourLength;
        for(int i = 0; i < ant.cityIndexes.length-1; i++) {
            int a = ant.cityIndexes[i];
            int b = ant.cityIndexes[i+1];
            trails[a][b] += delta;
            trails[b][a] = trails[a][b];
        }
    }
}

/**
 * Metoda koja obavlja isparavanje feromonskih tragova.
 */
private void evaporateTrails() {
    // Isparavanje feromonskog traga
    for(int i = 0; i < this.cities.length; i++) {
        for(int j = i+1; j < this.cities.length; j++) {
            trails[i][j] = trails[i][j]*(1-ro);
            trails[j][i] = trails[i][j];
        }
    }
}

/**
 * Metoda provjerava je li pronađeno bolje rješenje od
 * prethodno najboljeg.
 */
private void checkBestSolution() {
    // Nadi najbolju rutu
    if(!haveBest) {
        haveBest = true;
        TSPSolution ant = ants[0];
        System.arraycopy(
            ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
        best.tourLength = ant.tourLength;
    }
    double currentBest = best.tourLength;
}

```

```
int bestIndex = -1;
for(int antIndex = 0; antIndex < ants.length; antIndex++) {
    TSPSolution ant = ants[antIndex];
    if(ant.tourLength < currentBest) {
        currentBest = ant.tourLength;
        bestIndex = antIndex;
    }
}
if(bestIndex!=-1) {
    TSPSolution ant = ants[bestIndex];
    System.arraycopy(
        ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
    best.tourLength = ant.tourLength;
}
}

/**
 * Ulazna točka u program.
 *
 * @param args argumenti komandne linije
 */
public static void main(String[] args) throws IOException {
    String fileName = args.length<1 ?
        "data/gradovi01.txt"
        : args[0];
    List<City> cities = TSPUtil.loadCities(fileName);
    if(cities==null) return;
    new SimpleACO(cities).go();
}
}
```

### 7.2.2. Razred AntSystem

```
package hr.fer.zemris.aco;

import hr.fer.zemris.graphics.tsp.PrepareTSP;
import hr.fer.zemris.tsp.City;
import hr.fer.zemris.tsp.TSPUtil;
import hr.fer.zemris.tsp.TSPSolution;
import hr.fer.zemris.util.ArraysUtil;

import java.io.IOException;
import java.util.List;
import java.util.Random;

/**
 * Razred modelira rad algoritma AntSystem na problemu trgovačkog
 * putnika.
 *
 * @author marcupic
 */
public class AntSystem {

    // Polje gradova
    private City[] cities;

    // Generator slučajnih brojeva
    private Random rand;

    // Polje indeksa radova (uvijek oblika 0, 1, 2, 3, ...).
    private int[] indexes;

    // Feromonski tragovi - simetrična matrica
    private double[][] trails;

    // Udaljenosti između gradova - simetrična matrica
    private double[][] distances;

    // Heurističke vrijednosti
    private double[][] heuristics;

    // Populacija mrava koji rješavaju problem
    private TSPSolution[] ants;
```

```

// Pomoćno polje indeksa mravu dostupnih gradova
private int[] reachable;

// Pomoćno polje vjerojatnosti odabira grada
private double[] probabilities;

// Konstanta isparavanja
private double ro;

// Konstanta alfa
private double alpha;

// Konstanta beta
private double beta;

// Pomoćno rješenje koje pamti najbolju pronađenu turu - ikada.
private TSPSolution best;
private boolean haveBest = false;

/**
 * Konstruktor.
 *
 * @param cities lista gradova
 */
public AntSystem(List<City> cities) {
    this.cities = new City[cities.size()];
    cities.toArray(this.cities);
    ro = 0.2;
    rand = new Random();
    indexes = new int[this.cities.length];
    ArraysUtil.linearFillArray(indexes);
    probabilities = new double[this.cities.length];
    reachable = new int[this.cities.length];
    distances = new double[this.cities.length][this.cities.length];
    heuristics = new double[this.cities.length][this.cities.length];
    trails = new double[this.cities.length][this.cities.length];
    double initTrail = 1.0/5000.0;
    int m = 30;
    alpha = 3;
    beta = 2;
    for(int i = 0; i < this.cities.length; i++) {
        City a = this.cities[i];
        distances[i][i] = 0;
        trails[i][i] = initTrail;
        for(int j = i+1; j < this.cities.length; j++) {
            City b = this.cities[j];
            double dist = Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
            distances[i][j] = dist;
            distances[j][i] = dist;
            trails[i][j] = initTrail;
            trails[j][i] = initTrail;
            heuristics[i][j] = Math.pow(1.0 / dist, beta);
            heuristics[j][i] = heuristics[i][j];
        }
    }
    ants = new TSPSolution[m];
    for(int i = 0; i < ants.length; i++) {
        ants[i] = new TSPSolution();
        ants[i].cityIndexes = new int[this.cities.length];
    }
    best = new TSPSolution();
    best.cityIndexes = new int[this.cities.length];
}

/**
 * Glavna metoda algoritma.
 */
public void go() {
    System.out.println("Zapocinjem s populacijom:");
    System.out.println("=====");
    int iter = 0;
    int iterLimit = 500;

    // ponavljaj dozvoljeni broj puta
    while(iter < iterLimit) {
        iter++;
        for(int antIndex = 0; antIndex < ants.length; antIndex++) {

```

```

        // S kojim mravom radim?
        TSPSolution ant = ants[antIndex];
        doWalk(ant);
    }
    updateTrails();
    evaporateTrails();
    checkBestSolution();
}
System.out.println("Best length: "+best.tourLength);
System.out.println(best);
PrepareTSP.visualize(TSPUtil.reorderCities(cities, best.cityIndexes));
}

