Prirodom inspirirani optimizacijski algoritmi. Metaheuristike.

Marko Čupić

30. prosinca 2013.

Sadržaj

Popis slika

Popis tablica

Izvorni kodovi programa

Predgovor

Ovaj dokument predstavlja radnu verziju knjige: *Prirodom inspirirani optimizacijski algoritmi. Metaheuristike*. Molimo sve pogreške, komentare, nejasnoće te sugestije dojaviti na Marko.Cupic@fer.hr.

Verzija dokumenta: 0.1.2013-12-30.

Poglavlje 1

Uvod

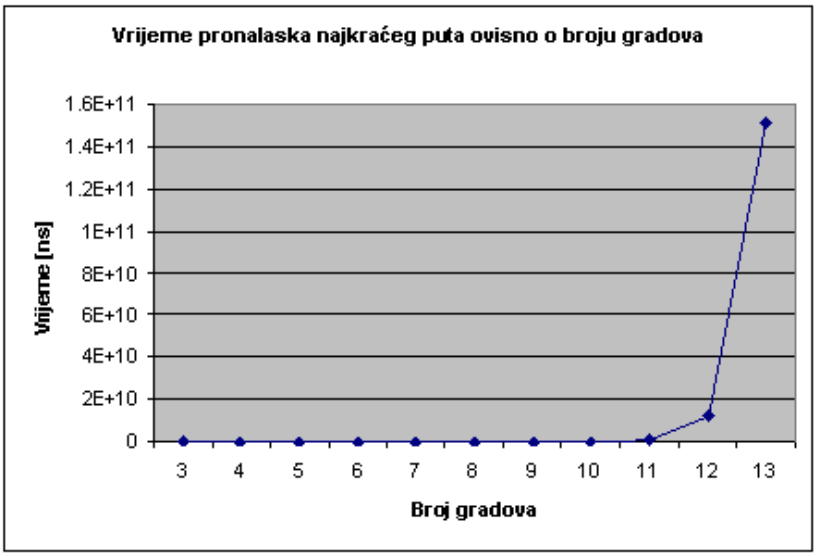
Razvojem računarstva i povećanjem procesne moći računala, ljudi su počeli rješavati izuzetno kompleksne probleme koji su do tada bili nerješivi – i to naprosto tehnikom grube sile (engl. *brute-force*, koristi se još i termin *iscrpna pretraga*), tj. pretraživanjem cjelokupnog prostora rješenja. Uobičajeni algoritmi koji se koriste u tu svrhu su algoritam pretraživanja u širinu, algoritam pretraživanja u dubinu [Russell and Norvig, 2010] te algoritam iterativnog pretraživanja u dubinu [Korf, 1985]. Svi navedeni algoritmi spadaju u algoritme slijepog pretraživanja s obzirom da ne uzimaju u obzir nikakve informacije vezane uz problem koji rješavaju. Kako bi se pretraga ubrzala, razvijena je i porodica algoritama usmjerenog pretraživanja, u koju spadaju algoritmi koji su prilikom pretraživanja vođeni informacijama o problemu koji rješavaju i procjeni udaljenosti od trenutnog pa do ciljnog rješenja koje se traži. Ove se informacije uzimaju u obzir kako bi se pretraga usmjerila i time prije završila. Primjer ovakvog algoritma je algoritam A^* [Hart et al., 1968].

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, odnosno uporabom upravo spomenutih algoritama, dade riješiti. Nažalost, postoji čitav niz problema koji ne spadaju u ovu kategoriju, pa spomenimo samo neke od njih.

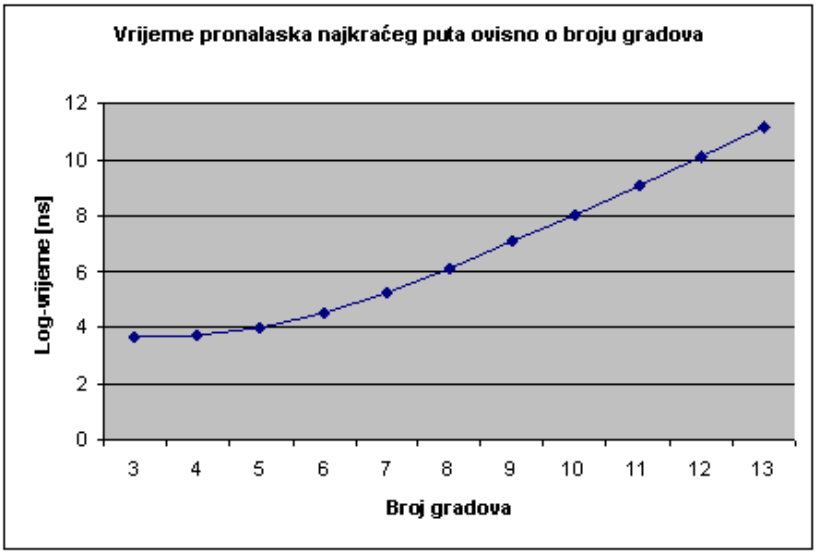
Problem trgovačkog putnika (engleski termin je *Traveling sales-person problem*, odnosno kraće *TSP*) [Lawler, 1985] definiran je ovako: potrebno je pronaći redoslijed kojim trgovački putnik mora obići sve gradove, a da pri tome niti jedan ne posjeti više puta, te da ukupno prevali najmanje kilometara. Na kraju, trgovački se putnik mora vratiti u onaj grad iz kojeg je krenuo. Danas je poznato da ovaj naoko trivijalan problem pripada razredu $\mathcal{NP}$-teških problema. *$\mathcal{NP}$-teški problemi* [Garey and Johnson, 1979] pripadaju u vrstu problema za koje danas još uvijek nemamo efikasnih algoritama kojima bismo ih mogli riješiti. A radi se, naravno, o ekstremnoj složenosti ove vrste problema. Kako bismo ilustrirali složenost problema trgovačkog putnika, napisan je jednostavan program u programskom jeziku Java koji ga rješava tehnikom grube sile, i potom je napravljeno mjerenje za probleme od 3 do 13 gradova. Rezultate prikazuju slike 1.1 i 1.2.

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 poprilici 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. Paktorijelna složenost naivnog načina rješavanja ovog problema proizlazi iz činjenice da se jedno rješenje problema može predstaviti kao jedna permutacija redosljeda obilaska gradova. Problem koji tada rješavamo jest pronalazak optimalne permutacije, koji, ako ga rješavamo tako da ispitamo svaku moguću permutaciju pa potom odaberemo optimalnu, ima faktorijelnu složenost (jer toliko ima permutacija). Međutim, pokazano je da nema potrebe ispitivati baš svaku permutaciju rješenja čime su pronađeni efikasniji algoritmi. Tako je u, primjerice, [Held and Karp, 1962] opisan algoritam koji se temelji na *dinamičkom programiranju* i koji ovaj problem rješava u eksponencijalnoj složenosti (konkretno, u složenosti $O(n^2 \cdot 2^n)$). No iako je ovo bitno prihvatljivije od



Slika 1.1: Ovisnost vremena pretraživanja kod iscrpne pretrage o broju gradova kod TSP-a



Slika 1.2: Ovisnost vremena pretraživanja kod iscrpne pretrage o broju gradova kod TSP-a uz logaritamsku skalu po ordinati

faktorijelne složenosti, jasno je da su takvi pristupi i dalje neupotrebljivi za veće primjerke problema. U teoriji grafova, problem trgovačkog putnika odgovara pronalasku *Hamiltonovog ciklusa* u grafu, za koji je [Karp, 1972] pokazao da je $\mathcal{NP}$*-potpun* problem. Za pregled tehnika egzaktnog rješavanja $\mathcal{NP}$-teških optimizacijskih problema čitatelj se upućuje na [Woeginger, 2003].

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 grupe u kojima se održavaju predavanja. Potrebno je za prvog studenta i njegov prvi kolegij provjeriti 4 grupe, pa za njegov drugi kolegij 4 grupa, itd. Za sve studente to ukupno daje:

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

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 Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu, u ljetnom semestru akademske godine 2008/2009. Broj termina bio 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} kombinacija. Opisani problem je 10^{174} puta složeniji od prethodno opisanog problema raspoređivanja neraspoređenih studenata. Za jesenski ispitni rok u akademskoj godini 2011/2012 brojke su bile još veće: u raspoređivanje je ušlo gotovo 200 kolegija.

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 $30^{200} \approx 10^{295}$. Zgodno je primjetiti da je ovo samo broj načina na koje možemo događaju pridijeliti termin. Međutim, tu još nismo uzeli u obzir da, primjerice, ako imamo kolegij sa 60 studenata čije su laboratorijske vježbe razpodijeljene u dva događaja (dvije grupe) od po 30 ljudi, negdje trebamo ispitati kojih 30 studenata treba staviti u koju grupu – što dovodi do daljnje eksplozije broja kombinacija koje treba ispitati.

Na sreću, *optimalno rješenje* nam često nije nužno; obično smo zadovoljni i s rješenjem koje je dovoljno dobro.

Definicija 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) nazivamo *približni algoritmi*, *heurističke metode*, *heuristički algoritmi* ili *jednostavno heuristike* [Dorigo and Stützle, 2004]. Dijelimo ih na konstrukcijske algoritme te algoritme koji koriste lokalnu pretragu.

Konstrukcijski algoritmi rješenje problema grade dio po dio (često 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*.

Definicija *Metaheuristika* [Glover and Kochenberger, 2003, Talbi, 2009] 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 [Dorigo and Stützle, 2004].

Primjeri metaheuristika su simulirano kaljenje, tabu pretraživanje, algoritmi evolucijskog računanja i slični. Pri tome ih možemo podijeliti u dvije velike porodice algoritama: algoritmi koji rade nad jednim rješenjem (gdje spadaju algoritam simuliranog kaljenja te algoritam tabu pretrage) te na populacijske algoritme koji rade sa skupovima rješenja (gdje spadaju algoritmi evolucijskog računanja).

Simulirano kaljenje [Kirkpatrick et al., 1983, Eglese, 1990, Fleischer, 1995] motivirano je procesom kaljenja metala. Ideja algoritma jest da se dobra rješenja $f(s^{'}) > f(s)$ automatski prihvaćaju (pretpostavimo da se traži maksimum funkcije; f je promatrana funkcija, s trenutno najbolje rješenje a $s^{'}$ razmatrani kandidat), a loša $f(s^{'}) < f(s)$ prihvaćaju s vjerojatnošću:

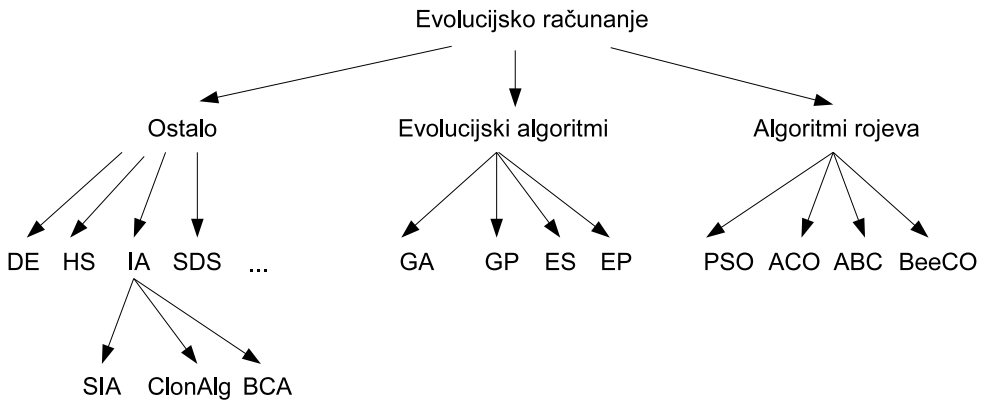
$$p_{\text{prihvati}}(s, s^{'}, T) = e^{\frac{f(s)-f(s^{'})}{T}}$$

gdje je T temperatura. Ovakva strategija bi algoritmu trebala osigurati da uvijek prihvati bolje novopronađeno rješenje, a povremeno i lošije rješenje od trenutnog najboljeg, kako bi se pomoglo u izbjegavanju lokalnih optimuma. Pri tome je vjerojatnost prihvaćanja 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 kreće od visokih temperatura čime se osigurava grubo pretraživanje prostora stanja, i potom temperaturu postupno smanjuje čim se pokušava osigurati fina pretraga u okolici najboljeg rješenja te u konačnici konvergencija.

Tabu pretraživanje [Glover, 1989, 1990, Glover and Laguna, 1997] je tehnika kod koje se pamti lista zadnjih n posjećenih rješenja, i ta su rješenje zabranjena (tabu). Prilikom pretraživanja prostora, iz susjedstva rješenja s bira se najbolji susjed $s^{'}$ koji pri tome nije u listi zabranjenih rješenja – čak i ako je taj lošiji od trenutno najboljeg rješenja. Ovom tehnikom algoritam nudi mogućnost izlaska iz lokalnih optimuma i napredak prema globalnom optimumu.

Osvrnimo se ovdje ukratko i na podskup algoritama *evolucijskog računanje* poznat pod nazivom *evolucijski algoritmi*. Danas ga uobičajeno dijelimo na četiri podpodručja: *genetske algoritme* [Holland, 1975, Goldberg, 1989, Liepins and Hillard, 1989, Muhlenbein, 1997], *genetsko programiranje* [Koza, 1992], *evolucijske strategije* [Rechenberg, 1973, Schwefel, 1981] te *evolucijsko programiranje* [Fogel et al., 1966]. 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: simulirano kaljenje, genetski algoritam, optimizaciju kolonijom mrava, optimizaciju rojem čestica, optimizaciju umjetnim imunološkim sustavom te algoritam diferencijske evolucije.



Slika 1.3: Podjela evolucijskog računanja na glavne grane. Uz svaku granu prikazani su i odabrani algoritmi.

1.1 Pregled algoritama evolucijskog računanja

U prethodnom odjeljku kratko smo se osvrnuli na evolucijske algoritme. No kako izgleda šira slika?

Evolucijsko računanje je grana umjetne inteligencije koja se, najvećim dijelom, bavi rješavanjem optimizacijskih problema. To je područje u razvoju već više od pola stoljeća: začetci sežu u kasne 50.-e [Back et al., 1997]. Evolucijsko računanje danas obuhvaća bogat skup raznorodnih algoritama od kojih ćemo se u ovom poglavlju osvrnuti na nekoliko. Evolucijsko računanje može se podijeliti u tri grane: *evolucijske algoritme*, *algoritme rojeva* te *ostale algoritme*. Ova podjela kao i odabrani algoritmi prikazani su na slici 1.3.

Područje evolucijskog računanja ujedno pripada i u područje metaheuristika [Michalewicz and Fogel, 2004, Talbi, 2009, Yang, 2008] – heuristika namijenjenih vođenju problemski specifičnijih heuristika u pokušaju da se pronađe potprostor u prostoru pretraživanja koji sadrži dobra rješenja.

U evolucijske algoritme danas uobičajeno ubrajamo evolucijske strategije, evolucijsko programiranje, genetski algoritam te genetsko programiranje. Utjecaj na ovo područje imali su radovi [Bremermann, 1962, Friedberg, 1958, Friedberg et al., 1959, Box, 1957], dok su temelj postavili radovi [Holland, 1962, Rechenberg, 1965, Schwefel, 1968, Fogel, 1962]. Danas o ovim algoritmima postoji obilje literature, a spomenut ćemo samo neke novije knjige: [Affenzeller et al., 2009, Deb, 2009, Sivanandam and Deppa, 2010, Langdon and Poli, 2010].

U algoritme rojeva ubrajamo *mravlje algoritme*, *algoritam roja čestica*, *algoritme pčela* i druge. Razvoj mravljih algoritama potaknut je eksperimentima biologa [Pasteels et al., 1987, Goss et al., 1989] o ponašanju mrava u prirodi. Popularizaciju te veliki poticaj za daljnja istraživanja mravlji algoritmi su doživjeli nakon objave rada [Colorni et al., 1991] te knjige [Dorigo and Stützle, 2004]. Opširni pregled mravljih algoritama može se pogledati u [Cordon et al., 2002]. Istaknutiji razvoj i primjena algoritma roja čestica počinje od sredine devedesetih, kada Eberhart i Kennedy definiraju sam algoritam [Kennedy and Eberhart, 1995] te knjigu [Kennedy et al., San Francisco, USA]. Od preglednih radova o algoritmu roja čestica valja istaknuti [Song and Gu, 2004, Banks et al., 2007, 2008]. Razvoj algoritama zasnovanih na pčelinjem ponašanju počinje 2004. godine objavom algoritma *Honey Bee Algorithm* [Nakrani and Tovey, 2004]; nakon toga se razvija algoritam virtualnih pčela [Yang, 2005], algoritam *Honey Bee Mating Optimization* [Haddad et al., 2006] te algoritam umjetne pčelinje kolonije [Karaboga and Basturk, 2007, 2008].

Granu *ostali algoritmi* čini još niz drugih algoritama koji ne pripadaju u prethodne dvije grane. Od značajnijih algoritama tu svakako pripadaju umjetni imunološki algoritmi, algoritam diferencijske evolucije te algoritam harmonijske pretrage. Podloga za definiranje umjetnih imunoloških sustava seže još u 1957. i čine je radovi Burneta [Burnet, 1957, 1959, 1978]. Najraširenije inačice algoritama, posebice onih temeljenih na principu klonske selekcije opisani su u [de Castro and Timmis, 2002, Cutello and Nicosia, 2005], a knjiga [Dasgupta and Niño, 2009] također nudi dobar uvod u područje. Algoritam diferencijske evolucije nastao je iz algoritma genetskog kaljenja (engl. *Genetic Annealing*) [Price, 1994].

Prva verzija algoritma opisana je u tehničkom izvještaju [Storn and Price, 1995], nakon čega su uslijedili i radovi [Storn and Price, 1996, Price and Storn, 1997, Storn and Price, 1997, Price, 1997, 1999]. Algoritam harmonijske pretrage osmislili su Lee i Geem 2004. godine, te su pokazali njegovu primjenu kako u optimizaciji funkcija kontinuiranih varijabli, tako i za rješavanje kombinatornih problema [Lee and Geem, 2004, 2005].

1.2 Najbolji optimizacijski algoritam

Nakon ovog kraćeg uvoda, evo i zanimljivog pitanja: *koji je od spomenutih optimizacijskih algoritama najbolji?*. Ovo je važno pitanje, jer ako ga odgovorimo, nadamo se da će nam odgovor omogućiti da se fokusiramo na samo jedan – onaj najbolji, i zaboravimo na sve ostale. To bi svakako bila značajna ušteda vremena. A evo i još jedno pitanje: mogu li se metaheuristički algoritmi iz porodice algoritama evolucijskog računanja uspješno koristiti za rješavanje svih problema?

Dodatno, postavlja se i pitanje je li dobro koristiti nasumičnu pretragu, te da li, kada i u kojoj mjeri uvesti dodatne informacije u algoritam? Uporaba dodatne informacije u postupku pretraživanja čini razliku između slijepog i usmjerenog pretraživanja. U nedostatku bilo kakvih smjernica, algoritmi mogu samo nasumično generirati moguća rješenja ili pak nasumično modificirati trenutna rješenja. Uvođenjem dodatnih informacija mogli bismo pomisliti da će algoritmi uvijek raditi bolje (ma što *bolje* značilo). Međutim, uvođenjem dodatnih informacija pretraga se fokusira čime se efektivno smanjuje prostor pretraživanja; to u nekim slučajevima može djelovati nepovoljno na postupak pretraživanja (primjerice, u prisustvu velikog broja lokalnih ekstrema gdje će algoritam ostati trajno zarobljen u lokalnom optimumu jer je previše fokusiran). S druge pak strane, ako problem nije takve vrste, fokusiranje pretrage može dovesti do značajnog skraćivanja vremena potrebnog za pronalazak dovoljno dobrog (ili čak optimalnog) rješenja.

Razmišljajući dalje u tom smjeru, vratimo se na početno pitanje: *koji je algoritam pretraživanja onda najbolji*? Na našu sreću (ili ne), danas znamo odgovor na ovo pitanje. Wolpert i Macready su u svojim radovima dokazali da nema najboljeg algoritma, što je poznato kao *no-free-lunch* teorem. Dapače, dokazali su da su svi algoritmi pretraživanja – upravo prosječno dobri [Wolpert and Macready, 1995, 1997]:

> *All algorithms that search for an extremum of a cost function perform exactly the same, according to any performance measure, when averaged over all possible cost functions. In particular, if algorithm A outperforms algorithm B on some cost functions, then loosely speaking there must exist exactly as many other functions where B outperforms A.*

No što to znači odnosno kakve to ima posljedice? Poruka je jasna: za različite probleme različiti će algoritmi biti "najbolji". Što više algoritama znamo, i što više razumijemo njihovo ponašanje i način rada, veće su nam šanse da odaberemo pravi algoritam za problem koji pokušavamo riješiti. Stoga nam ne preostaje ništa drugo nego se baciti na posao i krenuti redom.

Bibliografija

M. Affenzeller, S. Winkler, S. Wagner, and A. Beham. *Genetic algorithms and Genetic programming. Modern Concepts and Practical Applications.* CRC Press, Boca Raton, USA, 2009.

T. Back, U. Hammel, and H.-P. Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, 1997.

A. Banks, J. Vincent, and C. Anyakoha. A review of particle swarm optimization. part i: background and development. *Natural Computing*, 6:467–484, 2007. ISSN 1567-7818. URL `http://dx.doi.org/10.1007/s11047-007-9049-5`. 10.1007/s11047-007-9049-5.

A. Banks, J. Vincent, and C. Anyakoha. A review of particle swarm optimization. part ii: hybridisation, combinatorial, multicriteria and constrained optimization, and indicative applications. *Natural Computing*, 7:109–124, 2008. ISSN 1567-7818. URL `http://dx.doi.org/10.1007/s11047-007-9050-z`. 10.1007/s11047-007-9050-z.

G. E. P. Box. Evolutionary operation: A method for increasing industrial productivity. *Appl. Statistics*, 6(2):81–101, 1957.

H. J. Bremermann. Optimization through evolution and recombination. *Self-Organizing Systems*, 1962.

F. M. Burnet. A modification of jerne's theory of antibody production using the concept of clonal selection. *Australian Journal of Science*, 20:67–69, 1957.

F. M. Burnet. *The clonal selection theory of acquired immunity.* Vanderbilt University Press, Nashville, Tennessee, U.S.A., 1959.

F. M. Burnet. Clonal selection and after. *Theoretical Immunology*, pages 63–85, 1978.

A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of European conference on artificial life*, pages 134–142, Paris, France, 1991.

O. Cordon, F. Herrera, and T. Stützle. A review on the ant colony optimization metaheuristic: Basis, models and new trends. *Mathware and Soft Computing*, 9:141–175, 2002.

V. Cutello and G. Nicosia. The clonal selection principle for in silico and in vitro computing. In L. N. de Castro and F. J. V. Zuben, editors, *Recent Developments in Biologically Inspired Computing*, chapter IV, pages 104–146. Idea Group Publishing, Hershey, London, Melbourne, Singapore, 2005.

D. Dasgupta and L. F. Niño. *Immunological Computation. Theory and Applications.* CRC Press, Boca Raton, USA, 2009.

L. N. de Castro and J. Timmis. *Artificial Immune Systems: A new computational intelligence approach.* Springer-Verlag, Great Britain, 2002.

K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley, West Sussex, United Kingdom, 2009.

M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, Cambridge, MA, 2004.

R. W. Eglese. Simulated annealing: A tool for operational research. *European Journal of Operational Research, EJOR, Vol. 46, :271-281*, 1990.

M. Fleischer. Simulated annealing: Past, present, and future. In *Winter Simulation Conference*, pages 155–161, 1995. URL `http://doi.acm.org/10.1145/224401.224457`.

L. J. Fogel. Autonomous automata. *Ind. Res.*, 4:14–19, 1962.

L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley, New York, 1966.

R. M. Friedberg. A learning machine: Part i. *IBM J.*, 2(1):2–13, 1958.

R. M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part ii. *IBM J.*, 3(7):282–287, 1959.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Books in the Mathematical Sciences Series. W. H. Freeman Company, New York NY, 1979.

F. Glover. Tabu search, part I. *ORSA Journal on Computing*, 1:190–206, 1989.

F. Glover. Tabu search– part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

F. Glover and G. Kochenberger. *Handbook of Metaheuristics.* Kluwer Academic Publishers, Boston, MA, 2003.

F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

S. Goss, S. Aron, J.-L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

O. Haddad, A. Afshar, and M. Mariño. Honey-bees mating optimization (hbmo) algorithm: A new heuristic approach for water resources optimization. *Water Resources Management*, 20:661–680, 2006. ISSN 0920-4741. URL `http://dx.doi.org/10.1007/s11269-005-9001-3`. 10.1007/s11269-005-9001-3.

P. Hart, N. Nilsson, B., and Raphael. A formal basis for the heuristic determination of minimum-cost paths. *IEEE Trans. on Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

J. H. Holland. Outline for a logical theory of adaptive systems. *J. Assoc. Comput. Mach.*, 3:297–314, 1962.

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

D. Karaboga and B. Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of Global Optimization*, 39:459–471, 2007. ISSN 0925-5001. URL `http://dx.doi.org/10.1007/s10898-007-9149-x`. 10.1007/s10898-007-9149-x.

D. Karaboga and B. Basturk. On the performance of artificial bee colony (abc) algorithm. *Applied Soft Computing*, 8(1):687–697, 2008. ISSN 1568-4946. doi: DOI:10.1016/j.asoc.2007.05.007. URL `http://www.sciencedirect.com/science/article/B6W86-4NWCGRR-G/2/422ccff5df9d32a5bf8517068ca2a094`.

R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, Piscataway, NJ, 1995.

J. Kennedy, R. Eberhart, and Y. Shi. *Swarm intelligence.* Morgan Kaufmann, 2001, San Francisco, USA.

S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

R. E. Korf. Iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer, Berlin, Germany, 2010.

E. Lawler. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York, 1985.

K. S. Lee and Z. W. Geem. A new structural optimization method based on the harmony search algorithm. *Computers & Structures*, 82(9-10):781–798, 2004. ISSN 0045-7949. doi: DOI:10.1016/j.compstruc.2004.01.002. URL `http://www.sciencedirect.com/science/article/B6V28-4BWMP8Y-2/2/cd0c48f22f516cc7df98796923cbe99a`.

K. S. Lee and Z. W. Geem. A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 194(36-38):3902–3933, 2005. ISSN 0045-7825. doi: DOI:10.1016/j.cma.2004.09.007. URL `http://www.sciencedirect.com/science/article/B6V29-4DW3937-2/2/0b447356f1eea2414d356f14b137582b`.

G. E. Liepins and M. R. Hillard. Genetic algorithms: Foundations and applications. *Annals of Operations Research*, 21(1-4):31–57, Nov. 1989.

Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics.* Springer, Berlin, Germany, 2nd edition, 2004.

Muhlenbein. Genetic algorithms. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization.* Wiley, 1997.

S. Nakrani and C. Tovey. On honey bees and dynamic server allocation in internet hosting centers. *Adaptive Behavior*, 12(3–4):223–240, 2004.

J. M. Pasteels, J.-L. Deneubourg, and S. Goss. Self-organization mechanisms in ant societies (i): Trail recruitment to newly discovered food sources. *Experientia Supplementum*, 54:155–175, 1987.

K. Price. Genetic annealing. *Dr. Dobb's Journal*, 19(10):127–132, 1994.

K. Price and R. Storn. Differential evolution: A simple evolution strategy for fast optimization. *Dr. Dobb's Journal of Software Tools*, 22(4):18–24, 1997.

K. V. Price. Differential evolution vs. the functions of the 2nd iceo. In *Proc. of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 153–157, Indianapolis, IN, USA, April 1997.

K. V. Price. *New Ideas in Optimization*, chapter Differential Evolution. McGraw-Hill, London, 1999.

I. Rechenberg. Cybernetic solution path of an experimental problem. *Royal Aircraft Establishment, Library translation No. 1122*, 1965.

I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme und Prinzipien der biologischen Evolution.* Frommann-Holzboog, Stuttgart, 1973.

S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. international edition).* Pearson Education, 2010. ISBN 978-0-13-207148-2. URL `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html`.

H. P. Schwefel. Projekt mhd-staustrahlrohr: Experimentelle optimierung einer zweiphasendüse, teil i. Technical Report 35, AEG Forschungsinstitut, October 1968.

H. P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley and Sons, New York, New York, 1981.

S. Sivanandam and S. Deppa. *Introduction to Genetic Algorithms*. Springer, Berlin, Germany, 2010.

M. Song and G. Gu. Research on particle swarm optimization: A reviews. In *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, pages 2236–2241, 2004.

R. Storn and K. Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, International Computer Science Institute, Berkeley, CA, 1995. Technical Report TR-95-012.

R. Storn and K. Price. Minimizing the real functions of the icec'96 contest by differential evolution. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 842–844, 1996.

R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

E.-G. Talbi. *Metaheuristics. From Design to Implementation.* Wiley, New Jersey, USA, 2009.

G. Woeginger. *Exact Algorithms for NP-Hard Problems: A Survey*, volume 2570 of *Combinatorial Optimization – Eureka, You Shrink! Lecture notes in computer science*. Springer, 2003.

D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical report, Santa Fe Institute, Sante Fe, NM, USA, 1995. Technical Report SFI-TR-95-02-010.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions On Evolutionary Computation*, 1(1):67–82, 1997.

X.-S. Yang. Engineering optimizations via nature-inspired virtual bee algorithms. In J. Mira and J. Álvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, volume 3562 of *Lecture Notes in Computer Science*, pages 317–323. Springer Berlin / Heidelberg, 2005. URL `http://dx.doi.org/10.1007/11499305\_33`. 10.1007/11499305_33.

X.-S. Yang. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, Frome, United Kingdom, 2008.

Poglavlje 2

Optimizacija

Tekst koji upravo čitate bavi se načinima rješavanja optimizacijskih problema. I to ne bilo kakvim načinima, već prirodom inspiriranim optimizacijskim algoritmima. Stoga ćemo si uzeti malo vremena i pokušati definirati problem optimizacije, pogledati koje vrste postoje te smjestiti to sve u širi kontekst.

2.1 Optimizacijski problem

Prije no što se upustimo u razmatranje svojstava optimizacijskih problema, pogledajmo najprije nekoliko primjera optimizacijskih problema.

Primjer 1. Zadana je funkcija $f(x)$ nad domenom $[-300, 500] \subset \mathcal{R}$. Pronaći vrijednost varijable x za koju funkcija f poprima maksimalnu vrijednost.

Primjer 2. Zadana je funkcija $g(x, y, z)$ nad domenom $[-300, 500] \times [-300, 500] \times [-300, 500] \subset \mathcal{R} \times \mathcal{R} \times \mathcal{R}$. Pronaći točku (x, y, z) za koju funkcija g poprima maksimalnu vrijednost.

Primjer 3. Na karti Hrvatske označeno je 100 turističkih lokacija čiji obilazak turistička agencija želi staviti u svoju ponudu. Pronaći redosljed obilaska tih 100 lokacija tako da turisti krenu iz jedne od njih, obiđu sve ali niti jednu ne posjete više puta i na kraju se vrate na lokaciju s koje su krenuli. Ukupna cijena aranžmana koja je proporcionalna prijeđenim kilometrima mora biti što je moguće manja.

Primjer 4. Na raspolaganju su podatci o kolegijima koji se predaju, studentima koji su upisali kolegije, raspoloživim prostorijama u kojima se mogu održavati predavanja kolegija, željeni tjedni broj sati održavanja predavanja svakog od kolegija te o nastavnicima koji predaju pojedine kolegije. Generirajte tjedni raspored sati koji će garantirati da (i) svi studenti imaju zakazana sva predavanja i da ih mogu odslušati bez kolizija, (ii) da niti jedan nastavnik ne drži više predavanja istovremeno, (iii) da niti ujednu prostoriju nije smješteno više studenata no što je kapacitet prostorije te (iv) da niti u jednu prostoriju nisu smještena dva predavanja istovremeno. Poželjno je također da raspored u što većoj mjeri osigurava (v) da svaki student u danu ima barem dva predavanja ili niti jedno, (vi) da svaki student u danu ima minimalan broj rupa između predavanja, (vii) da nastavnik ima minimalni broj rupa u rasporedu te (viii) da nastavnik što je moguće rjeđe mijenja dvorane tijekom istog dana.

Primjer 5. Zadana je funkcija $g(x, y, z)$ nad domenom $\mathcal{R} \times \mathcal{R} \times \mathcal{R}$. Pronaći točku (x, y, z) za koju funkcija g poprima maksimalnu vrijednost. Pri tome mora vrijediti $x^2 - 7yz < 25$ te $xy + z^3 > 2$.

Što je zajedničko svim opisanim problemima?

1. Postoji *prostor rješenja* (engleski termin je *solution space*). Rješenja definiramo kao točke u prostoru rješenja. Prostor rješenja može biti jednodimenzijski ili višedimenzijski. U primjeru 1, prostor rješenja je jednodimenzijski: to je skup $[-300, 500] \subset \mathcal{R}$. Svako rješenje jedan je element tog skupa (npr. -250, 117, 243 i slično). U primjeru 2, prostor rješenja trodimenzijski: to je kartezijev produkt $[-300, 500] \times [-300, 500] \times [-300, 500] \subset \mathcal{R} \times \mathcal{R} \times \mathcal{R}$ pa je svako rješenje jedan element tog produkta: uređena trojka, primjerice $(-100, 217, 43)$.

2. Postoji *prostor ciljnih funkcija* (engleski termin je *objectivespace*). Prostor funkcija cilja (ili kriterijskih funkcija ili naprosto kriterija) može biti jednodimenzijski ili višedimenzijski. U primjeru

1 traži se maksimizacija eksplicitno zadane funkcije f – ta je funkcija u tom slučaju naša funkcija cilja. U primjeru 2 prostor funkcija cilja je također jednodimenzijski. Međutim, u primjeru 4 imamo zadan niz kriterija koje treba zadovoljiti. Teoretski, svaki od tih kriterija može biti zadovoljen u određenoj mjeri čime je prostor funkcija cilja 8-dimenzijski.

Funkcije cilja koje je potrebno minimizirati često prevodimo u komplementarne funkcije koje je potrebno maksimizirati. U tom slučaju govorimo o *funkcijama dobrote* (engleski termin je *fitness functions*) koje po definiciji daju to veću vrijednost što je rješenje bolje. Ako je zadana funkcija f koju je potrebno minimizirati, najjednostavniji način izgradnje pripadne funkcije dobrote jest uzeti funkciju $-f$. Minimizacija funkcije f jednaka je maksimizaciji funkcije $-f$.

3. Prostor rješenja može biti *kontinuiran* ili *diskretan*. Primjeri 1 i 2 su primjeri kod kojih je prostor rješenja kontinuirani – rješenje može poprimiti bilo koju vrijednost iz podskupa realnih brojeva. Primjeri 3 i 4 su primjeri s diskretnim prostorima rješenja – takve probleme nazivamo još i *kombinatoričkim problemima*.

4. Problem može imati postavljena ograničenja (engleski termin je *constraints*). Ograničenja mogu proizlaziti iz ograničene domene (kao u primjeru 1) ili pak iz eksplicitno zadanih ograda koje moraju vrijediti (primjer 4 te primjer 5).

5. Ograničenja mogu biti *tvrda* (engl. *hard*) ili *meka* (engl. *soft*). Tvrda ograničenja su ograničenja koja moraju biti zadovoljena da bi rješenje bilo prihvatljivo. Primjer su ograničenja dana u primjeru 5 te ograničenja (i) do (iv) u primjeru 4 – ako ona za neko rješenje ne vrijede, to rješenje je neprihvatljivo. Tvrda ograničenja dijele prostor rješenja u dva podprostora: *prostor prihvatljivih rješenja* (engl. *feasible solutions*) te u *prostor neprihvatljivih rješenja* (engl. *unfeasible solutions*). Zadaća optimizacijskog algoritma jest pronaći prikladno prihvatljivo rješenje.

   Meka ograničenja su ograničenja definiraju poželjna svojstva rješenja i ona mogu biti zadovoljena u određenoj mjeri (dakako, što više, to bolje). Primjer su ograničenja (v) do (viii) u primjeru 4.

Temeljem prethodne analize optimizacijske probleme dijelimo u dvije velike porodice: *problemi jednokriterijske optimizacije* te *problemi višekriterijske optimizacije*. Kod problema jednokriterijske optimizacije prostor funkcija cilja je jednodimenzijski: svako rješenje možemo direktno preslikati u broj koji govori o kvaliteti tog rješenja. Kod problema višekriterijske optimizacije prostor funkcija cilja je višedimenzijski: svako rješenje preslikava se u vektor brojeva (svaki element vektora je iznos jedne od definiranih funkcija cilja). Problem s višekriterijskom optimizacijom jest što rješenja ne moraju nužno biti međusobno usporediva. Pretpostavimo primjerice da rješavamo problem dvokriterijske optimizacije: svako rješenje mjerimo na dva načina, i cilj nam je pronaći rješenje koje maksimizira oba kriterija. Ako je kvaliteta rješenja $s_1 \rightarrow (2,5)$ a kvaliteta rješenja $s_2 \rightarrow (3,7)$, jasno je da je rješenje s_2 bolje od rješenja s_1 (jer je bolje po oba kriterija). No kako usporediti rješenja $s_3 \rightarrow (2,7)$ i $s_4 \rightarrow (3,5)$? Ta su rješenja međusobno neusporediva što predstavlja određen izazov u razvoju algoritama za višekriterijsku optimizaciju (o problemu višekriterijske optimizacije pričat ćemo kasnije). U prvom dijelu ovog teksta stoga ćemo se fokusirati na problem jednokriterijske optimizacije.

Problem jednokriterijske optimizacije. Općeniti problem jednokriterijske optimizacije definiran je na sljedeći način.

$$\begin{array}{lll} \text{Minimiziraj / maksimiziraj} & f(\vec{x}), & \\ \text{uz zadovoljenje ograničenja} & g_j(\vec{x}) \geq 0, & j = 1,2,\ldots,J \\ & h_k(\vec{x}) = 0, & k = 1,2,\ldots,K \\ & x_i^L \leq x_i \leq x_i^U, & i = 1,2,\ldots,n. \end{array}$$

Rješenje $\vec{x}$ je vektor decizijskih varijabli $\vec{x} = \{x_1, x_2, \ldots, x_n\}$. Prvi i drugi skup ograničenja prestavljaju skup nejednakosti odnosno jednakosti koje moraju biti zadovoljene za sva prihvatljiva rješenja. Treći skup ograničenja definira donju i gornju granicu na vrijednosti koje decizijske varijable smiju poprimiti. Ova tri skupa ograničenja definiraju prostor prihvatljivih rješenja. U tom prostoru svako rješenje $\vec{x}$ je jedna točka.

Sada, nakon što smo definirali problem jednokriterijske optimizacije, možemo definirati i pojam globalnog optimuma.

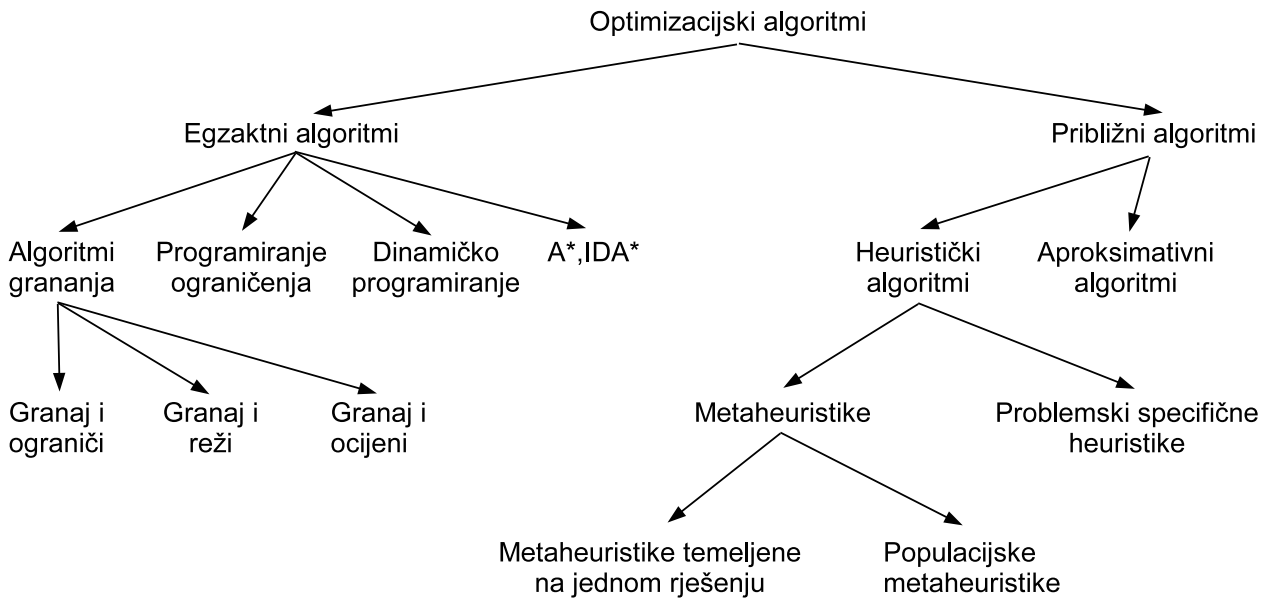
Globalni optimum. Kod jednokriterijske optimizacije, rješenje $\vec{x}^*$ naziva se globalnim optimumom ako i samo ako rješenje $\vec{x}^*$ pripada prostoru prihvatljivih rješenja i ako za svako drugo rješenje $\vec{x}$ iz prostora prihvatljivih rješenja vrijedi $f(\vec{x}^*) \geq f(\vec{x})$, gdje je f funkcija dobrote rješenja.

Uočimo da globalni optimum nije nužno jedinstven – prethodna definicija dozvoljava postojanje više različitih rješenja u kojima funkcija dobrote poprima svoju maksimalnu vrijednost. Također, osim ako funkcija nije strogo konveksna, moguće je postojanje jednog ili više lokalnih optimuma.

Lokalni optimum. Kod jednokriterijske optimizacije, rješenje $\vec{x}^*$ naziva se lokalnim optimumom ako i samo ako rješenje $\vec{x}^*$ pripada prostoru prihvatljivih rješenja i ako za svako drugo rješenje $\vec{x}$ iz δ-okoline od $\vec{x}^*$, tj. uz $\delta > 0$ i $|\vec{x} - \vec{x}^*| \leq \delta$ vrijedi $f(\vec{x}^*) \geq f(\vec{x})$, gdje je f funkcija dobrote rješenja.

Lokalni optimumi predstavljaju ozbiljan problem za optimizacijske algoritme. Naime, kako su unutar δ okoline od lokalnog optimuma sva rješenja lošija od lokalnog optimuma, svaki algoritam koji radi dovoljno male korake i deterministički bira uvijek samo bolje rješenje od trenutnog zaglavit će u takvom lokalnom optimumu. Jedina šansa da se takav algoritam izvuče iz lokalnog optimuma leži ili u povećanju koraka kojim se pretražuje prostor ili u (moguće stohastičkom) prihvaćanju i lošijih rješenja od trenutno najboljeg, u nadi da će to na duge staze ipak dovesti do još boljih rješenja.

Optimizacijske postupke možemo dijeliti na više načina. Jedna takva podjela prikazana je na slici 2.1. Naš fokus će biti na metaheuristikama: bilo na onima koje rade s jednim rješenjem poput algoritma simuliranog kaljenja, bilo na populacijskim metaheuristikama, poput genetskog algoritma i drugih.



Slika 2.1: Podjela optimizacijskih algoritama

2.2 Proces pretraživanja

Svi metaheuristički algoritmi dijele jednu zajedničku osobinu: u pretraživanje prostora kreću od jednog ili više početnih rješenja i potom temeljem postojećih rješenja (ili njihovog indirektnog utjecaja) generiraju nova rješenja. Prostor pretraživanja (tj. prostor rješenja) pri tome je dovoljno velik da provedba iscrpne pretrage u dostupnom vremenu nije moguća. Funkcija cilja (ili funkcije cilje) su uobičajeno takve da su visokonelinearne, odnosno uz globalni optimum imaju i mnoštvo različito kvalitetnih lokalnih optimuma.

Da bi optimizacijski proces u takvim uvjetima mogao biti uspješan, postupak pretraživanja prostora rješenja provodi se u dvije faze.

1. *Gruba pretraga.* U fazi grube pretrage algoritam nasumično uzorkuje rješenja iz (nadamo se) čitavog prostora pretraživanja kako bi pronašao podprostor koji sadrži obećavajuća rješenja.

2. *Fina pretraga.* Nakon lociranja obećavajućeg podprostora nastupa faza fine pretrage u kojoj se pretraživanje fokusira. U ovom fazi istražuje se okolica pronađenog podprostora u potrazi za globalnim optimumom.

Na papiru, ovo zvuči vrlo uvjerljivo. Međutim, u praksi se tu javlja niz problema. Kako definirati granicu između grube i fine pretrage? Koliko dugo raditi grubu pretragu a koliko dugo finu pretragu? Što ako je gruba pretraga locirala pogrešan podprostor? U tom slučaju dobit ćemo suboptimalno rješenje (možda čak i jako loše). Kako se boriti protiv ovoga?

Različiti optimizacijski algoritmi kroz ove faze prolaze na različite načine. Dapače, velik broj optimizacijskih algoritama nema postupak pretraživanja izveden ovako u fazama, već opseg pretrage (gruba ili fina) ovisi o trenutnom stanju algoritma. Uzmimo za primjer populacijske algoritme. Populacijski algoritam kreće s inicijalnom populacijom rješenja koja je, po pretpostavci, nastala uniformnim uzorkovanjem čitavog prostora pretraživanja. Populacijski algoritmi kombiniranjem i modificiranjem rješenja stvaraju populaciju za sljedeću generaciju i postupak ciklički ponavljaju. Kombiniranjem rješenja međutim smanjuje se raznolikost u populaciji – populacija se skuplja. Zamislite to ovako: populacija se sastoji od niza točaka u 2D ravnini. Neka je operator kombiniranja (tj. operator koji stvara djecu) temeljem dvaju roditelja operator aritmetičke sredine. Ako zamislite roditelja kao vektore $r_1 = (r_{1,x}, r_{1,y})$ i $r_2 = (r_{2,x}, r_{2,y})$, neka dijete bude definirano s $d = (\frac{r_{1,x}+r_{2,x}}{2}, \frac{r_{1,y}+r_{2,y}}{2})$. Lako je vidjeti da će se, uz pretpostavku da svi roditelji imaju jednaku šansu stvoriti dijete, populacija iz generacije u generaciju sve više sažimati prema zamišljenom centru populacije. Nova djeca koja time nastaju sve su međusobno bliža i bliža – od grube pretrage na početku nakon nekog vremena pretraga postaje fina i fokusirana jer su se prosječne udaljenosti između roditelja smanjile.