/**
 * Metoda koja obavlja hod jednog mrava.
 *
 * @param ant mrav
 */
private void doWalk(TSPSolution ant) {
    // Svi su gradovi dostupni
    System.arraycopy(indexes, 0, reachable, 0, indexes.length);
    // Permutirajmo redosljed gradova tako da krenemo iz slučajnog
    ArraysUtil.shuffleArray(reachable, rand);
    // Neka je prvi grad fiksiran
    ant.cityIndexes[0] = reachable[0];
    // trebamo utvrditi kamo iz drugoga pa na dalje:
    for(int step = 1; step < cities.length-1; step++) {
        int previousCityIndex = ant.cityIndexes[step-1];
        // Koji grad biram u koraku "step"?
        // Mogu ici u sve gradove od step do cities.length-1
        double probSum = 0.0;
        for(int candidate = step; candidate < cities.length; candidate++) {
            int cityIndex = reachable[candidate];
            probabilities[cityIndex] =
                Math.pow(trails[previousCityIndex][cityIndex], alpha) *
                heuristics[previousCityIndex][cityIndex];
            probSum += probabilities[cityIndex];
        }
        // Normalizacija vjerojatnosti:
        for(int candidate = step; candidate < cities.length; candidate++) {
            int cityIndex = reachable[candidate];
            probabilities[cityIndex] = probabilities[cityIndex] / probSum;
        }
        // Odluka kuda dalje?
        double number = rand.nextDouble();
        probSum = 0.0;
        int selectedCandidate = -1;
        for(int candidate = step; candidate < cities.length; candidate++) {
            int cityIndex = reachable[candidate];
            probSum += probabilities[cityIndex];
            if(number <= probSum) {
                selectedCandidate = candidate;
                break;
            }
        }
        if(selectedCandidate===-1) {
            selectedCandidate = cities.length-1;
        }
        int tmp = reachable[step];
        reachable[step] = reachable[selectedCandidate];
        reachable[selectedCandidate] = tmp;
        ant.cityIndexes[step] = reachable[step];
    }
    ant.cityIndexes[ant.cityIndexes.length-1] = reachable[ant.cityIndexes.length-1];
    TSPUtil.evaluate(ant, distances);
}

/**
 * Metoda koja obavlja ažuriranje feromonskih tragova
 */
private void updateTrails() {
    // Koliko mrava radi ažuriranje?
    int updates = ants.length;
    // Ako zelim samo da najbolji rade ažuriranje...
    if(false) {
        updates = 5;
        //ili updates = ants.length / 10;
    }
}

```

```

        TSPUtil.partialSort(ants, updates);
    }
    // Azuriranje feromonskog traga:
    for(int antIndex = 0; antIndex < updates; antIndex++) {
        // S kojim mravom radim?
        TSPSolution ant = ants[antIndex];
        double delta = 1.0 / ant.tourLength;
        for(int i = 0; i < ant.cityIndexes.length-1; i++) {
            int a = ant.cityIndexes[i];
            int b = ant.cityIndexes[i+1];
            trails[a][b] += delta;
            trails[b][a] = trails[a][b];
        }
    }
}

/**
 * Metoda koja obavlja isparavanje feromonskih tragova.
 */
private void evaporateTrails() {
    // Isparavanje feromonskog traga
    for(int i = 0; i < this.cities.length; i++) {
        for(int j = i+1; j < this.cities.length; j++) {
            trails[i][j] = trails[i][j]*(1-ro);
            trails[j][i] = trails[i][j];
        }
    }
}

/**
 * Metoda provjerava je li pronađeno bolje rješenje od
 * prethodno najboljeg.
 */
private void checkBestSolution() {
    // Nadi najbolju rutu
    if(!haveBest) {
        haveBest = true;
        TSPSolution ant = ants[0];
        System.arraycopy(
            ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
        best.tourLength = ant.tourLength;
    }
    double currentBest = best.tourLength;
    int bestIndex = -1;
    for(int antIndex = 0; antIndex < ants.length; antIndex++) {
        TSPSolution ant = ants[antIndex];
        if(ant.tourLength < currentBest) {
            currentBest = ant.tourLength;
            bestIndex = antIndex;
        }
    }
    if(bestIndex!=-1) {
        TSPSolution ant = ants[bestIndex];
        System.arraycopy(
            ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
        best.tourLength = ant.tourLength;
    }
}

/**
 * Ulazna točka u program.
 */
@params argumentsi komandne linije
*/
public static void main(String[] args) throws IOException {
    String fileName = args.length<1 ?
        "data/gradovi01.txt"
        : args[0];
    List<City> cities = TSPUtil.loadCities(fileName);
    if(cities==null) return;
    new AntSystem(cities).go();
}
}

```

### 7.3. Algoritam roja čestica

Algoritam roja čestica sastoji se od nekoliko razreda smještenih u paket `hr.fer.zemris.pso`. Razred `ParticleSwarmOptimization` je jezgra algoritma. Razred `Particle` predstavlja jednu česticu (rješenje). Sučelje `Neighborhood` opisuje pojam susjedstva, a implementirana su dva: globalno susjedstvo u razredu `GlobalNeighborhood` te lokalno susjedstvo u razredu `LocalNeighborhood`.

#### 7.3.1. Razred `ParticleSwarmOptimization`

```
package hr.fer.zemris.pso;

import hr.fer.zemris.numeric.IFunkcija;

import java.text.DecimalFormat;
import java.util.Random;

/**
 * Primjer uporabe algoritma roja čestica za optimizaciju.
 *
 * @author marcupic
 */
public class ParticleSwarmOptimization {

    // Funkcija koju optimiramo
    private IFunkcija funkcija;

    // Mimimumi varijabli u prostoru pretraživanja
    private double[] varMin;

    // Maksimumi varijabli u prostoru pretraživanja
    private double[] varMax;

    // Maksimalna promjene varijable odjednom, za svaku dimenziju
    private double[] velBounds;

    // Postotak raspona prostora pretraživanja koji se koristi za
    // izračun ograničenja pomaka u jednom koraku
    double velBoundsPercentage;

    // Globalni brojač iteracija
    private int iteracija;

    // Težina koju korisimo na početku
    private double linWeightStart;

    // Težina koju korisimo na kraju smanjivanja
    private double linWeightEnd;

    // Iteracija u kojoj težina pada na krajnju i dalje se ne mijenja
    private int linWeightTreshold;

    // Broj čestica s kojima radimo
    int VEL_POP;

    // Generator slučajnih brojeva
    Random rand;

    // Broj dimenzija funkcije
    int dims;

    // Konstanta c1
    double c1;

    // Konstanta c2
    double c2;

    // Čestice
    Particle[] particles;
```



```

// Susjedstvo
Neighborhood neighborhood;

// Pomoć u formatiranu brojeva
DecimalFormat df = new DecimalFormat("0.00000");

/**
 * Glavni program.
 *
 * @param args argumenti komandne linije - ne koriste se.
 */
public static void main(String[] args) throws Exception {
    final ParticleSwarmOptimization t = new ParticleSwarmOptimization();
    t.go();
}

/**
 * Konstruktor.
 */
public ParticleSwarmOptimization() {

    // Definiranje parametara algoritma
    VEL_POP = 20;
    dims = 2;
    c1 = 2;
    c2 = 2.5;

    iteracija = 0;
    linWeightStart = 0.9;
    linWeightEnd = 0.4;
    linWeightTreshold = 50;

    // Definiranje minimalnih i maksimalnih vrijednosti po dimenzijama,
    // te definiranje maksimalne promjene varijable u jednom koraku.
    varMin = new double[dims];
    varMax = new double[dims];
    velBounds = new double[dims];
    velBoundsPercentage = 0.05; // dozvoli pomak od 5% raspona
    for(int d = 0; d < dims; d++) {
        varMin[d] = -5;
        varMax[d] = 5;
        velBounds[d] = (varMax[d]-varMin[d])*velBoundsPercentage;
    }

    // Definiranje funkcije koju optimiramo.
    funkcija = new IFunkcija() {
        @Override
        public double izracunaj(double[] varijable) {
            int n = varijable.length;
            double vrijednost = 10*n;
            for(int i = 0; i < n; i++) {
                vrijednost += varijable[i]*varijable[i]
                    - 10*Math.cos(2*Math.PI*varijable[i]);
            }
            return vrijednost;
        }
    };

    // Generator slučajnih brojeva.
    rand = new Random();

    // Inicijalizacija
    particles = new Particle[VEL_POP];
    for(int i = 0; i < VEL_POP; i++) {
        particles[i] = new Particle(dims);
        for(int d = 0; d < dims; d++) {
            particles[i].vars[d] = rand.nextDouble()*(varMax[d]-varMin[d])+varMin[d];
            particles[i].oldVars[d] = particles[i].vars[d];
            particles[i].bestVars[d] = particles[i].vars[d];
            particles[i].velocity[d] = rand.nextDouble() *
                (2*velBounds[d])-velBounds[d];
        }
        particles[i].bestValue = funkcija.izracunaj(particles[i].vars);
    }

    // Definiranje susjedstva koje koristimo - lokalno veličine 5

```

```
neighborhood = new LocalNeighborhood(VEL_POP, dims, 5, true);

// Za odabir globalnog susjedstva može se iskoristiti sljedeće:
//neighborhood = new GlobalNeighborhood(VEL_POP, dims, true);

// Ispiši inicijalnu statistiku.
updateStatistics();
}

/**
 * Glavna metoda optimizacijskog algoritma.
 */
public void go() {
    // Ponavlja zadani broj puta
    for(int iter = 0; iter < 100; iter++) {
        nextIteration();
    }
}

/**
 * Jedna iteracija algoritma PSO.
 */
protected void nextIteration() {
    iteracija++;

    // Koju težinu koristimo? Težina linearno pada s iteracijama do
    // neke male zadane, i dalje ostaje konstantna.
    double w;
    if(iteracija > linWeightTreshold) {
        w = linWeightEnd;
    } else {
        w = linWeightStart + (linWeightEnd - linWeightStart) *
            (iteracija - 1.0) / linWeightTreshold;
    }

    // Ažurirajmo "znanje" susjedstva
    neighborhood.scan(particles);

    // Ažuriraj pozicije i brzine svake čestice
    for(int i = 0; i < particles.length; i++) {
        double[] socialBest = neighborhood.findBest(i);
        for(int d = 0; d < dims; d++) {
            particles[i].oldVars[d] = particles[i].vars[d];
            particles[i].velocity[d] =
                w * particles[i].velocity[d]
                + c1 * rand.nextDouble() * (particles[i].bestVars[d] - particles[i].vars[d])
                + c2 * rand.nextDouble() * (socialBest[d] - particles[i].vars[d]);
            ;
            if(particles[i].velocity[d] < -velBounds[d]) {
                particles[i].velocity[d] = -velBounds[d];
            } else if(particles[i].velocity[d] > velBounds[d]) {
                particles[i].velocity[d] = velBounds[d];
            }
            particles[i].vars[d] = particles[i].vars[d] + particles[i].velocity[d];
        }
    }

    // Izračunaj vrijednost funkcije u novim pozicijama svih čestica,
    // i po potrebi ažuriraj najbolje rješenje čestice
    for(int i = 0; i < particles.length; i++) {
        particles[i].value = funkcija.izracunaj(particles[i].vars);
        if(particles[i].value < particles[i].bestValue) {
            particles[i].bestValue = particles[i].value;
            for(int d = 0; d < dims; d++) {
                particles[i].bestVars[d] = particles[i].vars[d];
            }
        }
    }

    // Ispiši statistiku na ekran
    updateStatistics();
}

/**
 * Pomoćna metode koja na ekran ispisuje statističke podatke o populaciji,
 * te najbolje pronađeno rješenje.
 */
}
```

```

private void updateStatistics() {
    int bestindex = 0;
    double bestValue = particles[bestindex].bestValue;
    double sum = bestValue;
    for(int i = 1; i < particles.length; i++) {
        if(particles[i].bestValue < bestValue) {
            bestValue = particles[i].bestValue;
            bestindex = i;
        }
        sum += particles[i].bestValue;
    }
    StringBuilder sb = new StringBuilder();
    sb.append("Iter: ");
    sb.append(iteracija);
    sb.append(", Average: ");
    sb.append(df.format(sum/particles.length));
    sb.append(", (");
    for(int d = 0; d < dims; d++) {
        if(d>0) {
            sb.append(", ");
        }
        sb.append(df.format(particles[bestindex].bestVars[d]));
    }
    sb.append(")=");
    sb.append(df.format(bestValue));
    System.out.println(sb.toString());
}
}

```

### 7.3.2. Razred Particle

```

package hr.fer.zemris.pso;

/**
 * Čestica algoritma PSO.
 *
 * @author marcupic
 */
public class Particle {

    // Najbolje rješenje
    double[] bestVars;
    // Dobrota najboljeg rješenja
    double bestValue;
    // Prethodno rješenje
    double[] oldVars;
    // Trenutno rješenje
    double[] vars;
    // Vektor brzine
    double[] velocity;
    // Vrijednost funkcije u trenutnom rješenju
    double value;

    /**
     * Konstruktor čestice. Prima broj dimenzija.
     *
     * @param dimensions broj dimenzija prostora
     */
    public Particle(int dimensions) {
        vars = new double[dimensions];
        oldVars = new double[dimensions];
        velocity = new double[dimensions];
        bestVars = new double[dimensions];
    }
}

```

### 7.3.3. Sučelje Neighborhood

```

package hr.fer.zemris.pso;

/**
 * Sučelje koje apstrahira pojam susjedstva.

```

```
* Različite implementacije ponudit će konkretne
* definicije susjedstva.
*
* @author marcupic
*
*/
public interface Neighborhood {