Umjesto dopuštanja da se pretraga fokusira prema centru populacije, pretragu je moguće usmjeriti na način da se boljim roditeljima da veća šansa da stvaraju potomke: u tom slučaju oni će češće ulaziti u križanje i u svojoj okolici će stvoriti veći broj djece čime će se populacija pomicati prema boljim jedinkama. Međutim, samo uz ovako opisani operator kombiniranja sa svakom sljedećom generacijom pretraga će se malo po malo fokusirati. Ako je tijekom grube pretrage populacija napipala dobro područje, ovo će fokusiranje dovesti do pronalaska dobrih (moguće i globalno optimalnih) rješenja. Ako to nije slučaj, algoritam će zaglaviti u suboptimalnom rješenju.

Što dovodi do zaglavljivanja u lokalnom optimumu? S jedne strane, to može biti neprikladna reprezentacija rješenja – više o ovome će biti riječi u sljedećem poglavlju. S druge strane, problem može biti u prebrzom fokusiranju pretrage. Vratimo se na prethodni primjer naše populacije u 2D prostoru. Ako kao roditelje uvijek biramo dva nasumična rješenja iz populacije, nekako očekujemo da će populacija duže vremena ostati raširena na širokom području. S druge pak strane, ako konstruiramo scenarij u kojem je jedan od roditelja uvijek fiksiran i nasumice biramo samo drugog roditelja, populacija će se vrlo brzo fokusirati oko tog izabranog roditelja. Možemo zaključiti da na brzinu fokusiranja populacije utjeće način kako biramo roditelje. Napravimo li algoritam kod kojeg deterministički uvijek odabiremo najboljeg roditelja za kombiniranje, pretraživanje će odgovarati algoritmu uspona na vrh – algoritmu koji pronalazi prvi lokalni optimum i ostaje zaglavljen u njemu. S druge pak strane, ako svim rješenjima damo jednaku šansu da postanu roditelji, algoritam se pretvara u nasumično pretraživanje, što je opet krajnost koju želimo izbjeći. U dobrom algoritmu pretraživanja, vjerojatnost odabira boljih roditelja je upravo takva da ne izaziva prebrzo fokusiranje populacije (jer ćemo time zapeti u prvom lokalnom optimumu) a da je dovoljna da postupak pretraživanja vodi prema kvalitetnijim rješenjima i globalnom optimumu. Parametar koji je vezan uz opisanu brzinu fokusiranja populacije naziva se *selekcijski pritisak* – što je pritisak veći, fokusiranje je brže.

Populacijski optimizacijski algoritmi tipično imaju dodatni mehanizam koji djeluje kao protusila selekcijskom pritisku i time vraća raznolikost u populaciju: *operator mutacije*. Zadaća operatora mutacije jest diverzificiranje populacije. Pri tome opet treba paziti na fini balans: ako je operator mutacije preagresivan, unosit će prevelike nasumične izmjene u rješenja čime će poništavati djelovanje operatora kombiniranja rješenja i pretragu ponovno prevesti u nasumično pretraživanje. Ako je djelovanje operatora mutacije nedovoljno agresivno, neće biti dovoljno da očuva raznolikost populacije i populacija će se fokusirati, najčešće oko nekog lokalnog optimuma. Stoga je kod populacijskih algoritama od

izričitog značaja uspostaviti odgovarajući odnos između intenziteta kojim djeluje operator mutacije i selekcijskog pritiska.

Bibliografija

Poglavlje 3

Prikaz rješenja i operacije nad rješenjima

Imate optimizacijski problem koji želite riješiti? Odlično! Prvi korak u rješavanju svakog optimizacijskog problema jest odlučiti na koji ćete način predstaviti jedno konkretno rješenje u računalu. Ova je odluka od izuzetnog značaja – ako odaberete prikladno rješenje, moći ćete s njime raditi brzo i efikasno; u suprotnom to neće biti moguće. Što se sve radi s rješenjima u optimizacijskim algoritmima?

- *Rješenje se vrednuje.* Svi optimizacijski algoritmi u konačnici korisniku trebaju vratiti jedno ili skup nekoliko najboljih rješenja. Da bi to bilo moguće, mora postojati mehanizam kojim se svakom pojedinom rješenju pridjeljuje mjera njegove dobrote (engl. *fitness*). Ovisno o načinu prikaza rješenja ova operacija može biti računski vrlo skupa.
- *Rješenje se modificira.* Velik broj optimizacijskih algoritama zahtjeva mogućnost izvođenja male promjene nad postojećim rješenjem. Primjerice, kod genetskog algoritma te kod algoritma klonske selekcije koji spada u umjetne imunološke algoritme koristi se operator mutacije čijom se primjenom nad postojećim rješenjem dobiva neko drugo rješenje koje je modifikacija postojećeg rješenja.
- *Rješenja se kombiniraju.* Neki optimizacijski algoritmi imaju potrebnu stvarati nova rješenja temeljem dva ili više postojećih koja se koriste kao predložak za izgradnju novog rješenja. Tako primjerice kod genetskog algoritma i kod algoritma diferencijske evolucije postoje operatori križanja koji novo rješenje grade od dijelova odabranih postojećih rješenja u nadi da će takva kombinacija dati novo rješenje koje je još bolje.
- *Generira se susjedstvo rješenja.* Kod nekih optimizacijskih algoritama jedna od važnih operacija jest generiranja susjednog rješenja ili čak generiranje čitavog susjedstva (dakle skupa rješenja). Kod algoritama koji se koriste susjedstvom važno je moći ga generirati što efikasnije.
- *Rješenja se odabiru.* Odabir rješenja čest je kod populacijskih optimizacijskih algoritama. Odabir može biti deterministički (odaberi iz skupa rješenja ono najbolje) ili može biti vjerojatnosni: za svako se rješenje na neki način način definira vjerojatnost da rješenje bude odabrano i potom se posredstvom slučajnog mehanizma a u skladu s definiranim vjerojatnostima odabire jedno ili više rješenja. Dodijeljena vjerojatnost može biti proporcionalna dobroti rješenja, relativnoj dobroti rješenja (razlici između dobrote rješenja i dobrote najgoreg rješenja), rangu rješenja (u populaciji koja je sortirana po dobrotama, na kojem je mjestu promatrano rješenje), nekoj funkciji od prethodno spomenutih vrijednosti i slično. Kod genetskog algoritma upravo operator selekcije igra centralnu ulogu koja uvelike određuje ponašanje algoritma.

Stoga ćemo se u nastavku najprije upoznati s načinima prikaza rješenja i provedbe modifikacija nad njima, kao i s operatorima selekcija.

3.1 Prikaz rješenja optimizacijskog problema

Iz prethodnog uvoda bi trebalo biti jasno: rješenje optimizacijskog problema uvijek se može prikazati na više načina. Način koji koristimo imat će utjecaj na sve operacije koje nad njim provodimo a time

i na sam optimizacijski algoritam. Prikaze rješenja možemo podijeliti u nekoliko najčešćih:

- prikaz nizom bitova,
- prikaz poljem ili vektorom brojeva,
- prikaz permutacijama i matricama,
- prikaz složenijim strukturama podataka te
- prikaz stablima.

Pogledajmo svaki od njih.

3.1.1 Prikaz nizom bitova

Prikaz rješenja nizom bitova (engl. *bit-string*) spada u najjednostavnije prikaze. Svako rješenje je predstavljeno kao niz nula i jedinica. Taj je niz najčešće fiksne duljine.

Ponekad je ovakav prikaz upravo prirodan način za predstavljanje rješenja. Evo primjera koji to jasno ilustrira. Neka je s $\mathcal{O}$ označen skup objekata $\{o_1, o_2, \ldots, o_n\}$. Neka je definirana funkcija $f(o)$ koja za svaki podskup $o \in \mathcal{O}$ vraća mjeru kvalitete tog podskupa; "mjeru kvalitete" ovdje namjerno ostavljamo tako apstraktno definiranu jer konkretna definicija nije važna za razmatranje. Zadatak optimizacijskog algoritma je pronaći podskup $o^* \in \mathcal{O}$ za koji vrijedi: $\forall o, o^* \in \mathcal{O}, f(o^*) \geq f(o)$. Rješenja opisanog problema možemo prikazati kao niz nula i jedinica upravo duljine n, pri čemu bi nula na poziciji i značila da objekt o_i nije uključen u podskup a jedinica na poziciji i bi značila da je objekt o_i uključen u podskup. Konkretno, ako je $\mathcal{O} = \{o_1, o_2, o_3, o_4\}$, tada bi zapis `0011` predstavljao podskup $\{o_3, o_4\}$ a zapis `1011` podskup $\{o_1, o_3, o_4\}$.

Generiranje male promjene nad ovakvim zapisom je trivijalno: možemo primjerice slučajno odabrati jednu ili više pozicija i na tim pozicijama okrenuti vrijednost bita. primjerice, ako je trenutno rješenje `0011` i ako smo slučajno odabrali prvi i zadnji bit dobit ćemo novo rješenje `1010` koje predstavlja podskup $\{o_1, o_3\}$.

Za generiranje susjednog rješenja ili čitavog susjedstva moramo najprije definirati pomoćni operator $\mathcal{W}^*$ koji radi neku ograničenu promjenu nad postojećim rješenjem. Neka je $\mathcal{W}^*$ definiran tako da na jednoj (bilo kojoj) poziciji okrene vrijednost bita. Uporabom operatora $\mathcal{W}^*$ tada nad rješenjem `0011` možemo dobiti skup rješenja {`1011`, `0111`, `0001`, `0010`}. Kažemo da taj skup s obzirom na operator $\mathcal{W}^*$ čini susjedstvo rješenja `0011`. Uočite: da smo operator $\mathcal{W}^*$ definirali kao operator koji može promijeniti 1 ili 2 bita, generirano susjedstvo bilo bi veće.

Primjena na numeričke probleme jedne varijable

Optimizacijske algoritme često koristimo kako bismo tražili minimume ili maksimume funkcija jedne ili više varijabli koje su definirane nad decimalnim brojevima. Binarni prikaz rješenja primjenjiv je i na takve probleme, pri čemu je potrebno definirati *postupak dekodiranja* – postupak koji će niz nula i jedinica prevesti u decimalne brojeve.

Pretpostavimo da koristimo binarni prikaz rješenja koji se sastoji od n bitova. Tim prikazom predstavljamo x koji je rješenje funkcije $f(x)$. Da bismo mogli utvrditi koju vrijednost predstavlja neki konkretni niz bitova, moramo definirati preslikavanje (odnosno postupak dekodiranja). Jedna mogućnost jest proglasiti da su legalne vrijednosti varijable x vrijednosti iz raspona $[x_{min}, x_{max}]$ te da se koristi prirodni binarni kod. Postupak dekodiranja tada se provodi na sljedeći način.

1. Binarni niz se protumači kao cijeli broj. Označimo njegovu vrijednost s k. Ako se koriste n-bitovni binarni nizovi, tada će k poprimiti vrijednost iz skupa cijelih brojeva $\{0, 1, \ldots, 2^n - 1\}$.
2. Vrijednost k preslika se na interval $[x_{min}, x_{max}]$. Pri tome se najčešće $k = 0$ preslika u vrijednost x_{min}, $k = 2^n - 1$ preslika u vrijednost x_{max} a ostale vrijednosti se linearno preslikaju na interval $[x_{min}, x_{max}]$. U tom slučaju se koristi izraz:

$$x = x_{min} + \frac{k}{2^n - 1} \cdot (x_{max} - x_{min}). \tag{3.1}$$

Pogledajmo ovo na sljedećem primjeru.

Primjer: 1

Varijabla x može poprimiti vrijednosti iz intervala $[-5, 5]$. Za prikaz rješenja se koristi 10-bitovni binarni prikaz uz prirodni binarni kod i linearno preslikavanje. Koju vrijednost tada predstavlja rješenje `00001101`?

Rješenje:

Kod prirodnog binarnog koda vrijednost zapisanu kao niz bitova $b_{n-1}b_{n-2}\dots b_1b_0$ tumačimo kao $k = \sum_{i=0}^{n-1} b_i \cdot 2^i$. U konkretnom slučaju imamo $k = 2^3 + 2^2 + 2^0 = 13$. Također, kako je $n = 10$, $2^n - 1 = 1023$ pa je vrijednost x pridružena rješenju `00001101` prema izrazu (3.1) jednaka:

$$\begin{aligned} x &= x_{min} + \frac{k}{2^n - 1} \cdot (x_{max} - x_{min}) \\ &= -5 + \frac{13}{1023} \cdot (5 - (-5)) \\ &= -5 + \frac{13}{1023} \cdot 10 \\ &= -4.87292. \end{aligned}$$

Uz opisano dekodiranje vezan je i pojam preciznosti kojom algoritam koji koristi opisani binarni prikaz uz linearno preslikavanje i prirodni binarni kod može pretraživati prostor. Sinonim za preciznost pretraživanja će biti i termin kvant pretrage. *Kvant pretrage* je minimalni korak kojim algoritam može promijeniti trenutnu vrijednost rješenja. Ovo je ilustrirano kroz sljedeća dva primjera.

Primjer: 2

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

Rješenje:

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, s obzirom na linearnost preslikavanja, razlika između bilo koja dva susjedna rješenja pa možemo pisati:

$$\begin{aligned} \tilde{x} &= x_{k+1} - x_k \\ &= x_{min} + \frac{k+1}{2^n - 1} \cdot (x_{max} - x_{min}) - \left\{ x_{min} + \frac{k}{2^n - 1} \cdot (x_{max} - x_{min}) \right\} \\ &= x_{min} + \frac{k}{2^n - 1} \cdot (x_{max} - x_{min}) + \frac{1}{2^n - 1} \cdot (x_{max} - x_{min}) - x_{min} - \frac{k}{2^n - 1} \cdot (x_{max} - x_{min}) \\ &= \frac{1}{2^n - 1} \cdot (x_{max} - x_{min}) \\ &= \frac{1}{7} \cdot 4 \\ &= 0.57. \end{aligned}$$

Primjer: 3

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

Rješenje:

Preciznost od 10^{-2} znači da je kvant jednak $10^{-2} = 0.01$ odnosno 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 koji će se preslikati u to rješenje, pa trebamo utvrditi koliko zapravo različitih rješenja želimo biti u mogućnosti prikazati, i potom utvrditi koliko nam za to minimalno treba bitova. Broj različitih rješenja uz zadanu preciznost određen je izrazom:

$$\frac{x_{max} - x_{min}}{\tilde{x}} + 1$$

što u ovom primjeru iznosi 401. Želimo pronaći prvi n za koji vrijedi:

$$2^n \geq \frac{x_{max} - x_{min}}{\tilde{x}} + 1.$$

Logaritmiranjem slijedi:

$$n \geq \frac{log\left(\frac{x_{max}-x_{min}}{\tilde{x}}+1\right)}{log(2)}. \tag{3.2}$$

što u ovom primjeru daje $n = 9$.

Spomenimo još i da kodiranje prirodnim binarnim kodom sa sobom donosi nezgodan problem: minimalna promjena na razini binarnog prikaza može uzrokovati drastičnu promjenu u pridijeljenoj vrijednosti. Također, minimalnom promjenom na razini binarnog prikaza nije uvijek moguće napraviti minimalnu promjenu u pridijeljenoj vrijednosti. Ilustrirat ćemo i jedno i drugo na primjeru.

Pretpostavimo da je optimizacijski algoritam došao do rješenja `0000000000` pretražujući interval $[-2, 2]$. To se rješenje preslikava u vrijednost $x = -2$. Pretpostavimo sada da smo koristeći operator promjene to rješenje modificirali tako da smo mu promijenili samo jedan bit – prvi, i time smo dobili novo rješenje `1000000000`. To se rješenje preslikava u 0.001955 što je, s obzirom da je prostor pretraživanja $[-2, 2]$ ogroman pomak od preko 50% intervala pretraživanja. Minimalna promjena od jednog bita na razini binarnog prikaza uzrokovala je ogromnu promjenu u dodijeljenoj vrijednosti.

Pretpostavimo sada da je optimizacijski algoritam došao do rješenja `0111111111` što se uz interval pretraživanja od $[-2, 2]$ preslikava u vrijednost -0.001955. Pretpostavimo također da je optimalno rješenje samo jedan kvant dalje. U ovom primjeru gdje je kvant jednak 0.00391 to bi značilo da je optimalna vrijednost jednaka $-0.001955 + 0.00391 = 0.001955$. Binarni niz koji se preslikava u tu vrijednost je `1000000000`. Sada je važno uočiti da optimizacijski algoritam nikakvom minimalnom promjenom na razini binarnog prikaza ne može rješenje `0111111111` pretvoriti u `1000000000` – ta pretvorba zahtjeva promjenu svih bitova binarnog prikaza što je gotovo nevjerojatan slučaj. Time će optimizacijski algoritam ostati zaglavljen u lokalnom optimumu `0111111111` koji je isključivo rezultat *odabranog prikaza rješenja* i definiranih operatora.

Moguće rješenje uočenog problema jest napuštanje prirodnog binarnog koda i uporaba binarnih kodova s minimalnom promjenom: primjerice, *Grayevog* koda. Sjetite se da je važna karakteristika Grayevog koda svojstvo da se svake dvije susjedne binarne riječi razlikuju u točno jednom bitu. To pak znači da iz bilo kojeg rješenja možemo doći do onog koje je za jedan kvant veće ili manje (dakle susjedno u smislu vrijednosti) promjenom samo jednog bita (dakako, ne bilo kojeg). Stoga je uporaba Grayevog koda umjesto prirodnog binarnog koda često bolje rješenje. Negativna strana uporabe Grayevog koda jest dodatni utrošak vremena koji je potreban kako bi se došlo do cjelobrojne vrijednosti koja je kodirana binarnim uzorkom koji tumačimo u skladu s Grayevim kodom. Jednom kad smo očitali vrijednost k, dalje je postupak identičan prethodno opisanom (linearno skaliranje vrijednosti na interval $[x_{min}, x_{max}]$).

Primjena na numeričke probleme više varijabli

Pretpostavimo da tražimo optimum funkcije od više varijabli. Postupak prikaza rješenja jednostavno je proširenje prethodno opisanog postupka za prikaz rješenja problema s jednom varijablom: za svaku je varijablu potrebno definirati prostor pretraživanja i željeni broj bitova (ili kvant pretrage iz kojeg se izračuna potreban broj bitova). Nakon toga binarni prikaz naprosto je spoj binarnih prikaza za svaku od varijabli. Evo primjera.

Primjer: 4

Pretpostavimo da tražimo optimum funkcije $f(x, y, z)$ od 3 varijable. Neka je područje pretrage za sve tri varijable isto: $[-2, 2]$. Kvant pretrage za varijable x i y pri tome treba biti 10^{-1} dok za varijablu z treba biti za red veličine finiji: 10^{-2}. Koliko bitova trebamo za binarni prikaz? Dajte jedan primjer binarnog prikaza i očitajte preslikane vrijednosti uz pretpostavku uporabe prirodnog binarnog koda.

Rješenje:

Imamo 3 varijable. To znači da će ukupni broj bitova biti jednak

$$n = n_x + n_y + n_z$$

gdje su n_x, n_y i n_z brojevi bitova potrebni za varijable x, y i z. Za varijable x i y kvant je 10^{-1} što uz prostor pretraživanja od $[-2, 2]$ zahtjeva 6 bitova po varijabli (prema izrazu (3.2)). Za varijablu z kvant je 10^{-2} što uz prostor pretraživanja od $[-2, 2]$ zahtjeva 9 bitova. Slijedi da je $n_x = 6$, $n_y = 6$ i $n_z = 9$, odnosno $n = 6 + 6 + 9 = 21$ bit. Stoga će binarni prikaz imati 21 bit.

Prvih 6 bitova će predstavljati varijablu x, sljedećih 6 varijablu y i posljednjih 9 varijablu z. Struktura binarnog prikaza bit će dakle `xxxxxxyyyyyyzzzzzzzzz`.

Neka je sada jedno konkretno rješenje `001100100011001100110`. Grupirajmo bitove da nam bude lakše očitati vrijednosti: `001100 100011 001100110`. Očitavamo: $k_x = 2^3 + 2^2 = 12$, $k_y = 2^5 + 2^1 + 2^0 = 35$ te $k_z = 2^6 + 2^5 + 2^2 + 2^1 = 102$.

$$\begin{aligned} x &= x_{min} + \frac{k_x}{2^{n_x} - 1} \cdot (x_{max} - x_{min}) \\ &= -2 + \frac{12}{63} \cdot (2 - (-2)) \\ &= -2 + \frac{12}{63} \cdot 4 \\ &= -1.23810. \end{aligned}$$

$$\begin{aligned} y &= y_{min} + \frac{k_y}{2^{n_y} - 1} \cdot (y_{max} - y_{min}) \\ &= -2 + \frac{35}{63} \cdot (2 - (-2)) \\ &= -2 + \frac{35}{63} \cdot 4 \\ &= 0.22222. \end{aligned}$$

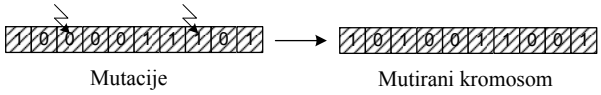
$$\begin{aligned} z &= z_{min} + \frac{k_z}{2^{n_z} - 1} \cdot (z_{max} - z_{min}) \\ &= -2 + \frac{102}{511} \cdot (2 - (-2)) \\ &= -2 + \frac{102}{511} \cdot 4 \\ &= -1.20157. \end{aligned}$$

Modificiranje rješenja

Pogledajmo sada kako se može raditi modificiranje postojećeg rješenja. Primjerice, kod genetskog algoritma te algoritma klonske selekcije ovo će odgovarati operatoru mutacije rješenja. Uobičajena izvedba jest definirati vjerojatnost promjene bita p_m, koja uobičajeno iznosi između 1% i 5% (a često je i manja). Potom se svaki bit rješenja mutira s tom vjerojatnošću. Pseudokod 3.1 ilustrira izvedbu ovog postupka uz pretpostavku da je funkcija `slucajni_broj(0,1)` funkcija koja vraća slučajni broj iz intervala $[0, 1]$ i to s uniformnom razdiobom. Djelovanje opisanog postupka prikazano je na slici 3.1.

Pseudokod 3.1 Izvedba mutiranja rješenja s binarnim prikazom.

```
mutiraj(rjesenje[1 do n]: bit, pm: double)
ponavljaj za i iz 1 do n
  double r = slucajni_broj(0,1);
  ako r < pm tada
    rjesenje[i] = not rjesenje[i];
  kraj ako
kraj
```

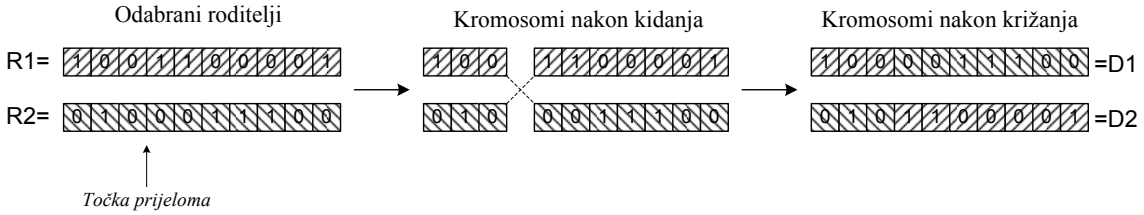


Slika 3.1: Djelovanje mutacije.

Kombiniranje rješenja

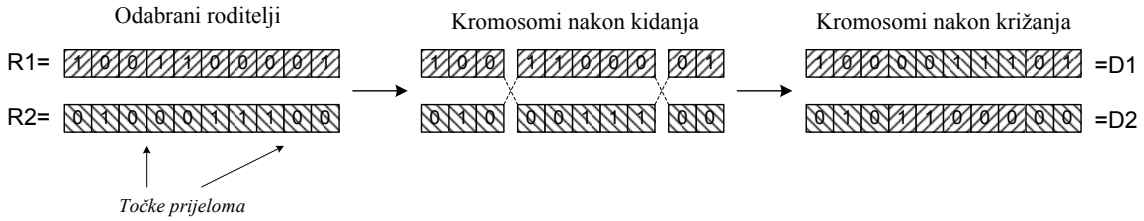
U svrhu kombiniranja rješenja razvijen je niz postupaka. U kontekstu evolucijskih algoritama ovi se postupci nazivaju operatorima križanja. Za potrebe primjera pretpostavimo da smo odabrali dva rješenja (koje ćemo nazvati roditeljima): $R1 = 1001100001$ i $R2 = 0100011100$.

Križanje s jednom točkom prekida Roditelji se poslože jedan ispod drugoga – vidi sliku 3.2. Posredstvom slučajnog mehanizma odabere se točka kidanja zajednička za oba roditelja. Potom se stvaraju dva nova rješenja - djeteta $D1$ i $D2$: prvo dobiva prvi dio bitova roditelja $R1$ i drugi dio bitova roditelja $R2$, a drugo dijete dobiva prvi dio bitova roditelja $R2$ i drugi dio bitova roditelja $R1$. Postupak je ilustriran na slici 3.2. Što ćemo napraviti s to dvoje djece (hoćemo li ih uzeti oba, jedno od njih ili niti jedno) nije definirano operatorom križanja – ta odluka ovisi o konkretnoj inačici optimizacijskog algoritma koju implementiramo.



Slika 3.2: Križanje s jednom točkom prijeloma.

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 korištenih za prikaz rješenja). Primjerice, slučaj za $t = 2$, te s točkama prijeloma nakon 3. i 8. bita prikazan je na slici 3.3. I ova verzija operatora križanja uobičajemo stvara dva djeteta uzimanjem cik-cak segmenata bitova iz jednog pa drugog roditelja.



Slika 3.3: Križanje s t-točaka prijeloma.

Uniformno križanje Uniformno križanje može se promatrati kao "oslabljeno" križanje s $k-1$ točkom prekida (gdje je k broj bitova korištenih za prikaz rješenja). Bitovni prikazi rješenja oba roditelja raspadnu se nakon svakog bita, i potom svako dijete bira što će uzeti od kojeg roditelja; pri tome se alterniranje ne treba raditi nakon svakog bita već je moguće uzeti i veće segmente iz svakog roditelja. Ovo se u računalnim programima najčešće izvodi na sljedeći način:

```
D1 = R1 · R2 + R · (R1 ⊕ R2)
D2 = R1 · R2 + !R · (R1 ⊕ R2)
```

pri čemu znak '$\cdot$' predstavlja operator *logičko-I* nad pojedinim bitovima, znak '$+$' *logičko-ILI* nad pojedinim bitovima, znak '$\oplus$' *logičko-isključivo-ILI* nad pojedinim bitovima, te uskličnih 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.

3.1.2 Prikaz poljem decimalnih brojeva

Umjesto uporabe binarnog prikaza, rješenja optimizacijskih problema ponekad je praktičnije prikazivati direktno decimalnim brojevima. Primjerice, ako je zadatak pronaći maksimum funkcije $f(x, y, z)$, tada se kao prikaz rješenja direktno može koristiti polje (ili u nekim programskim jezicima lista poduprta poljem) od tri elementa. Za funkciju od n varijabli može se direktno koristiti polje od n elemenata. U tom slučaju nestaje potreba za trošenjem vremena na dekodiranje rješenja čime algoritam postaje efikasniji. Međutim, kako tada više ne radimo s binarnim prikazom, potrebno je definirati na koji se način mogu izvesti potrebni operatori.

Modificiranje rješenja

Modificiranje rješenja kod ovog se zapisa najčešće izvodi dodavanjem malog slučajnog pomaka svakoj varijabli. Primjerice, neka nam je na raspolaganju funkcija `rand_normal(mi,sigma)` koja vraća slučajni broj iz normalne distribucije s parametrima `mi` (srednja vrijednost) i `sigma` (standardna devijacija), mutiranje možemo izvesti kako je prikazano u pseudokodu 3.2.

Pseudokod 3.2 Izvedba mutiranja rješenja prikazanog decimalnim brojevima.

```
mutiraj(rjesenje[1 do n]: double, s: double)
ponavljaj za i iz 1 do n
  rjesenje[i] += rand_normal(0,s);
kraj
```

Parametar `s` je pri tome standardna devijacija uz koju generiramo slučajne brojeve oko srednje vrijednosti, a prikazani algoritam to radi oko vrijednosti 0. Kontroliranjem parametra `s` možemo dakle određivati koliko jako će, u prosjeku, mutacija mijenjati trenutno rješenje.

Alternativna izvedba mutacije mogla bi svakom elementu polja dodati slučajno generirani broj iz intervala $[m, M]$ prema uniformnoj distribuciji. Tako primjerice, ako imamo na raspolaganju funkciju `rand_uniform()` koja vraća slučajni broj iz intervala $[0, 1]$ prema uniformnoj distribuciji, mutaciju možemo izvesti kako je prikazano pseudokodom 3.3.

Pseudokod 3.3 Izvedba mutiranja rješenja prikazanog decimalnim brojevima.

```
mutiraj(rjesenje[1 do n]: double, m,M: double)
ponavljaj za i iz 1 do n
  rjesenje[i] += rand_uniform()*(M-m) + m;
kraj
```

Prednost uporabe normalne distribucije nad uniformnom distribucijom leži u tom što normalna distribucija u prosjeku radi mala odstupanja oko srednje vrijednosti, no može generirati i veliko odstupanje. U tom smislu, mutiranje normalnom distribucijom će češće generirati nova rješenja koja su u okolici trenutnog rješenja, dok će povremeno generirati rješenja koja su dosta daleko. To nemamo s uniformnom distribucijom: njome ćemo generirati s jednakom vjerojatnošću rješenja koja su od trenutnog udaljena do $[m, M]$ i nikada rješenja koja su dalje od toga.

Kombiniranje rješenja

Kombiniranje rješenja također je moguće raditi na više načina. Ovdje ćemo spomenuti samo neke. Neka su odabrana dva roditelja: $R_1 = (c_1^1, \ldots, c_n^1)$ i $R_2 = (c_1^2, \ldots, c_n^2)$, gdje je c_i^j i-ta komponenta j-tog roditelja.

Diskretno križanje Poznato je pod engleskim nazivom *Discrete crossover* [Mühlenbein and Schlierkamp-Voosen, 1993]; stvara jedno dijete. Provodi se tako da se i-ta komponenta djeteta postavi na slučajno odabranu vrijednost iz skupa $\{c_i^1, c_i^2\}$; drugim riječima, i-ta komponenta djeteta će se izravno preuzeti ili od prvog roditelja ili od drugog roditelja.

Ravno križanje Poznato je pod engleskim nazivom *flat crossover* [Radcliffe, 1991] i provodi se tako da se i-ta komponenta djeteta postavi na vrijednost koja je izvučena iz uniformne razdiobe nad intervalom $[\min(c_i^1, c_i^2), \max(c_i^1, c_i^2)]$.

Jednostavno križanje Poznato je pod engleskim nazivom *simple crossover* [Wright, 1991, Michalewicz, 1992] i provodi se tako da se odabere jedna točka prekida $i \in \{1, 2, \ldots, n-1\}$, i potom se generiraju dva djeteta:

$$H_1 = (c_1^1, c_2^1, \ldots, c_i^1, c_{i+1}^2, \ldots, c_n^2),$$
$$H_2 = (c_1^2, c_2^2, \ldots, c_i^2, c_{i+1}^1, \ldots, c_n^1).$$

Aritmetičko križanje Poznato je pod engleskim nazivom *arithmetical crossover* [Michalewicz, 1992]; stvara dva djeteta:

$$H_1 = (h_1^1, \ldots, h_n^1) \quad \text{i}$$
$$H_2 = (h_1^2, \ldots, h_n^2),$$

pri čemu se definira konstanta $\lambda \in [0, 1]$ i potom svaka komponenta djeteta H_1 i H_2 postaje linearna kombinacija komponenata roditelja:

$$h_i^1 = \lambda \cdot c_i^1 + (1 - \lambda) \cdot c_i^2,$$
$$h_i^2 = \lambda \cdot c_i^2 + (1 - \lambda) \cdot c_i^1.$$

***BLX-α* križanje** Poznato je pod engleskim nazivom *BLX-α crossover* [Eshelman and Schaffer, 1993]; stvara jedno dijete. Označimo najprije $c_{i,min} = min(c_i^1, c_i^2)$, $c_{i,max} = max(c_i^1, c_i^2)$, $I_i = c_{i,max} - c_{i,min}$. Dijete:

$$H = (h_1, \ldots, h_n)$$

se gradi tako da je komponenenta h_i definirana izrazom:

$$h_i = \text{izaberi uniformno iz intervala } [c_{i,min} - I_i \cdot \alpha, c_{i,max} + I_i \cdot \alpha].$$

Linearno križanje Poznato je pod engleskim nazivom *linear crossover* [Wright, 1991]; stvara tri dijeteta: H_1, H_2 i H_3. Pri tome je $h_i^1 = \frac{1}{2}c_i^1 + \frac{1}{2}c_i^2$, $h_i^2 = \frac{3}{2}c_i^1 - \frac{1}{2}c_i^2$ te $h_i^3 = -\frac{1}{2}c_i^1 + \frac{3}{2}c_i^2$.

Prošireno linijsko križanje Poznato je pod engleskim nazivom *Extended line crossover* [Mühlenbein and Schlierkamp-Voosen, 1993]; stvara jedno dijete. Provodi se tako da se i-ta komponenta djeteta postavi na slučajno odabranu vrijednost $h_i = c_i^1 + \alpha \cdot (c_i^2 - c_i^1)$, gdje je α vrijednost koja je slučajno izvučena iz uniformne distribucije nad intervalom $[-0.25, 1.25]$. Potrebno je uočiti da se za provođenje križanja najprije bira vrijednost za α i potom se sve komponente djeteta križaju uporabom te fiksirane vrijednosti.

Prošireno komponentno križanje Poznato je pod engleskim nazivom *Extended intermediate crossover* [Mühlenbein and Schlierkamp-Voosen, 1993]; stvara jedno dijete. Provodi se tako da se i-ta komponenta djeteta postavi na slučajno odabranu vrijednost $h_i = c_i^1 + \alpha_i \cdot (c_i^2 - c_i^1)$, gdje je α_i vrijednost koja je slučajno izvučena iz uniformne distribucije nad intervalom $[-0.25, 1.25]$. Ovo je osnovna razlika u odnosu na prethodno križanje: za svaku komponentu i slučajno se bira vrijednost α_i.

Osim navedenih, postoji još i niz drugih operatora.

3.1.3 Prikaz permutacijama i matricama

Postoji niz problema koji se mogu svesti na problem permutacija. Možda je najpoznatiji primjer te vrste problema upravo problem trgovačkog putnika. Ako indeksiramo sve gradove (neka ih ima n) koje trgovački putnik mora obići, tada je svaki Hamiltonov ciklus prikaziv kao jedna permutacija n-torke $(1, 2, \ldots, n)$. Primjerice, n-torka $(1, 3, 2, \ldots, 26, 4)$ bi značila da putnik kreće iz grada 1, odlazi u grad 3, potom u grad 2, potom ..., potom u grad 26 pa grad 4 i obilazak završava povratkom u grad 1. Permutacijski prikaz je specifičan po tome što zahtjeva potpuno novi skup operatora koji mogu djelovati nad rješenjima. Primjerice, jednostavna mutacija koja b indeks grada koji je na prvom mjestu slučajno promijenio u neki drugi indeks bi automatski stvorila nevažeće rješenje: izgubili bismo iz obilaska grad 1 (u našem slučaju) a dva puta bismo obišli neki drugi grad. Klasični operatori križanja također bi svi redom generirali nevažeća rješenja. Stoga je za ovakav prikaz razvijen novi skup operatora. Spomenimo samo da se modifikacija postojećeg rješenja može provoditi operatorom zamjene: slučajno odabereno dvije pozicije u polju i njihov sadržaj zamijenimo. Time imamo garanciju da je nastalo rješenje opet valjana permutacija. Uz slične ograde provode se i operatori križanja.

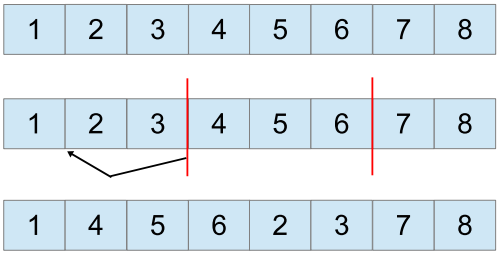
Matrični prikaz također je primjer prikaza koji se u praksi zna koristiti za rješavanje različitih problema. Primjenjiv je na problem trgovačkog putnika. U tom slučaju radili bismo s matricom $n \times n$ gdje element a_{ij} ima vrijednost 1 ako trgovački putnik iz grada i treba otići u grad j. Odmah se može uočiti da postoje ograničenja na legalne matrice: u retku i samo na jednom mjestu smije biti vrijednost i jer trgovački putnik iz grada i smije otići u samo jedan sljedeći grad. Drugi primjer uporabe matričnog prikaza jest rješavanje problema evolucije arhitekture i težinskih faktora umjetnih neuronskih mreža.

Pogledajmo sada nekoliko načina provedbe izmjena i kombiniranja permutacijskih prikaza.

Modificiranje rješenja

Sve primjere u nastavku dat ćemo nad početnim rješenje koje je permutacija od osam elemenata: $(1, 2, 3, 4, 5, 6, 7, 8)$.

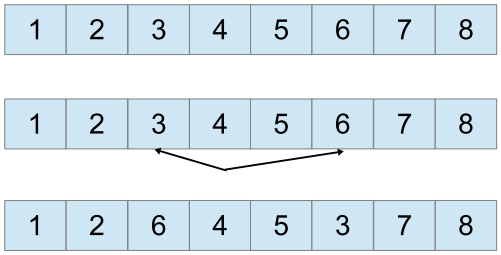
Mutacija premještanjem Poznata je pod engleskim nazivom *Displacement mutation* (DM) [Michalewicz, 1992]. Nad početnim rješenjem se slučajno odaberu dva mjesta. Potom se taj segment ukloni iz rješenja i ubaci nakon neke slučajno odabrane pozicije u preostalom rješenju. Primjer je prikazan na slici 3.4; u izvornom rješenju odabran je podniz $(4, 5, 6)$ i odlučeno je da se taj podniz premjesti nakon prvog elementa. Ovaj operator poznat je i pod nazivom *Cut mutation*.



Slika 3.4: Djelovanje mutacije DM.

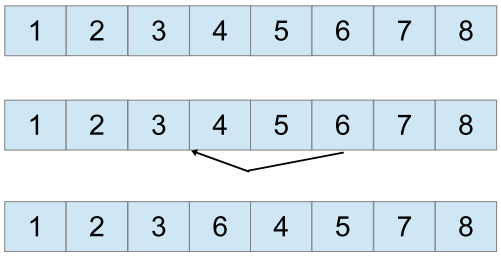
Mutacija zamjenom Poznata je pod engleskim nazivom *Exchange mutation* (EM) [Banzhaf, 1990]. Nad početnim rješenjem se slučajno odaberu dvije pozicije i elementi koji su na tim pozicijama se zamijene. Primjer je prikazan na slici 3.5; u izvornom rješenju odabrane su pozicija 3 i 6 i elementi koje se nalaze na tim pozicijama (također 3 i 6) su zamijenjeni. Ovaj operator poznat je i pod nazivom *Swap mutation*, *Point mutation*, *Reciprocal exchange mutation* te *Order based mutation*.

Mutacija umetanjem Poznata je pod engleskim nazivom *Insertion mutation* (ISM) [Fogel, 1988, Michalewicz, 1992]. Nad početnim rješenjem se slučajno odaberu jedna pozicija i element koji je na toj poziciji se ukloni i umetne na neku drugu slučajno odabranu poziciju. Primjer je prikazan na slici



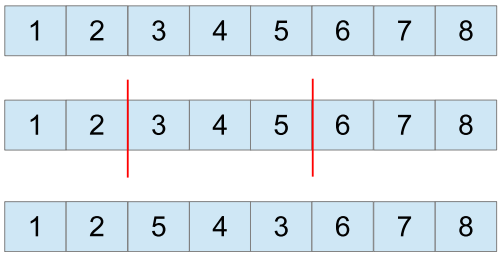
Slika 3.5: Djelovanje mutacije EM.

3.6; u izvornom rješenju odabrana je pozicija 6 na kojoj se nalazi element 6; taj je element uklonjen i umetnut je nakon slučajno odabrane treće pozicije. Ovaj operator poznat je i pod nazivom *Position based mutation*.



Slika 3.6: Djelovanje mutacije ISM.

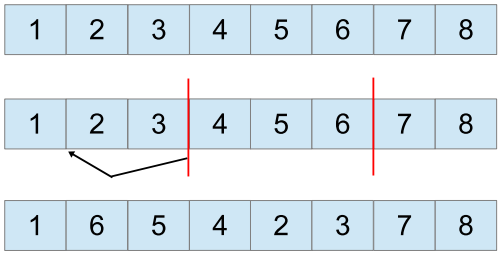
Jednostavna mutacija inverzijom Poznata je pod engleskim nazivom *Simple inversion mutation* (SIM) [Holland, 1975]. Nad početnim rješenjem se slučajno odaberu dva mjesta i tako selektirani podniz se prepiše obrnutim poretkom (reverzira se). Primjer je prikazan na slici 3.7; u izvornom rješenju odabran je podniz $(3, 4, 5)$ i zamijenjen je reverznim podnizom $5, 4, 3$.



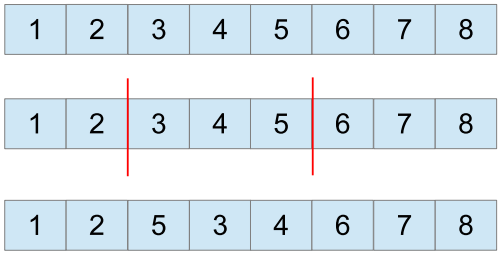
Slika 3.7: Djelovanje mutacije SIM.

Mutacija inverzijom Poznata je pod engleskim nazivom *Inversion mutation* (IVM) [Fogel, 1990, 1993]. Nad početnim rješenjem se slučajno odaberu dva mjesta. Potom se taj segment ukloni iz rješenja i obrnutim poretkom ubaci nakon neke slučajno odabrane pozicije u preostalom rješenju. Primjer je prikazan na slici 3.8; u izvornom rješenju odabran je podniz $(4, 5, 6)$ i odlučeno je da se njegov reverzni podniz $6, 5, 4$ ubaci nakon prvog elementa.

Mutacija miješanjem Poznata je pod engleskim nazivom *Scramble mutation* (SM) [Syswerda, 1991]. Nad početnim rješenjem slučajno se odabiru dva mjesta i svi elementi između se na slučajni način izmiješaju. Primjer je prikazan na slici 3.9; u izvornom rješenju odabran je podniz $(3, 4, 5)$ koji je potom izmiješan u podniz $(5, 3, 4)$.



Slika 3.8: Djelovanje mutacije IVM.



Slika 3.9: Djelovanje mutacije SM.

Kombiniranje rješenja

Sve primjere u nastavku dat ćemo nad dva početna rješenja (roditelj 1 i roditelj 2) koji su svaki permutacija od osam elemenata.

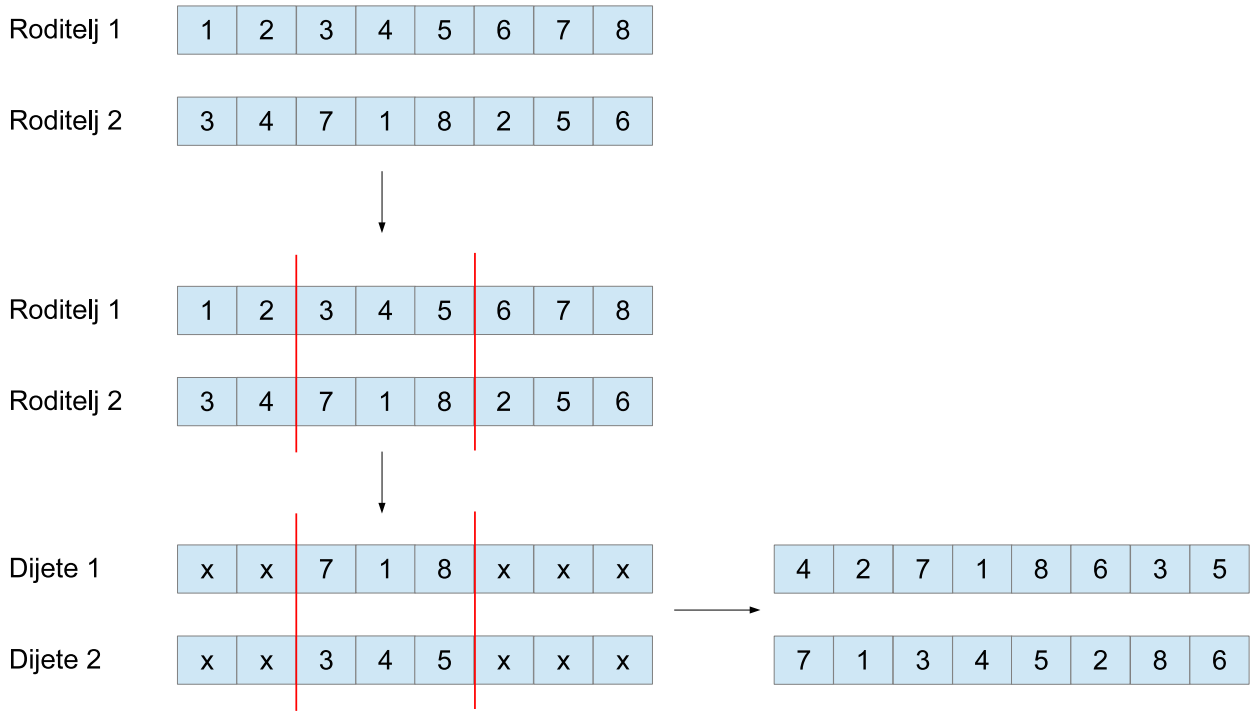
Djelomično-preslikano križanje Poznato je pod engleskim nazivom *Partially-mapped crossover* (PMX) [Goldberg and Lingle, 1985]. Slučajno se odaberu dva mjesta koja definiraju segment u oba roditelja – primjer je prikazan na slici 3.10 gdje je slučajno odabran segment koji počinje nakon druge pozicije i završava nakon pete pozicije; duljina segmenta je 3. Svaki element segmenta u prvom i drugom roditelju sada definira preslikavanje: prvi element segmenta definira da se vrijednost 3 preslikava u vrijednost 7, tj. $3 \leftrightarrow 7$; drugi element segmenta definira preslikavanje $4 \leftrightarrow 1$ dok treći element segmenta definira preslikavanje $5 \leftrightarrow 8$.

Grade se dva djeteta. U prvo dijete se iskopira odabrani segment drugog roditelja a u drugo dijete odabrani segment prvog roditelja. Sve ostale pozicije u oba djeteta ostaju prazne. Postupak se sada nastavlja tako da se prazna mjesta djeteta i popunjavanju vrijednostima koje se na tim pozicijama nalaze u roditelju i – osim ako bi se time unio duplikat. U tom slučaju na poziciju se kopira element koji je određen definiranim preslikavanjem.

Napravimo to za dijete 1. Početno, u djetetu 1 se nalaze vrijednosti $7, 1, 8$. Na prvu poziciju djeteta 1 treba kopirati vrijednost koja se nalazi na prvoj poziciji roditelja 1: to je 1. Međutim, dijete 1 već ima tu vrijednost: stoga se umjesto nje upisuje vrijednost u koju se 1 preslikava – to je vrijednost 4. Na drugu poziciju u djetetu 1 kopira se vrijednost s druge pozicije roditelja 1: to je 2 i tu nema problema. Pozicije 3, 4 i 5 u djetetu 1 već su popunjene. Na poziciju 6 djeteta 1 kopira se vrijednost s pozicije 6 roditelja 1: to je 6. Na poziciju 7 trebalo bi kopirati vrijednost 7 no ta već postoji, stoga se upisuje preslikani par: 3. Slično, na posljednju poziciju nije moguće upisati vrijednost 8 jer ona već postoji; umjesto nje upisuje se preslikani par 5.

Popunjavanje djeteta 2 ide na sličan način samo što se kopiraju vrijednosti iz roditelja 2. Konačni rezultat kao i međukoraci ovog križanja prikazani su na slici 3.10.

Potrebno je napomenuti da je moguća situacija u kojoj se identificira duplikat, pogleda se na što je on preslikan i ta vrijednost je opet duplikat (odnosno i ona je već prisutna u djetetu); razrješavanje ide iterativno – opet se pogleda u što je ta vrijednost preslikana i tako dalje sve dok se ne dođe do vrijednosti koja u djetetu još ne postoji.



Slika 3.10: Djelovanje križanja PMX.

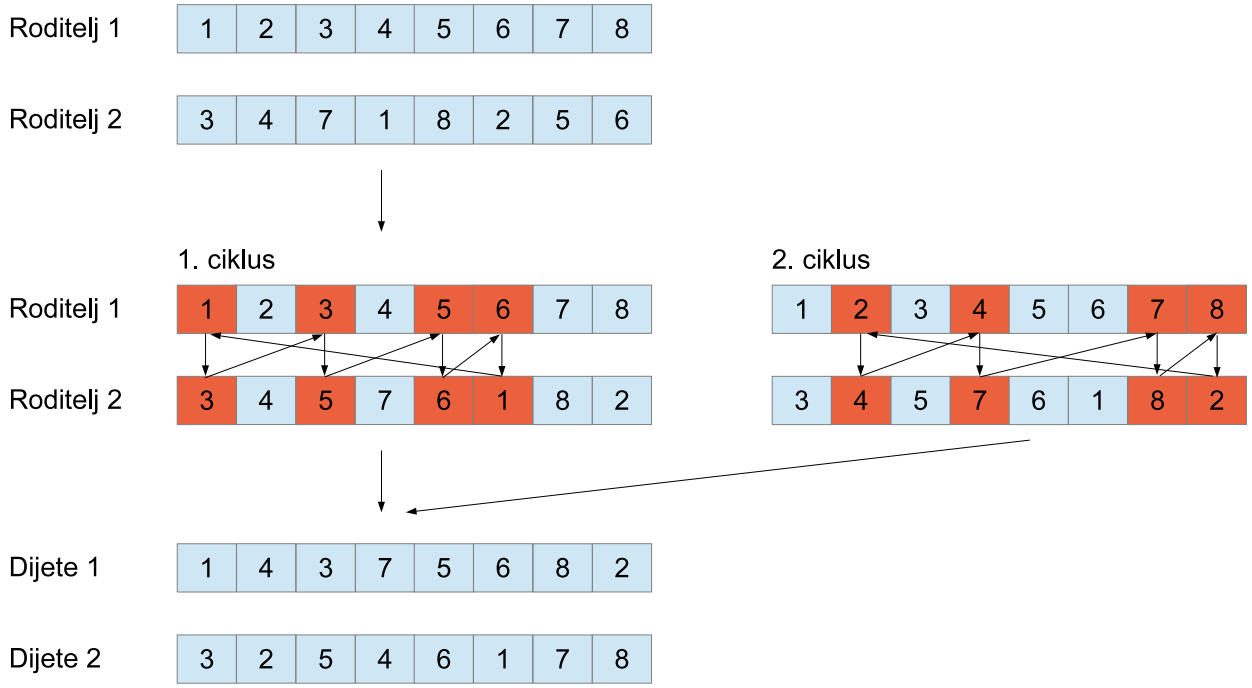
Križanje ciklusa Poznato je pod engleskim nazivom *Cycle crossover* (CX) [Oliver et al., 1987]. Provođenje križanja temelji se na identificiranju ciklusa, a prikazano je na slici 3.11. Dva se roditelja potpišu jedan ispod drugog, i potom se krene s prim elementom u prvom roditelju. Pogleda se koji se element u drugom roditelju nalazi na istoj poziciji (vertikalna strelica prema dolje); potom se u prvom roditelju potraži na kojoj je poziciji taj element; s te se pozicije opet pogleda što je na na toj istoj poziciji u drugom roditelju; opet se traži pozicija u prvom roditelju koja ima istu vrijednost i postupak se ponavlja. Ovaj će postupak završiti u konačnom broju koraka, i u oba će roditelja selektirati neke elemente. Time je pronađen jedan ciklus. Postupak se nastavlja od prvog sljedećeg neselektiranog elementa u prvom roditelju kako bi pronašli drugi ciklus, potom treći ciklus, itd. Broj ciklusa koji će nastati ovisi o konkretnim roditeljima koji se križaju. Prvo dijete se puni elementima (i pozicijama) prvog ciklusa iz prvog roditelja, drugog ciklusa iz drugog roditelja, trećeg ciklusa iz prvog roditelja, četvrtog ciklusa iz drugog roditelja, itd. Drugo dijete se puni obratno: elementima prvog ciklusa iz drugog roditelja, drugog ciklusa iz prvog roditelja, itd.

Pogledajmo to na konkretnom primjeru ilustriranom na slici 3.11. Tražimo prvi ciklus. Gledamo prvu poziciju prvog roditelja: vrijednost je 1, i na istoj poziciji u roditelju 2 nalazi se vrijednost 3. Stoga tražimo u prvom roditelju poziciju na kojoj se nalazi upisana vrijednost 3, i onda opet gledamo što je na toj poziciji u roditelju 2: to je vrijednost 5. Tražimo u prvom roditelju poziciju na kojoj se nalazi upisana vrijednost 5, i onda gledamo što je na toj poziciji u roditelju 2: to je vrijednost 6. Tražimo u prvom roditelju poziciju na kojoj se nalazi upisana vrijednost 6, i onda gledamo što je na toj poziciji u roditelju 2: to je vrijednost 1. Kako smo traženje ciklusa upravo započeli s vrijednosti 1 iz prvog roditelja, ciklus je zatvoren – ponavljanjem postupka zavrtili bismo se u krug. Utvrdili smo da ciklusu 1 pripadaju elementi na pozicijama 1, 3, 5 i 6. Za prvog roditelja to su vrijednosti 1, 3, 5 i 6 dok su za drugog roditelja to vrijednosti 3, 5, 6 i 1 (uočite da je to upravo permutacija elemenata identificiranih kod prvog roditelja).

Nastavljamo pretragu sljedećeg ciklusa. Prvi nesektirani element prvog roditelja je na poziciji 2: praćenjem dolazimo do ciklusa koji čine pozicije 2, 4, 7 i 8. Za prvog roditelja to su vrijednosti 2, 4, 7 i 8 dok su za drugog roditelja to vrijednosti 4, 7, 8 i 2. Kako smo ovime pokupili sve pozicije, treći ciklus ne postoji.

Prvo dijete sada popunjavamo tako da na pozicije koje pripadaju prvom ciklusu upišemo elemente

od iz prvog roditelja: na pozicije 1, 3, 5 i 6 upisujemo 1, 3, 5 i 6 te na pozicije koje pripadaju drugom ciklusu upišemo elemente iz drugog roditelja: na pozicije 2, 4, 7 i 8 upisujemo 4, 7, 8 i 2. Drugo dijete sada popunjavamo tako da na pozicije koje pripadaju prvom ciklusu upišemo elemente od iz drugog roditelja: na pozicije 1, 3, 5 i 6 upisujemo 3, 5, 6 i 1 te na pozicije koje pripadaju drugom ciklusu upišemo elemente iz prvog roditelja: na pozicije 2, 4, 7 i 8 upisujemo 2, 4, 7 i 8.

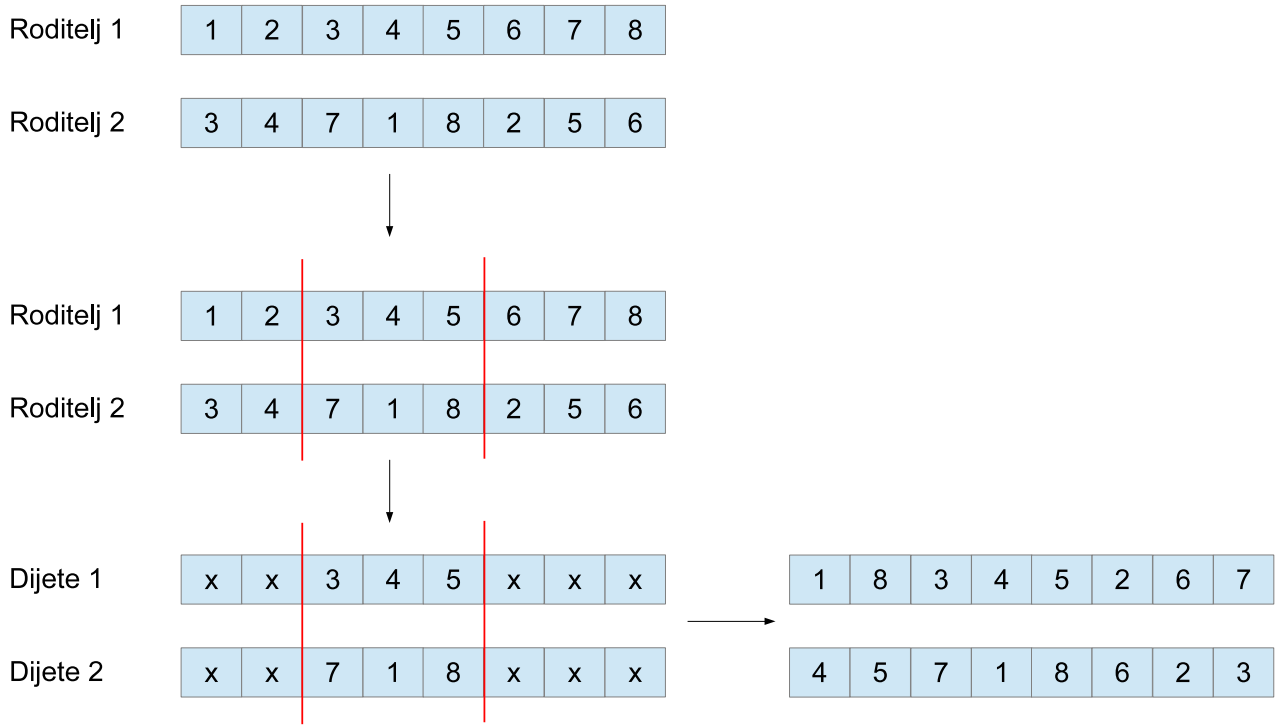


Slika 3.11: Djelovanje križanja CX.

Križanje poretka Poznato je pod engleskim nazivom *Order crossover* (OX1) [Davis, 1985]. Križanje se temelji na svojstvu koje vrijedi ako se rješava problem trgovačkog putnika: ono što je važno jest redosljed kojim se obilaze gradovi a ne njihova apsolutna pozicija u rješenju. Postupak kreće tako da se nasumično odaberu dvije pozicije koje će u oba roditelja označiti segment. Postupak je prikazan na slici 3.12. Stvore se dva djeteta; u prvo se prekopira samo označeni segment iz prvog roditelja a u drugo se prekopira označeni segment iz drugog roditelja. Potom se ostatak i-tog djeteta, počev od desnog kraja označenog segmenta, pokuša nadopuniti preostalim pozicijama iz onog suprotnog roditelja (pozicije se prolaz od kraja segmenta do kraja rješenja pa cirkularno od početka prema segmentu).

Pogledajmo to na konkretnom primjeru prikazanom na slici 3.12. Odabran je segment koji uključuje pozicije 3, 4 i 5. U prvom roditelju na tim pozicijama su vrijednosti 3, 4 i 5 i to je prekopirano u prvo dijete; u drugom roditelju na tim pozicijama su vrijednosti 7, 1 i 8 i to je prekopirano u drugo dijete. Prvo dijete sada popunjavamo počev od kraja segmenta: od pozicije 6. Kandidati za umetanje su vrijednosti u roditelju 2 od pozicije 6 ciklički na dalje, upravo tim redosljedom: 2, 5, 6, 3, 4, 7, 1, 8. Stoga u prvo dijete na poziciju 6 upisujemo vrijednost 2, vrijednost 5 preskačemo jer je već prisutna, na poziciju 7 upisujemo 6, vrijednosti 3 i 4 preskačemo jer ih već imamo, na poziciju 8 upisujemo 7, na poziciju 1 upisujemo 1 i posljednji preostali element 8 upisujemo na poziciju 2.

Drugo dijete također popunjavamo počev od kraja segmenta: od pozicije 6. Kandidati za umetanje su vrijednosti u roditelju 1 od pozicije 6 ciklički na dalje, upravo tim redosljedom: 6, 7, 8, 1, 2, 3, 4, 5. Stoga u drugo dijete na poziciju 6 upisujemo vrijednost 6, vrijednosti 7, 8 i 1 preskačemo jer su već prisutne, na poziciju 7 upisujemo 2, na poziciju 8 upisujemo 3, na poziciju 1 upisujemo 4 i na poziciju 2 upisujemo 5.



Slika 3.12: Djelovanje križanja OX1.

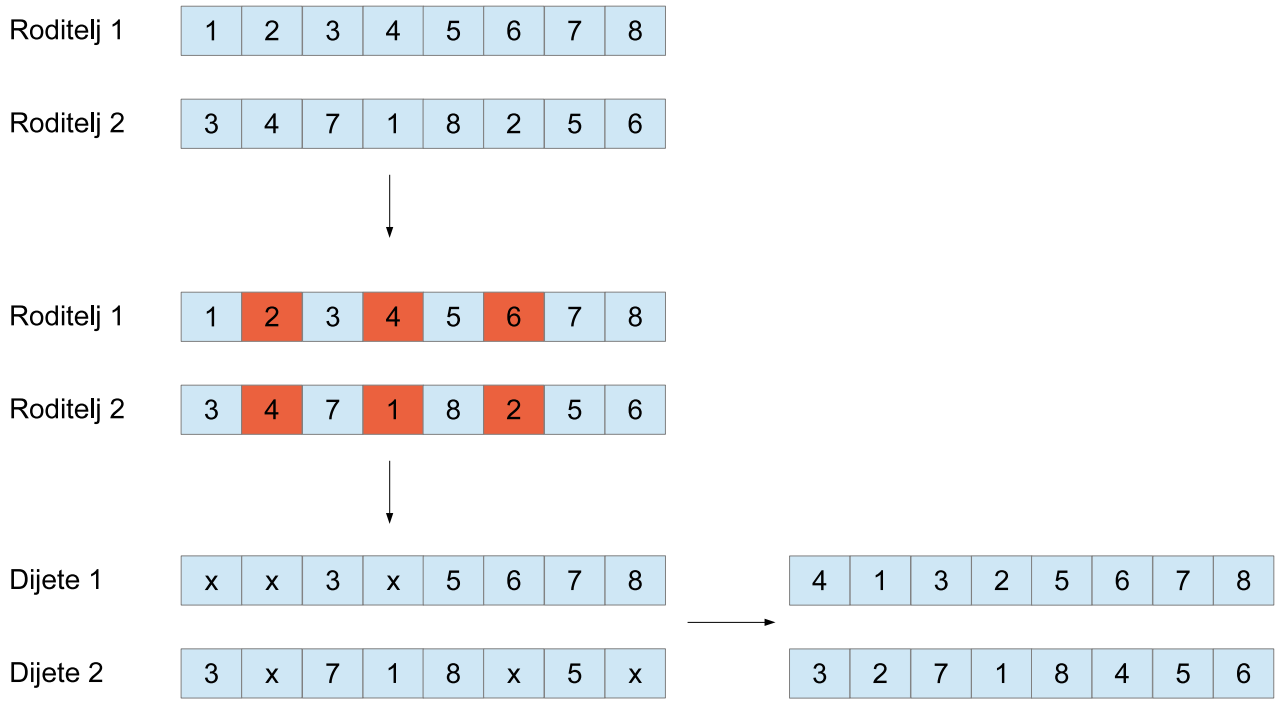
Križanje temeljeno na poretku Poznato je pod engleskim nazivom *Order-based crossover* (OX2) [Syswerda, 1991]. Postupak je prikazan na slici 3.13. Slučajno se odabere nekoliko pozicija. Da bi se dobilo prvo dijete, u njega se prekopira prvi roditelj osim onih pozicija koje sadrže vrijednosti koje se u drugom roditelju nalaze na selektiranim pozicijama. U ta prazna mjesta djeteta 1 prekopiraju se selektirane vrijednosti iz roditelja 2 redosljedom kojim su tamo prisutne. Drugo dijete stvara se sličan način, pri čemu se kopira drugi roditelj a gledaju se selektirane vrijednosti u prvom roditelju.

Pogledajmo to na konkretnom primjeru koji prikazuej slika 3.13. Neka su nasumice odabrane pozicije 2, 4 i 6. U drugom roditelju na tim su pozicijama vrijednosti 4, 1 i 2 (tim redosljedom); u prvom roditelju te su vrijednosti prisutne na pozicijama 4, 1 i 2. Prvo dijete bit će tada kopija prvog roditelja osim što će pozicije 1, 2 i 4 ostati prazne: $xx3x5678$. Prazne pozicije popunit ćemo vrijednostima iz roditelja 2 redosljedom kojim su u njmu navedene: na prvo slobodno mjesto upisujemo 4, na drugo slobodno mjesto 1 te na posljednje slobodno mjesto upisujemo 2.

Izgradimo sada i drugo dijete. U prvom roditelju se na selektiranim pozicijama nalaze vrijednosti 2, 4 i 6. U drugom su roditelju te vrijednosti na pozicijama 6, 2 i 8. Stoga će drugo dijete biti kopija drugog roditelja, ali bez iskopiranih pozicija 2, 6 i 8: $3x718x5x$. Prazna mjesta popunjavamo selektiranim vrijednostima iz roditelja 2 redosljedom kojim su u njemu navedene: 2, 4 i 6; stoga na poziciji 2 upisujemo 2, na poziciju 6 upisujemo 4 i na poziciji 8 upisujemo 6.

Križanje temeljeno na poziciji Poznato je pod engleskim nazivom *Position-based crossover* (POS) [Syswerda, 1991]. Postupak je prikazan na slici 3.14. Slučajno se odabere nekoliko pozicija. Postupak zahtjeva da u prvom djetetu na selektiranim pozicijama budu upravo elementi koji su na tim pozicijama u drugom roditelju; sva preostala mjesta popunjavaju se elementima prvog roditelja redosljedom kojim su u njemu prisutni. Kod drugog djeteta se zahtjeva da na selektiranim pozicijama budu upravo elementi koji su na tim pozicijama u prvom roditelju; sva preostala mjesta popunjavaju se elementima drugog roditelja redosljedom kojim su u njemu prisutni.

Pogledajmo to na konkretnom primjeru koji prikazuej slika 3.14. Neka su nasumice odabrane pozicije 2, 4 i 6. U drugom roditelju na tim su pozicijama vrijednosti 4, 1 i 2 (tim redosljedom). Stoga se u prvo dijete na te pozicije upisuju te vrijednosti. Time su u prvom djetetu preostale nepopunjene pozicije 1, 3, 5, 7 i 8. U njih redom upisujemo vrijednosti iz prvog roditelja: 1, 2, 3, 4, 5, 6, 7 i 8 pri



Slika 3.13: Djelovanje križanja OX2.

čemu izbacujemo vrijednosti koje su već prisutne; posljedica je da ćemo na te pozicije redom upisati: 3, 5, 6, 7 i 8.

U prvom roditelju na selektiranim su pozicijama vrijednosti 2, 4 i 6 (tim redosljedom). Stoga se u drugo dijete na te pozicije upisuju te vrijednosti. Time su u drugom djetetu preostale nepopunjene pozicije 1, 3, 5, 7 i 8. U njih redom upisujemo vrijednosti iz drugog roditelja: 3, 4, 7, 1, 8, 2, 5 i 6 pri čemu izbacujemo vrijednosti koje su već prisutne; posljedica je da ćemo na te pozicije redom upisati: 3, 7, 1, 8 i 5.

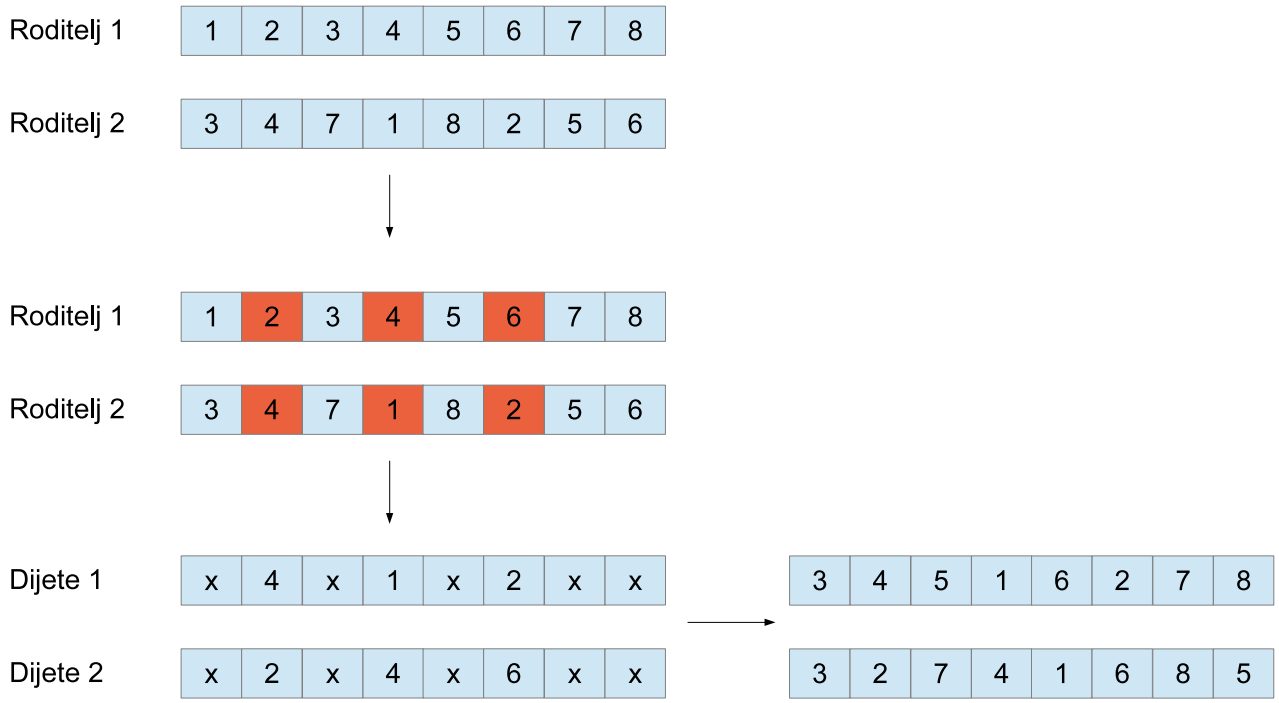
Osim spomenutih operatora, danas se u literaturi može pronaći još mnoštvo drugih operatora koji su razvijeni imajući na umu različite specifičnosti problema koji se rješavaju permutacijskim prikazom. Zainteresiranog čitatelja se upućuje na samostalnu pretragu.

3.1.4 Prikaz složenijim strukturama podataka

Ponekad su problemi čija tražimo rješenja takvi da su konceptualno vrlo daleko od domene brojeva. U tom slučaju su binarni prikaz kao i prikaz cijelim ili decimalnim brojevima neprikladni, i preporuča se razvoj prikaza rješenja složenijim strukturama podataka. Prednost razvoja takvog prikaza jest mogućnost "pametnog" razvoja operatora koji će brzo i efikasno djelovati nad rješenjem. Mana jest upravo poreba da se razvija još jedan novi skup operatora umjesto da se koriste postojeći. Razvoj vlastitog prikaza ilustrirat ćemo na problemu izrade rasporeda međuispita: postoji p termina u koje je potrebno smjestiti o kolegija kako bi svaki kolegij mogao održati međuispit. Tipično je $p << o$.

Najjednostavniji način prikaza rješenja jest uporabom vektora duljine o gdje je o upravo broj kolegija koje treba rasporediti. Elementi vektora su pri tome skalari koji mogu poprimiti vrijednosti iz skupa $\{1, \ldots, p\}$ gdje je p broj termina koji sudjeluju u izradi rasporeda. Ovakav način predstavljanja rješenja pogodan je i za često korištene evolucijske operatore: primjerice, uniformno križanje koje možemo provoditi po parovima elemenata dvaju vektora, mutaciju koju možemo provoditi nad elementima jednog vektora i sl. Međutim, iako je predloženi način predstavljanja rješenja najjednostavniji, on nije najpogodniji za vrednovanje rješenja. Naime, tijekom vrednovanja nas često zanimaju sljedeće informacije:

- koji su još kolegiji smješteni u isti termin kao i kolegij c_i,



Slika 3.14: Djelovanje križanja POS.

- koliko je ukupno studenata smješteno u termin t_j,
- koji su kolegiji smješteni u termin koji je odmah prije termina t_j (ili odmah nakon termina t_j) i mnoge druge.

Problemu prikaza rješenja bismo mogli pristupiti i s druge strane: umjesto da ga orijentiramo prema vektoru kolegija, možemo ga orijenitrati prema vektoru termina. Pri tome svaki termin pamti kolekciju kolegija koji su mu pridruženi. Ovakva struktura podataka može nam brzo dati odgovor na pitanja koja su orijentirana na termine (poput koji su sve kolegiji u menom terminu i sl.). Međutim, ovim pristupom postaju problematični upiti i operatori koji rade s kolegijima; primjerice, ako kolegij c_i želimo ubaciti u termin t_j, najprije moramo pretražiti sve termine kako bismo utvrdili u koji je termin taj kolegij bio prethodno smješten, potom ga trebamo ukloniti i tek tada ga možemo ubaciti u novi termin. Time je složenost izvođenja takve operacije često neprihvatljiva. Dodatno, ako se struktura podataka napravi loše, postoji mogućnost da će se pri tome često koristiti memorijski podsustav što bi moglo usporiti izvođenje programa.

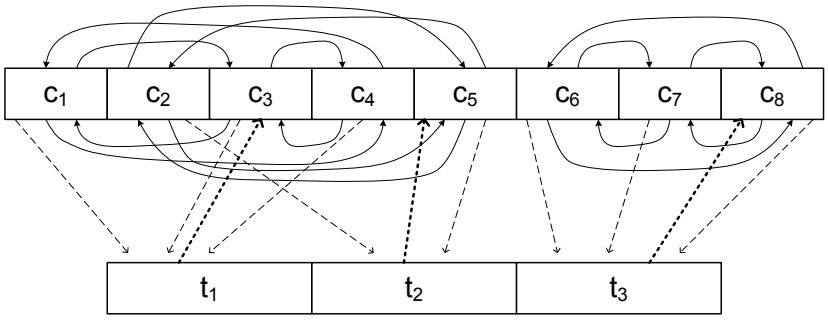
Stoga se kao jedno moguće rješenje nameće uporaba redundantnih struktura podataka koje će osigurati brzo izvođenje često korištenih operacija. Jedno takvo rješenje inspirirano radom [Talbi and Weinberg, 2007] te iskorišteno u radu [Čupić et al., 2009] prikazuje izvorni kod 3.1.

Ideja je sljedeća. Kromosom će paralelno voditi dvije povezane strukture podataka: vektor kolegija te vektor termina. Elementi vektora kolegija su objekti tipa KCourse, koji se mogu povezivati u dvostruko povezanu listu zahvaljujući varijablama previous i next. Te varijable nisu po tipu reference već cijeli brojevi koji imaju mogućnost pamćenja indeksa prethodnog/sljedećeg kolegija u polju svih kolegija kcourse jednog kromosoma. Vrijednost -1 tada služi kao zamjena za **null**-referencu. Osim ove dvije varijable, kolegij pamti i indeks termina u koji je smješten (varijabla term). Za pamćenje svih termina kromosom koristi polje objekata tipa KTerm. Svaki objekt tipa KTerm pamti indeks (varijabla someCourse) jednog od kolegija koji je smješten u taj termin (bilo kojeg) te koliko je ukupno kolegija smješteno u taj termin.

Uporabom ovakve strukture moguće je dobiti izvođenje najčešćih upita i operacija u složenosti O(1). Ovo ćemo pojasniti na primjeru koji pokazuje slika 3.15. Slika prikazuje raspored u kojem je osam kolegija smješteno u tri termina. Gornje polje je polje objekata KCourse. Vrijednosti varijabli

Ispis 3.1: Struktura rješenja za problem rasporeda provjera znanja.

```
1  public class Kromosom {
2      KCourse[] kcourse;
3      KTerm[] kterms;
4      int[] clusterTerms;
5      int[] eval;
6  }
7
8  public class KTerm {
9      int someCourse = -1;
10     int coursesCount = 0;
11 }
12
13 public class KCourse {
14     int previous = -1;
15     int next = -1;
16     int term = -1;
17     int index;
18 }
```



Slika 3.15: Struktura podataka za problem izrade rasporeda obaveznih provjera znanja.

prikazane su u sljedećoj tablici.

Pozicija	previous	next	term
1	3	4	1
2	5	5	2
3	4	1	1
4	1	3	1
5	2	2	2
6	7	8	3
7	8	6	3
8	6	7	3

Polje na dnu slike je polje kterms. Elementi tog polja prikazani su u tablici u nastavku.

Pozicija	someCourse	coursesCount
1	3	3
2	5	2
3	8	3

Zanima li nas u koji je termin smješten kolegij s indeksom 2, to jednostavno pogledamo u polju kcourse na poziciji 2 (varijabla term). Zanima li nas koji su sve kolegiji smješteni u termin 3, dovoljno je u polju kterms na poziciji 3 pogledati varijablu someCourse – to je referenca na prvi kolegij; do ostalih se direktno dolazi praćenjem reference next odgovarajućih objekata. Operaciju mutacije jednog kolegija sada možemo provesti u složenosti O(1): kolegij c_i dohvatimo u polju kcourse na poziciji i, izbacimo

ga iz dvostruko povezane liste, promijenimo mu termin na željeni i na kraju ga ubacimo u dvostruko povezanu listu tog termina (koju direktno dohvatimo u polju kterms) na poziciji promatranog termina. Kako nema potrebe za bilo kakvim pretraživanjem ili iteriranjem po kolekcijama, sve ove operacije obavljaju se u konstantnom vremenu. Operaciju križanja i druge također možemo izvesti na sličan način, što znači da će ukupna složenost križanja i mutacije biti linearno proporcionalna broju kolegija.

3.1.5 Prikaz stablima

Prikaz stablima uobičajena je tehnika za prikaz aritmetičkih izraza ili programskih odječaka i koristiti se kada je cilj optimizacijskog procesa pronaći formulu koja najbolje opisuje mapiranje između zadanih podataka ili pak kada je protrebno evoluirati program koji rješava zadani problem. Više o ovoj vrsti prikaza rješenja biti će riječi u poglavlju 7 u okviru *genetskog programiranja*.

3.2 Genotipski i fenotipski prikaz rješenja

Kada govorimo o načinima prikaza rješenja i prirodom inspiriranim optimizacijskim algoritmima, važno je osvrnuti se na još jedan detalj vezan uz prikaz rješenja: razlikujemo genotipski prikaz i fenotipski prikaz rješenja. *Genotipski prikaz* rješenja direktna je analogija genima u molekuli DNA: jedan ili više gena u molekuli DNA odgovoran je za ono što zovemo *izražena svojstva* jedinke (engl. *trait*): primjerice, koliko je jedinka visoka, koju ima boju očiju, kakvi su joj refleksi, koliko je otporna na zračenje i slično. Sva ova svojstva rezultat su međudjelovanja gena, i ta svojstva nazivamo *fenotipom*.

Je li prikaz genotipski ili fenotipski, ovisi o problemu koji rješavamo. U primjeru u kojem smo binarnim prikazom rješavali problem pronalaska optimalnog podskupa, binarni prikaz možemo tumačiti kao fenotipski: svaki bit u tom prikazu ima jasnu interpretaciju – dotični element univerzalnog skupa ili pripada ili ne pripada promatranom podskupu. Međutim, binarni prikaz korišten za reprezentaciju rješenja problema pronalaska maksimuma funkcije od tri varijable na zadanom podskupu je genotipski prikaz: poznavanjem pojedinog bita ne znamo ništa o rješenju – najprije je potrebno provesti postupak dekodiranja koji rješenja iz genotipskog prikaza prevodi u fenotipski prikaz (npr. 36 bitova u 3 decimalna broja), nakon čega možemo obaviti vrednovanje tog rješenja.

Uz ove pojmove važan je i pojam *lokalnosti prikaza rješenja* (engl. *locality of a representation*) koji opisuje koliko dobro genotipsko susjedstvo odgovara fenotipskom susjedstvu. Naime, dosta istraživanja je pokazalo da je za efikasan rad optimizacijskih algoritama koji se temelje na evoluciji rješenja nužno imati visoku lokalnost prikaza [Gottlieb and Raidl, 1999, 2000, Rothlauf and Goldberg, 2000]. Prilikom rada s genotipskim i fenotipskim prikazima nužno je uočiti da radimo u dva nezavisna prostora [Lewontin, 1974]: genotipski prikaz definira informacijski prostor (engl. *informational space*) dok fenotipski prikaz definira ponašajni prostor (engl. *behavioral space*). U svakom prostoru moguće je definirati susjedstvo.

Uzmimo za primjer traženje maksimuma funkcije $f(x)$ gdje se kao genotipski prikaz koristi binarno kodiranje uz interpretaciju u skladu s prirodnim binarnim kodom, područjem pretraživanja od $[0, 6.3]$ i neka radimo s 6-bitnim prikazom; tada će kvant pretrage biti 1. Fenotipski, imamo prostor rješenja koji se sastoji od decimalnih brojeva oblika $0.1 \cdot i$ gdje je $i \in 0, \ldots, 63$.

Susjedstvo rješenja s u genotipskom prikazu možemo definirati kao sva rješenja s^* koja je nekom malom promjenom moguće dobiti iz rješenja s. Npr. ako je $s = 001101$ i ako je "mala promjena" promjena jednog bita, tada je genotipsko susjedstvo od zadanog s skup rješenja {101101, 011101, 000101, 001001, 001111, 001100}, što je skup svih 6-bitovnih binarnih uzoraka čija je Hammingova udaljenost do s jednaka 1. Uočimo da je ovo susjedstvo definirano u informacijskom prostoru.

Susjedstvo rješenja u fenotipskom prikazu definiramo na isti način: skup svih rješenja koja je moguće dobiti nekom malom promjenom. Međutim, ovdje smo u drugom prostoru. Radimo s decimalnim brojevima, pa ako kao susjedstvo od $s = 1.3$ (taj broj odgovara genotipu 001101) definiramo sve decimalne brojeve koji su na udaljenosti od kvanta do tog rješenja, susjedstvo od 1.3 je $\{1.2, 1.4\}$.

S obzirom da, kada se koristi genotipski prikaz, algoritmi pretraživanja direktno rade manipulacije nad genotipskim prikazom, poželjno je da si genotipska i fenotipska susjedstva međusobno odgovaraju. Evo zašto je to važno. Optimizacijski će algoritam nad genotipskim prikazom kontrolirati koliko jako želi "razdesiti" trenutno rješenje: ako su rješenja kvalitetna, radit će se male promjene a ako su rješenja

nekvalitetna, radit će se veće promjene. Međutim, ako prikaz nema visoku lokalnost, tada je moguće da mala promjena u genotipskom prikazu izazove drastičnu promjenu u fenotipskom prikazu, čime umjesto fine pretrage algoritam radi nasumične skokove po prostoru rješenja. Klasični prirodni binarni prikaz nema visoku lokalnost: promjenom bita na "krivom kraju" (dakle onog najveće težine) fenotipska vrijednost će se drastično mijenjati.

Za više detalja o utjecaju lokalnosti na težinu problema koji se rješava i na performanse algoritama čitatelji se upućuju na rad [Rothlauf, 2003].

Bibliografija

W. Banzhaf. The "molecular" traveling salesman. *Biological Cybernetics*, 64:7–14, 1990.

L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 162–164, 1985.

L. J. Eshelman and J. D. Schaffer. Real–coded genetic algorithms and interval-schemata. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 187–202, San Mateo, 1993. Morgan Kaufmann.

D. B. Fogel. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics*, 60: 139–144, 1988.

D. B. Fogel. A parallel processing approach to a multiple traveling salesman problem using evolutionary programming. In L. H. Canter, editor, *Proceedings on the Fourth Annual Parallel Processing Symposium*, pages 318–326, Fullterton, CA, USA, 1990. California State University, Fullterton.

D. B. Fogel. Applying evolutionary programming to selected traveling salesman problems. *Cybernetics and Systems*, 24:27–36, 1993.

D. E. Goldberg and J. R. Lingle. Alleles, loci and the tsp. In J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 154–159, Hillsdale, New Jersey, USA, 1985. Lawrence Erlbaum Associates.

J. Gottlieb and G. R. Raidl. Characterizing locality in decoder-based eas for the multidimensional knapsack problem. In C. Fonlupt, J.-K. Hao, E. Lutton, E. M. A. Ronald, and M. Schoenauer, editors, *Proceedings of Artificial Evolution*, volume 1829 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 1999. ISBN 3-540-67846-8.

J. Gottlieb and G. R. Raidl. The effects of locality on the dynamics of decoder-based evolutionary search. In L. D. Whitley, D. E. Goldberg, E. Cantú-Paz, L. Spector, I. C. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference 2000*, pages 283–290, San Francisco, CA, 2000. Morgan Kaufmann. ISBN 1-55860-708-0.

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

R. Lewontin. *The Genetic Basis of Evolutionary Change.* Columbia University Press, New York, USA, 1974.

Z. Michalewicz. *Genetic algorithms + data structures = evolution programs.* Artificial Intelligence. Springer-Verlag, New York, 1992.

H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm, I: Continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993. URL `http://dx.doi.org/10.1162/evco.1993.1.1.25`.

I. M. Oliver, D. Smith, and J. R. C. Holland. A study of permutation crossover operators on the tsp. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 224–230, Hillsdale, New Jersey, USA, 1987. Lawrence Erlbaum Associates.

N. J. Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183–205, 1991.

F. Rothlauf. On the locality of representations. In E. Cantú-Paz, J. A. Foster, K. Deb, L. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. K. Standish, G. Kendall, S. W. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, and J. F. Miller, editors, *GECCO*, volume 2724 of *Lecture Notes in Computer Science*, pages 1608–1609. Springer, 2003. ISBN 3-540-40603-4.

F. Rothlauf and D. E. Goldberg. Pruefer numbers and genetic algorithms: A lesson on how the low locality of an encoding can harm the performance of gas. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. M. Guervós, and H.-P. Schwefel, editors, *PPSN*, volume 1917 of *Lecture Notes in Computer Science*, pages 395–404. Springer, 2000. ISBN 3-540-41056-2.

G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms.*, pages 332–349. Van Nostrand Reinhold, New York, 1991.

E.-G. Talbi and B. Weinberg. Breaking the search space symmetry in partitioning problems: An application to the graph coloring problem. *Theoretical Computer Science (TCS)*, 378(1):78–86, 2007.

A. H. Wright. Genetic algorithms for real parameter optimizations. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 205–218, San Mateo, 1991. Morgan Kaufmann.

M. Čupić, M. Golub, and D. Jakobović. Exam timetabling using genetic algorithm. In V. Luzar-Stiffler, I. Jarec, and Z. Bekić, editors, *Proceedings of the 31st International Conference on Information Technology Interfaces*, pages 357–362, 2009.

Poglavlje 4

Selekcije

4.1 Uvod

Jedan od centralnih mehanizama, kako kod većine populacijskih algoritama pa i općenito, jest mehanizam selekcije. Kod populacijskih algoritama zadaća operatora selekcije jest osigurati mehanizam koji će češće boljim rješenjima dati priliku da sudjeluju u produkciji novih rješenja, čime će se proces pretraživanja prostora rješenja voditi u područja koja više obećavaju.

Operatori selekcije i njihov utjecaj može se okarakterizirati kroz nekoliko parametara:

- vrijeme preuzimanja,
- selekcijski intenzitet,
- momenti razdiobe dobrote i drugi.

Vrijeme preuzimanja (engl. *takeover time*) je vrijeme koje je potrebno operatoru selekcije da generira populaciju u kojoj se nalazi samo najbolje rješenje [Goldberg and Deb, 1991]. Ideja je jednostavna: pretpostavimo da imamo populaciju rješenja i da novu generaciju generiramo samo uporabom operatora selekcije nad postojećom generacijom. Kad je nova populacija spremna, postupak se ponavlja kako bi stvoriti sljedeću generaciju, i tako dalje. Ako je operator selekcije takav da boljim rješenjima daje veću šansu da budu odabrani, za očekivati je da će u svakoj sljedećoj generaciji biti sve više i više kopija najboljeg rješenja. Vrijeme (odnosno broj generacija) od prve generacije pa do nastanka generacije koja je popunjena isključivo kopijama najboljeg rješenja naziva se vrijeme preuzimanja.

Utjecaj operatora selekcije može se mjeriti i kroz selekcijski intenzitet. *Selekcijski intenzitet I* [Bäck, 1995, Miller and Goldberg, 1995, Mühlenbein and Schlierkamp-Voosen, 1993, Thierens, 1997] računa se kao povećanje srednje dobrote populacije prije i nakon primjene operatora selekcije, podijeljenog sa standardnom devijacijom populacije:

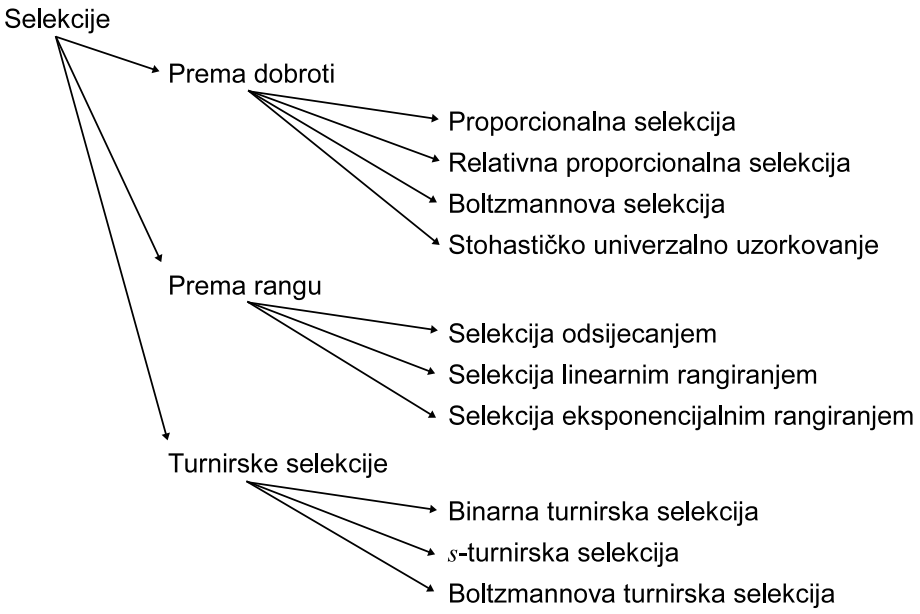
$$I = \frac{\bar{f}_s - \bar{f}}{\sigma}.$$

U ovom izrazu $\bar{f}$ predstavlja prosječnu dobrotu rješenja u populaciji roditelja, $\bar{f}_s$ predstavlja prosječnu dobrotu rješenja u populaciji djece koja je dobivena samo uporabom operatora selekcije, a σ predstavlja standardno odstupanje u populaciji roditelja. Moguće je pokazati da je kod, primjerice, genetskog algoritma, pod određenim uvjetima brzina konvergencije obrnuto proporcionalna selekcijskom intenzitetu [Mühlenbein and Schlierkamp-Voosen, 1993]. Selekcijski intenzit moguće je povezati i s optimalnim iznosom mutacije [Bäck, 1996] kao i veličinom populacije [Mühlenbein and Schlierkamp-Voosen, 1993, Harik et al., 1999].

Uporaba momenata razdiobe dobrote proučavana je u okviru [Prügel-Bennett and Shapiro, 1994]. r-ti centralni moment razdiobe dobrote populacije rješenja veličine n računa se prema izrazu:

$$\mu_r = \frac{1}{n}\sum_{i=1}^{n}(f_i - \bar{f})^r$$

gdje je f_i dobrota i-tog rješenja a $\bar{f} = \frac{1}{n}\sum_{i=1}^{n} f_i$ srednja vrijednost dobrota za čitavu populaciju. Prvi centralni moment jednak je srednjoj vrijednosti dobrota: $\mu_1 = \bar{f}$. Drugi centralni moment jednak je



Slika 4.1: Podjela operatora selekcije.

varijanci dobrote odnosno kvadratu standardnog odstupanja: ${}_m u_2 = \sigma_f^2$. Treći centralni moment mjeri asimetričnost razdiobe. Uporabom opisanih momenata može se proučavati kakve razdiobe dobrote generiraju pojedini operatori selekcije.

Upoznajmo se sada s često korištenim vrstama selekcije. Jedna moguća (i svakako nepotpuna) podjela operatora selekcija prikazana je na slici 4.1.

4.2 Proporcionalna selekcija

Proporcionalna selekcija [Holland, 1975] svakom rješenju u populaciji pridružuje vjerojatnost odabira koja je proporcionalna dobroti rješenja. Selekcija još poznata pod nazivom engl. *Roulette-wheel selection*. Vjerojatnost odabira i-te jedinke definirana je izrazom:

$$p_i = \frac{f_i}{\sum_{i=1}^{n} f_j} = \frac{f_i}{n \cdot \bar{f}} \tag{4.1}$$

pri čemu je f_i iznos dobrote i-tog rješenja. Valja napomenuti da za ispravno funkcioniranje ove selekcije je nužno da sve dobrote budu nenegativne: $\forall i \in \{1, \ldots, n\}, f_i \geq 0$ te srednja vrijednost dobrote mora biti veća od nule: $\bar{f} > 0$. Za ovu selekciju može se pokazati da je selekcijski intenzitet jednak:

$$I = \frac{\sigma}{\bar{f}}.$$

Iz izraza je vidljivo da se selekcijski intenzitet smanjuje kako se smanjuje standardno odstupanje, a također pada s porastom srednje vrijednosti dobrote populacije.