    /**
     * Metoda koja se mora pozvati nad populacijom
     * kako bi se napunili podaci o rješenjima iz
     * susjedstva. Ovo mora biti napravljeno prije
     * uporabe funkcije {@linkplain #findBest(int)}
     * i svakako prije bilo kakvih izmjena u česticama.
     *
     * @param particles populacija čestica
     */
    void scan(Particle[] particles);

    /**
     * Metoda koja za česticu određenu indeksom vraća
     * poziciju najboljeg rješenja pronađenog u njezinom
     * susjedstvu.
     *
     * @param forIndex indeks čestice
     * @return najbolje rješenje u susjedstvu te čestice
     */
    double[] findBest(int forIndex);
}
```

### 7.3.4. Razred *GlobalNeighborhood*

```
package hr.fer.zemris.pso;

/**
 * Razred implementira pojam globalnog susjedstva. Kod ove
 * vrste susjedstva, svaka čestica svjesna je najboljeg rješenja
 * koje je pronašla cjelokupna populacija.
 *
 * @author marcupic
 */
public class GlobalNeighborhood implements Neighborhood {

    // Broj čestica
    int particlesCount;

    // Dimezija prostora
    int dims;

    // Najbolje globalno rješenje
    double[] best;

    // Radi li se minimizacija (true) ili maksimizacija (false)
    boolean minimize;

    /**
     * Konstruktor susjedstava.
     *
     * @param particlesCount broj čestica
     * @param dims dimenzija
     * @param minimize true ako se radi minimizacija, false inače
     */
    public GlobalNeighborhood(int particlesCount, int dims, boolean minimize) {
        this.particlesCount = particlesCount;
        this.dims = dims;
        this.minimize = minimize;
        best = new double[dims];
    }

    /**
     * Pronalazi globalno najbolje rješenje populacije.
     *
     * @see hr.fer.zemris.pso.Neighborhood#scan(hr.fer.zemris.pso.Particle[])
     */
    @Override
```

```

public void scan(Particle[] particles) {
    int bestindex = 0;
    double bestValue = particles[bestindex].bestValue;
    for(int i = 1; i < particles.length; i++) {
        if(particles[i].bestValue < bestValue) {
            bestValue = particles[i].bestValue;
            bestindex = i;
        }
    }
    for(int d = 0; d < dims; d++) {
        best[d] = particles[bestindex].bestVars[d];
    }
}

/**
 * Vraća najbolje rješenje za zadanu česticu.
 * @see hr.fer.zemris.pso.Neighborhood#findBest(int)
 */
@Override
public double[] findBest(int forIndex) {
    return best;
}
}

```

### 7.3.5. Razred LocalNeighborhood

```

package hr.fer.zemris.pso;

/**
 * Razred implementira pojam lokalnog susjedstva određene širine.
 *
 * @author marcupic
 */
public class LocalNeighborhood implements Neighborhood {

    // Broj čestica
    int particlesCount;

    // Dimezija prostora
    int dims;

    // Veličina susjedstva
    int nSize;

    // Najbolja rješenje za susjedstvo svake čestice
    double[][] best;

    // Radi li se minimizacija (true) ili maksimizacija (false)
    boolean minimize;

    /**
     * Konstruktor.
     *
     * @param particlesCount broj čestica
     * @param dims broj dimenzija
     * @param nSize veličina susjedstva
     * @param minimize true ako se radi minimizacija, false inače
     */
    public LocalNeighborhood(int particlesCount, int dims, int nSize,
        boolean minimize) {
        this.particlesCount = particlesCount;
        this.dims = dims;
        this.minimize = minimize;
        this.nSize = nSize;
        best = new double[particlesCount][dims];
    }

    /**
     * Pronalazi najbolja rješenja susjedstva za sve jedinke.
     *
     * @see hr.fer.zemris.pso.Neighborhood#scan(hr.fer.zemris.pso.Particle[])
     */
    @Override
    public void scan(Particle[] particles) {
        for(int index = 0; index < particles.length; index++) {

```

```
    int startFrom = index - nSize/2;
    int endAt = index + nSize/2;
    if(startFrom < 0) startFrom = 0;
    if(endAt >= particles.length) endAt = particles.length-1;

    int bestindex = startFrom;
    double bestValue = particles[bestindex].bestValue;
    for(int i = startFrom+1; i <= endAt; i++) {
        if(particles[i].bestValue < bestValue) {
            bestValue = particles[i].bestValue;
            bestindex = i;
        }
    }
    for(int d = 0; d < dims; d++) {
        best[index][d] = particles[bestindex].bestVars[d];
    }
}

/**
 * Vraća najbolje rješenje za zadanu česticu.
 * @see hr.fer.zemris.pso.Neighborhood#findBest(int)
 */
@Override
public double[] findBest(int index) {
    return best[index];
}
}
```

## 7.4. Algoritmi umjetnog imunološkog sustava

Ovi algoritmi smješteni su u paket `hr.fer.zemris.ais`. Napravljene su dvije implementacije. Razred `SimpleIA` sadrži implementaciju jednostavnog imunološkog algoritma dok razred `ClonAlg` sadrži implementaciju istoimenog algoritma.

### 7.4.1. Razred `SimpleIA`

```
package hr.fer.zemris.ais;

import hr.fer.zemris.graphics.tsp.PrepareTSP;
import hr.fer.zemris.tsp.City;
import hr.fer.zemris.tsp.TSPSolution;
import hr.fer.zemris.tsp.TSPSolutionPool;
import hr.fer.zemris.tsp.TSPUtil;

import java.io.IOException;
import java.util.List;
import java.util.Random;

/**
 * Razred prikazuje implementaciju algoritma SimpleIA (jednostavan
 * imunološki algoritam) na problemu trgovačkog putnika.
 *
 * @author marcupic
 */
public class SimpleIA {

    // Polje gradova
    private City[] cities;

    // veličina populacije (broj antitijela)
    private int paramD;

    // broj klonova svakog rješenja (antitijela)
    private int paramDup;

    // priručna memorija s rješenjima
    private TSPSolutionPool pool;

    // Populacija rješenja
    private TSPSolution[] population;

    // Populacija klonova
    private TSPSolution[] clonedPopulation;

    // Unija obiju populacija
    private TSPSolution[] unionPopulation;

    // Generator slučajnih brojeva
    private Random rand;

    /**
     * Konstruktor.
     *
     * @param cities lista gradova
     */
    public SimpleIA(List<City> cities) {
        this.cities = new City[cities.size()];
        cities.toArray(this.cities);
        paramD = 50;
        paramDup = 30;
        this.pool = new TSPSolutionPool(this.cities.length);
        rand = new Random();
        population = new TSPSolution[paramD];
        clonedPopulation = new TSPSolution[paramD*paramDup];
        unionPopulation = new TSPSolution[paramD*paramDup + paramD];
        initialize();
        TSPUtil.evaluate(population, this.cities);
    }
}
```

```
/**
 * Inicijalizacija rješenja (antitijela).
 */
private void initialize() {
    for(int i = 0; i < paramD; i++) {
        TSPSolution s = pool.get();
        population[i] = s;
        TSPUtil.randomInitializeSolution(s, rand);
    }
}

/**
 * Glavna petlja optimizacijskog algoritma.
 */
public void go() {

    // Postavi parametre
    int iter = 0;
    int iterLimit = 2000;

    // Ponavljaj zadani broj puta
    while(iter < iterLimit) {
        iter++;
        cloning();
        hyperMutation();
        TSPUtil.evaluate(clonedPopulation, cities);
        select();
    }
    // Najbolje rješenje je prvo zbog sortiranja!
    System.out.println("Best length: "+population[0].tourLength);
    System.out.println(population[0]);
    PrepareTSP.visualize(TSPUtil.reorderCities(cities, population[0].cityIndexes));
}

/**
 * Operator kloniranja. Za svaku jedinku iz glavne populacije stvara
 * zadani broj klonova i time generira populaciju klonova.
 */
private void cloning() {
    int index = 0;
    for(int i = 0; i < population.length; i++) {
        TSPSolution s = population[i];
        for(int j = 0; j < paramDup; j++) {
            TSPSolution c = pool.get();
            System.arraycopy(
                s.cityIndexes, 0, c.cityIndexes, 0, s.cityIndexes.length);
            clonedPopulation[index] = c;
            index++;
        }
    }
}

/**
 * Operator hipermutacije. Svaku jedinku iz populacije klonova mutira
 * tako da zamijeni redosljed dva slučajno odabrana grada.
 */
private void hyperMutation() {
    for(int index = 0; index < clonedPopulation.length; index++) {
        TSPSolution c = clonedPopulation[index];
        int a = rand.nextInt(c.cityIndexes.length);
        int b = rand.nextInt(c.cityIndexes.length);
        if(a==b) {
            if(b==c.cityIndexes.length-1) {
                b--;
            } else {
                b++;
            }
        }
        int tmp = c.cityIndexes[a];
        c.cityIndexes[a] = c.cityIndexes[b];
        c.cityIndexes[b] = tmp;
    }
}

/**
 * Operator selekcije. U uniju dodaje izvornu populaciju i klonove,
```



```

* traži najbolja rješenja za novu populaciju a ostalo otpušta u
* priručnu memoriju.
*/
private void select() {
    int index = 0;
    for(int i = 0; i < population.length; i++) {
        unionPopulation[index++] = population[i];
    }
    for(int i = 0; i < clonedPopulation.length; i++) {
        unionPopulation[index++] = clonedPopulation[i];
    }
    TSPUtil.partialSort(unionPopulation, population.length);
    for(int i = 0; i < population.length; i++) {
        population[i] = unionPopulation[i];
    }
    for(int i = population.length; i < unionPopulation.length; i++) {
        pool.release(unionPopulation[i]);
    }
}

/**
 * Ulazna točka u program.
 *
 * @param args argumenti komandne linije
 */
public static void main(String[] args) throws IOException {
    String fileName = args.length < 1 ?
        "data/gradovi03.txt"
        : args[0];
    List<City> cities = TSPUtil.loadCities(fileName);
    if(cities == null) return;
    new SimpleIA(cities).go();
}
}

```

### 7.4.2. Razred ClonAlg

```

package hr.fer.zemris.ais;

import hr.fer.zemris.graphics.tsp.PrepareTSP;
import hr.fer.zemris.tsp.City;
import hr.fer.zemris.tsp.TSPSolution;
import hr.fer.zemris.tsp.TSPSolutionPool;
import hr.fer.zemris.tsp.TSPUtil;

import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

/**
 * Razred prikazuje implementaciju algoritma ClonAlg
 * na problemu trgovačkog putnika.
 *
 * @author marcupic
 */
public class ClonAlg {

    // Polje gradova
    private City[] cities;

    // Parametar koji određuje veličinu populacije klonova
    private int paramBeta;

    // broj novih rješenja (antitijela) koje ćemo dodavati
    private int paramD;

    // Broj rješenja u populaciji (broj antitijela)
    private int paramN;

    // priručna memorija s rješenjima
    private TSPSolutionPool pool;

    // Populacija rješenja
    private TSPSolution[] population;
}

```

```
// Populacija klonova
private TSPSolution[] clonedPopulation;

// Generator slučajnih brojeva
private Random rand;

// Veličina populacije klonova
private int clonedPopulationSize;

// Rangovi u populaciji klonova
private int[] clonedPopulationRanks;

/**
 * Konstruktor.
 *
 * @param cities lista gradova
 */
public ClonAlg(List<City> cities) {
    this.cities = new City[cities.size()];
    cities.toArray(this.cities);
    paramN = 100;
    paramD = 10;
    paramBeta = 10;
    this.pool = new TSPSolutionPool(this.cities.length);
    rand = new Random();
    population = new TSPSolution[paramN];
    clonedPopulationSize = 0;
    for(int i = 1; i <= paramN; i++) {
        clonedPopulationSize += (int) ((paramBeta*paramN)/((double)i)+0.5);
    }
    clonedPopulation = new TSPSolution[clonedPopulationSize];
    clonedPopulationRanks = new int[clonedPopulationSize];
    initialize();
}

/**
 * Inicijalizacija rješenja (antitijela).
 */
private void initialize() {
    for(int i = 0; i < paramN; i++) {
        TSPSolution s = pool.get();
        population[i] = s;
        TSPUtil.randomInitializeSolution(s, rand);
    }
}

/**
 * Glavna petlja optimizacijskog algoritma.
 */
public void go() {

    // Postavi parametre
    int iter = 0;
    int iterLimit = 2000;

    // Ponavljaj zadani broj puta
    while(iter < iterLimit) {
        iter++;
        TSPUtil.evaluate(population, cities);
        cloning();
        hyperMutation();
        TSPUtil.evaluate(clonedPopulation, cities);
        select();
        birthAndReplace();
    }
    // Najbolje rješenje je prvo zbog sortiranja!
    System.out.println("Best length: "+population[0].tourLength);
    System.out.println(population[0]);
    PrepareTSP.visualize(TSPUtil.reorderCities(cities, population[0].cityIndexes));
}