Proporcionalna selekcija podložna je problemu skale, što je direktna posljedica prethodnog izraza za selekcijski intenzitet. Problem možemo ilustrirati na sljedećem primjeru. Zamislimo dvije populacije rješenja od po tri jedinke. U jednoj populaciji dobrote rješenja su redom 1, 2 i 3. U drugoj populaciji dobrote rješenja su redom 1000001, 1000002 i 1000003. U obje populacije treće rješenje je najbolje: ima najveću dobrotu. Međutim, dok je u prvoj populaciji vjerojatnost odabira najbolje jedinke jednaka:

$$p_3 = \frac{3}{1+2+3} = \frac{3}{6} = 0.5$$

odnosno čak 50%, u drugoj populaciji ona iznosi:

$$p_3 = \frac{1000003}{1000001 + 1000002 + 1000003} = \frac{1000003}{3000006} = 0.33333366666600000133333066667 2 \approx \frac{1}{3}$$

čime je vjerojatnost izbora najgore i najbolje jedinke praktički ista. Kako ovaj postupak dobro funkcionira za male brojeve a sve lošije kako raste skala (magnituda broja), problem je poznat pod nazivom *problem skale*.

4.3 Relativna proporcionalna selekcija

Problem skale može se ublažiti a ograničenje da funkcija dobrote mora biti nenegativna zaobići uvođenjem relativne dobrote. Umjesto da se vjerojatnosti odabira rješenja računaju temeljem dobrote f_i, za svako se rješenje uvodi relativna dobrota $g_i = f_i - f_{min}$, gdje je s f_{min} označena dobrota najlošijeg rješenja u populaciji. U tom slučaju vjerojatnosti se računaju prema izrazu:

$$p_i = \frac{g_i}{\sum_{i=1}^{n} g_j} = \frac{f_i - f_{min}}{\sum_{i=1}^{n}(f_j - f_{min})} = \frac{f_i - f_{min}}{n \cdot (\bar{f} - f_{min})}. \tag{4.2}$$

4.4 Boltzmannova selekcija

Boltzmannova selekcija (engl. *Boltzmann selection*) [de la Maza and Tidor, 1993] je selekcija kod koje je vjerojatnost odabira rješenja eksponencijalno proporcionalna dobroti rješenja. Vjerojatnost odabira rješenja i definirana je izrazom:

$$p_i = \frac{e^{\beta \cdot f_i}}{\sum_{j=1}^{n} e^{\beta \cdot f_j}}. \tag{4.3}$$

Da bi selekcija radila u skladu s očekivanjima, vrijednosti dobrota trebaju biti nenegativne. U izrazu (4.3) β je parametar koji utječe na razdiobu vjerojatnosti. Ova selekcija osjetljiva je na problem skale.

4.5 Stohastičko univerzalno uzorkovanje

Proporcionalna selekcija, relativna proporcionalna selekcija te Boltzmanova selekcija definiraju način dodjele vjerojatnosti rješenjima u populaciji. Ako trebamo k jedinki, postupak izvlačenja ćemo ponavljati k puta što računski može biti vrlo skupo. Ideja stohastičkog univerzalnog uzorkovanja (engl. *Stochastic universal sampling*) [Baker, 1985b] jest odjednom iz populacije izvuči svih potrebnih k jedinki; implementacijski, to bi značilo koristiti samo jedan poziv generatora slučajnih brojeva za izvlačenje svih k jedinki.

Ova selekcija izvodi se na sljedeći način. Neka je n broj rješenja u populaciji, f_i dobrota i-tog rješenja (dobrote moraju biti nenegativne) i neka je $F = \sum_{i=1}^{n} f_i > 0$ ukupna dobrota. Generira se slučajni broj r iz raspona $[0, \frac{F}{k}]$ gdje je k broj rješenja koje želimo izvući.

Zamislimo sada da na brojevni pravac slažemo rješenja na način da krenemo od nule, i prvom rješenju damo segment brojevnog pravca od nule pa do f_1, nastavimo sa sljedećim rješenjem koje zauzima segment pravca od f_1 do $f_1 + f_2$, nastavimo sa sljedećim itd. Ukupna duljina svih zauzetih segmenata je upravo F. Promotrimo sada na tom istom brojevnom pravcu točke:

$$\{r_0, r_1, \ldots, r_{k-1}\} = \{r, r + 1 \cdot \frac{F}{k}, r + 2 \cdot \frac{F}{k}, \ldots, r + (k-1) \cdot \frac{F}{k}\}.$$

Svaka točka r_i upada u segment nekog od rješenja – to rješenje je selektirano i postaje jedno od rješenja koje vraćamo.

Evo konkretnog primjera. Neka su dobrote rješenja redom 1, 4, 3, 5 i 8. Vrijedi $F = 1+4+3+5+8 = 21$. Pretpostavimo da trebamo odabrati $k = 3$ rješenja. Računamo: $\frac{F}{k} = \frac{21}{3} = 7$. Stoga biramo slučajni broj iz intervala $[0, 7]$. Neka je to $r = 2.3$.

Složimo rješenja na brojevni pravac. Interval na kojem se proteže rješenje 1 je $[0, 1]$, slijedi ga rješenje 2 kojem pripada interval $[1, 5]$, pa rješenje 3 kojem pripada interval $[5, 8]$, rješenje 4 kojem pripada interval $[8, 13]$ te rješenje 5 kojem pripada interval $[13, 21]$.

Temeljem odabranog broja r imamo skup $\{r_0, r_1, r_2\} = \{2.3, 2.3 + 7, 2.3 + 2 \cdot 7\} = \{2.3, 9.3, 16.3\}$. Rješenja u čije segmente upadaju ovi brojevi su drugo, četvrto i peto.

4.6 Selekcija odsijecanjem

Selekcija odsijecanjem (engl. *truncation selection*) [Mühlenbein and Schlierkamp-Voosen, 1993] je deterministička selekcija kod koje se vjerojatnost odabira za najboljih τ rješenja postavlja ja $\frac{1}{\tau}$, a za sva ostala rješenja na 0. Time će samo τ najboljih rješenja moći sudjelovati u postupcima stvaranja novih rješenja. Konceptualno, ovo se može izvesti tako da se populacija rješenja sortira po padajućim vrijednostima dobrote i potom se za prvih τ rješenja vjerojatnost postavi na $\frac{1}{\tau}$ a za preostale na 0. Ako s i označimo rang (poziciju) rješenja u populaciji koja je sortirana po padajućim vrijednostima funkcije dobrote, vjerojatnost odabira jedinke je:

$$p_i = \begin{cases} \frac{1}{\tau} & \text{ako je } i \in \{1, \ldots, \tau\} \\ 0 & \text{ako je } i \in \{\tau + 1, \ldots, n\} \end{cases} . \tag{4.4}$$

Jasno je da selekcijski intenzitet raste sa smanjenjem parametra τ. Primjerice, postavimo li $\tau = 1$, samo će najbolje rješenje biti izabirano, i već sljedeća generacija će sadržavati samo njegove kopije. Povećavanjem vrijednosti parametra τ selekcijski se intenzitet smanjuje čime se duže čuva raznolikost u populaciji.

4.7 Selekcija linearnim rangiranjem

Kod selekcije linearnim rangiranjem (engl. *linear ranking selection*) [Baker, 1985a] rješenja u populaciji se sortiraju prema rastućim vrijednostima dobrote rješenja i potom im se pridjeljuje vjerojatnost odabira koja je linearno proporcionalna rangu (poziciji) rješenja. Rang 1 pripada najgorem rješenju a rang n najboljem rješenju. Da bi se do kraja definirala dodjela vjerojatnosti potrebno je definirati dva parametra:

- n^+ – koliko se kopija najboljeg rješenja očekuje u novoj populaciji te
- n^- – koliko se kopija najgoreg rješenja očekuje u novoj populaciji.

Tada se vjerojatnost odabira jedinke ranga i računa prema sljedećem izrazu:

$$p_i = \frac{n^- + \frac{n^+ - n^-}{n-1}(i-1)}{n}. \tag{4.5}$$

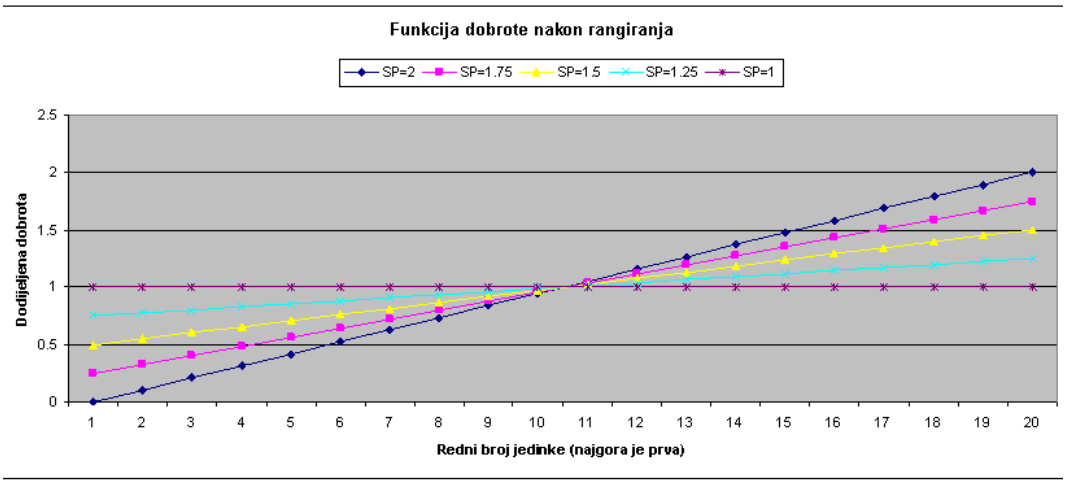
Pri tome suma $n^+ + n^-$ mora biti jednaka 2 kako bi se dobile vjerojatnosti (tj. kako bi vrijedilo $\sum_{i=1}^{n} p_i = 1$. Kako su parametri $n^+ + n^-$ vezani, i kako ne želimo situaciju u kojoj je $n^+ < n^-$, supstitucijom $n^- = 2 - n^+$ te preimenovanjem n^+ u SP prethodni izraz možemo zapisati i ovako:

$$p_i = \frac{2 - SP + 2(SP-1)\frac{i-1}{n-1}}{n}. \tag{4.6}$$

pri čemu su za parametar SP dozvoljene vrijednosti iz intervala $[1, 2] \subset \mathcal{R}$. Parametar SP direktno opisuje selekcijski pritisak. Evo pojašnjenja. Slika 4.2 prikazuje vrijednosti $n \cdot p_i$ za primjer populacije veličine $n = 20$ i različite vrijednosti parametra SP.

Uz $SP = 1$, svim se rješenjima, neovisno o njihovoj stvarnoj dobroti pridjeljuje ista vjerojatnost odabira. Ovo će rezultirati time da sva rješenja – i dobra i loša – u prosjeku budu odabrana jednak broj puta, i to neće rezultirati usmjerenim pretraživanjem prostora mogućih rješenja. Potpuno se suprotna situacija pojavljuje uz $SP = 2$: najgorem rješenju tada je pridjeljena vjerojatnost 0, čime se to rješenje nikada neće birati kao potencijalni roditelj, dok najbolje rješenje ima najveću vjerojatnost odabira. Za vrijednosti parametra SP između 1 i 2 omjer vjerojatnosti dodijeljene najboljem i najgorem rješenju $\frac{p_n}{p_1}$ mijenja se od 1 (potpuno slučajna pretraga) do ∞ (jako usmjerena pretraga). Stoga se ovim parametrom može fino podešavati koliki naglasak algoritam stavlja na najbolje pronađeno rješenje.

Selekcija linearnim rangiranjem, generalno govoreći, ima relativno mali selekcijski intenzitet.



Slika 4.2: Ovisnost dobrote jedinke o parametru SP kod rangiranja. Primjer ilustrira slučaj $n = 20$ te nekoliko različitih vrijednosti parametra SP.

4.8 Selekcija eksponencijalnim rangiranjem

Kod selekcije eksponencijalnim rangiranjem (engl. *exponential ranking selection*) [Baker, 1985a] rješenja u populaciji se sortiraju prema rastućim vrijednostima dobrote rješenja i potom im se pridjeljuje vjerojatnost odabira koja je eksponencijalno proporcionalna rangu (poziciji) rješenja. Rang 1 pripada najgorem rješenju a rang n najboljem. Vjerojatnost odabira jedinke ranga i računa prema sljedećem izrazu:

$$p_i = \frac{c^{n-i}}{\sum_{j=1}^{n} c^{n-j}} = \frac{c-1}{c^n - 1} c^{n-i} \tag{4.7}$$

gdje je $c \in [0, 1]$ parametar koji utječe na način dodjele vjerojatnosti. Za vrijednosti parametra c koje su bliske 1, dobiveni selekcijski intenzitet bit će vrlo mali. U krajnjem slučaju gdje je $c = 1$, sva će rješenja imati jednaku vjerojatnost odabira $(\frac{1}{n})$ čime će selekcijski intenzitet biti minimalan ali će se čuvati raznolikost populacije. Smanjenjem vrijednosti c selekcijski intenzitet vrlo brzo raste a raznolikost populacije pada čime ovaj operator postaje puno jači od linearnog rangiranja. Međutim, kako smanjenjem vrijednosti parametra c raznolikost vrlo rapidno pada, ovu selekciju nije moguće koristiti ako se u algoritam ne ugrade drugi operatori čija bi namjena bila povećavanje raznolikosti kako bi se dobila ravnoteža. U praksi se ovaj operator koristi s vrijednostima c koje nisu baš predaleko od 1.

4.9 Turnirska selekcija

Turnirska selekcija (engl. *Tournament selection*) [Goldberg and Deb, 1991, Blickle and Thiele, 1995b,a] je selekcija kod koje se iz populacije izvlači slučajni uzorak od s rješenja i rješenje s najvećom dobrotom u izvučenom uzorku se odabire. Selekcijski intenzitet kod turnirske selekcije približno je jednak $\sqrt{\sqrt{2} \ln s}$ [Bäck, 1995].

Turnirska selekcija koja iz populacije od n rješenja izvlači slučajni uzorak od s rješenja naziva se s-turnirska selekcija. U praksi postoje dvije varijante s-turnirskih selekcija:

- varijanta kod koje se s puta ponavlja biranje rješenja iz čitave populacije – kod ove izvedbe je moguće da isto rješenje bude više puta odabrano (odnosno da uzorak od s rješenja sadrži duplikate) te
- varijanta kod koje se biranje ponavlja s puta ali na način da se nakon svakog odabira dalje bira iz podskupa populacije iz kojeg su uklonjene do tada odabrana rješenja (pa uzorak od s rješenja neće imati duplikata ako originalna populacija nije imala duplikata).

4.9.1 Turnirska selekcija s ponavljanjima

U nastavku ćemo najprije dati izraz za vjerojatnost odabira jedinke uz izvedbu opisanu u prvoj varijanti. Zamislimo da imamo populaciju od n rješenja i da smo je sortirali po rastućem poretku (rješenje čija je dobrota najlošija je na prvom mjestu odnosno ima rang 1, a rješenje čija je dobrota najbolja je na posljednjem mjestu, odnosno ima rang n). Pretpostavimo sada da nasumično izvlačimo jedno rješenje prema uniformnoj distribuciji. Vjerojatnost da smo izvukli najgore rješenje, najbolje rješenje ili bilo koje između je isto: $\frac{1}{n}$. Vjerojatnost da smo izvučemo rješenje koje je na mjestu 1 ili na mjestu 2 je $\frac{1}{n} + \frac{1}{n} = \frac{2}{n}$. Poopćavanjem možemo zaključiti da je vjerojatnost da smo izvukli rješenje koje je na bilo kojem od mjesta $1, 2, \ldots, i$ jednaka sumi $\frac{1}{n}$ (za mjesto 1) i $\frac{1}{n}$ (za mjesto 2) pa sve do $\frac{1}{n}$ (za mjesto i). Ovih članova ukupno ima i što znači da je vjerojatnost da smo izvukli rješenje koje je na bilo kojoj poziciji počev od 1. pa do i-te jednaka:

$$i \cdot \frac{1}{n} = \frac{i}{n}.$$

Ako izvlačenje ponavljamo s puta, vjerojatnost da je u svih s izvlačenja odabrano rješenje koje je na prvih i pozicija jednaka je umnošku vjerojatnosti da je u svakom izvlačenju izvučeno takvo rješenje:

$$\left(\frac{i}{n}\right)^s.$$

Sada imamo sve što je potrebno da odgovorimo na pitanje: kolika je vjerojatnost da u turniru od s rješenja pobjedi rješenje ranga i? Rješenje na poziciji i će pobjediti:

1. ako je svaki puta u s izvlačenja izvučeno neko od prvih i rješenja te
2. ako je rješenje i doista izvučeno, što znači da se nije dogodio slučaj da je u svih s izvlačenja izvučeno neko od prvih $i-1$ rješenja.

Prvi uvjet je važan jer osigurava da nije izvučeno niti jedno rješenje koje je na poziciji većoj od i – prisjetimo se, populacija je sortirana po rastućim dobrotama, što znači da se na kasnijoj poziciji nalazi bolje rješenje. Stoga, da bi rješenje i bilo pobjednik turnira, u svih s izvlačenja smiju biti izabrana samo rješenja na poziciji i ili manjoj. Drugi uvjet osigurava da je rješenje i barem jednom izvučeno – inače ne može biti pobjednik.

Tražena vjerojatnost stoga je jednaka vjerojatnosti da smo u s izvlačenja izvukli rješenje koje je na poziciji $1 \ldots i$ što je $\left(\frac{i}{n}\right)^s$ minus vjerojatnost da smo u svih s izvlačenja izvukli rješenje koje je na poziciji $1 \ldots i-1$ koja iznosi $\left(\frac{i-1}{n}\right)^s$. Slijedi da je tražena vjerojatnost jednaka:

$$p_i = \left(\frac{i}{n}\right)^s - \left(\frac{i-1}{n}\right)^s. \tag{4.8}$$

U literaturi se može pronaći i drugačiji izraz koji vrijedi ako se rangovi jedinkama dodjeljuju obrnuto: ako najbolja jednika ima rang 1 a najlošija jedinka rang n. Analiza opet kreće od najlošije jedinke samo što je ona sada s desne strane: vjerojatnost da nasumičnim izvlačenjem odaberemo rješenje koje je ranga i ili lošije (dakle na pozicijama $i, i+1, \ldots, n-1, n$) je jednako $\frac{n-i+1}{n}$ jer tih pozicija ima $n-i+1$. Vjerojatnost da u s izvlačenja svaki puta odaberemo rješenje koje je ranga i ili gore jednaka je

$$\left(\frac{n-i+1}{n}\right)^s.$$

Tada je vjerojatnost da je rješenje ranga i pobjednik s-turnira jednaka vjerojatnosti da je u s izvlačenja svaki puta odabrano rješenje ranga i ili gore minus vjerojatnost da je u svih s izvlačenja odabrano rješenje koje je većeg ranga od i (odnosno uvijek gore, na pozicijama $i+1, \ldots, n$), pa možemo pisati:

$$p_i = \left(\frac{n-i+1}{n}\right)^s - \left(\frac{n-i}{n}\right)^s.$$

4.9.2 Turnirska selekcija bez ponavljanja

Vjerojatnost za drugu verziju turnirske selekcije dade se izvesti uz malo kombinatorike. Pretpostavimo opet da je populacija sortirana po rastućim dobrotama: rang 1 pripada najlošijem rješenju a rang n najboljem. Radimo turnir sa s rješenja, ali bez duplikata. To znači da iz skupa od n rješenja nasumice odaberemo jedno rješenje, pa iz preostalog skupa od $n-1$ rješenja sljedeće, i tako redom dok ne izaberemo svih s rješenja. Broj načina na koje možemo odabrati s rješenja je dakle:

$$\frac{n \cdot (n-1) \cdot \cdots \cdot (n-s+1)}{s!} = \frac{n!}{s!(n-s)!} = \binom{n}{s}$$

(dijelimo sa $s!$ jer redosljed izabranih elemenata nije bitan). Rješenje i može biti pobjednik samo ako su u svih s izvlačenja izvučena rješenja koja su ranga i ili manjeg. Broj načina na koje iz skupa $\{1, 2, \ldots, i\}$ možemo izvući s rješenja bez duplikata jednak je $\binom{i}{s}$. To znači da je vjerojatnost da smo u s izvlačenja iz skupa od n rješenja izvukli rješenja čiji su rangovi i ili manji jednaka omjeru $\binom{i}{s}/\binom{N}{s}$. Da bi rješenje i bilo pobjednik turnira, ne smije se dogoditi da je u s izvlačenja svaki puta izabrano rješenje ranga manjeg od i čija je vjerojatnost $\binom{i-1}{s}/\binom{N}{s}$.

Konačna vjerojatnost da u turniru u kojem se izvlači s različitih rješenja pobijedi rješenje ranga i tada je jednaka:

$$p_i = \frac{\binom{i}{s}}{\binom{n}{s}} - \frac{\binom{i-1}{s}}{\binom{n}{s}} = \frac{\binom{i}{s} - \binom{i-1}{s}}{\binom{n}{s}} \tag{4.9}$$

Do istog se izraza može doći direktno primjenom hipergeometrijske razdiobe koja opisuje vjerojatnost da se izvlačenjem n kuglica iz skupa koji sadrži m bijelih kuglica i $N-m$ crnih kuglica izvuče k bijelih kuglica, i koja je definirana izrazom:

$$P(X = k) = \frac{\binom{m}{k}\binom{N-m}{n-k}}{\binom{N}{n}}.$$

Konkretno, zanima nas vjerojatnost da od n rangova (ukupni broj kuglica) u s izvlačenja izvučemo s rangova koji su iz skupa $\{1, 2, \ldots, i\}$ (što predstavlja bijele kuglice, {i+1,. . .,n} su crne). Ta je vjerojatnost prema prethodnom izrazu uz supstitucije $m \to i$, $k \to s$, $n \to s$ te $N \to n$ jednaka:

$$\frac{\binom{i}{s}\binom{n-i}{s-s}}{\binom{n}{s}} = \frac{\binom{i}{s}}{\binom{n}{s}}$$

jer je $\binom{n-i}{s-s} = \binom{n-i}{0} = 1$ čime smo došli do prethodno izvedene vjerojatnosti. Vjerojatnost da u svih s pokušaja izvučemo rangove iz skupa $\{1, 2, \ldots, i-1\}$ primjenom istog izraza daje: $\frac{\binom{i-1}{s}}{\binom{n}{s}}$ pa je konačna vjerojatnost razlika tih vjerojatnosti.

U oba slučaja (varijante 1 i 2) selekcijski intenzitet raste s porastom veličine turnira s. To je i za očekivati, jer što je uzorak veći, to je veća šansa da se u njemu nađu bolja rješenja koja će se potom prenijeti dalje. Selekcijski intenzitet je najmanji za $s = 2$ gdje takva selekcija odgovara linearno rangirajućoj selekciji uz $n_- = 0$ (odnosno $SP = 2$).

4.10 (μ, λ)-selekcija i $(\mu + \lambda)$-selekcija

Ove dvije selekcije uobičajeno se sreću kod evolucijskih strategija; međutim, znaju se pojaviti i kod drugih algoritama. Osnovna zadaća ovih selekcija jest smanjivanje veličine populacije, i obje selekcije ujedno definiraju i djelomičnu implementaciju optimizacijskog algoritma.

Kod (μ, λ)-selekcije pretpostavlja se da će iz populacije veličine μ nastati privremena međupopulacija potomaka veličine $\lambda > \mu$. Tu preveliku populaciju selekcija smanjuje tako što za odabire upravo μ najboljih jedinki.

Kod $(\mu + \lambda)$-selekcije također se pretpostavlja se da će iz populacije veličine μ nastati privremena međupopulacija potomaka veličine $\lambda > \mu$. Međutim, kod ove selekcije se potom odabire μ najboljih

jedinki iz unije populacije djece i populacije starih roditelja. Time se osigurava da najbolje rješenje koje je algoritam pronašao ne može biti izgubljeno jer čak ako je populacija djece sastavljena od lošijih jedinki, unija roditelja i djece će sadržavati najbolju jedinku koja će sigurno biti odabrana i tako sačuvana.

4.11 Druge selekcije

Uz navedene, razvijen je još čitav niz drugih, rijeđe korištenih selekcija, poput Boltmannove turnirske selekcije, uniformnog rangiranja koje je vjerojatnosna izvedba (μ, λ)-selekcije, Whitleyevo linearno rangiranje itd. Njih međutim nećemo ovdje posebno obrađivati.

4.12 Zaključne misli

Selekcijski pritisak u populacijskom algoritmu direktna je posljedica primijenjenog operatora selekcija. Kao što je vidljivo iz opisanih operatora selekcije, svaki se operator selekcije ponaša na različit način i generira drugačiji selekcijski pritisak. Svaki od operatora također nudi parametrizaciju koja omogućava podešavanje selekcijskog pritiska.

Ponekad se u populacijskim algoritmima koriste operatori selekcije s jakim selekcijskim pritiskom – tipično su upareni su operatorima poput operatora mutacije koji agresivno diverzificira rješenja. Uz manje agresivne operatore mutacije treba koristiti i operatore selekcije koji generiraju manji selekcijski pritisak. Ono što je važno zapamtiti jest da je u dobrom optimizacijskom algoritmu presudno postići dobru ravnotežu između operatora koji smanjuju raznolikost populacije (poput operatora selekcije) i operatora koji povećavaju raznolikost.

Uz preslabi selekcijski pritisak rješenja koja i posjeduju dobre dijelove i koja bi mogla usmjeriti pretragu prema globalno optimalnom rješenju neće doći do izražaja – zbog premalog pritiska vjerojatnost odabira takvih rješenja će biti nedovoljno velika i takva će se rješenja često izgubiti (a s njima i dijelovi potrebni za izgradnju optimalnog rješenja); pretraga će odgovarati skoro pa nasumičnom pretraživanju. Kod genetskog algoritma ova je pojava poznata pod naziv *genetski drift*.

Uz prejaki selekcijski pritisak rješenje koje posjeduje dobre dijelove bit će prenaglašeno – u vrlo kratkom vremenu to će rješenje postati dominantno u populaciji čime će se algoritmu onemogućiti da istraži druga područja i moguće kvalitetnija rješenja; ovo će dovesti do pojave koju nazivamo *prerana konvergencija* (engl. *premature convergence*). Jedna od karakteristika ove pojave je i gubitak raznolikosti u populaciji – čitava populacija komprimirana je u okolicu jednog rješenja.

Uz selekcije važno je spomenuti još jedan blizak pojam – elitizam. *Elitizam* kod optimizacijskog algoritma označava svojstvo algoritma koje garantira da najbolje pronađeno rješenje ne može biti izgubljeno, odnosno da može nestati iz populacije rješenja samo ako ga zamijeniti neko još bolje rješenje. Algoritmi koji žele biti elitistički, ovisno o korištenim operatorima selekcije ponekad se mogu osloniti na operatore selekcije za garanciju ovog svojstva a ponekad to trebaju osigurati sami (kad operator selekcije to ne garantira).

Kakve su implikacije uprabe elitizma u algoritmima koji rade nad konačnim populacijama i s ograničenim brojem koraka, još uvijek nije jasno. S praktične strane, imati elitizam je zgodno jer je time napredak algoritma monoton – čuva najbolje rješenje tako dugo dok ga ne uspije zamijeniti još boljim. Međutim, s teorijske strane, trenutno najbolje rješenje u populaciji jest rješenje koje se ponaša kao atraktor čitave populacije koja se komprimira oko njega – ako je to lokalni optimum, to može uzrokovati zaglavljivanje algoritma. U takvim bi situacijama potencijalni gubitak takvog rješenja mogao potaknuti populaciju na istraživanje drugih dijelova potprostora rješenja i time doprinijeti pronalasku boljih rješenja.

Bibliografija

T. Bäck. Generalized convergence models for tournament- and (mu, lambda)-selection. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 2–8. Morgan Kaufmann, 1995. ISBN 1-55860-370-0.

T. Bäck. *Evolutionary algorithms in theory and practice - evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, 1996. ISBN 978-0-19-509971-3.

J. Baker. Adaptive selection methods for genetic algorithms. In J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Application*, pages 101–111, Hillsdale, NJ, USA, 1985a. Lawrence Erlbaum Associates.

J. Baker. Reducing bias and inefficiency in the selection algorithm. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, Hillsdale, New Jersey, USA, 1985b. Lawrence Erlbaum Associates.

T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 9–16. Morgan Kaufmann, 1995a. ISBN 1-55860-370-0.

T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. TIK-Report 11, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, December 1995b.

M. de la Maza and B. Tidor. An analysis of selection procedures with particular attention paid to proportional and boltzmann selection. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 124–131, San Mateo, CA, USA, 1993. Morgan Kaufmann.

D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms (FOGA 1)*, volume 1, pages 69–93, San Francisco, CA, USA, 1991. Morgan Kaufmann.

G. Harik, E. Cantu-Paz, D. Goldberg, and B. Miller. The gamblerś ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999.

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

B. Miller and D. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995.

H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm, I: Continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993. URL `http://dx.doi.org/10.1162/evco.1993.1.1.25`.

H. Mühlenbein and D. Schlierkamp-Voosen. The science of breeding and its application to the breeder genetic algorithm (bga). *Evolutionary Computation*, 1(4):335–360, 1993.

A. Prügel-Bennett and J. Shapiro. An analysis of a genetic algorithm using statistical mechanics. *Physical Review Letters*, 72(9):1305–1309, 1994.

D. Thierens. Selection schemes, elitist recombination, and selection intensity. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 152–159, San Francisco, CA, USA, 1997. Morgan Kaufmann.

Poglavlje 5

Jednostavni algoritmi i lokalne pretrage

U okviru ovog poglavlja pogledat ćemo najprije nekoliko jednostavnih optimizacijskih algoritama koji se temelje na pretpostavci da algoritam već raspolaže nekim početnim rješenjem. U tu svrhu od izuzetnog će nam značaja biti pojam susjedstva. Osvrnut ćemo se također na primjere optimizacije kombinatoričkih problema te kontinuiranih problema kao i na jedan od načina poboljšanja uspješnosti generiranja početnog rješenja.

Susjedstvo Neka je $x \in \mathcal{X}$ neko rješenje iz skupa mogućih rješenja $\mathcal{X}$. Susjedstvo rješenja x, oznaka $\mathcal{N}(x)$, definirano je funkcijom susjedstva $\mathcal{N}$ koja predstavlja preslikavanje $\mathcal{N} : \mathcal{X} \to 2^{\mathcal{X}}$. Ova funkcija svakom rješenju x pridružuje podskup skupa $\mathcal{X}$ koji čini skup svih rješenja koja zovemo susjedima od rješenja x.

Kod problema optimizacije nad kontinuiranim prostorom, susjedstvo rješenja x najčešće se definira kao skup svih rješenja x' koja su od rješenja x udaljena za ne više od nekog fiksnog iznosa ϵ; u tom slučaju susjedstvo je definirano kao:

$$\mathcal{N}(x) = \{x' : |x' - x| \leq \epsilon, \;\; x' \in \mathcal{X}\}.$$

Kod problema kombinatoričke optimizacije uobičajeno je definirati operator pomaka $\pi(x)$ koji iz rješenja x stvara novo rješenje x' na način da provodi ograničenu modifikaciju rješenja x. Primjerice, ako kao reprezentaciju rješenja koristimo binarno kodiranje, operator π bi mogao biti operator koji nad zadanim rješenjem promjeni točno jedan bit na proizvoljno odabranoj poziciji. Tada možemo definirati i udaljenost $d(x, x')$ koja predstavlja broj primjena operatora π koje su potrebne da bi se iz rješenja x došlo do rješenja x'. Primjerice, ako je rješenje x kodirano kao `00000`, rješenje x' kao `01001` te ako je operator π definiran kao operator koji na proizvoljno odabranom mjestu mijenja jedan bit, tada je $d(x, x') = 2$ jer operator π trebamo primijeniti jednom kako bismo iz `00000` dobili `01000`, i potom ga primijeniti još jednom nad tim međurezultatom kako bismo iz `01000` dobili `01001` (moguć je i drugačiji redosljed: `00000`→`00001`→`01001`). Treba napomenuti da je $d(x, x')$ direktno vezano uz definiciju operatora π. Da smo operator π definirali kao operator koji nad zadanim rješenjem može na najviše dva proizvoljno odabrana mjesta promijeniti vrijednost bita, vrijedilo bi $d(x', x) = 1$ jer uz tako definirani operator π iz rješenja `00000` u jednom koraku možemo dobiti rješenje `01001`. Evo još jednog primjera operatora π: pretpostavimo da je rješenje problema permutacija brojeva, i da je trenutno rješenje $x = (2, 1, 3)$. Ako je operator π definiran kao operator zamjene dva elementa, i ako susjedstvo generiramo uz $d(x, x') = 1$, vrijedi: $\mathcal{N}(x) = \{(3, 1, 2), (2, 3, 1), (1, 2, 3)\}$.

Uočimo odmah i da, ovisno o načinu kako je definiran operator π, rješenje čije susjedstvo razmatramo može ili ne mora sadržavati to rješenje. Ako operator π definiramo navođenjem gornje ograde, takav će operator generirati susjedstvo koje sadrži i razmatrano rješenje; primjerice, neka je π operator koji u trenutnom rješenju mijenja *najviše jedan* bit na nasumično odabranoj poziciji. S druge strane, ako zadamo i donju netrivijalnu granicu, tada originalno rješenje neće biti uključeno u susjedstvo; primjerice, neka je π operator koji u trenutnom rješenju mijenja *točno jedan* bit na nasumično odabranoj poziciji. Na koju se vrstu susjedstva misli, obično je jasno iz konteksta. U algoritmima u nastavku pretpostavka je da susjedstvo uključuje i trenutno rješenje (osim ako nije drugačije zadano).

Uz definiran operator π susjedstvo rješenja x tada je uobičajeno definirati kao

$$\mathcal{N}(x) = \{x' : d(x', x) \leq k, \;\; x' \in \mathcal{X}\}$$

gdje je k pozitivan cijeli broj. Možemo uočiti da će, očekivano, uz veći k i susjedstvo biti veće.

Jednom kada smo definirali pojam susjedstva, možemo definirati iterativni algoritam pretraživanja kako to prikazuje pseudokod 5.1.

Pseudokod 5.1 Iterativni algoritam pretraživanja.

```
x(0) ... početno rješenje
t = 0
ponavljaj
  Generiraj N(x(t)) tj. susjedstvo rješenja x(t)
  x(t+1) = odaberi neko rješenje iz generiranog susjedstva N(x(t))
  t = t+1
dok nije zadovoljen uvjet zaustavljanja
vrati x(t)
```

Algoritam u svakoj iteraciji temeljem trenutnog rješenja generira susjedstvo i potom iz tog susjedstva bira neko rješenje koje će postati novo rješenje algoritma. Ako ovaj odabir napravimo tako da se odabire uvijek najbolje rješenje, dobit ćemo pohlepni algoritam uspona na vrh koji je prikazan pseudokodom 5.2.

Pseudokod 5.2 Pohlepni algoritam uspona na vrh.

```
x(0) ... početno rješenje
t = 0
ponavljaj
  Generiraj N(x(t)) tj. susjedstvo rješenja x(t)
  x(t+1) = odaberi najbolje rješenje iz generiranog susjedstva N(x(t))
  t = t+1
  ako je x(t)=x(t-1) prekini petlju
dok nije zadovoljen uvjet zaustavljanja
vrati x(t)
```

Pretpostavka prethodnog algoritma je da je u susjedstvo od x uključeno i rješenje x tako da odabirom najboljeg rješenja iz generiranog susjedstva sigurno ne odabiremo rješenje koje je manje kvalitete od trenutnog. Opisani algoritam neizravno postavlja ograničenje na definiciju operatora π kojim se generira susjedstvo. Kada bismo taj operator definirali kao operator koji iz zadanog rješenja može generirati bilo koje drugo rješenje iz skupa $\mathcal{X}$, opisani algoritam bi se u samo jednom prolazu pretvorio u algoritam iscrpne pretrage: susjedstvo zadanog rješenja postalo bi jednako čitavom prostoru mogućih rješenja čime bi njegovo generiranje, vrednovanje i odabir najboljeg rješenja postalo vremenski neprihvatljivo. Primjerice, ako za prikaz rješenja koristimo binarni prikaz duljine 1024 bita, susjedstvo svakog rješenja sadržavalo bi svih 2^{1024} mogućih binarnih riječi koje bi trebalo vrednovati, usporediti i odabrati najbolje. Na računalu koje u jednoj nanosekundi uspije generirati i vrednovati jedno rješenje te ga usporediti s do tada pronađenim najboljim iz tog susjedstva čitav postupak trajao nešto manje od 10^{292} godina.

Stoga je nužno da operator π i pripadno susjedstvo budu definirani na način koji će osigurati da je susjedstvo dovoljno male veličine da ga se može u razumnom vremenu generirati i pretraživati, ili pak postupak odabira rješenja iz susjedstva treba biti takav da susjedstvo samo uzorkuje i radi s njegovim podskupom. U oba slučaja, nažalost, općenito govoreći, gubi se garancija pronalaska globalnog optimuma te se uvode lokalni optimumi – ne kao inherentno svojstvo funkcije koju optimiramo već kao posljedica načina na koji definiramo susjedstvo ili načina na koji ga uzorkujemo.

Lokalni optimum Lokalni optimum s obzirom na susjedstvo $\mathcal{N}(x^*)$ definiramo kao rješenje x^* za koje vrijedi da su sva rješenja $x \in \mathcal{N}(x^*)$ i $x \neq x^*$ lošija od rješenja x^*. Pohlepni algoritam pretraživanja stoga će zaglaviti u rješenju x^*.

Zgodno je još i uočiti zanimljivo svojstvo globalnog optimuma: globalni optimum ujedno je i lokalni optimum neovisno o načinu kako je definiran operator pomaka π i pripadno susjedstvo (jasno je zašto je tome tako).

U algoritmu 5.2 kompletno susjedstvo može se, a i ne mora generirati. Uobičajene su tri inačice.

- Generira se cjelokupno susjedstvo te se pronalazi najbolje rješenje. U slučaju da je susjedstvo veliko, ovaj pristup će biti računski izuzetno zahtjevan.
- Susjedstvo se generira rješenje po rješenje i postupak se zaustavlja čim se pronađe prvo bolje rješenje. U slučaju da je trenutno rješenje lokalni optimum, postupak se pretvara u prethodni slučaj u kojem se radi iscrpna pretraga cjelokupnog susjedstva.
- Posredstvom slučajnog mehanizma odabire se neko od rješenja iz susjedstva koja su bolja od trenutnog (dakle, nije nužno da će biti odabrano najbolje rješenje iz susjedstva).

Koja god od strategija da se odabere, opisani algoritmi podložni su zaglavljivanju u lokalnim optimumima. Stoga postoji nekoliko varijanti algoritama koje za cilj imaju povećati robusnost algoritma s obzirom na lokalne optimume. Tri najčešće navedene su u nastavku.

Iterativno pokretanje algoritma s različitih početnih rješenja. Postupak pretraživanja, posebice ako se iz algoritma uklone stohastičke odluke, određen je rješenjem od kojeg je pretraga pokrenuta. Stoga je jedna mogućnost postuzanja robusnosti iterativno pokretanje algoritma i to svaki puta s novim početnim rješenjem. Tim pristupom nadamo se da algoritam neće svaki puta zaglaviti u istom lokalnom optimumu.

Prihvaćanje i rješenja koja nisu bolja od trenutnog. S obzirom na danu definiciju lokalnog optimuma kao najboljeg rješenja u njegovom susjedstvu, jasno je da algoritam koji kao sljedeće rješenje prihvaća samo bolja rješenja od trenutnog neće uspjeti napustiti lokalni optimum. Ako se u algoritam doda mogućnost da algoritam "zaboravi" trenutno najbolje rješenje i u nekim slučajevima prihvati i lošije rješenje, moguće je da će se u susjedstvu tog novog lošijeg rješenja zateći i neko rješenje koje je bolje od prethodnog lokalnog optimuma (ili koje će voditi do rješenja koje će voditi do takvog rješenja), pa će u tom slučaju algoritam uspjeti napustiti lokalni optimum i krenuti prema još boljim rješenjima.

Promjena susjedstva. Lokalni optimum x može biti inherentno svojstvo funkcije koja se optimira i koja u toj točki doista ima ekstrem ili pak može biti posljedica načina na koji je definirano susjedstvo odnosno odabranog operatora π. Ukoliko je u x nastupio ovaj drugi slučaj, promjenom operatora π moguće je da rješenje x više neće biti lokalni optimum.

5.1 Primjeri algoritama lokalne pretrage

5.1.1 Višekratno pokretanje lokalne pretrage

Algoritam je poznat pod nazivom engl. *Multistart local search*. Osnovna ideja prikazana je pseudokodom 5.3.

U svakom prolazu vanjske petlje u algoritmu se na slučajan način generira početno rješenje koje će potom biti predano algoritmu lokalne pretrage. Pri tome se početna rješenja u svakom koraku generiraju nezavisno, odnosno jedan prolaz vanjske petlje algoritma niti na koji način ne utječe na početno rješenje koje će biti generirano za sljedeći prolaz.

Pseudokod 5.3 Višekratno pokretanje lokalne pretrage.

```
x'' = null ... još nema najboljeg rješenja
ponavljaj
  x = generiraj slučajno početno rješenje
  x' = primjeni lokalnu pretragu na rješenje x
  ako je x''=null ili ako je x' bolji od x'' tada
   x'' = x'
  kraj
dok nije zadovoljen uvjet zaustavljanja
vrati x''
```

5.1.2 Iterativna lokalna pretraga

Poboljšanje prethodnog algoritma je algoritam poznat pod nazivom engl. *Iterated local search*. Osnovna ideja prikazana je pseudokodom 5.4.

Pseudokod 5.4 Iterativno pokretanje lokalne pretrage.

```
x0 = generiraj slučajno početno rješenje ili ga preuzmi izvana
x = primjeni lokalnu pretragu na x0
ponavljaj
  x' = perturbiraj(x, povijesni podatci)
  x'' = primjeni lokalnu pretragu na x'
  x = prihvati(x, x'', memorija)
dok nije zadovoljen uvjet zaustavljanja
vrati najbolje pronađeno rješenje
```

Algoritam kao sastavni dio koristi neku drugu lokalnu pretragu. Prethodno opisani algoritam višekratnog pokretanja lokalne pretrage može davati loše rezultate kod određene vrste problema kod kojih je prostor pretraživanja vrlo veliki. Kod takvih problema čest je slučaj da imaju mnoštvo lokalnih optimuma slične kvalitete, pa nasumično generiranje početnog rješenja u tom slučaju obično rezultira pronalaskom lokalnih optimuma slične kvalitete. Kako bi se tome doskočilo, ovaj algoritam najprije uporabom algoritma lokalne pretrage pronalazi prvi lokalni optimum i potom koristi operator perturbiranja čija je zadaća stvoriti novo rješenje koje će se potom koristiti kao početno rješenje za novo pokretanje lokalne pretrage, nakon čega će se proces ponavljati. Operator perturbiranja pri tome ima zadaću modificirati trenutno rješenje tako da dio rješenja sačuva a dio rješenja drastično modificira. Ako je udio rješenja koji će se modificirati premali, moguće je da će se primjenom lokalne pretrage opet dobiti isti lokalni optimum. Ako je udio rješenja koji će se modificirati preveliki, izgubit će se kvalitetni dijelovi već pronađenog rješenja, i u ekstremu algoritam će se pretvoriti u algoritam višekratnog pokretanja lokalne pretrage. Stoga optimalni udio i način provođenja perturbacije rješenja ovisi o problemu koji se rješava. Nakon što se nad perturbiranim rješenjem x' pokrene lokalna pretraga i time dobije rješenje x'', potrebno je odlučiti hoće li se to novo rješenje prihvatiti kao trenutno rješenje ili će se sačuvati staro rješenje, i to radi funkcija `prihvati`. Prihvaćanjem samo boljih rješenja, algoritam će imati više tendencije brže zaglaviti u lokalnom optimumu. S druge strane, prihvaćanje i lošijih rješenja (tada je pitanje pod kojim uvjetima i koliko lošijih) može pomoći općenitoj robusnosti algoritma ali isto tako može usporiti brzinu kojom se dobivaju bolja rješenja što ponekad može biti vremenski neprihvatljivo.

5.1.3 Pohlepna randomizirana adaptivna procedura pretrage

Algoritam *pohlepna randomizirana adaptivna procedura pretrage* poznat pod nazivom GRASP, što je kratica od engl. *Greedy randomized adaptive search procedure*. Osnovna ideja prikazana je pseudoko-

dom 5.5.

Pseudokod 5.5 Pohlepna randomizirana adaptivna procedura pretrage.

```
x = null
ponavljaj
  x' = pohlepnim slučajnim algoritmom konstruiraj novo početno rješenje
  x'' = primjeni lokalnu pretragu na x'
  x = prihvati(x, x'')
dok nije zadovoljen uvjet zaustavljanja
vrati najbolje pronađeno rješenje, tj. x
```

Ovaj algoritam sličan je algoritmu višekratnog pokretanja lokalne pretrage, uz važnu razliku: umjesto da se početna rješenja stvaraju sasvim slučajno, početna se rješenja konstruiraju element po element koristeći neku pohlepnu (i po mogućnosti randomiziranu) proceduru. Ovime se osigurava da rješenja od kojih se kreće nisu skroz slučajna, već u sebi sadrže dijelove rješenja koja se vjerojatno nalaze i u rješenju koje je globalni optimum, a također zbog načina na koji su konstruirana, već imaju kvalitetu koja je bitno bolja od tipičnih nasumično stvorenih rješenja. Funkcija `prihvati` najčešće je implementirana tako da deterministički prihvaća novo rješenje samo ako je ono bolje od trenutnog.

Izgradnja početnog rješenja radi se element po element. Primjerice, da se radi o problemu trgovačkog putnika, uz fiksirani prvi grad obilaska trebali bismo odabrati koji ćemo grad posjetiti kao drugi, pa koji ćemo grad posjetiti kao treći, i tako redom sve dok ima neposjećenih gradova. U svakom koraku konstrukcije rješenja definira se skup C kao skup elemenata koje je u tom koraku moguće dodati u trenutno rješenje (npr. gradovi koje još nismo posjetili a do njih postoji direktna veza iz posljednjeg grada koji smo prethodno dodali u rješenje). Potom se za svaki element $c \in C$ računa doprinos kvaliteti trenutnog rješenja ako mu se nadoda element c (za TSP doprinos će biti to veći što je grad-kandidat bliži posljednjem dodanom gradu). Vrednovanje se obavlja pohlepnom funkcijom $g(c)$ (npr. recipročna vrijednost udaljenosti za slučaj TSP-a). Ako se radi o klasičnoj pohlepnoj konstrukciji, iz skupa C uzima se onaj element koji najviše pridonosi povećanju kvalitete rješenja i taj se element nadodaje u trenutno rješenje nakon čega se postupak ponavlja, sve dok rješenje nije u potpunosti izgrađeno. U slučaju randomizirane pohlepne konstrukcije iz skupa C gradi se ograničena lista kandidata, RCL, engl. *rescricted candidate list* i potom se iz te liste odabire posredstvom slučajnog mehanizma element koji će biti nadodan u trenutno rješenje. Listu RCL moguće je graditi temeljem dva kriterija: ograničenjem na kardinalnost liste gdje se u listu uključuje najboljih k elemenata s obzirom na funkciju g, ili pak uzimajući u obzir vrijednosti funkcije g. U ovom posljednjem slučaju uobičajeno je u skupu pronaći minimalnu vrijednost funkcije g (oznaka $g_{\min}$) i maksimalnu vrijednost funkcije g (oznaka $g_{\max}$), i potom uz zadanu vrijednost $\alpha \in [0, 1]$ u listu uključiti sve elemente čija je kvaliteta od kvalitete najboljeg elementa udaljena za najviše $\alpha \cdot (g_{\max} - g_{\min})$, tj.

$$RCL = \{c \in C | g(c) \geq g_{\max} - \alpha \cdot (g_{\max} - g_{\min})\}.$$

Opisani kriterij vrijedi uz pretpostavku da je funkcija g doista funkcija koja govori koliko će se dodavanjem elementa c u trenutno rješenje povećati njegova kvaliteta. Ako kao funkciju g koristimo funkciju koja mjeri pogoršanje kvalitete, odnosno koja predstavlja kaznu, RCL listu možemo definirati kako slijedi:

$$RCL = \{c \in C | g(c) \geq g_{\min} + \alpha \cdot (g_{\max} - g_{\min})\}.$$

Možemo uočiti da će se uz odabir $\alpha = 0$ ograničena lista kandidata svesti samo na podskup elemenata koji maksimalno doprinose poboljšanju kvalitete rješenja čime dobivamo klasični pohlepni konstrukcijski algoritam. Uz odabir $\alpha = 1$ postupak konstrukcije se pak pretvara u čisto nasumično generiranje rješenja. Uz vrijednosti koje su između ova dva ekstrema algoritam radi kompromis između ova dva ponašanja. Točan iznos parametra α koji je optimalan ovisi o problemu koji se rješava. Osim uporabe fiksne vrijednosti parametra postoje i varijante algoritma kod kojih se ova vrijednosti dinamički mijenja.

5.2 Numeričke optimizacije

Optimizacijski algoritmi često se primjenjuju na visoko-nelinearne funkcije koje su definirane nad kontinuiranim domenama. Ako su te funkcije derivabilne, tada napredak algoritama često možemo pospješiti prikladnom lokalnom pretragom koja će ubrzati konvergenciju ka lokalnom optimumu koji se nalazi u blizini trenutnog rješenja koje je pronašao optimizacijski algoritam. U tu svrhu na raspolaganju nam stoji čitav niz algoritama. Kako ova knjiga ne predstavlja udžbenik iz navedenog područja matematičke optimizacije, ovdje ćemo navesti samo podskup metoda, a zainteresirani čitatelj se upućuje na odgovarajuću dopunsku literaturu.

Pretpostavimo da je zadana funkcija $f(\vec{x})$ čiji tražimo minimum. To je skalarna funkcija koja je definirana nad n-dimenzijskim vektorom $\vec{x}$. Primjerice, funkcija

$$f(\vec{x}) = (x_1 - 4)^2 + (x_2 + 8)^2 - (x_3 + 5)^2$$

je primjer funkcije koja je definirana nad $\mathcal{R}^3$.

Gradijent funkcije iz primjera je vektor stupac:

$$\begin{aligned}\nabla f(\vec{x}) &= \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \end{bmatrix} \\ &= \begin{bmatrix} 2x_1 - 8 \\ 2x_2 + 16 \\ 2x_3 + 10 \end{bmatrix}\end{aligned}$$

Funkcija poprima ekstremnu vrijednost (ili sedlo) u točkama u kojima je $\nabla f(\vec{x}) = \vec{0}$. Na primjeru zadane funkcije ovo se svodi na sustav od tri jednadžbe koje je trivijalno riješiti egzaktno:

$$\begin{aligned}2x_1 - 8 &= 0 \\ 2x_2 + 16 &= 0 \\ 2x_3 + 10 &= 0\end{aligned}$$

iz čega slijedi:

$$\begin{aligned}x_1 &= 4 \\ x_2 &= -8 \\ x_3 &= -5\end{aligned}$$

što predstavlja jedini minimum ove funkcije: $\vec{x} = [4, -8, -5]^T$.

5.2.1 Pretraživanje u zadanom smjeru

Optimizacija funkcije koja je definirana nad višedimenzijskim prostorom inherentno je problematična. Stoga je najjednostavniji pokušaj kroćenja ove složenosti iterativni postupak koji problem optimizacije nad višedimenzijskim prostorom svodi na optimizaciju nad jednom dimenzijom. Pretpostavimo ponovno da je $f(\vec{x})$ funkcija čiji se traži minimum, pri čemu je $\vec{x} \in \mathcal{R}^n$. Neka je $\vec{x}^{(0)}$ odabrano početno rješenje (primjerice, točka koju smo generirali nasumično). Pretpostavimo također da je $\vec{d}^{(0)} \in \mathcal{R}^n$ vektor koji pokazuje u smjeru u kojem je potrebno modificirati trenutno rješenje kako bi vrijednost funkcije pala, gledano iz trenutne točke $\vec{x}^{(0)}$.

Problem ćemo pretvoriti u jednodimenzijski tako što ćemo uvesti parametar λ te definirati novu funkciju:

$$\theta(\lambda) = f(\vec{x}^{(0)} + \lambda \cdot \vec{d}^{(0)}).$$

Funkcija θ ovisi samo o *lambda*; za nju su $\vec{x}^{(0)}$ i $\vec{d}^{(0)}$ konstante. Sada je zadatak pronaći vrijednost λ^* koja minimizira funkciju $\theta(\lambda)$ i potom kao novo rješenje uzeti vrijednost:

$$\vec{x}^{(1)} = \vec{x}^{(0)} + \lambda \cdot \vec{d}^{(0)}.$$

Pogledajmo to na konkretnom primjeru gdje je $f(\vec{x}) = (x_1 - 4)^2 + (x_2 + 8)^2 - (x_3 + 5)^2$, $\vec{x}^{(0)} = [6, -10, -9]^T$ te $\vec{d} = [-1, 2, 1]^T$. Lagano se možemo uvjeriti da će za male vrijednosti λ funkcija padati te će potom u jednom trenutku početi rasti. Za vrijednosti λ iz skupa $\{0, 1, 2, 3, 4\}$ sljedeća tablica prikazuje vrijednosti koje poprima funkcija $\theta(\lambda)$.

λ	$\vec{x}^{(0)} + \lambda \cdot \vec{d}^{(0)}$	$\theta(\lambda)$
0	$[6, -10, -9]^T$	24
1	$[5, -8, -8]^T$	10
2	$[4, -6, -7]^T$	8
3	$[3, -4, -6]^T$	18
4	$[2, -2, -5]^T$	40

Pitanje koje treba razriješiti jest kako na sistematičan način utvrditi traženu vrijednost λ^*. Naime, vidimo da uz premali korak λ funkciju nećemo dovoljno minimizirati – da smo napravili veći korak, dobili bismo bolje rješenje. S druge pak strane, ako napravimo preveliki korak, iako se krećemo u dobrom smjeru, vrijednost funkcije će rasti a ne padati. Dakako, što je "preveliko" a što "premalo" ovisi o konkretnoj funkciji koju rješavamo: nekada te vrijednosti mogu biti reda veličine 10^{-5} a nekada 10^5.

Jedan od pristupa jest pogledati derivaciju funkcije $\theta(\lambda)$. Vrijedi:

$$\frac{d\theta(\lambda)}{d\lambda} = \nabla f(\vec{x})^T \cdot \vec{d}.$$

Kako je u točki $\vec{x}^{(0)}$ vektor $\vec{d}^{(0)}$ odabran tako da pokazuje u smjeru pada funkcije f, slijedi da je

$$\left.\frac{d\theta(\lambda)}{d\lambda}\right|_{\lambda=0} < 0.$$

Definirat ćemo stoga da je $\lambda_{lower} = 0$.

Ono što trebamo jest gornju ogradu za $\lambda = \lambda_{upper}$ za koju po prvi puta vrijedi

$$\left.\frac{d\theta(\lambda)}{d\lambda}\right|_{\lambda_{upper}} > 0.$$

Ako ne znamo pametniji način kako doći do vrijednosti λ_{upper}, možemo pokušati s $\lambda_{upper} = 1$, pa izračunamo $\left.\frac{d\theta(\lambda)}{d\lambda}\right|_{\lambda_{upper}}$; ako je vrijednost manja od 0, postavimo $\lambda_{upper} \leftarrow \lambda_{upper} \cdot 2$ i ponovimo provjeru; procjenu za λ_{upper} na opisani način eksponencijalno povećavamo sve dok se nastupi uvjet za prekid.

Na ovaj način uspjeli smo ograditi vrijednosti za λ s donje (λ_{lower}) i gornje (λ_{upper}) strane, i sigurni smo da se minimum nalazi negdje unutar tog intervala. Sve što je još potrebno jest pronaći vrijednost $\lambda^* \in [\lambda_{lower}, \lambda_{upper}]$ koja minimizira funkciju θ. Nakon stvaranja nove procjene pozicije optimuma $\vec{x}^{(1)} = \vec{x}^{(0)} + \lambda \cdot \vec{d}^{(0)}$ čitav se postupak može iterativno ponoviti, što nas vodi na općeniti algoritam pretraživanja u zadanom smjeru koji je dan pseudokodom 5.6.

Pseudokod 5.6 Pretraživanje u zadanom smjeru.

```
Ponavljaj za k = 1, 2, ...
    Ako je x^(k) optimum, prekini s izvođenjem i vrati x^(k).
    Inače
        Utvrdi d^(k) – smjer pretrage
        Pronađi λ* > 0 – korak
        Definiraj x^(k+1) = x^(k) + λ* · d^(k) – novo rješenje
```

Za pronalazak optimalne vrijednosti parametra λ^* uz pretpostavku da je funkcija $\theta(\lambda)$ konveksna te da smo minimum ogradili i imamo vrijednosti λ_{lower} i λ_{upper} možemo koristiti niz metoda. Metoda bisekcije prikaza ne pseudokodom 5.7.

Pseudokod 5.7 Metoda bisekcije.

Korak 0. Postavi $k = 0$. Postavi $\lambda_l = \lambda_{lower}$ te $\lambda_u = \lambda_{upper}$.

Korak k. Postavi $\lambda = \frac{\lambda_l + \lambda_u}{2}$ i izračunaj $\left.\frac{d\theta(\lambda)}{d\lambda}\right|_\lambda$.

Ako je $\left.\frac{d\theta(\lambda)}{d\lambda}\right|_\lambda > 0$, redefiniraj $\lambda_u = \lambda$, $k = k + 1$.

Ako je $\left.\frac{d\theta(\lambda)}{d\lambda}\right|_\lambda < 0$, redefiniraj $\lambda_l = \lambda$, $k = k + 1$.

Ako je $\left.\frac{d\theta(\lambda)}{d\lambda}\right|_\lambda \approx 0$, stani jer smo došli do minimuma dovoljno blizu.

U pseudokodu 5.7 se kao uvjet zaustavljanja koristi operator $\approx$ koji nam omogućava da stanemo kada smo dovoljno zadovoljni pronađenim rješenjem. Iz opisanog algoritma također se može uočiti da algoritam u svakom koraku interval dijeli na pola čime širina intervala pretraživanja pada eksponencijalno.

Za funkciju iz primjera te zadani vektor pretraživanja analitički možemo izračunati optimalnu vrijednost od λ^*. Naime, kako tražimo minimum funkcije $\theta(\lambda)$, slijedi da mora vrijediti: $\left.\frac{d\theta(\lambda)}{d\lambda}\right|_\lambda = 0$, što nas vodi na sustav $-20 + 12\lambda = 0$ iz čega slijedi $\lambda^* = \frac{5}{3} \approx 1.666666$.

Označimo s ϵ vrijednost koliko izračunata derivacija u pseudokodu 5.7 smije odstupati o nule da bismo prekinuli postupak. Sljedeća tablica prikazuje broj iteracija te izračunata rješenja ovisno o vrijednosti ϵ. Pretraživanje je rađeno unutar intervala $[0.0, 2.0]$ pri čemu je gornja granica utvrđena iterativnim množenjem s 2 dok derivacija funkcije θ nije postala pozitivna.

ϵ	napravljeni broj koraka	λ^*
1	4	1.625
0.5	5	1.6875
0.1	7	1.671875
0.05	8	1.6640625
0.01	10	1.666015625
0.005	11	1.6669921875
0.001	13	1.666748046875

Osim prikazane metode, postoji još niz drugih poput metode zlatnog reza, pravila *Armija*, pravila *Armijo-Wolfe* itd.

5.2.2 Gradijentni spust

U prethodnom podpoglavlju nismo odgovorili na pitanje kako odabrati vektor $\vec{d}$ u čijem smjeru će se raditi pretraga. Algoritam gradijentnog spusta daje jedan mogući odgovor na to pitanje. Kako bi dao odgovor na to pitanje, algoritam gradijentnog spusta u okolici trenutnog rješenja $\vec{x}$ gradi linearni model funkcije koja se optimira – drugim riječima, algoritam pretpostavlja da se u okolici trenutnog rješenja funkcija ponaša kao hiperravnina. Neka je zadana funkcija $f(\vec{x})$ čiji tražimo minimum. Taylorov razvoj funkcije f u okolici rješenja $\vec{x}$ glasi:

$$f(\vec{x} + \vec{\tau}) = f(\vec{x}) + \nabla f(\vec{x})^T \cdot \vec{\tau} + \frac{1}{2!} \cdot \vec{\tau}^T \cdot \nabla^2 f(\vec{x}) \cdot \vec{\tau} + \zeta \tag{5.1}$$

pri čemu su s ζ označene pogreške višeg reda. Za potrebe izgradnje algoritma gradijentnog spusta u okolici točke $\vec{x}$ funkcija $f(\vec{x})$ modelira se linearnom funkcijom $g(\vec{x})$ koja je definirana na sljedeći način:

$$g(\vec{x} + \vec{\tau}) = f(\vec{x}) + \nabla f(\vec{x})^T \cdot \vec{\tau}. \tag{5.2}$$

Pri tome je $\nabla f(\vec{x})$ vektor koji pokazuje u smjeru maksimalnog povećanja funkcije g a $-\nabla f(\vec{x})$ pokazuje u smjeru smanjenja vrijednosti funkcije g. Stoga se kao vektor u čijem će se smjeru raditi pretraga može uzeti $-\nabla f(\vec{x})$.

Ako se ne želi raditi pretraživanje u smjeru zadanog vektora, jednostavna varijanta algoritma gradijentnog spusta može se dobiti tako da se pomak uvijek radi tako da se trenutnoj procjeni rješenja

doda skalirana negativna vrijednost gradijenta pri čemu se kao faktor skaliranja koristi mali pozitivni broj (npr. 0.1). Treba međutim biti svjestan da odabrani faktor može biti premali (ili preveliki) pa postupak optimizacije izvedene na ovaj način može biti vrlo spor ili može čak divergirati. ovo je način kako je u praksi izvedeno nekoliko poznatih algoritama, od kojih možemo istaknuti algoritam *Backpropagation* koji se koristi za učenje unaprijednih neuronskih mreža.

5.2.3 Newtonova metoda

Za potrebe Newtonove metode u okolici trenutnog rješenja $\vec{x}$ gradi se kvadratni model funkcije koja se optimira – pretpostavka algoritma je da se u okolici trenutnog rješenja funkcija ponaša kvadratno. Neka je zadana funkcija $f(\vec{x})$ čiji tražimo minimum. Taylorov razvoj funkcije f u okolici rješenja $\vec{x}$ glasi:

$$f(\vec{x}+\vec{\tau}) = f(\vec{x}) + \nabla f(\vec{x})^T \cdot \vec{\tau} + \frac{1}{2!}\vec{\tau}^T \nabla^2 f(\vec{x})\vec{\tau} + \zeta \tag{5.3}$$

pri čemu su s ζ označene pogreške višeg reda. Za potrebe izgradnje Newtonovog algoritma u okolici točke $\vec{x}$ funkcija $f(\vec{x})$ modelira se kvadratnom funkcijom $g(\vec{x})$ koja je definirana na sljedeći način:

$$g(\vec{x}+\vec{\tau}) = f(\vec{x}) + \nabla f(\vec{x})^T \cdot \vec{\tau} + \frac{1}{2!}\vec{\tau}^T \nabla^2 f(\vec{x})\vec{\tau}. \tag{5.4}$$

Pri tome je $H(f) = \nabla^2 f(\vec{x})$ Hesseova matrica odnosno matrica drugih parcijalnih derivacija. Ova matrica računa se na sljedeći način:

$$\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{5.5}$$

Funkciju $g(\vec{x}+\vec{\tau})$ tretirat ćemo kao funkciju od $\vec{\tau}$. Naime, u i-toj iteraciji algoritma $\vec{x}$ predstavlja trenutnu procjenu optimuma i za funkciju g to je konstanta. Funkcija g kvadratna je funkcija s obzirom na $\vec{\tau}$. Da bismo utvrdili vrijednost $\vec{\tau}$ za koju funkcija g postiže optimum, trebamo izračunati njezinu derivaciju s obzirom na $\vec{\tau}$ i potom to izjednačiti s nulom. Deriviranjem funkcije 5.5 s obzirom na $\vec{\tau}$ dobivamo:

$$\nabla g = \nabla f(\vec{x}) + H(\vec{x}) \cdot \vec{\tau}. \tag{5.6}$$

Uvjet za optimum je da je gradijent jednak 0:

$$\nabla f(\vec{x}) + H(\vec{x}) \cdot \vec{\tau} = \vec{0} \tag{5.7}$$

iz čega slijedi:

$$H(\vec{x}) \cdot \vec{\tau} = -\nabla f(\vec{x}) \tag{5.8}$$

odnosno

$$\vec{\tau} = -H(\vec{x})^{-1} \cdot \nabla f(\vec{x}) \tag{5.9}$$

Izraz $H(\vec{x})^{-1}$ predstavlja matrični inverz Hesseove matrice i upravo potreba za računom inverza čini ovaj postupak računski vrlo zahtjevnim. Dobiveni vektor $\vec{\tau}$ dalje se koristi kao vektor u čijem smjeru se radi pretraga prethodno opisanim algoritmom. Treba odmah uočiti: ako je funkcija f kvadratna funkcija, tada joj njezin model g savršeno odgovara i postupak minimizacije će završiti u jednom koraku.

5.2.4 Druge metode

Uz prethodne dvije osnovne metode, razvijen je još niz drugih metoda, a u neke od značajnijih još spada i porodica kvazi-Newtonovih metoda, metoda Levenberg-Marquardt i druge.

5.3 Povećanje učinkovitosti generiranja početnih rješenja

U prisustvu tvrdih ograničenja često uopće nije trivijalno stvoriti čak i neko početno rješenje, a ona su nam najčešće potrebna kako bismo mogli krenuti popravljati ih nekim drugim algoritmom. Osim navedenog slučaja, često pišemo i populacijske algoritme na način da ne dozvoljavamo da se u populaciju ubace rješenja koja krše tvrda ograničenja. Problem s kojim smo tada suočeni jest kako uopće doći do početne populacije. Stoga ćemo u nastavku pogledati jedan od mogućih načina kako pristupiti rješavanju takvih problema.

Primjer s kojim ćemo raditi jest jednostavan problem raspoređivanja. Pretpostavimo da imamo pet kolegija: K_0, K_1, K_2, K_3 i K_4 koje je potrebno razmjestiti u pet raspoloživih termina T_0, T_1, T_2, T_3 i T_4. U bilo koji od termina moguće je smjestiti najviše jedan kolegij jer ih sve drži isti nastavnik. Također, iz određenih razloga koji nam sada nisu važni, svaki kolegij ne može biti smješten u svaki termin. Umjesto toga, potrebno je osigurati da kolegij K_0 bude smješten u termin T_1, kolegij K_1 u termin T_0, T_1, T_2, T_3 ili T_4, kolegij K_2 u termin T_0 ili T_2, kolegij K_3 u termin T_0, T_3 ili T_4 te kolegij K_4 u termin T_0, T_1 ili T_2. Ova ograničenja nazvat ćemo tvrda ograničenja – ako ih raspored ne poštuje, raspored je neupotrebljiv. Jednom kada imamo raspored koji je upotrebljiv, dalje ćemo ga vrednovati temeljem niza drugih karakteristika koje ovdje nećemo razmatrati.

Rješavanju ovog problema pristupit ćemo tako da definiramo polje courseTerm; to će polje imati onoliko elemenata koliko ima kolegija. i-ti element polja sadržavat će oznaku termina koji je dodijeljen i-tom kolegiju. Primjerice, polje $(4, 0, 3, 1, 2)$ bit označavalo sljedeći raspored: $K_0 \to T_4$, $K_1 \to T_0$, $K_2 \to T_3$, $K_3 \to T_1$, $K_4 \to T_2$.

Najjednostavniji pokušaj generiranja ovakvih raspored prikazan je u kodu 5.1, gdje se za svaki kolegij nasumice odabere neki od postojećih termina. Jednom izgrađen, raspored se šalje funkciji za provjeru valjanosti koja analizira je li u svaki termin smješten najviše jedan kolegij te je li termin odabran za kolegij na popisu dozvoljenih termina za taj kolegij.

Ispis 5.1: Generiranje rasporeda, pokušaj 1 - primjer u programskom jeziku Java.

```
1  public static void main(String[] args) {
2      int[][][] acceptableTerms = new int[][][] {
3              {1},
4              {0,1,2,3,4},
5              {0,2},
6              {0,3,4},
7              {0,1,2}
8      };
9      int numberOfCourses = acceptableTerms.length;
10     int numberOfTerms = 5;
11
12     Random rand = new Random();
13     int[] courseTerm = new int[numberOfCourses];
14     int numberOfAttempts = 100000;
15
16     int failuresCount = 0;
17     for(int i = 0; i < numberOfAttempts; i++) {
18         // Za sve kolegije postavi termine na nevaljale
19         for(int k = 0; k < numberOfCourses; k++) {
20             courseTerm[k] = -1;
21         }
22         for(int k = 0; k < numberOfCourses; k++) {
23             int picked = rand.nextInt(numberOfTerms);
24             courseTerm[k] = picked;
25         }
26         if(!check(courseTerm, acceptableTerms, numberOfTerms)) {
27             failuresCount++;
28         }
29     }
30     System.out.println("Uspješnost stvaranja: "+
31         ((double)(numberOfAttempts-failuresCount)/(double)numberOfAttempts*100)+
32         "%, broj uspjeha: "+(numberOfAttempts-failuresCount));
33 }
```

Program prikazan u ispisu 5.1 ponavlja postupak stvaranja rješenja 100 000 puta. Ovo sve pokrenuli smo sto puta kako bismo prikupili statističke podatke o uspješnosti stvaranja rješenja koja zadovoljavaju postavljena tvrda ograničenja. Uspješnost dobivanja valjanih rješenja bila je 0%. Iako je ovaj problem naoko izuzetno jednostavan i lagano čak i ručno rješiv s obzirom na zadanu veličinu. Malo detaljnija analiza pokazala je da funkcija za vrednovanje rješenja u poprilici 20% slučajeva rješenja ruši zbog toga što je više kolegija smješteno u isti termin a u poprilici 80% slučajeva rješenja ruši zbog toga što su kolegiji smješteni u neprihvatljive termine.

Prvi korak prema poboljšanju učinkovitosti generiranja rješenja jest pokušati riješiti onih 20% neuspjeha: umjesto da za svaki kolegij generiramo neki nasumični termin iz popisa mogućih termina, promijenit ćemo kod tako da za svaki kolegij odabiremo nasumično neki od još neiskorištenih termina čime ćemo osigurati da nam rješenja ne padaju zbog više kolegija smještenih u isti termin. Jedan od načina kako to možemo napraviti jest u petlji za svaki kolegij nasumice odabirati neki od mogućih termina te potom provjeravati jesmo li taj termin već odabrali za neki od prethodnih kolegija i postupak ponavljati sve dok je to slučaj. Međutim, možemo i bolje, što je ilustrirano u ispisu 5.2.

Ispis 5.2: Generiranje rasporeda, pokušaj 2 - primjer u programskom jeziku Java.

```
1  public static void main(String[] args) {
2      int[][] acceptableTerms = new int[][] {
3              {1},
4              {0,1,2,3,4},
5              {0,2},
6              {0,3,4},
7              {0,1,2}
8      };
9      int numberOfCourses = acceptableTerms.length;
10     int numberOfTerms = 5;
11
12     Random rand = new Random();
13     int[] courseTerm = new int[numberOfCourses];
14     int[] terms = new int[numberOfTerms];
15     int numberOfAttempts = 100000;
16
17     int failuresCount = 0;
18     for(int i = 0; i < numberOfAttempts; i++) {
19         // Za sve kolegije postavi termine na nevaljale
20         for(int k = 0; k < numberOfCourses; k++) {
21             courseTerm[k] = -1;
22         }
23         // Pripremi polje s popisom svih termina
24         for(int t = 0; t < numberOfTerms; t++) {
25             terms[t] = t;
26         }
27         for(int k = 0; k < numberOfCourses; k++) {
28             // Koliko je termina preostalo za biranje?
29             int left = numberOfTerms - k;
30             // Sto je odabrano?
31             int picked;
32             if(left==1) {
33                 picked = terms[0];
34             } else {
35                 int pos = rand.nextInt(left);
36                 int tmp = terms[pos];
37                 terms[pos] = terms[left-1];
38                 terms[left-1] = tmp;
39                 picked = tmp;
40             }
41             courseTerm[k] = picked;
42         }
43         if(!check(courseTerm, acceptableTerms)) {
44             failuresCount++;
45         }
46     }
47     System.out.println("Uspješnost stvaranja: "+
```

```
48          ((double)(numberOfAttempts-failuresCount)/(double)numberOfAttempts*100)+
49          "%, broj uspjeha: "+(numberOfAttempts-failuresCount));
50  }
```

Ideja je jednostavna: koristit ćemo pomoćno polje terms koje ćemo inicijalizirati s popisom svih mogućih termina. Pri tome ćemo osigurati da uvijek znamo koliko je termina ostalo a da nisu odabrani. Također, osigurat ćemo da su takvi termini uvijek smješteni na početku polja. Jednom kada smo tako organizirali kod, postupak biranja je vrlo jednostavan: posredstvom slučajnog mehanizma biramo jednu od prvih left pozicija na kojima se nalaze indeksi neodabranih termina. Pamtimo odabrani termin i njega prebacujemo pred kraj polja a termin koji bi time ispao s popisa vraćamo na njegovo mjesto. Konkretno, neka je terms={0,1,2,3,4}, i neka su svi termini neodabrani (pa je left=5). Biramo slučajno poziciju iz intervala [0,4] – pretpostavimo da smo odabrali poziciju pos=1. Prvi termin koji smo odabrali je terms[pos]=terms[1] što je 1. Sada indeks tog termina stavljamo na poziciju zadnjeg neodabranog termina a ono što je pisalo na toj poziciji upisujemo na odabranu poziciju; slijedi da je sadržaj tako modificiranog polja terms={0,4,2,3,1}. Ako trebamo dalje birati, u sljedeći krug ulazimo s left=4, te sljedeću poziciju biramo slučajno iz intervala [0,3] – pretpostavimo da smo opet odabrali poziciju pos=1. Termin koji je time izabran je terms[pos]=terms[1] što je 4. Opet mijenjamo sadrža polja na izabranoj poziciji (1) i zadnjoj neodabranoj (4), čime dobivamo novi sadržaj polja terms={0,3,2,4,1}. Postupak možemo ponavljati još tri puta, s obzirom da polje sadrži 5 elementa.

Opisana modifikacija rezultira povećanjem uspješnosti stvaranja valjanih rješenja s 0% na 3.32687%. Jest da je ovaj postotak vrlo mali, ali postupak barem ponekad stvara legalna rješenja i može se koristiti za stvaranje inicijalne populacije. U opisanom slučaju metoda za provjeru legalnosti rješenja ista ruši samo ako su pojedinim kolegijima dodijeljeni nedozvoljeni termini. Stoga je naš sljedeći korak pokušati to uzeti u obzir.

Nova modifikacija postupka stvaranja rasporeda prikazana je u ispisu 5.3.

Ispis 5.3: Generiranje rasporeda, pokušaj 3 - primjer u programskom jeziku Java.

```
1  public static void main(String[] args) {
2      int[][][] acceptableTerms = new int[][][] {
3              {1},
4              {0,1,2,3,4},
5              {0,2},
6              {0,3,4},
7              {0,1,2}
8      };
9      int numberOfCourses = acceptableTerms.length;
10     int numberOfTerms = 5;
11
12     Random rand = new Random();
13     int[] courseTerm = new int[numberOfCourses];
14     int[] terms = new int[numberOfTerms];
15     int numberOfAttempts = 100000;
16
17     // Pomocno polje za kontrolu odabranih termina
18     boolean[] termUsed = new boolean[numberOfTerms];
19
20     int failuresCount = 0;
21     for(int i = 0; i < numberOfAttempts; i++) {
22         // Za sve kolegije postavi termine na nevaljale
23         for(int k = 0; k < numberOfCourses; k++) {
24             courseTerm[k] = -1;
25         }
26         // Ocisti zastavice odabranih termina
27         for(int k = 0; k < termUsed.length; k++) {
28             termUsed[k] = false;
29         }
30         boolean failedAttempt = false;
31         for(int k = 0; k < numberOfCourses; k++) {
32             int courseIndex = k;
33             int numberOfValidTerms = 0;
34             // Pripremi polje s popisom valjanih preostalih termina
```

```
35              for(int t = 0; t < acceptableTerms[courseIndex].length; t++) {
36                  if(termUsed[acceptableTerms[courseIndex][t]]) continue;
37                  terms[numberOfValidTerms] = acceptableTerms[courseIndex][t];
38                  numberOfValidTerms++;
39              }
40              // Sada odaberi jedan od tih termina
41              if(numberOfValidTerms==0) {
42                  failedAttempt=true;
43                  break;
44              }
45              int picked = terms[rand.nextInt(numberOfValidTerms)];
46              courseTerm[courseIndex] = picked;
47              termUsed[picked] = true;
48          }
49          if(failedAttempt || !check(courseTerm, acceptableTerms)) {
50              failuresCount++;
51          }
52      }
53      System.out.println("Uspješnost stvaranja: "+
54          ((double)(numberOfAttempts-failuresCount)/(double)numberOfAttempts*100)+
55          "%, broj uspjeha: "+(numberOfAttempts-failuresCount));
56  }
```

Kod prikazan u ispisu 5.3 prilikom stvaranja rasporeda vodi evidenciju o terminima koji su već iskorišteni (kao polje zastavica termUsed čime je evidentiranje/provjera izvediva u sliženosti $o(1)$). Za svaki kolegij konzultira se popis dozvoljenih termina te se stvara popis koji sadrži sve dozvoljene termine koji već nisu iskorišteni. Potom se termin bira s tog popisa. Ovakvom izmjenom uspješnost generiranja valjanih rješenja raste čak za red veličine i penje se na 37.50281% što je već sasvim pristojna brojka.

Međutim, temeljem informacija koje imamo, možemo pokušati još malo popraviti ovaj postotak. Ono što je sada važno uočiti jest da se svi kolegiji međusobno natječu za termine, i da postoje kolegiji koje je lakše zadovoljiti i oni koje je teže zadovoljiti. Stoga je ideja sljedeća: pokušat ćemo utvrditi koliko je koji kolegij "zahtjevan" te potom kolegije poslożiti prema zahtjevnosti – kolegij koji postavlja najviše ograničenja doći će na prvo mjesto a kolegij koji postavlja najmanje ograničenja doći će na posljednje mjesto. Potom ćemo tu listu iskoristiti kako bismo pustili kolegije da termine biraju redosljedom kojim se nalaze na listi.

Jedan od načina kako generirati ovu listu jest napraviti pažljivu analizu zahtjeva svih kolegija, što u ovom slučaju i nije pretjerano komplicirano. Međutim, u praksi, zadatci koje ćemo rješavati imat će čitav niz ograničenja i međuovisnosti te direktna analiza neće biti jednostavno (ili uopće) provediva. Stoga ćemo opisati postupak koji će inkrementalno korigirati početnu pretpostavku o izgledu liste. Pogledajte malo pažljivije ispis 5.3 i to redak 41. Ako je uvjet zadovoljen, nastupila je situacija u kojoj za trenutni kolegij više nije ostao niti jedan valjani termin – možemo zaključiti da je izgradnja rasporeda "pala" upravo na trenutnom kolegiju.

Modifikacija prikazana u ispisu 5.4 uvodi dva nova polja. Polje failures predstavlja brojače neuspjeha za svaki kolegij. Element na poziciji i je broj koji govori koliko je puta izgradnja rasporeda pukla upravo na kolegiju i. Drugo polje koje je uvedeno je courseOrder – to je polje koje čuva indekse kolegija i sortirano je prema broju pucanja kolegija. Primjerice, neka je sadržaj polja failures={17,3,21,5,8} (izgradnja je puknula na kolegiju K_0 17 puta, na K_1 3 puta, itd). Tada će vrijediti courseOrder={2,0,4,3,1}.

Inicijalno, krenut ćemo s pretpostavkom da je broj pucanja za sve kolegije jednak 0, te da izgradnja kreće od nultog pa do zadnjeg kolegija. U našem primjeru vrijedit će: failures={0,0,0,0,0} te courseOrder={0,1,2,3,4}. Uočimo: lista kolegija inicijalno jest sortirana prema broju pucanja, s obzirom da je taj broj jedna 0 za sve kolegije. Kada prilikom izgradnje rasporeda dođe do pucanja na i-tom kolegiju, odustajemo od tog pokušaja i za i-ti kolegij povećavamo brojač pucanja za jedan. Time je moguće da poredak kolegija po zahtjevnosti više nije očuvan pa je potrebno provesti sortiranje kolegija. Međutim, to je u ovom slučaju moguće provesti u linearnoj složenosti – ako postoji pogreška u redosljedu, ona je prisutna samo za i-ti kolegij na kojem je postupak izgradnje puknuo. Stoga i-ti kolegij treba pomicati prema vrhu liste tako dugo dok je brojač neuspjeha tog kolegija veći od brojača neuspjeha kolegija koji se nalazi neposredno ispred njega.

Ispis 5.4: Generiranje rasporeda, pokušaj 4 - primjer u programskom jeziku Java.

```
1  public static void main(String[] args) {
2      int[][] acceptableTerms = new int[][] {
3              {1},
4              {0,1,2,3,4},
5              {0,2},
6              {0,3,4},
7              {0,1,2}
8      };
9      int numberOfCourses = acceptableTerms.length;
10     int numberOfTerms = 5;
11
12     long[] failures = new long[numberOfCourses];
13     for(int i = 0; i < numberOfCourses; i++) {
14         failures[i] = 0;
15     }
16
17     int[] courseOrder = new int[numberOfCourses];
18     for(int i = 0; i < numberOfCourses; i++) {
19         courseOrder[i] = i;
20     }
21
22     Random rand = new Random();
23     int[] courseTerm = new int[numberOfCourses];
24     int[] terms = new int[numberOfTerms];
25     int numberOfAttempts = 100000;
26
27     // Pomocno polje za kontrolu odabranih termina
28     boolean[] termUsed = new boolean[numberOfTerms];
29
30     int failuresCount = 0;
31     for(int i = 0; i < numberOfAttempts; i++) {
32         // Za sve kolegije postavi termine na nevaljale
33         for(int k = 0; k < numberOfCourses; k++) {
34             courseTerm[k] = -1;
35         }
36         // Ocisti zastavice odabranih termina
37         for(int k = 0; k < termUsed.length; k++) {
38             termUsed[k] = false;
39         }
40         int failedCourse = -1;
41         for(int k = 0; k < numberOfCourses; k++) {
42             int courseIndex = courseOrder[k];
43             int numberOfValidTerms = 0;
44             // Pripremi polje s popisom valjanih preostalih termina
45             for(int t = 0; t < acceptableTerms[courseIndex].length; t++) {
46                 if(termUsed[acceptableTerms[courseIndex][t]]) continue;
47                 terms[numberOfValidTerms] = acceptableTerms[courseIndex][t];
48                 numberOfValidTerms++;
49             }
50             // Sada odaberi jedan od tih termina
51             if(numberOfValidTerms==0) {
52                 failedCourse=courseIndex;
53                 break;
54             }
55             int picked = terms[rand.nextInt(numberOfValidTerms)];
56             courseTerm[courseIndex] = picked;
57             termUsed[picked] = true;
58         }
59         if(failedCourse==-1) {
60             failedCourse = check(courseTerm, acceptableTerms);
61         }
62         if(failedCourse!=-1) {
63             failuresCount++;
64             failures[failedCourse]++;
65             // Pronadi poziciju na kojoj se nalazi kolegij na kojem je puklo
66             int pos = 0;
```

```
67                  while(courseOrder[pos]!=failedCourse) {
68                      pos++;
69                  }
70                  while(pos>0 && failures[courseOrder[pos]]>failures[courseOrder[pos-1]]) {
71                      int tmp = courseOrder[pos];
72                      courseOrder[pos] = courseOrder[pos-1];
73                      courseOrder[pos-1] = tmp;
74                      pos--;
75                  }
76              }
77          }
78          System.out.println("Uspješnost stvaranja: "+
79              ((double)(numberOfAttempts-failuresCount)/(double)numberOfAttempts*100)+
80              "%, broj uspjeha: "+(numberOfAttempts-failuresCount));
81          System.out.println("Poredak kolegija na kraju: "+Arrays.toString(courseOrder)+
82              ", failures="+Arrays.toString(failures));
83      }
```

Ovom modifikacijom konačna uspješnost generiranja rasporeda penje se čak na 99.996% na uzorku od 100000 pokušaja; dapače, većina neuspjeha se javlja na samom početku što znači da što više pokušaja radimo, u prosjeku ćemo biti uspješniji. Zanimljivo je pogledati konačnu listu courseOrder koja je izgrađena na kraju: {0, 4, 2, 1, 3} a pripadni brojač neuspjeha je failures={2, 0, 1, 0, 1}. Vidimo da je kolegij K_0 najzahtjevniji, što i jest u skladu s činjenicom da mu od svih termina odgovara samo jedan; slijedi ga kolegij K_4 pa kolegij K_2 koji oba efektivno imaju po dva moguća termina (iako K_4 navodi tri termina, jedan od njih mu uvijek uzme kolegij K_0 koji prvi bira), itd.

Spomenimo još i da se, osim ovakvog deterministčkog poretka biranja, brojači pucanja mogu koristiti i kako bi se kolegijima na vjerojatnosni način pridijelio poredak, pri čemu veću šansu da prije biraju imaju oni kolegiji koji su imali više neuspjeha.

Bibliografija

Dio I

Jednokriterijska optimizacija

Poglavlje 6

Algoritam simuliranog kaljenja

6.1 Uvod

Inspiracija za razvoj algoritma simuliranog kaljenja je postupak kaljenja metala koji se koristi u metalurgiji i kojim se postižu bolja mehanička svojstva metala (poput promjene tvrdoće metala te elastičnosti). Prilikom postupka kaljenja, metal se zagrijava preko kritične temperature koja se neko vrijeme održava i potom se postupno hladi (sporo hlađenje posebno je važno kod materijala poput željeza). Prilikom ovog postupka materijal se najprije dovodi do točke vrlo visoke energije pri kojoj se povećava gibljivost atoma koji se mogu kretati kroz materijal, nakon čega se energija postupno spušta. Zahvaljujući ovom postupku hlađenja i povećanoj gibljivosti atoma, u metalu će se postupno stvoriti pravilne kristalne strukture koje nemaju deformacija i koje metal dovode do stanja minimalne energije. Ako bi se metal hladio prebrzo, nastale bi nepravilne kristalne strukture, pojavile bi se deformacije i naprezanje; sustav bi ostao u višem energetskom stanju – u tom stanju metal bi iskazivao svojstva veće tvrdoće i manje elastičnosti što bi lakše dovelo do kidanja odnosno oštećivanja materijala. Dobro proveden postupak kaljenja povećava elastičnost metala, smanjuje mu tvrdoću i unutarnja naprezanja i stvara pravilnu kristalnu strukturu.

Prilikom hlađenja, metal neprestano prelazi iz jednog energetskog stanja u drugo. Pri tome su ti prijelazi vođeni zakonima termodinamike koji kažu da je pri temperaturi t vjerojatnost prelaska iz energetskog stanja E_1 u više energetsko stanje E_2 (čime dolazi do povećanja energije $\Delta E = E_2 - E_1 > 0$) određena vjerojatnošću:

$$P(\Delta E) = \exp(\frac{-\Delta E}{k \cdot t}) \tag{6.1}$$

gdje je k Boltzmannova konstanta ($1.3806503 \cdot 10^{-23} \ \frac{m^2 kg}{s^2 K}$); vjerojatnost prelaska u niže energetsko stanje (tj. prijelaz za koji je $\Delta E < 0$) je uvijek 1. Prilikom hlađenja metala, metal prolazi kroz niz stanja (i različitih konfiguracija atoma) koja u konačnici prevode metal iz stanja visoke energije u stanje vrlo niske energije. Na tom putu, međutim, često se događa da se energija monotono ne pada – javljaju se konfiguracije koje se jednostavno ne mogu presložiti u konfiguracije još niže energije već je potrebno najprije prijeći u međustanje više energije kako bi se nakon toga moglo prijeći u stanje još niže energije. Kod postupnog hlađenja metala to nije problem jer su takvi prijelazi mogući (što je jasno iz izraza (6.1)). Ono što je važno jest temperaturu smanjivati dovoljno polagano kako bi materijal imao dovoljno vremena proći kroz takva stanja i ne ostati zaglavljen.

Iz izraza (6.1) vidljivo je da temperatura direktno utječe na vjerojatnost prelaska u više energetsko stanje. Da bismo razumjeli kako, promotrimo dva ekstrema: $t \to \infty$ i $t \to 0$. Za ograničeni iznos

$\Delta E > 0$ vrijedi:

$$\begin{aligned}\lim_{t\to\infty} \frac{1}{\exp(\frac{\Delta E}{k\cdot t})} &= \frac{1}{\exp(\frac{\Delta E}{\infty})} \\ &= \frac{1}{\exp(0)} \\ &= \frac{1}{1} \\ &= 1\end{aligned}$$

dok je u drugom slučaju:

$$\begin{aligned}\lim_{t\to 0} \frac{1}{\exp(\frac{\Delta E}{k\cdot t})} &= \frac{1}{\exp(\frac{\Delta E}{0})} \\ &= \frac{1}{\exp(\infty)} \\ &= \frac{1}{\infty} \\ &= 0.\end{aligned}$$

Dakle, uz beskonačno visoku temperaturu sustav će s vjerojatnošću 1 prelaziti iz bilo kojeg stanja u bilo koje drugo više energetsko stanje (neovisno o inkrementu energije, uz pretpostavku da je on konačan). Pri temperaturi 0 situacija je skroz obrnuta: sustav će s vjerojatnošću 0 prelaziti u viša energetska stanja. Na temperaturi 0 sustav je zamrznut – jedini mogući prijelazi su prijelazi u niža energetska stanja; vjerojatnost prelaska u više energetsko stanja ma koliko malog inkrementa energije je 0.

U praksi, kaljenje metala nikada ne postiže temperaturu ∞ – no to nije niti potrebno; za svaki metal postoji kritična temperatura koja je dovoljno velika da dopusti i one najveće promjene energetskog stanja koje se u metalu mogu dogoditi. Od te točke temperatura se postupno spušta čime vjerojatnost većih promjena počinje padati – sustav se polako stabilizira, spušta energiju i radi sve manje i manje promjene energetskih stanja. Pri niskim temperaturama vjerojatnost svih promjena osim onih s vrlo malim inkrementom energije je praktički zanemariva. Konačno, spuštanjem temperature na nulu vjerojatnost bilo kakvog inkrementa pada na nulu. I opet, u praksi se temperatura ne spušta na nulu jer za time nema potrebe – beskonačno male promjene u energetskim razinama ionako nisu moguće i u jednom trenutku će sve promjene stati, čak i uz $t > 0$.

6.2 Primjena na optimizacijske probleme

Opisani postupak kaljenja metala moguće je direktno primijeniti i na optimizacijske probleme – kako na diskretne (tj. kombinatoričke), tako i na kontinuirane. Uočimo da vrijede odnosi prikazani u sljedećoj tablici [Dowsland, 1995].

Kaljenje metala	Kombinatorička optimizacija
Moguća stanja sustava	Prihvatljiva rješenja
Energija sustava	Funkcija kazne
Promjena stanja sustava	Prelazak u susjedno rješenje
Temperatura	Parametar koji simulira temperaturu
Zamrznuto stanje	Optimum (lokalni ili globalni)

Primjetite da se u tablici kao ekvivalent energiji sustava nalazi funkcija kazne, a ne funkcija dobrote. Naime, kako je u prirodi proces kaljenja proces koji prevodi sustav iz viših stanja viših energija u stanja nižih energija, njegov pandan u optimizacijskim procesima ne može biti funkcija dobrote (jer želimo postići što veću dobrotu). Stoga se kao alternativa koristi funkcija kazne – funkcija suprotna funkciji dobrota koja govori koliko je neko rješenje loše – što je iznos funkcije kazne veći, rješenje je lošije.

Primjetite da je to nužno jer je proces kaljenja, kako se događa u prirodi, minimizacijski proces. Ovo se dade vrlo jednostavno zaobići na način da se dogovorimo da oznaka ΔE označava mjeru pogoršanja rješenja, odnosno da pozitivni iznos ΔE govori koliko je novo stanje/rješenje lošije od postojećeg. U tom slučaju, ako se radi minimizacija, ΔE ćemo računati kao $\Delta E = E_2 - E_1$; ako je to pozitivno, znači da je $E_2 > E_1$ što je doista pogoršanje. Ako se radi maksimizacija (što bi bio slučaj ako bismo ili rješavali problem maksimizacije ili rješavali problem za koji imamo definiranu funkciju dobrote čime je problem automatski problem maksimizacije), dovoljno je ΔE računati kao $\Delta E = -(E_2 - E_1) = E_1 - E_2$; ako je tako definiran ΔE pozitivan, to znači da je nastupio pad u funkciji dobrote što je opet mjera pogoršanja rješenja.