/**
 * Operator kloniranja koji broj klonova koje će stvoriti utvrđuje proporcionalno
 * dobroći samog rješenja i time generira populaciju klonova.
 */
private void cloning() {
    Arrays.sort(population, TSPUtil.solComparator);
}
```

```

    int index = 0;
    for(int i = 1; i <= population.length; i++) {
        TSPSolution s = population[i-1];
        int copies = (int)((paramBeta*paramN)/((double)i)+0.5);
        for(int j = 0; j < copies; j++) {
            TSPSolution c = pool.get();
            System.arraycopy(
                s.cityIndexes, 0, c.cityIndexes, 0, s.cityIndexes.length);
            clonedPopulation[index] = c;
            clonedPopulationRanks[index] = i;
            index++;
        }
    }
}

/**
 * Operator hipermutacije koji broj mutacija u jedinki ima obrnuto proporcionalan
 * dobroti jedinke. Što je jedinka bolja, to će se manje mutirati. Sama mutacija
 * provodi se tako da se zamijeni redosljed dva slučajno odabrana grada.
 */
private void hyperMutation() {
    // Kada dodem na zadnju jedinku, zelim obaviti 1+c.cityIndexes.length/4 mutacija
    double tau = 3.476 * (population.length-1);
    // Idem od 1 kako bih najbolje rjesenje, i to njegovu prvu kopiju, ostavio
    // netaknuto
    for(int index = 1; index < clonedPopulation.length; index++) {
        TSPSolution c = clonedPopulation[index];
        int rank = clonedPopulationRanks[index]-1;
        int mutations = (int)(1+c.cityIndexes.length*0.25*
            (1-Math.exp(-rank/tau))+0.5);
        for(int attempt = 0; attempt < mutations; attempt++) {
            int a = rand.nextInt(c.cityIndexes.length);
            int b = rand.nextInt(c.cityIndexes.length);
            if(a==b) {
                if(b==c.cityIndexes.length-1) {
                    b--;
                } else {
                    b++;
                }
            }
            int tmp = c.cityIndexes[a];
            c.cityIndexes[a] = c.cityIndexes[b];
            c.cityIndexes[b] = tmp;
        }
    }
}

/**
 * Operator selekcije. Radi nad populacijom klonova i iz nje odabire
 * najbolja rjesenja za novu populaciju. Preostala neiskorištena rjesenja
 * vraćaju se priručnoj memoriji.
 */
private void select() {
    Arrays.sort(clonedPopulation, TSPUtil.solComparator);
    for(int i = 0; i < population.length; i++) {
        pool.release(population[i]);
        population[i] = clonedPopulation[i];
    }
    for(int i = population.length; i < clonedPopulation.length; i++) {
        pool.release(clonedPopulation[i]);
    }
}

/**
 * Operator rađanja - D najgorih rjesenja nanovo slučajno generira.
 */
private void birthAndReplace() {
    int offset = population.length-paramD;
    for(int i=0; i < paramD; i++) {
        TSPUtil.randomInitializeSolution(population[offset+i], rand);
    }
}

/**
 * Ulazna točka u program.
 */

```

```
* @param args argumenti komandne linije
*/
public static void main(String[] args) throws IOException {
    String fileName = args.length < 1 ?
        "data/gradovi01.txt"
        : args[0];
    List<City> cities = TSPUtil.loadCities(fileName);
    if(cities == null) return;
    new ClonAlg(cities).go();
}
}
```

## 7.5. Pomoćni razredi

Svi opisani algoritmi u manjoj ili većoj mjeri oslanjaju se na pomoćne metode smještene u nekoliko razreda.

### 7.5.1. Sučelje IFunkcija

```
package hr.fer.zemris.numeric;

/**
 * Sučelje koje opisuje funkciju koja se
 * optimira.
 *
 * @author marcupic
 */
public interface IFunkcija {

    /**
     * Metoda na temelju varijabli računa vrijednost
     * funkcije.
     *
     * @param varijable varijable
     * @return vrijednost funkcije
     */
    public double izracunaj(double[] varijable);
}
```

### 7.5.2. Razred City

```
package hr.fer.zemris.tsp;

/**
 * Razred predstavlja jedan grad. Grad je određen svojim imenom te
 * koordinatama X i Y.<br>
 * <i>Važno:</i> Varijable {@linkplain #x}, {@linkplain #y}
 * i {@linkplain #name} su javne kako bi se omogućio minimalni
 * "overhead" prilikom izvođenja evolucijskih algoritama. Ovo ima
 * kao ružnu posljedicu da se neopreznim programiranjem vrijednosti
 * mogu mijenjati od bilo kuda, što može dovesti do pogrešnog rada
 * programa!
 *
 * @author marcupic
 */
public class City {
    // Koordinata X
    public int x;
    // Koordinata Y
    public int y;
    // Naziv grada
    public String name;

    /**
     * Konstruktor. Ime se postavlja na null.
     * @param x x koordinata
     * @param y y koordinata
     */
    public City(int x, int y) {
        this(null, x, y);
    }

    /**
     * Konstruktor.
     * @param name ime grada
     * @param x x koordinata
     * @param y y koordinata
     */
    public City(String name, int x, int y) {
        super();
        this.name = name;
        this.x = x;
        this.y = y;
    }
}
```

```
@Override
public String toString() {
    if(name!=null) {
        return name+" ("+x+", "+y+)";
    } else {
        return "City at ("+x+", "+y+)";
    }
}
}
```

### 7.5.3. Razred *TSPSolution*

```
package hr.fer.zemris.tsp;

import java.util.Arrays;

/**
 * Razred koji predstavlja jedno rješenje problema TSP.
 * Razred automatski nudi mogućnosti <i>pool</i>-anja
 * pomoću razreda {@linkplain TSPSolutionPool}.<br>
 * <i>Važno:</i> Varijable {@linkplain #cityIndexes},
 * {@linkplain #tourLength} i {@linkplain #next} su javne kako bi
 * se omogućio minimalni "overhead" prilikom izvođenja evolucijskih
 * algoritama. Ovo ima kao ružnu posljedicu da se neopreznim
 * programiranjem vrijednosti mogu mijenjati od bilo kuda, što može
 * dovesti do pogrešnog rada programa!
 *
 * @author marcupic
 */
public class TSPSolution {
    // Indeksi kojima treba obići gradove
    public int[] cityIndexes;
    // Ukupna duljina ture
    public double tourLength;
    // Sljedeće rješenje; koristi se uz razred TSPSolutionPool
    public TSPSolution next;

    /**
     * Konstruktor.
     */
    public TSPSolution() {
    }

    /**
     * Konstruktor.
     */
    public TSPSolution(TSPSolution next) {
        this.next = next;
    }

    @Override
    public String toString() {
        return Arrays.toString(cityIndexes)+" , len="+tourLength;
    }
}
```

### 7.5.4. Razred *TSPSolutionPool*

```
package hr.fer.zemris.tsp;

/**
 * Pomoćni razred koji služi za iznajmljivanje rješenja.
 * Uporaba ovog razreda preporuča se u slučaju kada algoritam
 * u petlji privremeno stvara veliku količinu novih rješenja
 * i potom ih odbacuje. Neprestana uporaba memorijskog alokatora
 * u takvom bi slučaju bila izuzetno neefikasna.
 * Umjesto toga, ovaj razred objekte iznajmljuje i stvara ih
 * po potrebi. Jednom kada je objekt stvoren, operacija
 * dohvata i vraćanja je o(1).
 *
 * @author marcupic
 */
public class TSPSolutionPool {
    int citiesNumber;
}
```

```

TSPSolution first;

/**
 * Konstruktor.
 *
 * @param citiesNumber broj gradova koji rješenja sadrže
 */
public TSPSolutionPool(int citiesNumber) {
    super();
    this.citiesNumber = citiesNumber;
}

/**
 * Metoda iznajmljuje jedno rješenje.
 *
 * @return rješenje
 */
public TSPSolution get() {
    if(first!=null) {
        TSPSolution s = first;
        first = (TSPSolution)first.next;
        return s;
    }
    TSPSolution s = new TSPSolution();
    s.cityIndexes = new int[citiesNumber];
    return s;
}

/**
 * Metoda preuzima vraćeno rješenje. To rješenje
 * kasnije se može iznajmiti, i originalni vlasnik
 * ga više NE SMIJE koristiti.
 *
 * @param sol rješenje koje se vraća
 */
public void release(TSPSolution sol) {
    sol.next = first;
    first = sol;
}
}

```

### 7.5.5. Razred TSPUtil

```

package hr.fer.zemris.tsp;

import hr.fer.zemris.util.ArraysUtil;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Random;

/**
 * Pomoćni razred koji sadrži metode vezane uz TSP.
 *
 * @author marcupic
 */
public class TSPUtil {

    /**
     * Metoda učitava listu gradova iz datoteke. Datoteka je tekstualna.
     * U svakom retku nalazi se x i y koordinata grada razdvojene znakom
     * tab.
     *
     * @param fileName naziv datoteke
     * @return listu gradova ili null ako dođe do pogreške
     * @throws IOException ako se dogodi pogreška u radu s datotekom
     */
    public static List<City> loadCities(String fileName) throws IOException {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader(fileName));
            List<City> cities = new ArrayList<City>();

```

```

        while(true) {
            String line = br.readLine();
            if(line==null) break;
            line = line.trim();
            if(line.isEmpty()) continue;
            String[] elems = line.split("\\t");
            cities.add(new City(
                Integer.parseInt(elems[0]), Integer.parseInt(elems[1])));
        }
        br.close();
        return cities;
    } catch (IOException ex) {
        System.out.println("Pogreška prilikom rada s datotekom "+fileName);
        if(br!=null) try { br.close(); } catch (Exception ignorable) {}
        return null;
    }
}

/**
 * Metoda koja služi parcijalnom sortiranju predanog polja rješenja TSP-a.
 * Zadatak metode je na početak polja staviti <code>number</code> najboljih
 * rješenja (to su ona s najmanjom duljinom ture); poredak preostalog dijela
 * polja nije bitan.
 *
 * @param population rješenja koja treba parcijalno sortirati
 * @param number broj najboljih rješenja koja treba staviti na početak polja
 */
public static void partialSort(TSPSolution[] population, int number) {
    for(int i = 0; i < number; i++) {
        int best = i;
        for(int j = i+1; j < population.length; j++) {
            if(population[best].tourLength > population[j].tourLength) {
                best = j;
            }
        }
        if(best != i) {
            TSPSolution tmp = population[i];
            population[i] = population[best];
            population[best] = tmp;
        }
    }
}

/**
 * Metoda za predano rješenje računa njegovu duljinu temeljem predane
 * matrice udaljenosti. U toj matrici, na mjestu [i,j] nalazi se udaljenost
 * od grada <code>i</code> do grada <code>j</code>.
 *
 * @param sol rješenje za koje treba izračunati duljinu ture
 * @param distanceMatrix matrica udaljenosti gradova
 */
public static void evaluate(TSPSolution sol, double[][] distanceMatrix) {
    int pocetni = sol.cityIndexes[0];
    double distance = 0;
    for(int i = 1; i < sol.cityIndexes.length; i++) {
        distance += distanceMatrix[pocetni][sol.cityIndexes[i]];
        pocetni = sol.cityIndexes[i];
    }
    distance += distanceMatrix[pocetni][sol.cityIndexes[0]];
    sol.tourLength = distance;
}

/**
 * Metoda za predano rješenje računa njegovu duljinu temeljem predanog
 * polja gradova i indeksa koji se nalaze u samom rješenju.
 *
 * @param sol rješenje
 * @param cities polje gradova
 */
public static void evaluate(TSPSolution sol, City[] cities) {
    int pocetni = sol.cityIndexes[0];
    double distance = 0;
    for(int i = 1; i < sol.cityIndexes.length; i++) {
        City a = cities[pocetni];
        City b = cities[sol.cityIndexes[i]];
        distance += Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
        pocetni = sol.cityIndexes[i];
    }
}

```



```

    }
    City a = cities[pocetni];
    City b = cities[sol.cityIndexes[0]];
    distance += Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
    sol.tourLength = distance;
}

/**
 * Pomoćna metoda koja računa dobrotu svih tura u predanom polju.
 *
 * @param population polje rješenja
 * @param cities polje gradova
 */
public static void evaluate(TSPSolution[] population, City[] cities) {
    for(int i = 0; i < population.length; i++) {
        TSPSolution s = population[i];
        evaluate(s, cities);
    }
}

/**
 * Metoda vraća novo polje gradova ne temelju izvornog polja gradova
 * i predanog redosljeda određenog indeksima.
 *
 * @param cities originalno polje gradova
 * @param indexes željeni poredak gradova
 * @return novo polje složeno prema indeksima
 */
public static City[] reorderCities(City[] cities, int[] indexes) {
    City[] array = new City[indexes.length];
    for(int i = 0; i < indexes.length; i++) {
        City c = cities[indexes[i]];
        array[i] = new City(c.x, c.y);
    }
    return array;
}