Koristeći preslikavanja dana u prethodnoj tablici svaki se problem kombinatoričke optimizacije može preslikati u problem prikladan za rješavanje algoritmom simuliranog kaljenja [Kirkpatrick et al., 1983, Cerny, 1985].

Uvedimo sada nekoliko oznaka. Pretpostavimo da rješavamo problem kombinatoričke optimizacije, i to minimizacijski problem. Neka je s Ω označen prostor rješenja. Neka je s $f : \Omega \to \mathcal{R}$ označena funkcija cilja definirana nad prostorom rješenja. Zadatak je pronaći globalni minimum, odnosno rješenje $\omega^* \in \Omega$ takvo da $\forall \omega \in \Omega,\ f(\omega) \geq f(\omega^*)$. Definirajmo još da je $N(\omega)$ *funkcija susjedstva* rješenja $\omega \in \Omega$. Funkcija susjedstva svakom rješenju $\omega \in \Omega$ pridružuje skup rješenja iz Ω koja su direktno dohvatljiva iz rješenja ω, tj. skup rješenja u koja se može direktno prijeći iz rješenja ω. Da bi algoritam simuliranog kaljenja imao garanciju pronalaska globalnog optimuma, nužno mora vrijediti sljedeće: ako rješenje $\omega^{'}$ pripada susjedstvu rješenja ω, tj. ako je $\omega^{'} \in N(\omega)$ što znači da se iz ω u jednom koraku može prijeći u rješenje $\omega^{'}$, tada rješenja ω također mora biti dohvatljivo iz rješenja $\omega^{'}$. Striktniji zahtjev bi glasio ovako: ako se iz ω u jednom koraku može prijeći u $\omega^{'}$, tada se iz $\omega^{'}$ u jednom koraku mora moći prijeći natrag u ω. To međutim nije nužno – dovoljno je da se iz $\omega^{'}$ kroz jedan ili više koraka može prijeći natrag u ω. O tome svakako treba voditi računa prilikom definiranja funkcije susjedstva odnosno prilikom izrade operatora koji će za zadano rješenje ω generirati nekog (slučajnog) susjeda $\omega^{'}$.

Algoritam simuliranog kaljenja započinje sa slučajno generiranim rješenjem $\omega \in \Omega$. Potom se iz skupa $N(\omega)$ odabire jedno rješenje $\omega^{'}$ koje postaje novi kandidat za trenutno rješenje. Taj se kandidat prihvaća u skladu s pravilom definiranom u [Metropolis et al., 1953]:

$$P(\text{prihvati } \omega^{'} \text{ kao novo rješenje}) = \begin{cases} \exp(-\frac{f(\omega^{'}) - f(\omega)}{t_k}), & \text{ako je } f(\omega^{'}) - f(\omega) > 0 \text{ te} \\ 1, & \text{ako je } f(\omega^{'}) - f(\omega) \leq 0, \end{cases} \tag{6.2}$$

gdje je s t_k označena temperatura u koraku k. Kako bi algoritam imao garanciju konvergencije ka globalnom optimumu, pretpostavlja se da će vrijediti:

$$t_k > 0 \quad \forall k \quad \wedge \quad \lim_{k \to \infty} t_k = 0.$$

Izvedba algoritma simuliranog kaljenja prikazana je pseudokodom 6.1.

Uočimo postojanje dviju petlji: vanjska petlja brine se za promjenu temperature u skladu s odabranim planom hlađenja. Plan hlađenja definira na koji se način temperatura mijenja kroz vrijeme; u literaturi je moguće pronaći čitav niz različitih planova kao i analize njihova utjecaja na rad algoritma. Unutarnja petlja pri fiksnoj temperaturi provodi određen broj koraka algoritma. Koliko će se koraka provesti pri pojedinim temperaturama definirano je planom M_k. Da bi opisani algoritam bio primjenjiv na problem maksimizacije, dovoljno je promijeniti predznak desne strane izraza kojim se računa $\Delta_{\omega,\omega'}$.

Spomenimo da je konvergencija algoritma simuliranog kaljenja i matematički dokazana – uz ograde koje nažalost u praksi nisu prihvatljive, ali ipak omogućavaju proučavanje ponašanja ovog algoritma; pretpostavke uz koje je konvergencija dokazana (uz prethodno spomenuti nužni uvjet o reverzibilnosti prijelaza) jest da se unutarnja petlja i vanjska petlja ponavljaju beskonačno mnogo puta.

6.3 Vjerojatnost prihvaćanja rješenja

Izračun eksponencijalne funkcije računski je vrlo skup. Stoga je dosta istraživanja provedeno kako bi se utvrdilo je li nužno da se kao vjerojatnost prihvaćanja rješenja koristi izraz (6.2). Tako u radu

Pseudokod 6.1 Algoritam simuliranog kaljenja.

Generiraj početno rješenje $\omega \in \Omega$
Postavi brojač za promjenu temperature na $k = 0$
Odaberi plan hlađenja t_k
Odaberi početnu temperaturu $t_0 \geq 0$
Odredi plan za M_k – broj ponavljanja petlje pri temperaturi t_k
Ponavljaj dok nije zadovoljen uvjet zaustavljanja
    Ponavljaj za m je 1 do M_k
        Generiraj susjedno rješenje $\omega' \in N(\omega)$
        Izračunaj $\Delta_{\omega,\omega'} = f(\omega') - f(\omega)$
        Ako je $\Delta_{\omega,\omega'} \leq 0$, prihvati ω', tj. postavi $\omega \leftarrow \omega'$
        Inače ako je $\Delta_{\omega,\omega'} > 0$, postavi $\omega \leftarrow \omega'$ s vjerojatnošću $\exp(\frac{-\Delta_{\omega,\omega'}}{t_k})$
    Kraj ponavljanja
Kraj ponavljanja
Vrati ω kao rješenje

[Ogbu and Smith, 1990] autori predlažu uporabu izraza kod kojeg se vjerojatnost prihvaćanja smanjuje geometrijski i koja ne ovisi o iznosu pogoršanja rješenja već samo o koraku k u kojem se računa:

$$P(\text{prihvati } \omega' \text{ kao novo rješenje}) = \begin{cases} a_1 x^{k-1} & \text{ako je } f(\omega') > f(\omega), \\ 1 & \text{inače.} \end{cases}$$

Pri tome je a_1 početna vjerojatnost prihvaćanja lošijih rješenja, dok je $x \in (0, 1)$ faktor koji definira brzinu smanjivanja vjerojatnosti.

6.4 Planovi hlađenja

Kod algoritma simuliranog kaljenja temperatura se mijenja od početne visoke pa sve do, u limesu, vrijednosti 0. Kod implementacije algoritma na računalu, s obzirom da se vanjska i unutarnja petlja ne ponavljaju beskonačno mnogo puta, loš plan hlađenja može rezultirati lošim pronađenim rješenjem. Pri tome uvijek postoji kompromis između brzine smanjivanja temperature i kvalitete dobivenih rješenja. Što se temperatura drastičnije smanjuje, algoritam će prije stići do niskih temperatura a time i stagnacije; pozitivni aspekt jest kratko vrijeme izvođenja a negativan potencijalno loša pronađena rješenja. Što je hlađenje sporije, algoritam će se duže izvoditi; međutim, pronađena rješenja će također biti bolja.

U praksi se koristi nekoliko različitih planova hlađenja koje ćemo navesti u nastavku.

6.4.1 Linearni plan hlađenja

Kod linearnog plana hlađenja temperatura se u svakom koraku umanjuje za fiksan iznos: $T \leftarrow T - \beta$. Stoga je u koraku k temperatura definirana izrazom:

$$T_k = T_0 - k \cdot \beta$$

gdje je T_0 početna temperatura; pretpostavlja se da u ovoj izvedbi k kreće od vrijednosti 0. Umanjivanje nema smisla nakon što temperatura dosegne vrijednost 0. Stoga, ako je unaprijed poznato da će se obaviti K iteracija vanjske petlje te da temperatura treba krenuti od vrijednosti T_0 i završiti na vrijednosti $T_K < T_0$, kao β se može uzeti vrijednost:

$$\beta = \frac{T_0 - T_K}{K - 1}.$$

6.4.2 Geometrijski plan hlađenja

Kod geometrijskog plana hlađenja temperatura se u svakom koraku umanjuje za fiksan faktor: $T \leftarrow T \cdot \alpha$. Stoga je u koraku k temperatura definirana izrazom:

$$T_k = \alpha^k \cdot T_0$$

gdje je T_0 početna temperatura; pretpostavlja se da u ovoj izvedbi k kreće od vrijednosti 0. α je faktor umanjenja trenutne temperature i vrijednosti mu se kreću u intervalu $(0, 1)$. U praksi se obično koriste vrijednosti iz intervala $[0.5, 0.99]$. Ako je unaprijed poznato da će se obaviti K iteracija vanjske petlje te da temperatura treba krenuti od vrijednosti T_0 i završiti na vrijednosti $T_K < T_0$, kao α se može uzeti vrijednost:

$$\alpha = \sqrt[K-1]{\frac{T_K}{T_0}} = \left(\frac{T_K}{T_0}\right)^{\frac{1}{K-1}}.$$

6.4.3 Logaritamski plan hlađenja

Kod logaritamskog plana hlađenja temperatura se u koraku k računa prema izrazu:

$$T_k = \frac{T_0}{\log k}.$$

U praksi se ovaj plan rijetko koristi jer je smanjivanje temperature vrlo sporo; međutim, iskorišten je u [Mitra et al., 1986] za dokazivanje konvergencije algoritma simuliranog kaljenja.

6.4.4 Plan vrlo sporog hlađenja

Kod plana vrlo sporog hlađenja osnovna ideja je osigurati da se temperatura mijenja izuzetno polako čime se otvara mogućnost izbacivanja unutarnje petlje, te se umjesto nje za svaku temperaturu napravi samo jedan korak algoritma. Kod ovog plana hlađenja u koraku $k{+}1$ temperatura je definirana izrazom:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}$$

gdje je T_k temperatura u koraku k a β kontrolni parametar.

Uz navedene planove u literaturi se mogu naći i druge izvedbe koje uključuju i nemonotone planove (koji temperaturu i spuštaju i dižu ali prema unaprijed zadanom planu) te adaptivne planove (koji ovisno o trenutnom stanju pretraživanja donose odluku što će se dalje događati s temperaturom).

6.5 Druge specifičnosti

Prvo ćega se trebamo prisjetiti jest posljedica *No-free-lunch* teorema: ovaj algoritam, kao i svi ostali, u prosjeku je jednako dobar kao i svi ostali. Kako bi algoritam simuliranog kaljenja dobro rješavao neki konkretan problem, potrebno je dobro podesiti parametre algoritma čime algoritam podešavamo problemu.

Do sada smo se osvrnuli na izraz za izračun vjerojatnosti prihvaćanja novog rješenja te na planove hlađenja. Spomenimo sada da i odabir početne temperature T_0 također može imati veliki utjecaj na kvalitetu pronađenih rješenja. Naime, ako se kao početna temperatura odabere previsoka temperatura, algoritam će puno vremena provoditi u nasumičnom skakanju kroz prostor rješenja jer će se sva rješenja, praktički neovisno o njihovoj kvaliteti, prihvaćati. Kako je vrijeme izvođenja algoritma ograničeno, to će pak značiti da će manje vremena ostati za smisleno pretraživanje koje bi moglo generirati kvalitetna rješenja.

S druge pak strane, ako se kao početna temperatura odabere premala temperatura, postoji mogućnost da će rješenje koje algoritam vrati biti jako ovisno o početno generiranom rješenju jer algoritam neće imati dovoljno "energije" kako bi iskočio iz tog podprostora i napravio širu pretragu – javlja se problem zapinjanja u lokalnim optimumima.

Stoga se u literaturi mogu pronaći različiti naputci kako odrediti početnu temperaturu. Primjerice, ako se rješava problem trgovačkog putnika, početna se temperatura može podesiti tako da se prihvaćaju čak i rješenja kod kojih se međusobno povezuju dva najudaljenija grada. Ako pak nemamo kvalitetnih informacija o problemu koji se rješava, autori u [Rayward-Smith et al., 1996] predlažu pristup u kojem se kreće s vrlo visokom temperaturom koja se potom rapidno spušta tako dugo dok se i dalje prihvaća više od 60% lošijih promjena stanja. Kada postotak prihvaćanja lošijih stanja padne na 60%, temperatura pri kojoj se se to dogodi uzima se kao početna temperatura i s njom se dalje ide u klasični algoritam simuliranog kaljenja.

U [Dowsland, 1995] predlaže se pristup koji vjernije oponaša kaljenje metala: algoritam se provodi u dva koraka. U prvom koraku se kreće od niske temperature koja je postupno povećava – emulira se zagrijavanje metala. Prilikom zagrijavanja prati se u kojem se postotku prihvaćaju loša rješenja. što se temperatura više podigne to će taj postotak postajati veći. Jednom kada se dođe do zadovoljavajućeg postotka, temperatura na kojoj se to dogodi postaje početna temperatura za provođenje drugog koraka algoritma, koji je upravo klasični algoritam simuliranog kaljenja.

6.6 Primjena na kombinatoričku optimizaciju

Ilustraciju kako algoritam simuliranog kaljenja primijeniti na kombinatoričku optimizaciju dat ćemo na primjeru trgovačkog putnika. Neka je zadano N gradova pri čemu trgovački putnik može putovati direktno iz svakog grada u svaki drugi grad.

- Kao prikaz ćemo koristiti permutacijski prikaz (vidi poglavlje 3). Gradove ćemo numerirati brojevima od 1 do N; jedna tura tada je jedna permutacija uređene n-torke $(1, \ldots, N)$.
- Defirat ćemo da funkcije susjedstva $N(\omega)$ generira sve ture koje se iz ture ω mogu dobiti mutacijom premještanjem ili pak jednostavnom mutacijom inverzije (obje su opisane u poglavlju 3). U praksi, za rješenje ω nećemo morati generirati sva susjedna rješenja već ćemo, kad će nam zatrebati jedno susjedno rješenje (za jedan korak algoritma) nasumice odabrati jednu od ove dvije mutacije i nasumice je primijeniti.
- Kod problema trgovačkog putnika prirodno je raditi s funkcijom kazne. Funkciju kazne definirat ćemo kao ukupnu duljinu ture za zadano rješenje.
- Kako koristimo funkciju kazne, algoritam simuliranog kaljenja primijenit ćemo za minimizaciju.
- Početnu temperaturu utvrdit ćemo kako je predloženo u [Press et al., 1992]. Generiramo jedno rješenje i napravimo nekoliko modifikacija rješenja kako bismo utvrdili kretanje ΔE. Biramo vrijednost T koja je bitno veća od najvećeg utvrđenog ΔE. Koristimo geometrijski plan hlađenja: počinjemo smanjivati temperaturu u svakom prolazu za faktor 0.9. Takvu temperaturu održavamo dok ne napravimo $100N$ koraka algoritma ili dok ih $10N$ ne bude uspješno (u smislu da generira bolje rješenje).

6.7 Primjena na optimizaciju nad kontinuiranom domenom

U ovom slučaju potrebno je definirati kako se algoritam koristi ako se traži minimum funkcije $f(\vec{x})$ gdje je $\vec{x}$ d-dimenzijski vektor definiran nad nekim podskupom od $\mathbb{R}^d$. Uočimo da problem možemo svesti na problem kombinatoričke optimizacije: za svaku komponentu vektora $\vec{x}$ koristimo binarni prikaz uz odabranu proceduru dekodiranja i zadani raspon pretraživanja. Kako se to svodi na prethodno opisanu kombinatoričku optimizaciju, ovdje ćemo pogledati drugačiju izvedbu.

- Kao rješenje ćemo koristiti vektor decimalnih brojeva.
- Vrijednost funkcije $f(\vec{x})$ koristi se kao funkcija kazne. Zadaća algoritma simuliranog kaljenja jest pronaći rješenje $\vec{x}$ koje minimizira zadanu funkciju kazne.

- Još je potrebno definirati način na koji se generira susjedno rješenje. Susjedno rješenje možemo definirati kao rješenje koje se dobije dodavanjem vektora $\Delta\vec{x}$, tj. $\vec{x}^{'} = \vec{x} + \Delta\vec{x}$. Pri tome se vrijednost $\Delta\vec{x}$ može generirati nasumično ili se može koristiti neki od heurističkih postupaka (primjerice, simpleks procedura).

- Možemo koristiti geometrijski plan hlađenja pri čemu početnu temperaturu određujemo slično kao kod prethodno opisane kombinatoričke optimizacije uzorkovanjem zadane funkcije.

6.8 Varijante algoritma

U nastavku ćemo opisati još dvije varijante algoritma koje su konceptualno vrlo slične algoritmu simuliranog kaljenja ali su računski dosta povoljnije. Algoritmi su definirani u radu [Wood and Downs, 1998] i izvedeni su iz *Demonskog algoritma* [Creutz, 1983]. Ideja demonskog algoritma opisanog u [Creutz, 1983] jest modelirati sustav u kojem vrijedi zakon očuvanja energije, za što se brine demon koji je uparen sa sustavom. Inicijalno, demonu se daje određena količina energije (označimo je s D). Generira se početno rješenje ω. U tom trenutku ukupna energija sustava je $C = f(\omega)+D$ što se tijekom simulacije ne mijenja. Za svako novogenerirano rješenje $\omega^{'}$ računa se ΔE. Ako je $\Delta E < 0$, sustav prelazi u novo stanje i time gubi energiju iznosa $|\Delta E|$; stoga se upravo ta količina energije dodaje demonu. Time je količina energije u sustavu jednaka $(f(\omega) - |\Delta E|) + (D + |\Delta E|) = C$. U slučaju da je $\Delta E > 0$, odnosno da je novo rješenje takvo da bi povećalo energiju sustava, ono se prihvaća samo ako demon može isporučiti toliku energiju; u tom slučaju energija sustava raste ali energija demona pada. Ako demon nema dovoljno energije za kompenzaciju povećanja energije sustava, novo rješenje se ne prihvaća.

Opisani algoritam demona direktno ne obavlja minimizaciju funkcije. Autori u radu [Wood and Downs, 1998] modificiraju opisani postupak kako bi se postigao optimizacijski efekt. Prvi algoritam koji ćemo opisani je *algoritam ograničenog demona*, prikazan pseudokodom 6.2.

Pseudokod 6.2 Algoritam ograničenog demona.

1. Odaberi početno rješenje ω.
2. Odaberi početnu energiju demona $D = D_0 > 0$.
3. Ponavljaj dok nije zadovoljen uvjet zaustavljanja:
   (a) Generiraj susjedno rješenje $\omega^{'} \in N(\omega)$.
   (b) Izračunaj promjenu energije $\Delta E = f(\omega^{'}) - f(\omega)$.
   (c) Ako je $\Delta E \leq D$ prihvati novo rješenje: $\omega \leftarrow \omega^{'}$ i korigiraj energiju demona $D \leftarrow D - \Delta E$.
   (d) U suprotnom odbaci rješenje $\omega^{'}$.
   (e) Ako je $D > D_0$, korigiraj $D \leftarrow D_0$.

U ovoj inačici algoritma više se ne inzistira na konstantnosti očuvanja ukupne energije već se postavlja gornja granica na maksimalnu količinu energije koju demon može imati i ukupna energija čuva se samo kada to ne premašuje količinu energije koju demon može čuvati. Jednostavnom analizom postaje jasno da će kroz vrijeme ukupna količina energije polako padati. Kako je energija demona ograničena, to znači da će energija sustava morati sve više i više padati – čime će sustav prelaziti u sve kvalitetnija i kvalitetnija rješenja.

Velika prednost ovakvog algoritma jest što ne treba generator slučajnih brojeva (treba ga za generiranje susjednih rješenja, ali ne za donošenje odluka o prihvaćanju rješenja) te što ne koristi računski skupe funkcije (poput eksponencijalne funkcije).

Inačica algoritma poznata pod nazivom *Algoritam kaljenog demona* prikazana je pseudokodom 6.3.

Jedina razlika između algoritma ograničenog demona i algoritma kaljenog demona jest u retku 3(e) gdje se kod algoritma kaljenog demona ukupna količina energije koju demon ima u određenim

Pseudokod 6.3 Algoritam kaljenog demona.

1. Odaberi početno rješenje ω.
2. Odaberi početnu energiju demona $D = D_0 > 0$.
3. Ponavljaj dok nije zadovoljen uvjet zaustavljanja:
   (a) Generiraj susjedno rješenje $\omega' \in N(\omega)$.
   (b) Izračunaj promjenu energije $\Delta E = f(\omega') - f(\omega)$.
   (c) Ako je $\Delta E \leq D$ prihvati novo rješenje: $\omega \leftarrow \omega'$ i korigiraj energiju demona $D \leftarrow D - \Delta E$.
   (d) U suprotnom odbaci rješenje ω'.
   (e) Ako je postignut ekvilibrij, umanji energiju demona prema planu; npr. $D \leftarrow \alpha \cdot D$.

trenutcima umanjuje geometrijski (kada će se to točno događati, ovisi o odabranom planu, što odgovara smanjivanju temperature kod algoritma simuliranog kaljenja).

Bibliografija

V. Cerny. A thermodynamical approach to the travelling salesman problem; an efficient simulation algorithm. *Journal of Optimization Theory and Application*, (45):41–55, 1985.

M. Creutz. Microcanonical monte carlo simulation. *Physical Review Letters*, 50(19):1411–1414, 1983.

K. Dowsland. Simulated annealing. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, pages 41–49, USA, 1995. McGraw-Hill.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 (4598):671–680, 1983. doi: 10.1126/science.220.4598.671.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli. Convergence and finite time behavior of simulated annealing. *Advances in Applied Probability*, 18:747–771, 1986.

F. A. Ogbu and D. K. Smith. The application of the simulated annealing algorithm to the solution of the n/m/cmax flowshop problem. *Computers & OR*, 17(3):243–253, 1990.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C. The art of scientific computing.* Cambridge: University Press, second edition, 1992.

V. Rayward-Smith, I. Osman, C. Reeves, and G. Smith. *Modern Heuristic Search Methods.* John Wiley and Sons, 1996.

I. A. Wood and T. Downs. Fast optimization by demon algorithms. In T. Downs, M. Frean, and M. Gallagher, editors, *Australian Conference on Neural Networks 1998*, pages 245–249, University of Queensland, Brisbane, QLD, Australia, 1998. University of Queensland.

Poglavlje 7

Genetski algoritam

L. J. Fogel, A. J. Owens i M. J. Walsh stvorili su 1966. evolucijsko programiranje [Fogel et al., 1966], I. Rechenberg 1973. i H.-P. Schwefel 1975. evolucijske strategije [Rechenberg, 1973, Schwefel, 1975], a H. J. Holland 1975. genetski algoritam [Holland, 1975]. Paralelno tome, tekao je i razvoj genetskog programiranja koji je 1992. popularizirao J. R. Koza [Koza, 1992]. Inspiracija za razvoj došla je proučavanjem Darwinove teorije o postanku vrsta [Darwin, 1859].

Darwin teoriju razvoja vrsta temelji na 5 postavki:

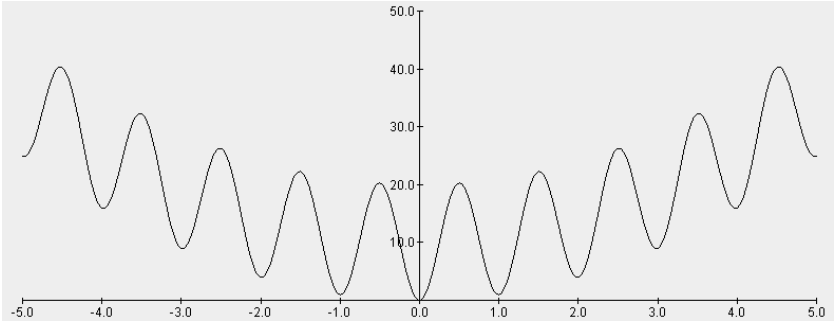
1. plodnost vrsta – potomaka uvijek ima više no što je potrebno,
2. veličina populacije je približno stalna,
3. količina hrane je ograničena,
4. kod vrsta koje se seksualno razmnožavaju, nema identičnih jedinki već postoje varijacije te
5. najveći dio varijacija prenosi se nasljeđem.

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 stvarati potomke. Pri tome su djeca roditelja u velikoj mjeri određena upravo 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, i možemo ih opisati na sljedeći način. Neka je zadana funkcija $f(\vec{x})$ gdje je $x = (x_1, x_2, \ldots, x_n)$. Potrebno je pronaći $\vec{x}^*$ koji maksimizira (ili minimizira) funkciju f. Pogledajmo ovo na primjeru funkcije jedne varijable (slika 7.1) koja je definirana izrazom:

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

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



Slika 7.1: Višemodalna funkcija jedne varijable.

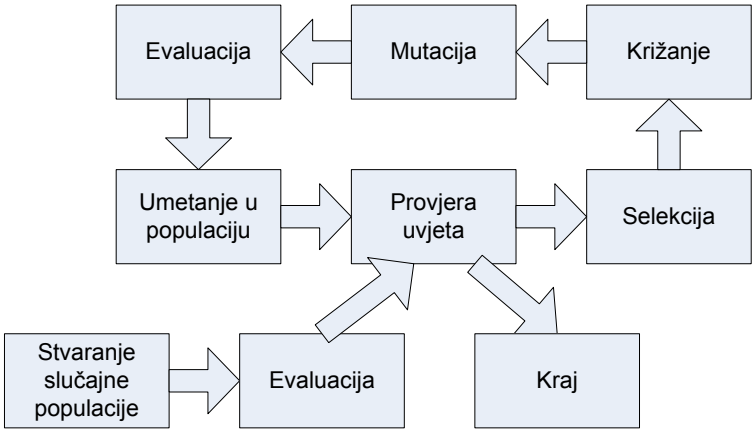
Kako genetski algoritam radi? Postoji puno različitih izvedbi a generalna ideja je sljedeća. Algoritam radi s *populacijom* jedinki. Svaka jedinka jedno je moguće rješenje zadanog problema. U našem primjeru svaka jedinka odgovara jednom vektoru $\vec{x}$). U kontekstu genetskog algoritma *jedinke* još nazivamo i *kromosomima* – to su sinonimi. Svakoj jedinki možemo odrediti njezinu *dobrotu* (engl. *fitness*). Dobrota jedinke određuje se tako da se izračuna vrijednost funkcije u točki koju jedinka predstavlja ($\vec{x}$). Uz pretpostavku da rješavamo minimizacijski problem, što je vrijednost funkcije veća, to je promatrana jedinka lošija. Operatorom *selekcije* (engl. *selection*) biraju se iz populacije jedinke koje će postati 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.

7.1 Vrste genetskog algoritma

Genetski algoritmi dijele se na *sekvencijske* i *paralelne*. Paralelni genetski algoritmi omogućavaju rad u višedretvenim i raspodijeljenim okruženjima, i njih ovaj tren nećemo razmatrati. Sekvencijski genetski algoritam možemo podijeliti na dvije tipične izvedbe [Alba and Dorronsoro, 2008]: *eliminacijski genetski algoritam* (engl. *steady-state genetic algorithm*) te *generacijski genetski algoritam* (engl. *generational genetic algorithm*). Oba su opisana u nastavku.

7.1.1 Eliminacijski genetski algoritam

Ova vrsta genetskog algoritma prikazana je na slici 7.2. U svakom koraku algoritma (može se još koristiti i termin *generacija*), iz čitave se populacije odabiru dva roditelja nad kojima se izvodi križanje, čime nastaje novo dijete koje se još i mutira. Na kraju se dijete ili ubacuje u populaciju ili odbacuje. Ako se ubacuje u populaciju, te kako veličina populacije mora biti stalna, ovo ubacivanje izvodi se tako da dijete zamijeni neku jedinku iz populacije (primjerice, najgoru).



Slika 7.2: Eliminacijski genetski algoritam.

Izvedba ove vrste genetskog algoritma dana je pseudokodom 7.1. Selekcija svakog od roditelja može se obaviti nekim od operatora selekcije; primjerice, proporcionalnom selekcijom ili pak turnirskom selekcijom (opis selekcija već je dan u prethodnim poglavljima). Isto tako, odabir jedinke koja će u populaciji biti zamijenjena nastalim djetetom također se može obaviti nekim od operatora selekcije, pri čemu se operator tako prilagodi da veću šansu odabira daje lošijim jedinkama; alternativa je za izbacivanje odabrati trenutno najgoru jedinku u populaciji, trenutno najstariju jedinku u populaciji ili pak nasumice odabrati neku jedinku. Konačno, moguće je odabrati uvjet pod kojim će dijete doista i zamijeniti odabranu jedinku: to može, primjerice, biti samo ako je dijete bolje od odabrane jedinke, ili pak može biti uvijek (bez uvjeta).

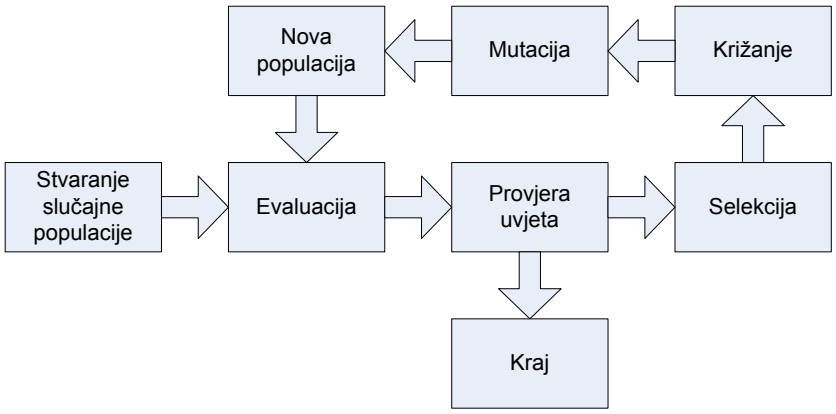
Uočimo da kod ove vrste genetskog algoritma nastalo dijete već u sljedećem trenutku može postati roditelj koji će se križati sa svojim pretcima i dalje stvarati potomke.

Pseudokod 7.1 Pseudokod eliminacijskog genetskog algoritma.

```
P = stvori_početnu_populaciju(VEL_POP)
evaluiraj(P)
ponavljaj_dok_nije_kraj
  odaberi R1 i R2 iz P
  D = križaj(R1, R2)
  mutiraj D
  evaluiraj D
  odaberi jednku iz populacije koja će eventualno biti zamijenjena stvorenim djetetom
  odluči hoće li dijete zamijeniti odabranu jedinku i po potrebi provedi zamjenu
kraj
```

7.1.2 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 7.3.



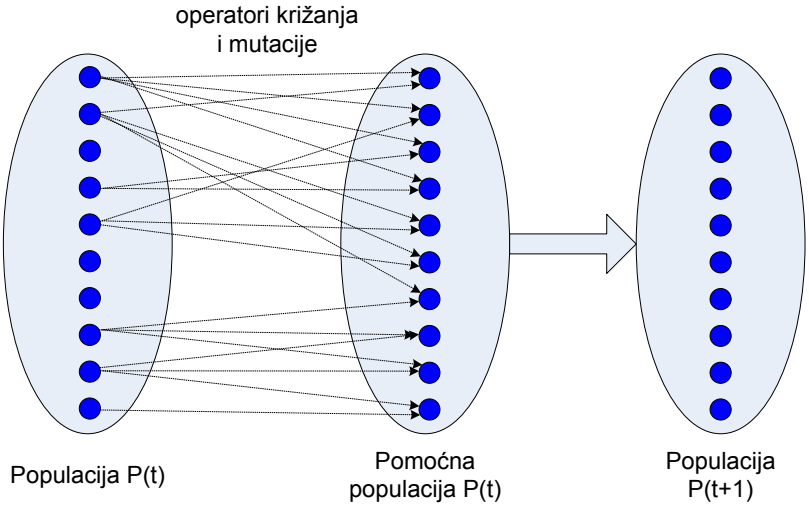
Slika 7.3: Generacijski genetski algoritam.

Generacijski genetski algoritam iz trenutne populacije operatorima selekcije, križanja i mutacije gradi novu pomoćnu populaciju sastavljenu isključivo iz djece (slika 7.4). Izvedba ove vrste genetskog algoritma dana je pseudokodom 7.2, dok pseudokod 7.3 pokazuje modificiranu verziju koja garantira elitizam.

Onog trenutka kada se izgradi nova populacija čija je veličina jednaka staroj, stara populacija roditelja se odbacuje, a novoizgrađena populacija djece postaje trenutna populacija. Moguće su i izvedbe kod kojih se gradi populacija djece koja je veća od populacije roditelja (primjerice, k puta veća) i gdje se na kraju iz te populacije probere potreban broj jedinki koje će postati roditelji u sljedećoj generaciji (višak djece se odbaci).

Uočimo da se direktnom implementacijom prikazanog pseudokôda 7.2 dobiva algoritam koji nema elitizma. *Elitizam* je svojstvo algoritma da čuva najbolje pronađeno rješenje (odnosno da ga ne može izgubiti). Naime, sasvim je moguće da sva nastala djeca budu lošija od najboljeg roditelja. Aktiviranjem populacije djece i brisanjem populacije roditelja u tom slučaju dolazi do gubitka najboljeg pronađenog rješenja. Ovome se može doskočiti tako da se najprije u populaciju djece iskopira najbolji roditelj (ili njih nekoliko), a ostatak populacije potom izgradi na uobičajeni način. Upravo je ta modifikacija prikazana pseudokôdom 7.3.

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



Slika 7.4: Korak generacijskog genetskog algoritama.

Pseudokod 7.2 Pseudokod generacijskog genetskog algoritma.

```
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
```

Pseudokod 7.3 Pseudokod generacijskog genetskog algoritma s elitizmom.

```
P = stvori_početnu_populaciju(VEL_POP)
evaluiraj(P)
ponavljaj_dok_nije_kraj
  nova_populacija P' = ∅
  ubaci u novu populaciju dvije najbolje jedinke iz populacije 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
```

odabiru jedinke za razmnožavanje. Krenimo redom.

Danas osim ove dvije "kanonske" verzije genetskog algoritma postoji i mnoštvo varijacija. Primjerice, moguće je iz populacije roditelja odabrati podskup roditelja koji će stvarati potomke pa samo njih koristiti za generiranje populacije djece. Moguće ja napraviti i varijantu u kojoj se generira više djece no što je potrebno za sljedeću populaciju, pa se potom nekim kriterijem proberu djeca koja će činiti populaciju za sljedeću generaciju (najjednostavniji kriterij jest sortirati populaciju djece po dobroti i uzeti najbolje jedinke). Također, moguće je napraviti i izvedbu kod koje se nakon generiranja djece napravi unija s populacijom roditelja i potom se iz te unije odaberu jednike koje će činiti populaciju u sljedećoj generaciji. Osim opisanih, moguć je još niz različitih varijacija, i sve njih ćemo zvati genetskim algoritmom.

7.2 Prikaz rješenja

Najjednostavniji način prikaza rješenja koji se koristi kod genetskih algoritama jest nizom bitova (engl. *bit-string*), pri čemu se svaki kromosom (tj. jedinka) sastoji se od jednakog i fiksnog broja bitova. Osim ovoga, mogući su i drugi prikazi (permutacijski prikaz, prikaz matricama, prikaz poljima cijelih brojeva, prikaz poljima decimalnih brojeva, i slično). U poglavlju 3 dan je osvrt na različite vrste prikaza rješenja kao i način provođenja operacija modificiranja rješenja (u kontekstu genetskog algoritma takav operator zovemo *mutacijom*) te operatora kombiniranja rješenja (u kontekstu genetskog algoritma takav operator zovemo *križanjem*).

7.3 Operatori križanja i mutiranja

U poglavlju 3 dan je detaljni prikaz niza operatora križanja i mutiranja za različite vrste prikaza rješenja; stoga se čitate upućuje na to poglavlje. U nastavku ćemo samo ukratko ponoviti popis najčešće korištenih operatora.

- Binarna mutacija uz zadanu vjerojatnost mutacije bita p_m (slika 3.1 na stranici 21).
- Križanje s jednom točkom prekida (slika 3.2 na stranici 22).
- Križanje s t točaka prekida (slika 3.3 na stranici 22).
- Uniformno križanje.

7.4 Operator selekcije

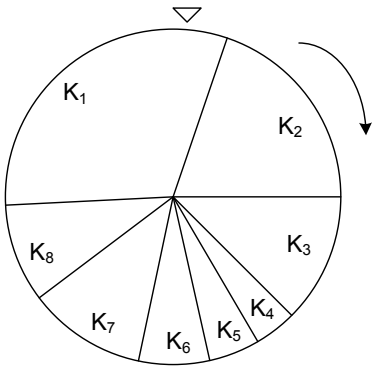
Selekcija je postupak kojim se odabiru jedinke iz populacije (najčešće u svrhu razmnožavanja). Vrste selekcija detaljno su obrađene u poglavlju 4. Stoga ćemo ovdje spomenuti samo tri najčešće.

- Proporcionalna selekcija.
- Selekcija linearnim rangiranje.
- Turnirske selekcije.

Za detalje je potrebno proučiti poglavlje 4, a u nastavku ćemo dati primjer izvedbe proporcionalne selekcije.

7.4.1 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 7.5). Potom zavrtimo kolo i pogledamo na kojoj se je jedinki kolo zaustavilo. S obzirom da jedinki pripada to veći dio dio oboda kola što ima veću dobrotu, ponavljamo li eksperiment puno puta, bolje će jedinke biti češće birane od lošijih, i ta će vjerojatnost biti upravo proporcionalna dobroti same jedinke.



Slika 7.5: Ilustracija proporcionalne selekcije za slučaj 8 jedinki čije su dobrote prikazane tablicom 7.1.

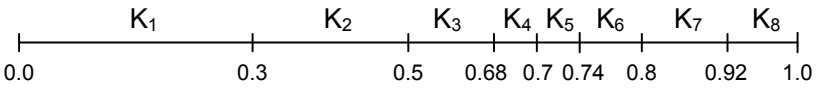
Tablica 7.1: Primjer razdiobe vjerojatnosti selekcije temeljem dobrote kod proporcionalne selekcije

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

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

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

Pogledajmo to na primjeru s 8 jedinki (tablica 7.1). Uočimo da na ovaj način normirana duljina segmenta direktno odgovara i vjerojatnosti odabira odnosne jedinke.

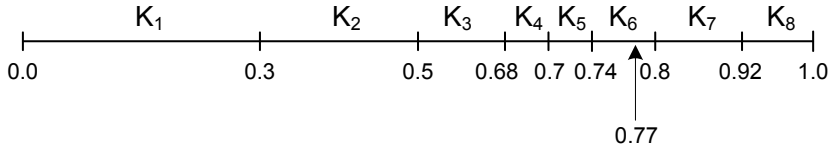


Slika 7.6: Ilustracija proporcionalne selekcije jediničnim pravcem za slučaj 8 jedinki čije su dobrote prikazane tablicom 7.1.

Stavimo li sve jedinke na segment pravca, dobit ćemo situaciju prikazanu na slici 7.6. 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 $K6$ (slika 7.7).

Spomenimo odmah i problem koji se javlja kod ovako izvedene proporcionalne selekcije i ilustrirajmo ga na primjeru prikazanom u tablici 7.2.

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 $2 \cdot 10^5$). U nekom trenutku populacija se sastoji od jedinki prikazanih u tablici 7.2. 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 dobrote relativno mali brojevi koji se pri tome dosta razlikuju, algoritam radi zadovoljavajuće. Međutim, kako vrijednosti dobrote 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



Slika 7.7: Ilustracija proporcionalne selekcije jediničnim pravcem za slučaj 8 jedinki čije su dobrote prikazane tablicom 7.1; odabran je broj $0,77$ što odgovara jedinki K_6.

Tablica 7.2: Problem skale kod proporcionalne selekcije

Jedinka	1	2	3	4
Dobrota jedinke	1007	1008	1005	1008
Vjerojatnost odabira	0.124768	0.124892	0.12452	0.124892

Jedinka	5	6	7	8
Dobrota jedinke	1003	1011	1017	1012
Vjerojatnost odabira	0.124272	0.125263	0.126007	0.125387

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.

7.4.2 k-turnirska selekcija

k-turnirska selekcija također je obrađena u poglavlju 4. Kod ove selekcije iz populacije posredstvom slučajnog mehanizma odabire k jedinki, i potom uzima najbolju (ili najgoru – ovisno za što je koristimo). Ukoliko trebamo n roditelja, naprosto ćemo n puta ponoviti turnirsku selekciju.

U praksi se jedna modifikacija k-turnirske selekcije često koristi za izvedbu jednostavne vrste eliminacijskog genetskog algoritma. Koristi se takozvana pojednostavljena 3-turnirska selekcija. Naime, prisjetimo se što je potrebno za jedan ciklus eliminacijskog genetskog algoritma: trebamo odabrati dva roditelja, križati ih i generirati dijete, dijete mutirati, iz populacije odabrati treću jedinku i odlučiti hoće li stvoreno dijete izbaciti tu jedinku iz populacije. Uporabom pojednostavljene 3-turnirska selekcija sve opisano riješi se odjednom: iz populacije se nasumice izvuku tri jedinke (kao kod 3-turnirske selekcije). Među te tri jedinke se pronađe najgora jedinka. Dvije preostale jedinke križaju se i generiraju dijete koje se mutira i iz populacije izbacuje upravo onu izabranu najgoru jedinku.

7.5 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 7.3).

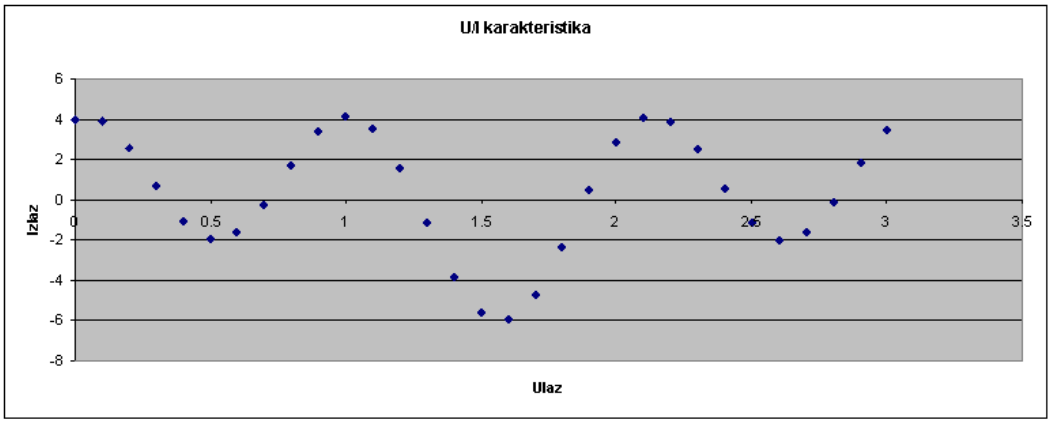
Zanima nas algebarski izraz koji najbolje opisuje tabeliranu funkciju (grafički prikaz dan je na slici 7.8).

Zadatak genetskog programiranja jest pronaći funkciju koja dobro aproksimira ove vrijednosti. Sjetimo se da se svaka funkcija može prikazati operatorskim stablom. Primjerice, funkciju:

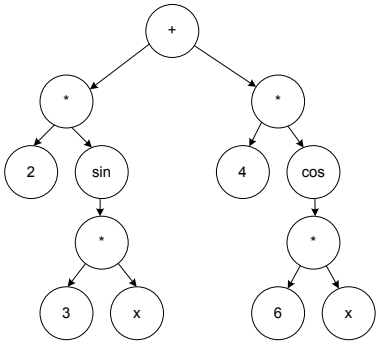
$$f(x) = 2 \cdot sin(3 \cdot x) + 4 \cdot cos(6 \cdot x)$$

Tablica 7.3: Primjer funkcije jedne varijable za aproksimaciju genetskim programiranjem

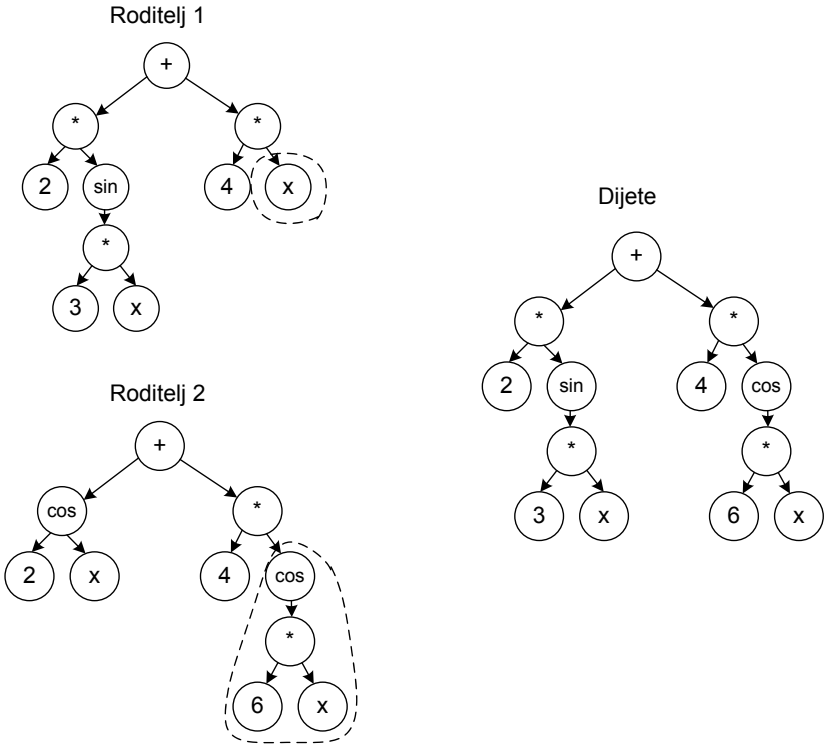
x	f(x)	x	f(x)
0	4	1.6	-5.93108
0.1	3.892383	1.7	-4.70869
0.2	2.578716	1.8	-2.32285
0.3	0.657845	1.9	0.472592
0.4	-1.0855	2	2.816585
0.5	-1.96498	2.1	4.031366
0.6	-1.63934	2.2	3.846619
0.7	-0.23462	2.3	2.480139
0.8	1.700922	2.4	0.548066
0.9	3.393531	2.5	-1.16275
1	4.122921	2.6	-1.97962
1.1	3.485439	2.7	-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		



Slika 7.8: Grafički prikaz ulazno-izlazne karakteristike sustava; vrijednosti odgovaraju onima prikazanima tablicom 7.3.



Slika 7.9: Prikaz funkcije operatorskim stablom.



Slika 7.10: Izvedba križanja kod genetskog programiranja.

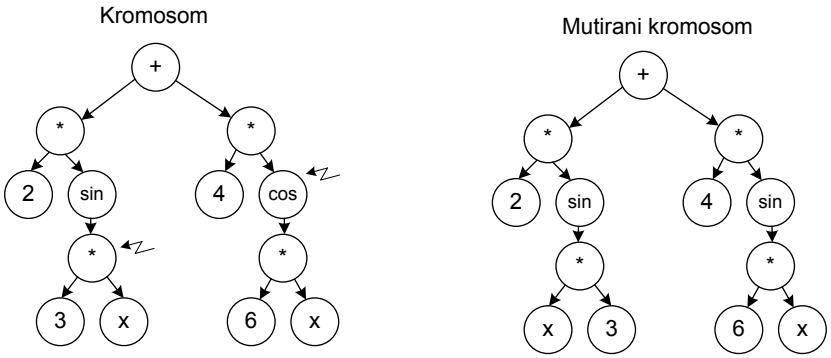
je moguće prikazati stablom prikazanim na slici 7.9.

Kromosomi kod genetskog programiranja upravo su operatorska stabla, čiji čvorovi mogu biti, primjerice, aritmetičke operacije ($+$, $-$, $\cdot$, $/$), 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 izmjerenog podatka (ili možemo računati srednje kvadratno odstupanje), po svim točkama za koje imamo zadane podatke. Križanje dvaju roditelja tipično se radi tako da se neko podstablo jednog roditelja zamijeni nekim podstablom drugog roditelja (slika 7.10).

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 na slici 7.11.

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).



Slika 7.11: Izvedba mutacije kod genetskog programiranja.

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.

Bibliografija

E. Alba and B. Dorronsoro. *Cellular Genetic Algorithms*, volume 42 of *Series Operations Research/Computer Science Interfaces Series*. Springer, 2008.

C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, 1st edition, 1859.

L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975.

J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme und Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973.

H.-P. Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technische Universität Berlin, 1975.

Poglavlje 8

Mravlji algoritmi

8.1 Uvod

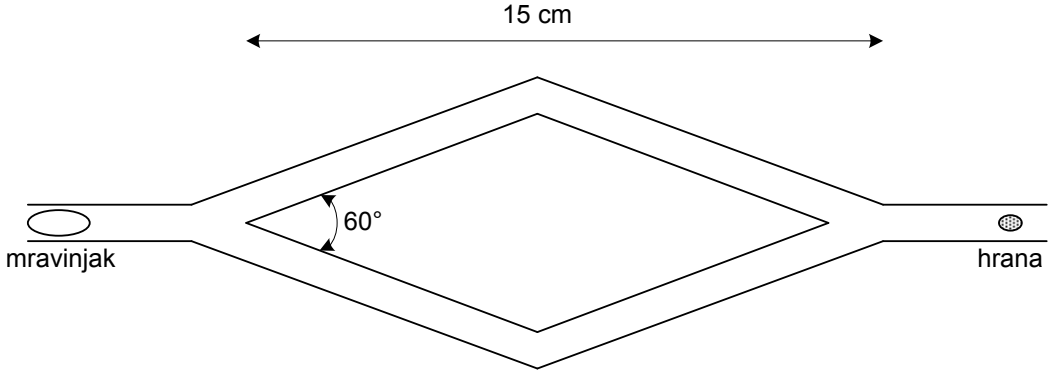
Mravi su izuzetno jednostavna bića – usporedimo li ih, primjerice, s čovjekom. No i tako jednostavna bića zahvaljujući socijalnim interakcijama postižu zadivljujuće rezultate. Sigurno ste puno puta u životu naišli na mravlju autocestu – niz mrava koji u koloni idu jedan za drugim prenoseći hranu do mravinjaka. 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?

Mnoštvo različitih znanstvenih disciplina iskazalo je interes za mrave. Nama su mravi posebno zanimljivi iz vrlo praktičnog razloga – mravi uspješno rješavaju optimizacijske probleme. Naime, uočeno je da će mravi 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.

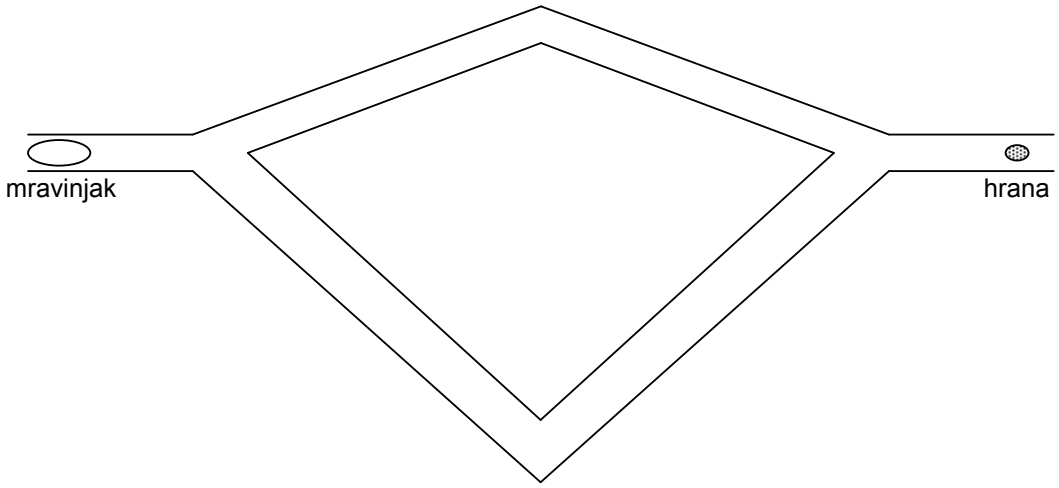
Kako bi istražili ponašanje mrava, Deneubourg i suradnici [Goss et al., 1989, Denebourg et al., 1990] napravili su niz eksperimenata koristeći dvokraki most postavljen između mravinjaka i hrane (slika 8.1).

Mravi prilikom kretanja ne koriste osjet vida, već je njihovo kretanje određeno socijalnim interakcijama s drugim mravima [Bonabeau et al., 1997a,b]. 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.

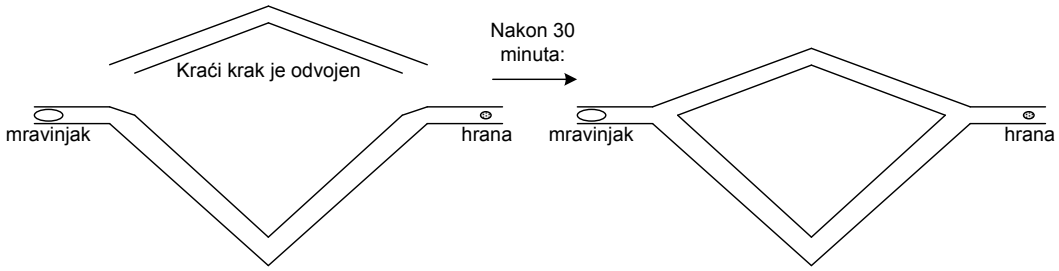
Deneubourg i suradnici prilikom eksperimenata varirali su duljine krakova mosta. U prvom eksperimentu oba su kraka jednako duga. Mravi su na početku u podjednakom broju krenuli preko oba kraka. Nakon nekog vremena, međutim, dominantni dio mrava kretao se je samo jednim krakom (slučajno odabranim). Objašnjenje je sljedeće. Na početku, mravi slučajno odabiru jedan ili drugi krak, i



Slika 8.1: Eksperiment dvokrakog mosta (prvi).



Slika 8.2: Eksperiment dvokrakog mosta (drugi).



Slika 8.3: Eksperiment dvokrakog mosta (treći).

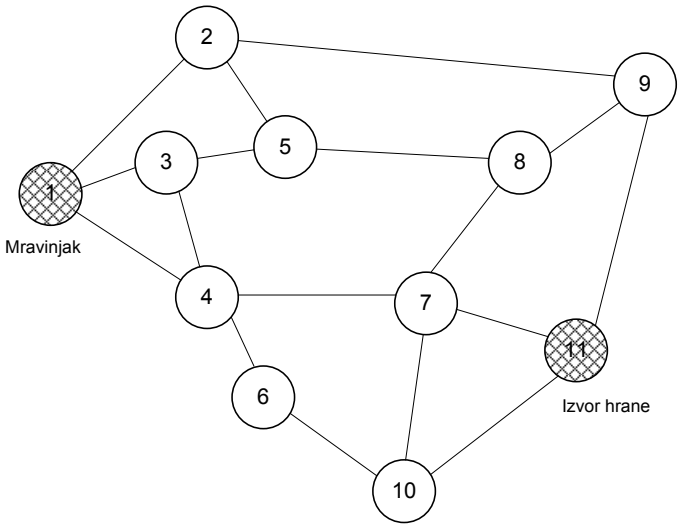
krećući se njima ostavljaju feromonski trag. U nekom trenutku dogodi se da nekoliko mrava više krene jednim od krakova (uslijed slučajnosti) i tu nastane veća koncentracija feromona. Privučeni ovom većom količinom feromona, još više mrava krene tim krakom što dodatno poveća količinu feromona. S druge 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 (slika 8.2). U svim pokušajima eksperimenta pokazalo se je da najveći broj mrava nakon nekog vremena bira kraći krak.

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" [Camazine et al., 2001, Haken, 1983, Nicolis and Prigogine, 1977]. 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 (slika 8.3).

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.



Slika 8.4: Primjer pronalaska hrane.

8.2 Pojednostavljeni matematički model

Matematički model koji opisuje kretanje mrava dan je u [Dorigo and Stützle, 2004]. 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 8.4, tuneli su modelirani bridovima grafa).

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 će čvor krenuti. Ovu odluku donosi na temelju vjerojatnosti odabira brida p_{ij}^k:

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

pri čemu τ_{ij} predstavlja vrijednost feromonskog traga na bridu između čvorova i i j, a α 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.

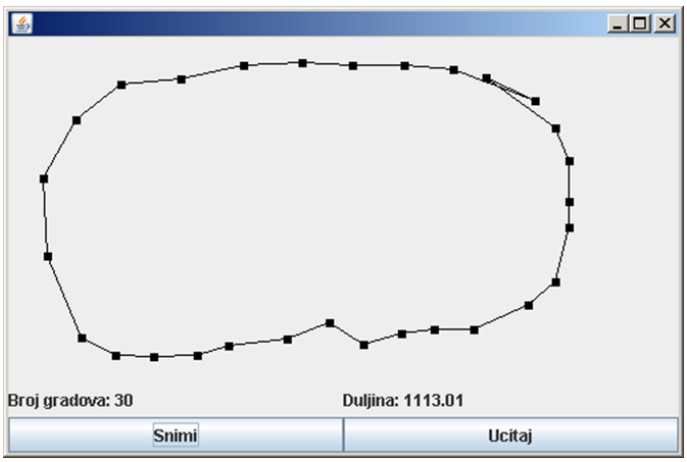
Nakon što odabere sljedeći čvor, mrav ponavlja proceduru sve dok ne stigne do izvora hrane (čvor 11). Uočimo kako se rješenje problema ovom tehnikom gradi dio po dio – mravlji algoritmi pripadaju porodici *konstrukcijskih algoritama*.

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:

$$\Delta\tau^k = \frac{1}{L},$$



Slika 8.5: Rješenje TSP-a dobiveno jednostavnim mravljim algoritmom.

pri čemu je L duljina pronađenog puta. Isparavanje feromona modelirano je izrazom:

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

gdje je ρ 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.

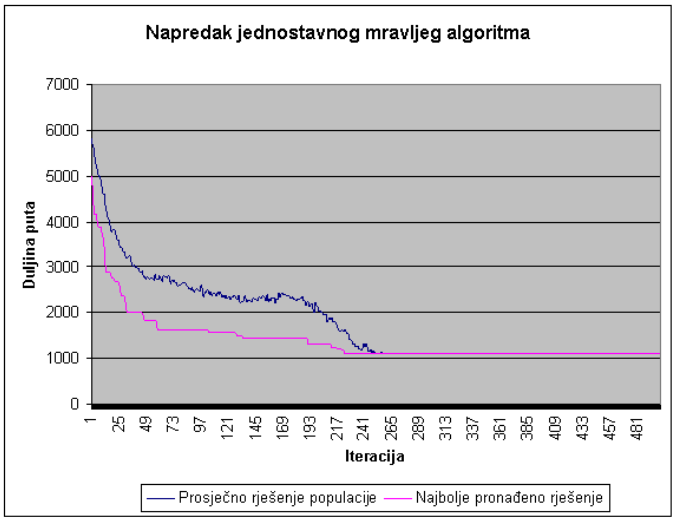
Temeljeći se na opisanim principima moguće je napisati jednostavan optimizacijski algoritam, što je prikazano pseudokodom 8.1.

Pseudokod 8.1 Pseudokod jednostavnog mravljeg algoritma.

```
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
```

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 (skupovi N_{ij}^k). Tek kada je svih m mrava stvorilo prijedloge rješenja, odabire se $n \leq m$ 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 najbolji mrav 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; dobiveno rješenje kao i rješavani problem prikazani su na slici 8.5).



Slika 8.6: Napredak jednostavnog mravljeg algoritma.

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 $\frac{1}{5000}$. Sa slike se može uočiti da pronađeno rješenje nije idealno, ali je vrlo blizu optimalnog. Kvaliteta pronađenih rješenja kroz epohe prikazana je na slici 8.6, gdje se može pratiti najbolje pronađeno rješenje kao i prosječno pronađeno rješenje.

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

8.3 Algoritam Ant System

Algoritam *Ant System* predložili su Dorigo i suradnici [Dorigo and Stützle, 2004, Dorigo et al., 1991, Colorni et al., 1992]. 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:

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

gdje je C^{nn} 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 [Dorigo and Stützle, 2004], uz premali τ_0 pretraga će brzo biti usmjerena prema području koje su mravi slučajno odabrali u prvoj iteraciji, a uz preveliki τ_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:

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

η_{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 $\eta_{ij} = \frac{1}{d_{ij}}$, gdje je d_{ij} udaljenost grada i od grada j. Parametri α i β 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 β 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:

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

Implementacija ovog algoritma dana je pseudokodom 8.2.

Pseudokod 8.2 Pseudokod algoritma Ant System.

```
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
```

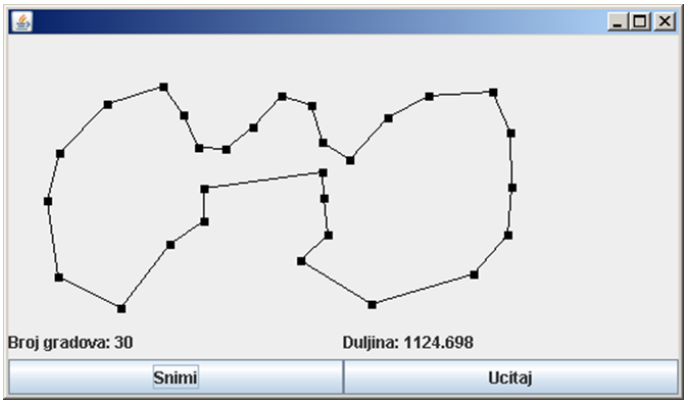
U svrhu ilustracije rada algoritma napravljen je program u programskom jeziku Java, i pokrenut nad problemom trgovačkog putnika s 30 gradova. Parametri algoritma su redom: $m = 30$, $\alpha = 1$, $\beta = 2$, $\rho = 0.5$. Algoritam je zaustavljen nakon 500 iteracija. Rezultati su prikazani na slikama 8.7 i 8.8.

Elitistička verzija algoritma (engl. *Elitist Ant System* – EAS) predložena je kao poboljšanje algoritma u [Dorigo et al., 1991, 1996]. 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:

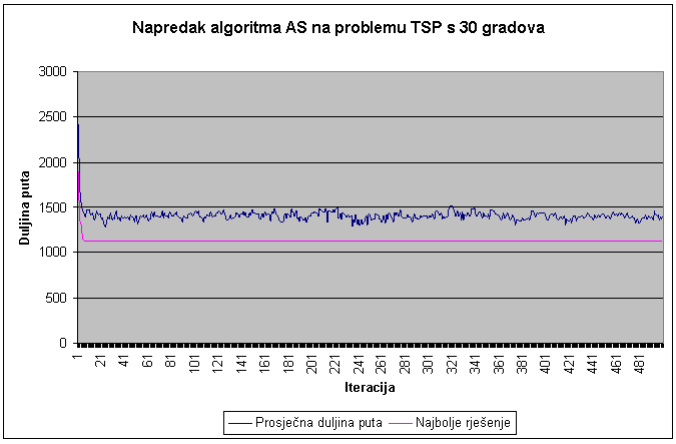
$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e \cdot \Delta\tau_{ij}^{bs}$$

U [Dorigo and Stützle, 2004] se kao vrijednost konstante e navodi upravo broj mrava m.

Prema [Bullnheimer et al., 1999], 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



Slika 8.7: Rješenje TSP-a dobiveno algoritmom Ant System.



Slika 8.8: Napredak algoritma Ant System.

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) [Stützle and Hoos, 1997, 2000, 1999] 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:

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

gdje je a parametar koji je potrebno izabrati. 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 [Dorigo and Stützle, 2004].

Bibliografija

E. Bonabeau, A. Sobkowski, G. Theraulaz, and J. L. Deneuborg. Adaptive task allocation inspired by amodel of division of labor in social insects. In D. Lundha, B. Olsson, and A. Narayanan, editors, *Bio-Computation and Emergent Computing*, pages 36–45, Singapore, 1997a. World Scientific Publishing.

E. Bonabeau, G. Theraulaz, J. L. Denebourg, S. Aron, and S. Camazine. Self-organization in social insects. *Tree*, 12(5):188–193, 1997b.

B. Bullnheimer, R. F. Hartl, and C. Strauss. A new rank-based version of the ant system: A computational study. *Central European Journal for Operations Research and Economics*, 7(1):25–38, 1999.

S. Camazine, J. L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau, editors. *Self-organization in Biological Systems*. Princeton University Press, Princeton, NJ, 2001.

A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 134–142, Cambridge, MA, 1992. MIT Press.

J. L. Denebourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behaviour*, 3:159–168, 1990.

M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. Technical report, Dipartimento di Elettronica, Politecnico di Milano, Milano, 1991. Technical report 91-016.

M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 26(1):29–41, 1996.

S. Goss, S. Aron, J.-L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

H. Haken. *Synergetics*. Springer-Verlag, Berlin, 1983.

G. Nicolis and I. Prigogine. *Self-Organisation in Non-Equilibrium Systems*. John Wiley and Sons, New York, 1977.

T. Stützle and H. Hoos. The max-min ant system and local search for the traveling salesman problem. In T. Bäck, Z. Michalewicz, and X. Yao, editors, *Proceedings of the 1997 IEEE International Conference on Evoultionary Computation (ICEC'97)*, pages 309–314, Piscataway, NJ, 1997. IEEE Press.

T. Stützle and H. Hoos. Max-min ant system and local search for combinatorial optimization problem. In S. Voss, S. Martello, I. Osmann, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trend sin Local Search Paradigms for optimization*, pages 137–154. Kluwer Academic Publishers, Dordrecht, Netherlands, 1999.

T. Stützle and H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

Poglavlje 9

Algoritam roja čestica

9.1 Uvod

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 [Reynolds, 1987] 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 [Kennedy and Eberhart, 1995, Eberhart and Kennedy, 1995]. Eberhart, Simpson i Dobbins 1996. godine izdaju knjigu [Eberhart et al., 1996] 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 [Kennedy and Eberhart, 1997]. 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.

9.2 Opis algoritma

Algoritam roja čestica je populacijski algoritam. Populacija se sastoji od niza jedinki (čestica) koje lete kroz višedimenzijski 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.

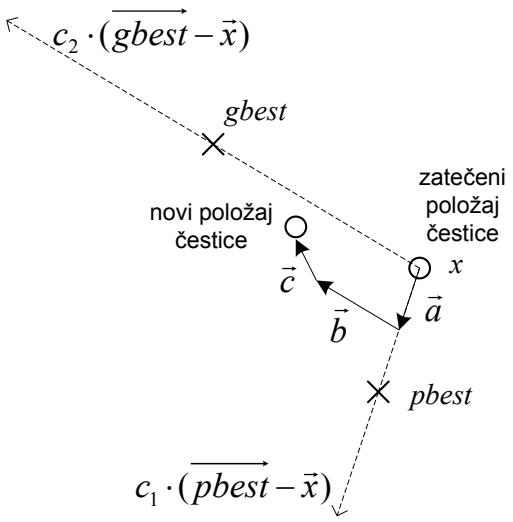
Pseudokod 9.1 prikazuje ideju algoritma roja čestica. Algoritam koristi populaciju veličine `VEL_POP`, te pretražuje `DIM`-dimenzijski prostor rješenja. Pri tome x sadrži trenutne pozicije svih čestica a v njihove brzine. Polja x i v su dvodimenzijska: imaju onoliko redaka koliko ima čestica, te onoliko 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.

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

Pseudokod 9.1 Pseudokod algoritma roja čestica.

```
// 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
```



Slika 9.1: Grafički prikaz pomaka čestice kod algoritma roja čestice.

- 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 dvodimenzijskom polju *pbest* (engl. *particles best solution*). Potrebno je pamtiti 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 pamtiti 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.

Kako vidimo iz opisanoga, ažuriranje se obavlja prema sljedećim izrazima:

$$v_{i,d} = v_{i,d} + c_1 \cdot \text{rand}() \cdot (\text{pbest}_{i,d} - x) + c_2 \cdot \text{rand}() \cdot (\text{gbest}_d - x)$$

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 9.1 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 $c_2 = 2.0$.

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

$$\begin{aligned} \vec{a} &= c_1 \cdot \text{rand}() \cdot \vec{(}\text{pbest} - \vec{x}), \\ \vec{b} &= c_2 \cdot \text{rand}() \cdot \vec{(}\text{gbest} - \vec{x}), \\ \vec{c} &= \vec{v}. \end{aligned}$$

Vektor $\vec{a}$ predstavlja pomak prema najboljem rješenju koje je pronašla sama jedinka, odgovarajuće skaliranom. Vektor $\vec{b}$ predstavlja pomak prema najboljem rješenju kolektiva, također odgovarajuće skaliranom. Vektor $\vec{c}$ 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.

9.3 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 se oteti utjecaju lokalnog optimuma. Prema [Eberhart and Shi, 2001], *vmax* 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.

9.3.1 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:

$$v_{i,d} = w(t) \cdot v_{i,d} + c_1 \cdot \text{rand}() \cdot (\text{pbest}_{i,d} - x) + c_2 \cdot \text{rand}() \cdot (\text{gbest}_d - x)$$

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.

9.3.2 Stabilnost algoritma

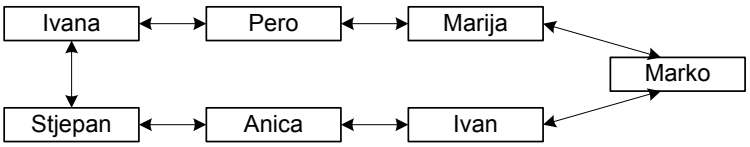
Već smo spomenuli da je jedan od načina da se pokuša osigurati stabilnost algoritma (u smislu da se spriječi divergencija) ograničavanje iznosa brzine. Matematička analiza samog algoritma (vidi [Clerc, 1999]) pokazuje da se stabilnost može postići ako se izraz za ažuriranje brzine modificira dodavanjem *faktora ograničenja* (engl. *constriction factor*) K:

$$v_{i,d} = K \cdot \left[v_{i,d} + c_1 \cdot \text{rand}() \cdot (\text{pbest}_{i,d} - x) + c_2 \cdot \text{rand}() \cdot (\text{gbest}_d - x) \right]$$

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

$$K = \frac{2}{\left| 2 - \phi - \sqrt{\phi^2 - 4 \cdot \phi} \right|}.$$

Pri tome je $\phi = c_1 + c_2$, $\phi > 4$. Prema [Eberhart and Shi, 2001], ϕ se tipično postavlja na 4.1 ($c_1 = 2.05$, $c_2 = 2.05$), što daje za K vrijednost 0.729.



Slika 9.2: Primjer definiranog susjedstva.

9.3.3 Lokalno susjedstvo

Kod algoritma roja čestica opisanog pseudokodom 9.1 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 9.2; susjedi su osobe direktno povezane strelicama).

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 n_s (veličina susjedstva), a čestice slože u niz. Ako je $n_s = 1$, svaka je čestica isključivo susjed sama sebi. Ako je $n_s = 2$, susjedi od `čestica(i)` su `čestica(i-1)`, `čestica(i)` te `čestica(i+1)`. Ako je $n_s = 3$, 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:

$$v_{i,d} = v_{i,d} + c_1 \cdot \text{rand}() \cdot (\text{pbest}_{i,d} - x) + c_2 \cdot \text{rand}() \cdot (\text{lbest}_{i,d} - x).$$

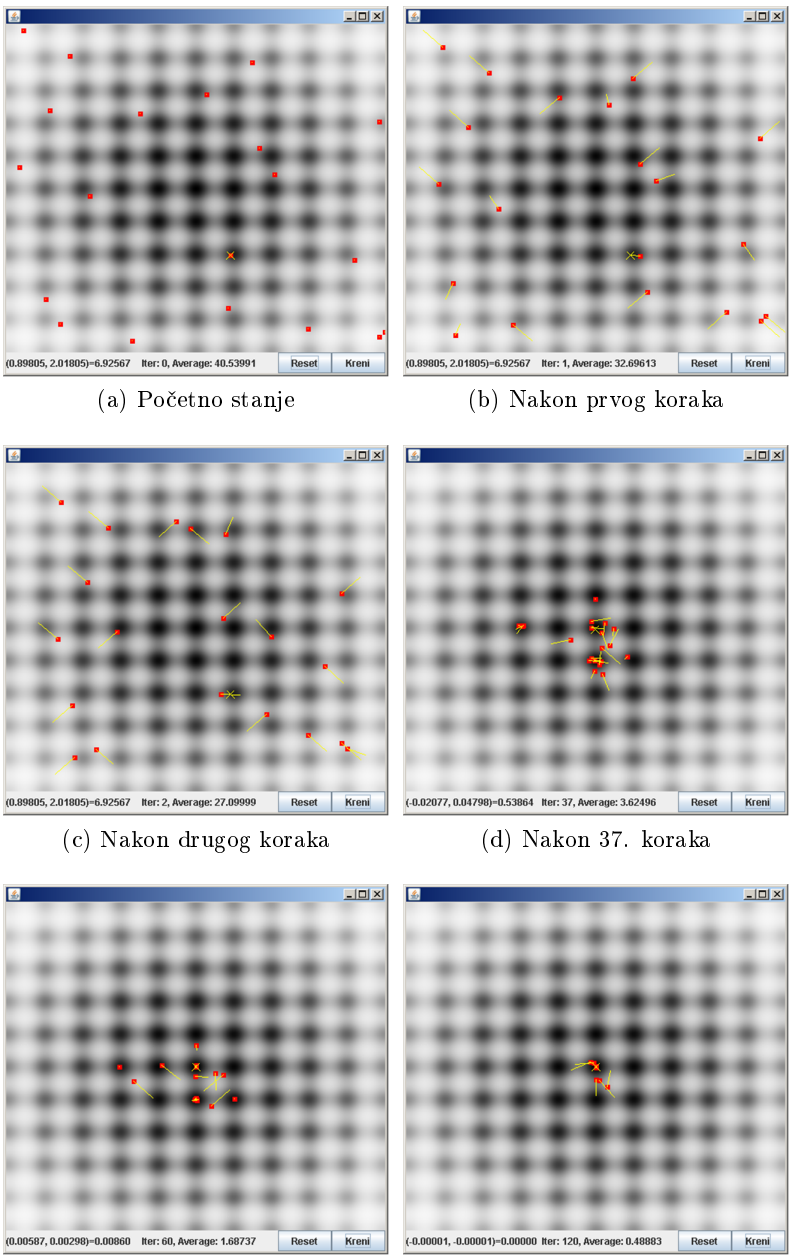
Prema [Eberhart and Shi, 2001], prikladne veličine susjedstva su oko 15% ukupne veličine populacije. Ako s `VEL_SUS` označimo ukupan broj čestica koje čine susjedstvo, vrijedi:

$$\text{VEL\_SUS} = 1 + 2 \cdot n_s.$$

Za više informacija na ovu temu čitatelj se upućuje na nedavno objavljeni rad [Montes de Oca et al., 2009].

9.4 Primjer rada algoritma

Primjer u nastavku prikazuje optimizaciju funkcije:



(a) Početno stanje (b) Nakon prvog koraka

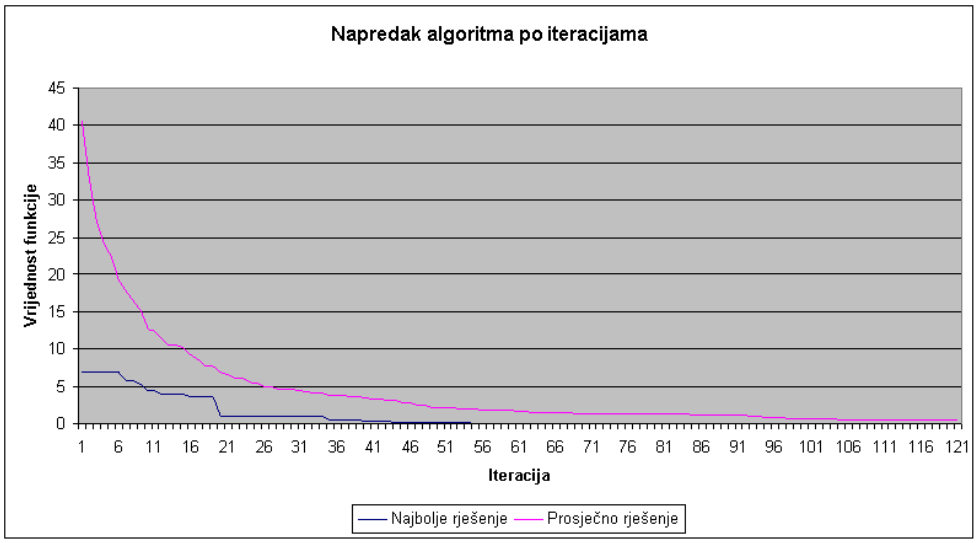
(c) Nakon drugog koraka (d) Nakon 37. koraka

(e) Nakon 60. koraka (f) Nakon 120. koraka

Slika 9.3: Prikaz rada algoritma roja čestica

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

Konkretno, tražimo vektor $\vec{x}$ 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. Prikaz rada algoritma za slučaj $D = 2$ prikazan je na slici 9.3, dok slika 9.4 prikazuje ovisnost najboljeg i prosječnog rješenja populacije u ovisnosti o epohi algoritma.



Slika 9.4: Ovisnost najboljeg i prosječnog rješenja o iteraciji kod algoritma roja čestica.

Bibliografija

M. Clerc. The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In *Proc. 1999 Congress on Evolutionary Computation, Washington, DC*, pages 1951–1957, Piscataway, NJ: IEEE Service Center, 1999.

R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995. Piscataway, NJ: IEEE Service Center.

R. C. Eberhart and Y. Shi. Particle swarm optimization: developments, applications and resources. In *Proc. Congress on Evolutionary Computation 2001, Seoul, Korea*, pages 81–86, Piscataway, NJ: IEEE Service Center, 2001.

R. C. Eberhart, P. K. Simpson, and R. W. Dobbins. *Computational Intelligence PC Tools*. Academic Press Professional, Boston, MA, 1996.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, Piscataway, NJ, 1995.

J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In *In Proceedings of the IEEE 1997 International Conference on Systems, Man and Cybernetics*, pages 4104–4109, 1997.

M. A. Montes de Oca, T. Stützle, M. Birattari, and D. M. Frankenstein's pso: A composite particle swarm optimization algorithm. *IEEE Trans. on Evolutionary Computation*, 13(5):1120–1132, 2009.

C. W. Reynolds. Flocks, herds and schools: a distributed behavioral model. *Computer Graphics*, 21 (4):25–34, 1987.

Poglavlje 10

Umjetni imunološki algoritmi

10.1 Uvod

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 razmožavanje 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 razmožavanje, 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 okruže antigen 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 naiđe 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 grupo pojednostavljenje 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 [Burnet, 1957, 1959, 1978], počev još davne 1957. godine. Područje *umjetnih imunoloških sustava* (engl. AIS – *Artificial Im-*

mune Systems) bavi se razvojem modela i apstrakcija imunološkog sustava i njihovom primjenom u algoritmima za rješavanje problema u znanosti i inženjerstvu [de Castro and Timmis, 2002]. 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*) [Cutello and Nicosia, 2005, de Castro and Timmis, 2002]. Važno je napomenuti da zbog jednostavnosti, većina algoritama ne uvodi distinkciju između pojmova B-stanica i antitijelo, te ta dva pojma tretira kao jedan.

10.2 Jednostavni imunološki algoritam

Jednostavni imunološki algoritam (engl. SIA – *Simple Immunological Algorithm*) autora Cutello i Nicosia razvijen je 2002. godine, i opisan u [Cutello and Nicosia, 2002a,b, 2005]. 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. Izvedba algoritma dana je pseudokodom 10.1.

Pseudokod 10.1 Pseudokod jednostavnog imunološkog algoritma.

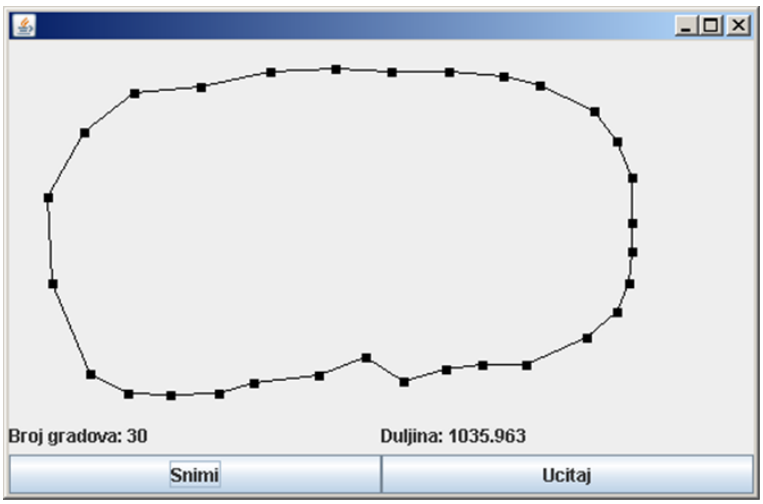
```
SIA(Ag, l, d, dup)
  t = 0
  inicijaliziraj P(0) = {x1, x2, ..., xd}
  evaluiraj(P(0), Ag)
  ponavljaj dok nije zaustavi(P(t))
    Pclo = kloniraj(P(t), dup)
    Phyp = hipermutiraj(Pclo)
    evaluiraj(Phyp, Ag)
    P(t+1) = odaberi(Phyp + P(t), d)
    t = t+1
  kraj ponavljanja
kraj
```

Parametri algoritma su:

- Ag – funkcija koju se optimira,
- l – broj bitova rješenja,
- d – veličina populacije rješenja, te
- 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).

Zatim 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.



Slika 10.1: Problem obilaska 30 gradova riješen algoritmom SIA.

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 $d \cdot dup$. Svako antitijelo iz populacije P^{clo} prolazi potom 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 jedinke 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. Uočimo 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 [Cutello and Nicosia, 2005].

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:

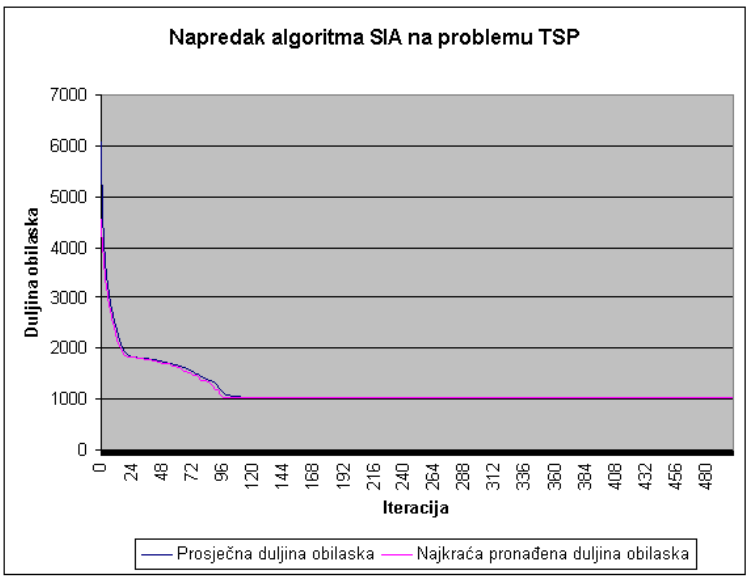
- broj antitijela u populaciji: 50,
- broj klonova za svaku stanicu: 10, te
- maksimalni broj iteracija: 500.

Na testnom primjeru (vidi sliku 10.1) 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 10.2.

U nastavku slijedi opis algoritma CLONALG.

10.3 Algoritam CLONALG

CLONALG je kratica od *Clonal Selection Algorithm*. Ovaj algoritam predložili su 1999. i kasnije razradili (2000., 2002.) De Castro i Von Zuben [de Castro and Zuben, 1999, 2000, 2002]. Opis algoritma dan je pseudokodom 10.2.



Slika 10.2: Napredak algoritma SIA na problemu TSP s 30 gradova.

Pseudokod 10.2 Pseudokod algoritma CLONALG.

```
CLONALG(Ag, n, d, beta)
  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), beta)
    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
```

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-dimenzijskom 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
- β – 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 fiksan 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 kloniranju 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 β 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 ρ 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.

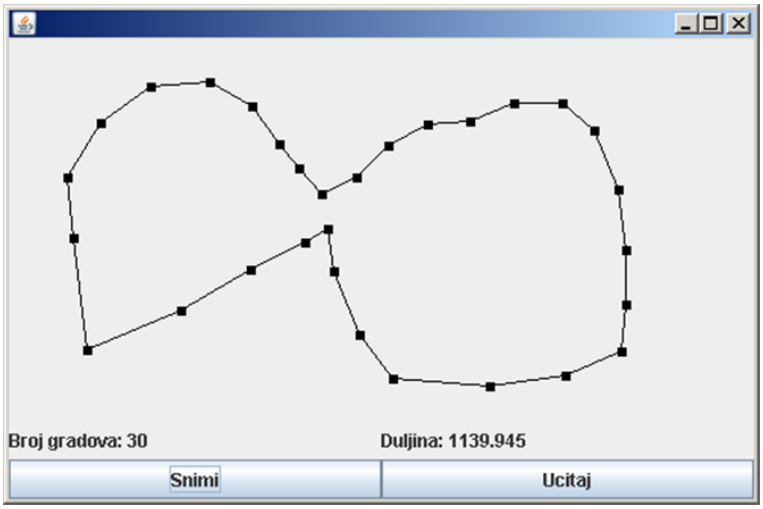
Analiza složenosti samog algoritma može se pogledati u [Cutello and Nicosia, 2005]. Spomenimo samo da je ona veća od prethodno opisanog algoritma jer uključuje i sortiranja kompletnih populacija.

Kako bi se provjerio rad algoritma, napravljena je implementacija u programskom jeziku Java. Rezultat izvođenja algoritma prikazan je na slici 10.3, dok je napredak algoritma po iteracijama prikazan na slici 10.4.

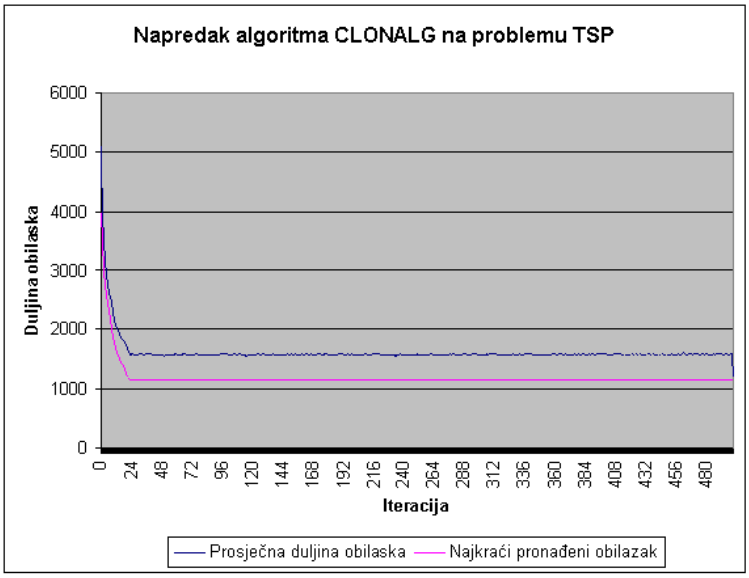
Rezultati su dobiveni uz $n = 200$, $d = 20$ i $\beta = 5$.

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, ρ pozitivna konstanta manja od 1). Postavljen je zahtjev da točan broj (k) bude određen izrazom:

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



Slika 10.3: Problem obilaska 30 gradova riješen algoritmom CLONALG.



Slika 10.4: Napredak algoritma CLONALG na problemu TSP s 30 gradova.

pri čemu je r rang antitijela (0 za najbolje antitijelo, 1 za sljedeće, itd). Promatramo li k kao funkciju od r, iz zahtjeva: $k(0) = 0$ 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 $\tau \approx 691.736$ (uočimo 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).

10.4 Pregled korištenih operatora

U nastavku ćemo ukratko nabrojati i opisati vrste operatora koji se danas uobičejeno koriste kod imunoloških algoritama. Detaljna razmatranja ovih operatora kao i dokaz konvergencije imunoloških algoritama nalazi se u [Cutello et al., 2007].

10.4.1 Operator kloniranja

Operator kloniranja zadužen je za stvaranje populacije klonova iz postojeće populacije antitijela. Operator statičkog kloniranja (engl. *static cloning operator*) [Cutello and Nicosia, 2002a] 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*) [de Castro and Zuben, 2002] za svako antitijelo stvara broj klonova koji je proporcionalan afinitetu antitijela (tako da bolja antitijela dobiju više klonova). Vjerojatnosno kloniranje (engl. *probabilistic cloning*) [Cutello et al., 2003] definira parametar p_c (engl. *clonal selection rate*) temeljem kojeg iz izvorne populacije bira antitijela koja će biti klonirana.

10.4.2 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*) [Cutello et al., 2005].

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. *hypermacromutation*) 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.

10.4.3 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 τ_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. *stohastic aging operator*) antitijelo iz populacije može biti izbrisano i prije isteka τ_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.

10.5 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*).

Bibliografija

F. M. Burnet. A modification of jerne's theory of antibody production using the concept of clonal selection. *Australian Journal of Science*, 20:67–69, 1957.

F. M. Burnet. *The clonal selection theory of acquired immunity.* Vanderbilt University Press, Nashville, Tennessee, U.S.A., 1959.

F. M. Burnet. Clonal selection and after. *Theoretical Immunology*, pages 63–85, 1978.

V. Cutello and G. Nicosia. An immunological approach to combinatorial optimization problems. In *Proceedings of the 8th Ibero-American Conference on AI: Advances in Artificial Intelligence, Seville, Spain*, pages 361–370, 2002a.

V. Cutello and G. Nicosia. Multiple learning using immune algorithms. In *Proceedings of 4th International Conference on Recent Advances in Soft Computing, RASC 2002, Nottingham, UK*, pages 102–107, 2002b.

V. Cutello and G. Nicosia. The clonal selection principle for in silico and in vitro computing. In L. N. de Castro and F. J. V. Zuben, editors, *Recent Developments in Biologically Inspired Computing*, chapter IV, pages 104–146. Idea Group Publishing, Hershey, London, Melbourne, Singapore, 2005.

V. Cutello, G. Nicosia, and M. Pavone. A hybrid immune algorithm with information gain for the graph coloring problem. In *GECCO '03, Chicago, IL, USA*, volume 2723, pages 171–182. Springer, 2003.

V. Cutello, G. Narzisi, G. Nicosia, and M. Pavone. Clonal selection algorithms: A comparative case study using effective mutation potentials. In *ICARIS 2005*, volume 3627, pages 13–28. Springer, 2005.

V. Cutello, G. Nicosia, P. S. Oliveto, and M. Romeo. On the convergence of immune algorithms. In *The First IEEE Symp. on Foundations of Computational Intelligence, FOCI 2007*, pages 409–415, Honolulu, Hawaii, USA, 2007. IEEE Press.

L. N. de Castro and J. Timmis. *Artificial Immune Systems: A new computational intelligence approach.* Springer-Verlag, Great Britain, 2002.

L. N. de Castro and F. J. V. Zuben. Artificial immune systems - part i: Basic theory and applications. Technical report, Department of Computer Engineering and Industrial Automation, School of Electrical and Computer Engineering, State University of Campinas, Brazil, December 1999. TR DCA 01/99.

L. N. de Castro and F. J. V. Zuben. The clonal selection algorithm with engineering applications. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00), Workshop on Artificial Immune Systems and Their Applications*, pages 36–37, Las Vegas, Nevada, USA, 2000.

L. N. de Castro and F. J. V. Zuben. Learning and optimization using the clonal selection principle. *IEEE Transactions on Evolutionary Computation*, 6:239–251, June 2002.

Poglavlje 11

Algoritam diferencijske evolucije

11.1 Uvod

Algoritam diferencijske evolucije, iako izravno ne pripada među biološki inspirirane algoritme, ima dosta sličnosti s genetskim algoritmom. To je također populacijski algoritam koji nove potomke stvara uporabom operatora *diferencijacije*.

Razvoj algoritma diferencijske evolucije može se pratiti još od 1994. godine, kada je Storn u časopisu *Dr. Dobbs Journal* opisao algoritam genetskog kaljenja (engl. *Genetic Annealing*) [Price, 1994]. Godinu dana kasnije, prva verzija algoritma diferencijske evolucije opisana je u tehničkom izvještaju [Storn and Price, 1995], nakon čega je uslijedio niz radova [Storn and Price, 1996, Price and Storn, 1997, Storn and Price, 1997, Price, 1997, 1999].

Prva izvedba algoritma bila je napravljena za rješavanje optimizacijskih problema funkcija u kontinuiranom prostoru. Stoga algoritam sva rješenja tretira kao D-dimenzijske vektore. Osnovna ideja algoritma jest modificirati jedinke trenutne populacije linearnom kombinacijom drugih jedinki iste populacije. Ovakav pristup ima često poželjno svojstvo *adaptivnog koraka pretraživanja*; naime, kako se modifikacija jedinki radi skaliranom diferencijom dviju slučajno odabranih jedinki, u slučaju kada je populacija dosta raspršena, i promjene će biti dosta velike. S druge strane, kada se populacija preseli u blizinu optimuma, diferencije između svih jedinki će biti male pa će i modifikacije biti male, čime se pospješuje brzo napredovanje prema optimumu, odnosno konvergencija.