/**
 * Pomoćna metoda koja stvara nasumični poredak gradova.
 *
 * @param sol rješenje
 * @param rand generator slučajnih brojeva
 */
public static void randomInitializeSolution(TSPSolution sol, Random rand) {
    ArraysUtil.linearFillArray(sol.cityIndexes);
    ArraysUtil.shuffleArray(sol.cityIndexes, rand);
}

/**
 * Komparator dvaju rješenja. Rješenje je manje ako je duljina ture manja.
 */
public static Comparator<TSPSolution> solComparator = new Comparator<TSPSolution>()
{
    @Override
    public int compare(TSPSolution o1, TSPSolution o2) {
        double razlika = o1.tourLength - o2.tourLength;
        return razlika < 0 ? -1 : (razlika > 0 ? 1 : 0);
    }
};
}

```

### 7.5.6. Razred ArraysUtil

```

package hr.fer.zemris.util;

import java.util.Random;

/**
 * Pomoćni razred s metodama za rad nad poljima.
 *
 * @author marcupic
 */
public class ArraysUtil {

    /**

```

```
* Metoda koja popunjava polje integera počev od 0 na dalje.
* Primjerice, ako je polje duljine 3, sadržaj će postati:
* 0, 1, 2.
*
* @param array polje koje treba popuniti
*/
public static void linearFillArray(int[] array) {
    for(int i = 0; i < array.length; i++) {
        array[i] = i;
    }
}

/**
 * Metoda koja permutira redosljed elemenata u predanom
 * polju posredstvom slučajnog mehanizma.
 *
 * @param array polje koje treba permutirati
 * @param rand generator slučajnih brojeva
 */
public static void shuffleArray(int[] array, Random rand) {
    for(int i = array.length; i>1; i--) {
        int b = rand.nextInt(i);
        if(b!=i-1) {
            int e = array[i-1];
            array[i-1] = array[b];
            array[b] = e;
        }
    }
}
}
```

### 7.5.7. Razred *PrepareTSP*

```
package hr.fer.zemris.graphics.tsp;

import hr.fer.zemris.tsp.City;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.GridLayout;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

/**
 * Razred koji obavlja vizualizaciju pronađene rute kod problema TSP.
 *
 * @author marcupic
 */
public class PrepareTSP extends JFrame {

    private static final long serialVersionUID = 1L;

    // Lista gradova; poredak već predstavlja rutu
    private List<City> cities = new ArrayList<City>();
    // Pomoćna labela za prikaz duljine rute
    private JLabel duljinaLabel;
    // Pomoćna labela za prikaz brja gradova
    private JLabel brojLabel;

    // Pomoćna komponenta koja obavlja iscrtavanje slike gradova
    private VisualizeComponent komponenta;

    /**
     * Konstruktor koji prima polje gradova. Redosljed elemenata automatski određuje
     * i samu turu.
     *
     * @param presetCities polje gradova
     */
}
```

```

*/
public PrepareTSP(City[] presetCities) {
    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    setSize(500, 500);
    setLocation(20, 20);

    if(presetCities!=null) {
        for(City c : presetCities) {
            cities.add(c);
        }
    }
    komponenta = new VisualizeComponent();
    this.getContentPane().setLayout(new BorderLayout());
    this.getContentPane().add(komponenta, BorderLayout.CENTER);

    JPanel p = new JPanel(new GridLayout(1,2));
    duljinaLabel = new JLabel();
    brojLabel = new JLabel();
    p.add(brojLabel);
    p.add(duljinaLabel);
    this.getContentPane().add(p, BorderLayout.PAGE_END);

    azurirajDuljinu();
    setVisible(true);
}

/**
 * Pomoćna funkcija koja računa duljinu ture i ažurira podatke u labelama
 * {@linkplain #duljinaLabel} i {@linkplain #brojLabel}.
 */
private void azurirajDuljinu() {
    double d = 0;
    if(!cities.isEmpty()) {
        City cc = null;
        City oldCc = null;
        for(int i = 0; i < cities.size(); i++) {
            cc = cities.get(i);
            if(oldCc!=null) {
                d += Math.sqrt((oldCc.x-cc.x)*(oldCc.x-cc.x) +
                    (oldCc.y-cc.y)*(oldCc.y-cc.y));
            }
            oldCc = cc;
        }
        cc = cities.get(0);
        if(cc!=oldCc) {
            d += Math.sqrt((oldCc.x-cc.x)*(oldCc.x-cc.x) +
                (oldCc.y-cc.y)*(oldCc.y-cc.y));
        }
    }
    duljinaLabel.setText("Duljina: "+((double)((int)(d*1000+0.5))/1000.0));
    brojLabel.setText("Broj gradova: "+cities.size());
}

/**
 * Pomoćna metoda koju smije pozvati proizvoljna dretva a služi za inicijalizaciju
 * prikaza ture. Metoda prima polje gradova i generira prikaz. Polje se pri tome
 * kopira pa je dretva-pozivatelj slobodna kasnije obavljati modifikacije nad
 * poljem; to više neće imati nikakvog utjecaja na prikaz.
 *
 * @param cities polje gradova čijim je poretkom u polju ujedno određena i tura
 */
public static void visualize(final City[] cities) {
    try {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new PrepareTSP(cities);
            }
        });
    } catch (Exception ignorable) {
        System.out.println("Prikaz nije moguć.");
    }
}

/**
 * Pomoćna komponenta koja crta turu.

```

```
*
* @author marcupic
*/
private class VisualizeComponent extends JComponent {

    private static final long serialVersionUID = 1L;

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        if(cities.isEmpty()) return;
        City cc = null;
        City oldCc = null;
        g.setColor(Color.BLACK);
        for(int i = 0; i < cities.size(); i++) {
            cc = cities.get(i);
            if(oldCc!=null) {
                g.drawLine(oldCc.x, oldCc.y, cc.x, cc.y);
            }
            oldCc = cc;
        }
        cc = cities.get(0);
        if(cc!=oldCc) {
            g.drawLine(oldCc.x, oldCc.y, cc.x, cc.y);
        }
        g.setColor(Color.BLACK);
        for(int i = 0; i < cities.size(); i++) {
            cc = cities.get(i);
            g.fillRect(cc.x-2, cc.y-2, 6, 6);
        }
    }
}
```