Da bismo objasnili kako algoritam radi, krenimo s pretpostavkom da rješavamo optimizacijski problem pronalaska vektora $\vec{x}$ koji minimizira zadanu funkciju $f(\vec{x})$; neka je pri tome vektor $\vec{x}$ D-dimenzijski, tj. $\vec{x} = (x_0, x_1, \ldots, x_{D-1})$.

Općeniti koraci algoritma prikazani su u nastavku.

1. Stvori početnu populaciju.

2. Ponavljaj (generacijska petlja)

   (a) Kreni po svim jedinkama (petlja po trenutnim rješenjima)

      i. Trenutnu jedinku prozovi *ciljnim vektorom*.

      ii. Generiraj *bazni vektor* i mutiraj ga: rezultat nazovimo *vektor mutant*. Generiranje mutanta radi se u skladu s nekom strategijom (opisano kasnije).

      iii. Križaj *ciljni vektor* i *vektor mutant*: rezultat nazovimo *probni vektor*; postoji više načina križanja (opisano kasnije).

      iv. Primijeni operator *selekcije*: u sljedeću generaciju pošalji *probni vektor* ako nije lošiji od *ciljnog vektora*; inače pošalji *ciljni vektor*.

   (b) Stvorenu populaciju proglasi novom "trenutnom" generacijom.

Sada ćemo detaljnije pogledati svaki od navedenih koraka.

11.2 Stvaranje početne populacije

Algoritam diferencijske evolucije je populacijski algoritam. Pretpostavimo da je broj jedinki koje je potrebno stvoriti jednak n. Jedinke su D-dimenzijski vektori pri čemu su dozvoljene vrijednosti koje j-ta komponenta vektora $\vec{x}_i$ može poprimiti ograničene s $b_{L,j} \leq (\vec{x}_i)_j \leq b_{U,j}$, $0 \leq j \leq D-1$. Uz takve uvjete stvaranje početne populacije opisano je pseudokodom 11.1. Oznaka `x(j,i)` pri tome označava j-tu komponentu vektora $\vec{x}_i$. Funkcija `slucajno(a,b)` za decimalne brojeve a i b vraća slučajni decimalni broj iz raspona $[a, b)$.

Pseudokod 11.1 Pseudokod algoritma za stvaranje početne populacije.

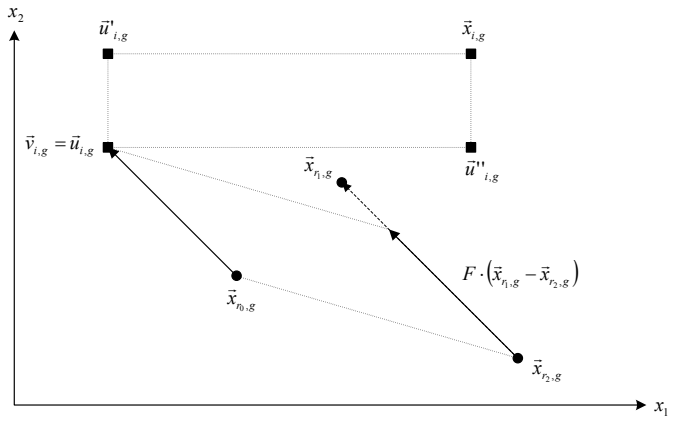
```
ponavljaj za i iz 0 do n-1
  ponavljaj za j iz 0 do D-1
    x(j,i)=bL(j) + slucajno(0.0,1.0) * (bU(j)-bL(j))
  kraj
kraj
```

11.3 Diferencijska mutacija

Prvi korak u stvaranju nove jedinke je stvaranje vektora *mutanta* (engl. *mutant vector*) za svaku jedinku trenutne generacije g. Taj će vektor potom biti iskorišten kako bi se križanjem stvorila nova jedinka koja će potom postati kandidat za ubacivanje u sljedeću generaciju $(g+1)$ – zvat ćemo je *probni vektor*. Pretpostavimo da promatramo i-tu jedinku populacije. Taj vektor $\vec{x}_{i,g}$ (i-to rješenje u generaciji g) zvat ćemo *ciljni vektor* (engl. *target vector*). Potrebno je stvoriti *bazni vektor* i mutirati ga kako bismo izgradili *vektor mutant*. Najjednostavnija strategija jest posredstvom slučajnog mehanizma odabrati još tri jedinke iz trenutne populacije u koraku g. Označimo njihove indekse r_0, r_1 i r_2. Sva četiri broja (i, r_0, r_1 i r_2) pri tome moraju biti različita. Prvi odabrani vektor ($\vec{x}_{r_0,g}$) proglasit ćemo *baznim vektorom* (engl. *base vector*). Druga dva slučajno odabrana vektora poslužit će nam za implementaciju mutacije baznog vektora: *vektor mutant* $\vec{v}_i, g$ definirat ćemo kao zbroj baznog vektora i skalirane razlike preostala dva slučajno odabrana vektora, što prikazuje izraz (11.1). Napomenimo da je ovo samo jedna moguća strategija generiranja vektora mutanta: detaljniji prikaz dan je nešto kasnije.

$$\vec{v}_{i,g} = \vec{x}_{r0,g} + F \cdot (\vec{x}_{r1,g} - \vec{x}_{r2,g}) \tag{11.1}$$

Grafički prikaz izgradnje vektora mutanta za slučaj dvodimenzijskih vektora prikazan je na slici 11.1; skalirana razlika vektora $F \cdot (\vec{x}_{r_1,g} - \vec{x}_{r_2,g})$ dodana je baznom vektoru $\vec{x}_{r_0,g}$ čime je izgrađen vektor mutant $\vec{v}_{i,g}$.



Slika 11.1: Djelovanje operatora diferencijske mutacije kod algoritma diferencijske evolucije.

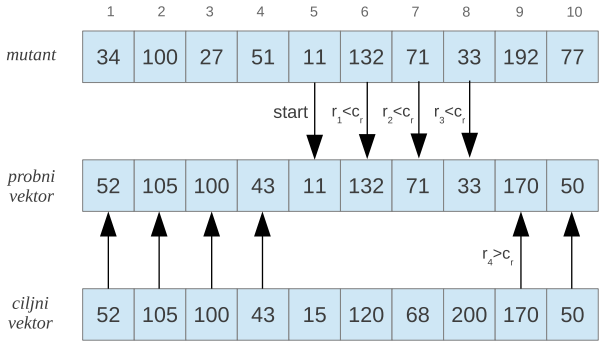
11.4 Križanje

Nakon što je za neki fiksirani ciljni vektor $\vec{x}_{i,g}$ izgrađen vektor mutant $\vec{v}_{i,g}$, provodi se postupak križanja kako bi se izgradio *probni vektor* $\vec{u}_{i,g}$ (engl. *trial vector*). Križanje se pri tome provodi između izgrađenog *vektora mutanta* i *ciljnog vektora*. Autori algoritma pri tome predlažu dvije izvedbe križanja: *eksponencijalno križanje* te *uniformno (binomno) križanje*. U oba slučaja definira se parametar C_r koji definira vjerojatnost križanja ($0 \leq C_r \leq 1$). Također, u oba slučaja inzistira se da u probni vektor uđe barem jedna komponenta vektora mutanta. Stoga obje implementacije započinju tako da se posredstvom slučajnog mehanizma odabere komponenta koja se iz vektora mutanta kopira u probni vektor i to se učini. Daljnje postupanje ovisi o vrsti križanja koje se provodi i opisano je u nastavku.

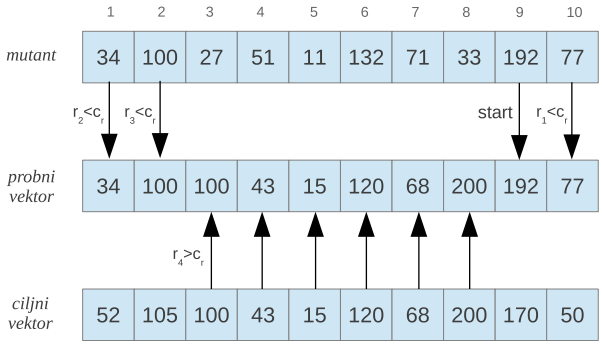
11.4.1 Eksponencijalno križanje

Postupak eksponencijalnog križanja započinje odabirom komponente koja će se sigurno prekopirati iz vektora mutanta u probni vektor. Tu ćemo poziciju na grafičkom prikazu označiti kao *start*. odabranu komponentu prekopiramo u probni vektor. Potom posredstvom slučajnog mehanizma uz vjerojatnost C_r odlučujemo hoćemo li i sljedeću komponentu prekopirati iz vektora mutanta; ako je odgovor da, kopiramo je i posredstvom slučajnog mehanizma opet uz vjerojatnost C_r odlučujemo hoćemo li i njoj sljedeću komponentu prekopirati iz vektora mutanta i tako redom. Prvi puta kada odluka bude *ne*, postupak kopiranja komponenata iz vektora mutanta prestaje i sve nepopunjene komponente preuzimaju se iz ciljnog vektora.

Posljedica ovakve izvedbe je da će probni vektor iz vektora mutanta naslijediti jedan kontinuirani niz komponenti, što je ilustrirano na slikama 11.2 i 11.3.



Slika 11.2: Primjer izvedbe eksponencijalnog križanja kod algoritma diferencijske evolucije.

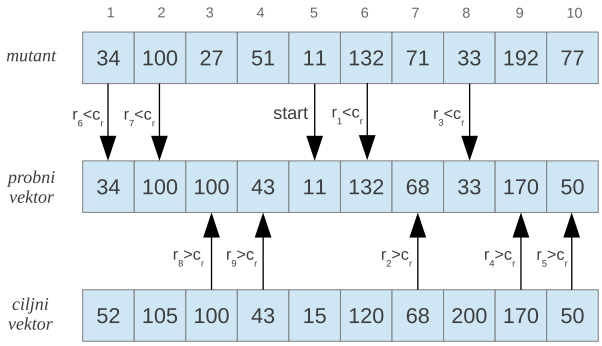


Slika 11.3: Drugi primjer izvedbe eksponencijalnog križanja kod algoritma diferencijske evolucije.

U oba ilustrirana slučaja, nakon slučajnog odabira prve komponente koja je preuzeta iz vektora mutanta u ciljni vektor, generirana su redom još četiri slučajna broja: r_1, r_2, r_3 i r_4. Brojevi r_1, r_2, r_3 bili su manji od C_r dok je r_4 prvi broj koji je bio veći od C_r: od tog trenutka sve daljnje komponente preuzete su iz ciljnog vektora.

11.4.2 Uniformno (binomno) križanje

Postupak uniformnog križanja konceptualno odgovara uniformnom križanju binarnih kromosoma kod genetskog algoritma; razlika je u tome što ovdje jedinice s kojima radimo nisu bitovi već komponente vektora. Postupak započinje odabirom komponente koja će se sigurno prekopirati iz vektora mutanta u probni vektor. Tu ćemo poziciju na grafičkom prikazu označiti kao *start*. odabranu komponentu prekopiramo u probni vektor. Potom se posredstvom slučajnog mehanizma uz vjerojatnost C_r za svaku preostalu komponentu pitamo hoćemo li je preuzeti iz vektora mutanta: ako je odgovor potvrdan, preuzimamo je iz vektora mutanta, inače je preuzimamo iz ciljnog vektora. Posljedica ovakve izvedbe križanja jest mogućnost izgradnje probnog vektora koji neke komponente preuzima iz vektora mutanta a neke iz ciljnog vektora ali potpuno slučajno; izvedba nije ograničena na preuzimanje jednog kontinuiranog podniza komponenata. Slika 11.4 ilustrira ovakvu izvedbu križanja.



Slika 11.4: Izvedba uniformnog križanja kod algoritma diferencijske evolucije.

Implementacijski, ovo križanje je prikazano s (11.2). Indeks komponente koja će sigurno biti preuzeta označena je s j_{rand}.

$$\vec{u}_{j,i,g} = \begin{cases} v_{j,i,g} & \text{ako je slucajno}(0.0, 1.0) \leq C_r \text{ ili } j == j_{rand} \\ x_{j,i,g} & \text{inace} \end{cases} \tag{11.2}$$

Ilustracija križanja ciljnog vektora $\vec{x}_{i,g}$ i vektora mutanta $\vec{v}_{i,g}$ za slučaj dvodimenzijskih vektora također je prikazan na slici 11.1. Teoretski, moguća su tri rezultata križanja – vektor $\vec{u}_{i,g}$ koji je sve komponente preuzeo iz vektora mutanta, vektor $\vec{u}'_{i,g}$ koji je prvu komponentu preuzeo iz vektora mutanta a drugu iz ciljnog vektora te vektor $\vec{u}''_{i,g}$ koji je prvu komponentu preuzeo iz ciljnog vektora a drugu iz vektora mutanta.

11.5 Operator selekcije

Nakon što se za svaki ciljni vektor vektor izgradi po jedan probni vektor, probni vektori se vrednuju. Operator selekcije primjenjuje se na parove (ciljni vektor, probni vektor), pri čemu se u sljedeću generaciju propušta probni vektor ako mu je dobrota bolja ili jednaka dobroti ciljnog vektora; u suprotnom propušta se ciljni vektor. Ako se obavlja postupak minimizacije funkcije f, djelovanje operatora selekcije možemo opisati izrazom (11.3).

$$\vec{x}_{i,g+1} = \begin{cases} u_{i,g} & \text{ako je } f(u_{i,g}) \leq f(x_{i,g}) \\ x_{i,g} & \text{inače} \end{cases} \tag{11.3}$$

Razlog propuštanja probnog vektora u sljedeću generaciju ako je dobrotom jednak ciljnom vektoru jest izbjegavanje stagnacije. Naime, ako je probni vektor jednako dobar kao i ciljni vektor, kvaliteta rješenja se ne mjenja, a u sljedeću generaciju se uvodi raznolikost koja će možda pridonijeti daljnjem napretku postupka optimizacije.

11.6 Čitav algoritam

Nakon prolaska kroz osnovne operatore koje koristi algoritam diferencijske evolucije, sada možemo opisati i konkrentu inačicu cjelokupnog algoritma; pretpostavit ćemo da izgradnju mutanta radimo kao što je prethodno opisano te da radimo uniformno križanje. Nakon što se izgradi početna populacija algoritam svaku jedinku trenutne populacije g proglašava ciljnim vektorom; za njega gradi vektor mutant te križanjem dobiva probni vektor. Ako se populacija sastoji od n jedinki, ovo će rezultirati s n probnih vektora. Nakon što su izgrađeni svi probni vektori, uspoređuju se dobrote odgovarajućih ciljnih vektora i probnih vektora (po parovima) te se u sljedeću generaciju propuštaju bolji. Time je izgrađena populacija $g + 1$ i postupak se ponavlja.

Implementacija ovog algoritma prikazana je pseudokodom 11.2, a grafički prikaz dan je na slici 11.5.

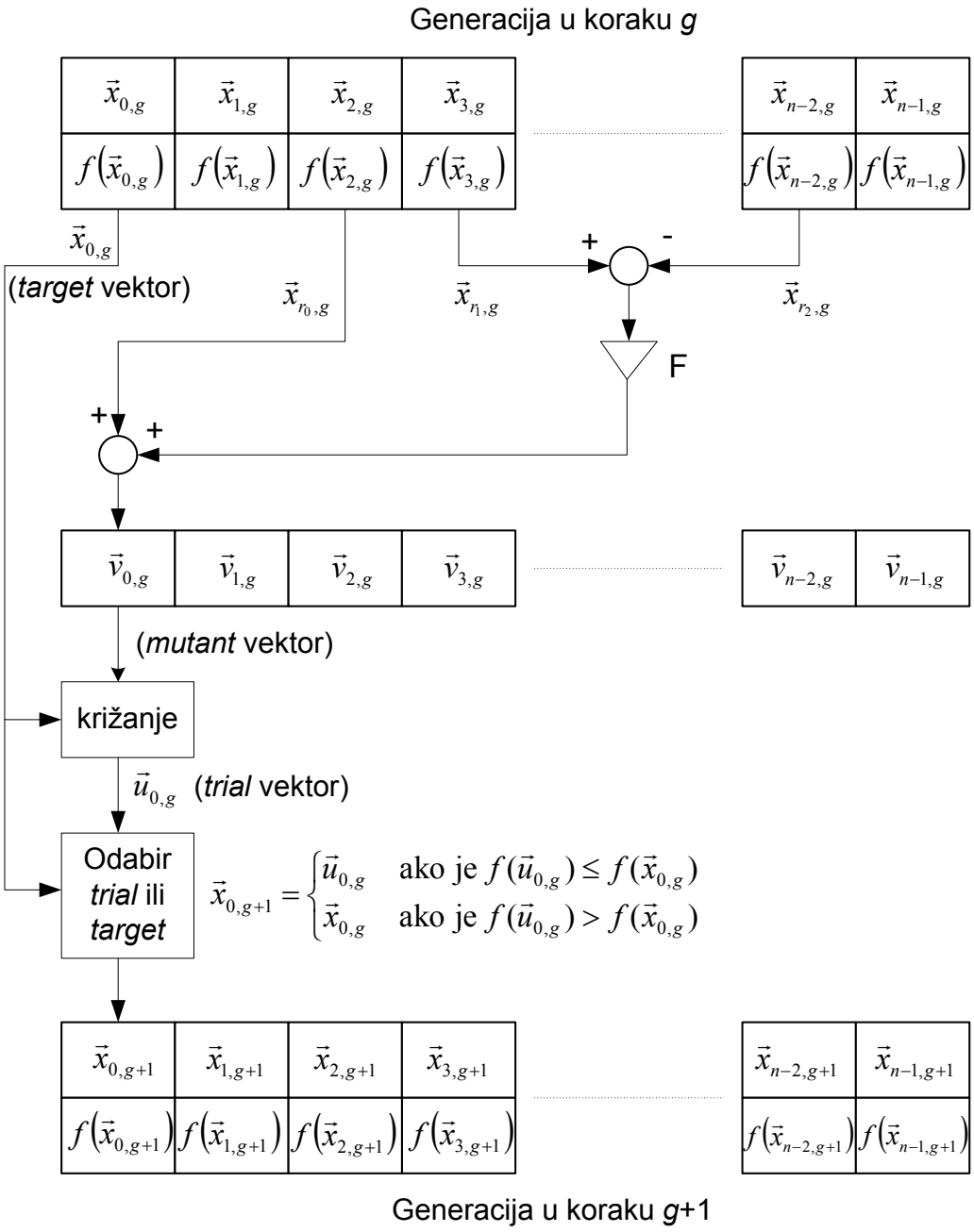
Pseudokod 11.2 Pseudokod diferencijske evolucije.

```
P = stvori_početnu_populaciju(n)
evaluiraj(P)
ponavljaj_dok_nije_kraj
  ponavljaj za i iz 0 do n-1
    ponavljaj r0=slucajno(0,n) dok r0 u {i}
    ponavljaj r1=slucajno(0,n) dok r1 u {i,r0}
    ponavljaj r2=slucajno(0,n) dok r2 u {i,r0,r1}
    jrand = slucajno(0,D)
    ponavljaj za j iz 0 do D
      v(j,i)=x(j,r0)+F*(x(j,r1)-x(j,r2))
    kraj
    ponavljaj za j iz 0 do D
      ako je slucajno(0,1)<=Cr ili j==jrand tada
        u(j,i)=v(j,i)
      inače
        u(j,i)=x(j,i)
      kraj
    kraj
  kraj
  ponavljaj za i iz 0 do n-1
    ako je f(u(i)) < f(x(i)) tada
      x(i) = u(i)
    kraj
  kraj
kraj
```

11.7 Strategije generiranja probnih vektora

Prethodno opisani način generiranja probnog vektora samo je jedan od niza mogućih. U svojoj knjizi iz 2005. Price i suautori navode četiri strategije generiranja probnih vektora [Price et al., 2005].

- DE/rand/1/bin
- DE/best/1/bin
- DE/target-to-best/1/bin
- DE/rand/1/either-or



Slika 11.5: Algoritam diferencijske evolucije – grafički prikaz.

Osim ovih, možemo razmatrati i inačice:

- DE/rand/1/exp
- DE/best/1/exp
- DE/target-to-best/1/exp
- DE/rand/2/bin
- DE/best/2/bin
- DE/target-to-best/2/bin
- DE/rand/2/exp
- DE/best/2/exp
- DE/target-to-best/2/exp
- ...

Da bismo razumjeli kakvu strategiju svaki od ovih naziva predstavlja, moramo pogledati opću strukturu naziva: DE/*bazni*/*broj-linearnih-kombinacija*/*vrsta-križanja*.

Komponenta *bazni* određuje način selekcije baznog vektora pri izgradnji mutanta. Ako je `rand`, bazni se vektor bira slučajno iz populacije; ako je `best`, kao bazni se vektor uzima najbolja jedinka trenutne populacije, ako je `target-to-best`, kao bazni vektor se uzima ciljni vektor koji je translatiran u smjeru najbolje jedinke populacije. Uz ove, mogli bismo raditi još niz drugih načina odabira baznog vektora.

Komponenta *broj-linearnih-kombinacija* određuje koliko linearnih kombinacija treba dodati baznom vektoru: ako je to 1, trebat ćemo još dvije jedinke iz populacije čiju ćemo skaliranu razliku nadodati baznom vektoru; ako je to 2, trebat ćemo još četiri jedinke iz populacije kako bismo mogli napraviti dvije skalirane razlike koje ćemo dodati baznom vektoru; ako je 3, trebat ćemo još šest jedinki iz populacije, ...

Konačno, poljednji parametar određuje kako se radi križanje mutanta i ciljnog vektora: ako je `bin`, radi se binomno (uniformno) križanje, ako je `exp`, radi se eksponencijalno križanje.

Strategije koje za izgradnju baznog vektora koriste najbolje rješenje populacije ili neko iz njega izvedenog odnosno strategije koje koriste mali broj linearnih kombinacija imat će veću brzinu konvergencije. Takve će strategije kod jednostavnih (posebice unimodalnih) funkcija imati odlične performanse; međutim, kod složenijih funkcija brzo i često će zapinjati u lokalnim optimumima. Strategije koje se oslanjaju na slučajni izbor baznog vektora te na dodavanje većeg broja linearnih kombinacija karakterzirat će sporija konvergencija ali i veća otpornost na zapinjanje u lokalnim optimumima. Stoga je za rješavanje konkretnih problema potrebno pronaći i odgovarajuću strategiju uz koju će ponašanje optimizacijskog algoritma biti zadovoljavajuće.

11.7.1 Strategija DE/rand/1/bin

Kod ove strategije kao bazni vektor odabire se slučajna jedinka iz populacije ($\vec{x}_{r_0,g}$). Koristi se jedna vektorska razlika a križanje se provodi uporabom zadane vjerojatnosti križanja C_r. Možemo pisati:

$$\vec{v}_{i,g} = \vec{x}_{r_0,g} + F \cdot (\vec{x}_{r_1,g} - \vec{x}_{r_2,g})$$

Križanje se provodi prema izrazu (11.2) koji je ponovno naveden u nastavku.

$$\vec{u}_{j,i,g} = \begin{cases} v_{j,i,g} & \text{ako je slucajno}(0.0, 1.0) \leq C_r \text{ ili } j == j_{rand} \\ x_{j,i,g} & \text{inače} \end{cases}$$

11.7.2 Strategija DE/best/1/bin

Kod ove strategije kao bazni vektor koristi se trenutno najbolja jedinka iz populacije ($\vec{x}_{\text{best}}$). Koristi se jedna vektorska razlika a križanje se provodi uporabom zadane vjerojatnosti križanja C_r. Preporuča se umjesto fiksne vrijednosti za F u svaku dimenziju uvesti malo odstupanje, tako da se prilikom za j-tu dimenziju koristi $F_j = F + 0.001 \cdot (\text{slucajno}_j(0.0, 1.0) - 0.5)$. Možemo pisati:

$$\vec{v}_{i,g} = \vec{x}_{\text{best}} + F \cdot (\vec{x}_{r_1,g} - \vec{x}_{r_2,g})$$

Križanje se provodi prema izrazu (11.2) koji je ponovno naveden u nastavku.

$$\vec{u}_{j,i,g} = \begin{cases} v_{j,i,g} & \text{ako je slucajno}(0.0, 1.0) \leq C_r \text{ ili } j == j_{rand} \\ x_{j,i,g} & \text{inače} \end{cases}$$

11.7.3 Strategija DE/target-to-best/1/bin

Kod ove strategije kao bazni vektor odabire se ciljni vektor kojem se dodaje skalirana razlika najbolje jedinke iz populacija i ciljnog vektora ($\vec{x}_{i,g} + F \cdot (\vec{x}_{\text{best}} - \vec{x}_{i,g})$). Koristi se jedna vektorska razlika a križanje se provodi uporabom zadane vjerojatnosti križanja C_r. Možemo pisati:

$$\vec{v}_{i,g} = \vec{x}_{i,g} + F \cdot (\vec{x}_{\text{best}} - \vec{x}_{i,g}) + F \cdot (\vec{x}_{r_1,g} - \vec{x}_{r_2,g})$$

Križanje se provodi prema izrazu (11.2) koji je ponovno naveden u nastavku.

$$\vec{u}_{j,i,g} = \begin{cases} v_{j,i,g} & \text{ako je slucajno}(0.0, 1.0) \leq C_r \text{ ili } j == j_{rand} \\ x_{j,i,g} & \text{inače} \end{cases}$$

11.7.4 Strategija DE/rand/1/either-or

Rad Price-a i drugih [Price et al., 2005] upućuje na postojanje problema kod koji je optimalan način generiranja problnih vektora uporaba čiste diferencijske mutacije; s druge pak strane postoje problemi kod kojih je optimalan način generiranja problnih vektora uporaba čiste rekombinacije. Stoga je definirana strategija *either-or* koja u skladu s propisanom vjerojatnosti P_F generira probne vektore uporabom diferencijske mutacije – izraz 11.4, a u $(1 - P_F)$ posto slučajeva probne vektore generira uporabom rekombinacije – izraz 11.5.

$$\vec{u}_{i,g} = x_{r_0,g} + F \cdot (x_{r_1,g} - x_{r_2,g}) \tag{11.4}$$

$$\vec{u}_{i,g} = x_{r_0,g} + K \cdot (x_{r_1,g} + x_{r_2,g} - 2 \cdot x_{r_0,g}) \tag{11.5}$$

Pri tome se preporuča koristiti $K = 0.5 \cdot (F + 1)$.

U svojoj knjizi Vitaliy Feoktistov daje još puno koncizniju podjelu strategija za generiranje vektora mutanata [of Solutions, 2006]. Prema njemu, sve se strategije mogu zapisati u općem obliku kao:

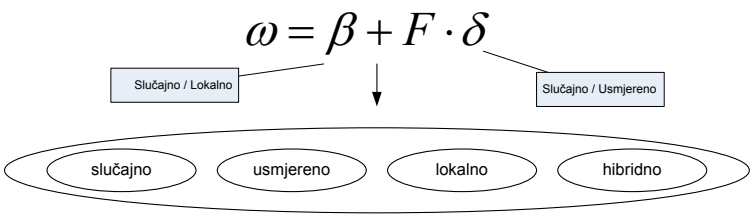
$$\omega = \beta + F \cdot \delta.$$

gdje je ω oznaka za vektor mutant, β oznaka za bazni vektor a δ oznaka za vektor diferencije. Različitim načinima odabira parametara β i δ moguće je dobiti različite razrede strategija, kao što je to ilustrirano na slici 11.6.

Četiri razreda koja definira Feoktistov navedena su u nastavku.

1. Razred *RAND* čine strategije kod kojih se pri stvaranju vektora mutanta ne uzima u obzir vrijednost evaluacijske funkcije.

2. Razred *RAND/DIR* čine strategije kod kojih se pri stvaranju vektora mutanta u obzir uzima vrijednost evaluacijske funkcije kako bi se odredio dobar smjer. Time se imitira gradijentni spust.



Slika 11.6: Algoritam diferencijske evolucije – opći oblik strategije generiranja mutanta.

3. Razred *RAND/BEST* čine strategije kod kojih se pri stvaranju vektora mutanta u obzir uzima trenutno najbolje rješenje. Često ovakvo pretraživanje sliči nasumičnom pretraživanju okolice najboljeg rješenja.

4. Razred *RAND/BEST/DIR* kombinira svojstva prethodne dvije grupe.

Primjere za svaku od ovih grupa čitatelj može potražiti u [of Solutions, 2006].

Bibliografija

D. E. I. S. of Solutions. *Feoktistov, Vitaliy*, volume 5 of *Optimization and Its Applications*. Springer, New York, Inc. Secaucus, NJ, USA, 2006.

K. Price. Genetic annealing. *Dr. Dobb's Journal*, 19(10):127–132, 1994.

K. Price and R. Storn. Differential evolution: A simple evolution strategy for fast optimization. *Dr. Dobb's Journal of Software Tools*, 22(4):18–24, 1997.

K. V. Price. Differential evolution vs. the functions of the 2nd iceo. In *Proc. of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 153–157, Indianapolis, IN, USA, April 1997.

K. V. Price. *New Ideas in Optimization*, chapter Differential Evolution. McGraw-Hill, London, 1999.

K. V. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution A Practical Approach to Global Optimization*. Natural Computing Series. Springer-Verlag, Berlin, Germany, 2005. URL `http://www.springer.com/west/home/computer/foundations?SGWID=4-156-22-32104365-0\&\#38;teaserId=68063\&\#38;CENTER\_ID=69103`.

R. Storn and K. Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, International Computer Science Institute, Berkeley, CA, 1995. Technical Report TR-95-012.

R. Storn and K. Price. Minimizing the real functions of the icec'96 contest by differential evolution. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 842–844, 1996.

R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

Poglavlje 12

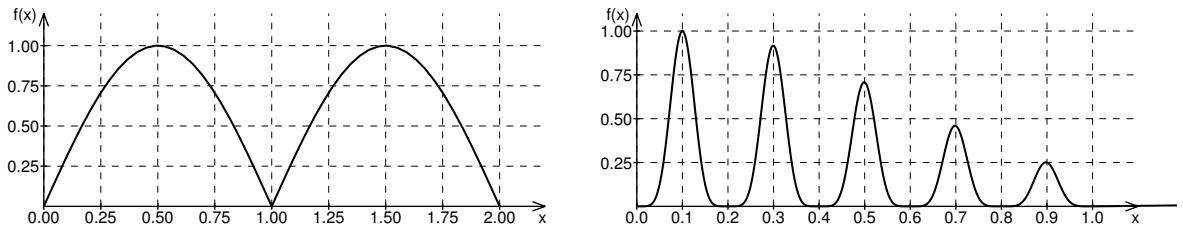
Optimizacija višemodalne funkcije

12.1 Uvod

Kako bi se osiguralo djelotvorno pretraživanje prostora rješenja, evolucijski algoritmi trebaju dva mehanizma: prikladno definiranu funkciju dobrote te selekcijski pritisak koji će osigurati da bolja rješenja imaju veću šansu producirati potomke i/ili preživjeti, čime direktno utječe na tijek pretraživanja. Bez selekcijskog pritiska (ili uz premali iznos) preraživanje se pretvara u nasumično. Prejaki selekcijski pritisak dovodi pak do fokusiranja populacije na vrlo mali potprostor čime se javlja gubitak genetske raznolikosti – a dovoljna genetska raznolikost nužna je kako bi se algoritmu omogućilo djelotvorno pretraživanje prostora rješenja. U suprotnom, nastupa prerana konvergencija i zapinjanje u lokalnom optimumu [Eshelman and Schaffer, 1991, Goldberg, 1989]. Stoga je jasno da su selekcijski pritisak i genetska raznolikost međusobno suprotstavljeni [Whitley, 1989]. U populacijskim optimizacijskim algoritmima stoga je vrlo važno pronaći dobar balans između selekcijskog pritiska i genetske raznolikosti – raznolikost mora biti dovoljna da podrži pronalazak dobrih potprostora u prostoru rješenja a selekcijski pritisak taman dovoljan da vodi populaciju u pravom smjeru ali ne i prejak kako se populacija ne bi prerano fokusirala i zaglavila u lokalnom optimumu [Mahfoud, 1995].

Optimizacija višemodalne funkcije predstavlja pak dodatni izazov pred optimizacijske algoritme. Višemodelane funkcije su funkcije koje imaju više globalnih optimuma (slika 12.1a) ili više dovoljno dobrih ekstrema (ne nužno globalnih optimuma, slika 12.1b). Kada govorimo o optimizaciji višemodalne funkcije, tada implicitno stavljamo naglasak na želju za pronalaskom više (ili čak svih) dovoljno dobrih rješenja. Ovaj zadatak neprikladan je za algoritme koji rade s jednim rješenjem, poput simuliranog kaljenja i slično (jasno je zašto). Međutim, populacijski algoritmi mogu se upogoniti za rješavanje optimizacije višemodalnih funkcija – kako imaju populaciju, pojedini dijelovi populacije mogu se fokusirati na pronalazak različitih optimuma.

Da bi ovo dobro funkcioniralo, očuvanje genetske raznolikosti još je važnije – ako se dopusti fokusiranje populacije, izgubit će se sva rješenja osim jednog. Većina populacijskih optimizacijskih algoritama u svojoj kanonskoj izvedbi loše se ponaša na problemima višemodalne optimizacije; naime, uslijed pojave poznate pod nazivom *genetski drift* relativno brzo će se izgubiti rješenja iz različitih dijelova



(a) Funkcija s dva globalna maksimuma

(b) Funkcija s jednim globalnim i više lokalnih maksimuma

Slika 12.1: Primjer višemodalnih funkcija

prostora pretraživanja i pojavit će se dominantno rješenje koje će biti jedan od optimuma. Kako bi se ovo spriječilo, razvijen je niz tehnika koje se direktno bave pokušajem očuvanja genetske raznolikosti, u želji da se različita dobra rješenja očuvaju. U nastavku ćemo opisati neke od tih postupaka. Kod višekriterijske optimizacije funkcija pozvat ćemo se na neke od ovih mehanizama jer će nam biti nužni za djelotvoran rad algoritama.

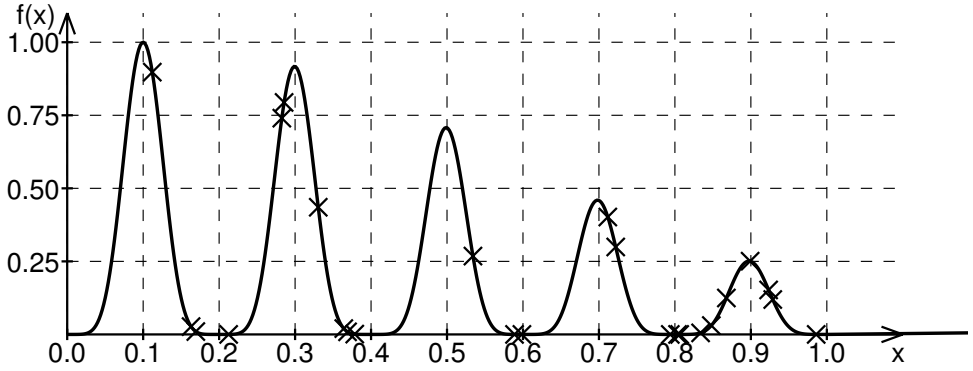
12.2 Postupci za očuvanje raznolikosti

U znanstvenoj literaturi danas se može pronaći niz algoritama razvijenih s ciljem očuvanja raznolikosti u populaciji. Čest primjer su različite vrste algoritama grupiranja (engl. *crowding algorithms*) [Jong, 1975, Mahfoud, 1995]. Osnovna ideja jest u populaciji dopuštati zamjenu samo onih jedniki koje su međusobno slične (genotipski, fenotipski) čime se pokušava spriječiti fokusiranje populacije i potiče razdioba populacije u više podpopulacija (takozvanih niša). Ovo je posebno prikladno za optimizaciju višemodalnih funkcija jer omogućava stvaranje podpopulacija oko pojedinih optimuma [Sareni and Krähenbühl, 1998]. Alternativa su postupci koji modificiraju dobrotu jedinki populacije temeljem gustoće populacije u okolici svake jedinke. Ako u okolici jedinke postoji više rješenja, dobrota takvih jedinki prividno će se smanjiti kako bi ih postupak izbora manje preferirao. Na taj način poticat će se razvoj populacija koje imaju veću raznolikost.

Za neke od tehnika navedenih u nastavku dat ćemo i prikaz rezultata optimizacije za funkciju:

$$f(x) = 2^{-2(\frac{x-0.1}{0.8})^2} \sin^6(5\pi x)$$

te početnu populaciju koja je prikazana na slici 12.2.



Slika 12.2: Početna populacija za postupke očuvanja raznolikosti

12.2.1 Zabrana unosa duplikata u populaciju

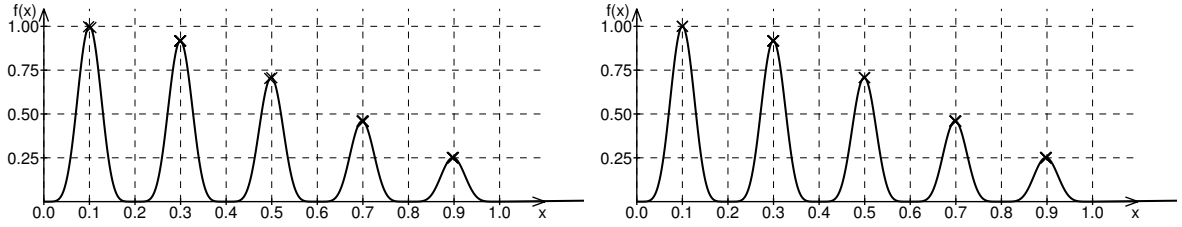
Jedan od najjednostavnijih pokušaja očuvanja genetske raznolikosti jest uporaba zabrane unosa duplikata u populaciju. Postupak uspoređuje jedinku koju treba dodati sa već postojećim jedinkama, i ako jedinka već postoji, ne dodaje se u populaciju. Ovaj postupak jednakost jedinki može promatrati bilo na genotipskoj razini, bilo na fenotipskoj razini.

12.2.2 Zabrana unosa jednakovrijednih jedinki u populaciju

Mala modifikacija prethodnog postupka jest uvođenje zabrane na unos više jedinki koje imaju identičnu vrijednost funkcije dobrote.

12.2.3 Ograničavanje roditelja

Ograničavanje roditelja (engl. *Mating restriction*) je pristup kod kojeg se prvi roditelj bira slučajno. Potom se traži kandidat za drugog roditelja. Iz populacije slučajno odabire neka jedinka i provjerava se je li genotipska (ili fenotipska) udaljenost do prvog roditelja prihvatljiva (tj. manja od nekog propisanog praga). Ako je, ta jedinka postaje drugi roditelj i može se proslijediti dalje na križanje i mutaciju. Ako nije, iz ostatka populacije ponovno se bira novi kandidat, i provjerava se udaljenost do prvog roditelja. Postupak se ponavlja sve dok se ili ne pronađe drugi roditelj, ili se provjeri čitav ostatak populacije i utvrdi da u njoj nema jedinke koja bi bila dovoljno blizu prvom roditelju. Ako nastupi ovaj posljednji slučaj, onda se kao drugi roditelj iz populacije slučajno odabire neka jedinka. Uz početnu populaciju prikazanu na slici 12.2 stanje nakon 50. i 100. generacija genetskog algoritma prikazano je na slikama 12.3a i 12.3b.

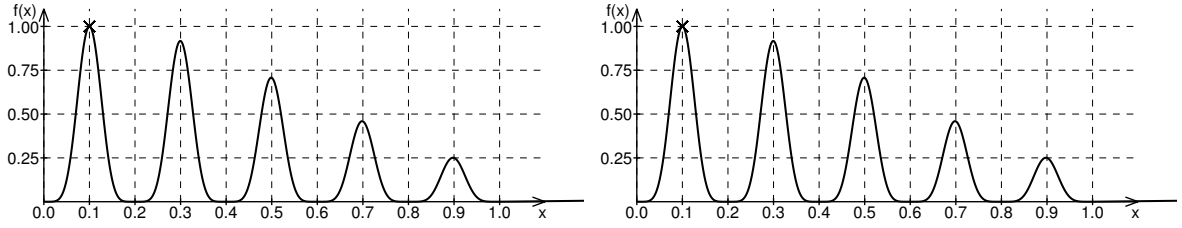


(a) Nakon 50 generacija (b) Nakon 100 generacija

Slika 12.3: Dinamika očuvanja raznolikosti kod algoritma ograničavanja roditelja

12.2.4 Algoritam s predodabirom

Algoritam s predodabirom (engl. *Preselection algorithm*) opisan je u [Cavicchio, 1970]. Iz populacije se odaberu dva roditelja R_1 i R_2 koji produciraju dijete D. Potom se dijete usporedi s oba roditelja. Ako dijete ima bolju vrijednost funkcije dobrote od lošijeg roditelja, zamijenit će lošijeg roditelja; u suprotnom se dijete odbacuje. Pretpostavka ovog algoritma jest da će roditelji davati genotipski (ili fenotipski) sličnu djecu pa se kandidati za zamjenu gledaju samo između roditelja. Uz početnu populaciju prikazanu na slici 12.2 stanje nakon 50. i 100. generacija genetskog algoritma prikazano je na slikama 12.4a i 12.4b.



(a) Nakon 50 generacija (b) Nakon 100 generacija

Slika 12.4: Dinamika očuvanja raznolikosti kod algoritma s predodabirom

Nešto kasnije dan je još čitav niz algoritama koji rade uz istu pretpostavku (vidi *Zamjena natjecanjem u obitelji*).

12.2.5 Ograničena turnirska selekcija

Ograničena turniska selekcija (engl. *Restricted tournament selection*, RTS) opisana je u [Harik, 1995]. Postupak se provodi na sljedeći način. Iz populacije se posredstvom slučajnog mehanizma odabiru dvije jedinke R_1 i R_2. Te jedinke postaju roditelji i produciraju dijete D uporabom križanja i mutacije. Potom se iz populacije nasumice odabere ω jedinki, pri čemu parametar ω predstavlja veličinu prozora.

Svih izvučenih ω jedinki uspoređuje se s djetetom D i odabire se ona jedinka R_3 koja je najsličnija djetetu D. Dijete D natječe se s odabranom jedinkom R_3 i ako je bolje, mijenja jedinku R_3 u populaciji (R_3 se izbacuje a D ubacuje u populaciju).

12.2.6 Zamjena najgore jedinke među najsličnijima

Zamjena najgore jedinke među najsličnijima (engl. *Worst among most similar replacement policy*, WAMS) opisana je u [Cedeño et al., 1994, Cedeño and Vemuri, 1999]. Postupak se provodi na sljedeći način. Neka je na neki način stvoreno dijete D. Iz populacije se slučajno izvlači C_f grupa rješenja (C_f – *crowding factor*), pri čemu je svaka grupa veličine C_s. Grupe pri tome ne moraju biti disjunktne – naprosto se C_f puta ponovi postupak izvlačenja C_s jedinki iz populacije. Rješenja iz svake grupe uspoređuju se s djetetom D i u svakoj se grupi identificira rješenje koje je najsličnije djetetu D. Time je identificirano C_f kandidata za zamjenu (u svakoj grupi po jedan). Sada se od tih C_f najsličnijih rješenja odabere ono koje ima najmanju vrijednost funkcije dobrote, i to se rješenje izbacuje iz populacije te se na njegovo mjesto ubacuje dijete D.

Kako se ova selekcija koristi kod eliminacijskog genetskog algoritma, vrijednost parametra C_f određuje selekcijski pritisak a vrijednost parametra C_s omjer natjecanja unutar niše i između niša [Cedeño and Vemuri, 1999]. Naime, porastom C_f bira se sve više jedinki (jer se bira više grupa) pa raste šansa da se odabere i potom eliminira najlošija jedinka; time prosječna dobrota populacije raste pa je i selekcijski pritisak veći. Porastom parametra C_s smanjuje se natjecanje između pojedinih grupa (a time i niša) jer imamo dosta jedinki pa je natjecanje između najlošijih. Smanjenjem parametra C_s raste jakost natjecanja između pojedinih grupa. Ova dva parametra nude mogućnost podešavanja omjera istraživanje nasuprot iskorištavanja trenutnih rješenja.

12.2.7 Zamjena natjecanjem u obitelji

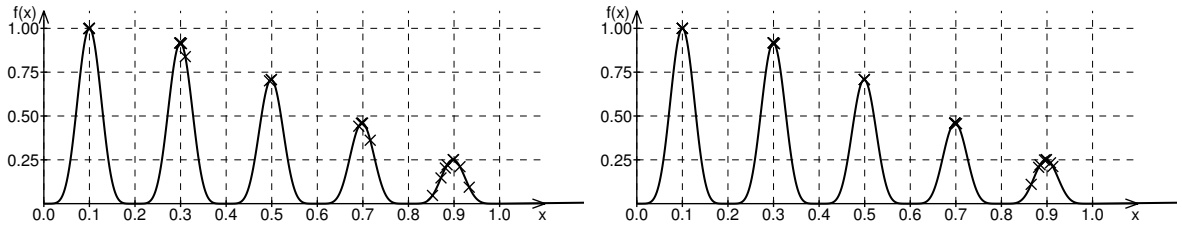
Zamjena natjecanjem u obitelji (engl. *Family competition replacement schemes*) predstavlja porodicu algoritama koji kreću od pretpostavke da će roditelji stvarati djecu koja su njima najsličnija, pa stoga provode natjecanje između djece i njihovih roditelja. Postupci koji pripadaju ovoj porodici su algoritam deterministčkog grupiranja, vjerojatnosti algoritam grupiranja, elitistička rekombinacija, reprodukcija sa zadržavanjem najboljeg te selekcija temeljena na korelaciji u obitelji.

12.2.8 Algoritam grupiranja

Algoritam grupiranja (engl. *Crowding model*) opisan je u [Jong, 1975]. Nakon što se stvori dijete, potrebno je odlučiti koju će jedinku iz populacije to dijete zamijeniti. U tu svrhu iz populacije se nasumice odabire CF (engl. *Crowding Factor*) jedinki, i dijete se uspoređuje s tim jedinkama. Dijete će zamijeniti onu jedinku od izvučenih CF koja mu je najsličnija. Ako je CF premali, javlja se velika pogreška uzorkovanja – često će se dogoditi da je svih CF izvučenih jedinki vrlo različito od djeteta (iako u populaciji postoje slične jedinke) i dijete će iz populacije izbaciti jedinku koja mu nije slična, čime će zapravo smanjiti genetsku raznolikost u populaciji. Postavljanjem CF na preveliku vrijednost (ekstreman slučaj je $CF = N$) gubi se previše vremena na usporedbe i algoritam postaje neefikasan. Uz početnu populaciju prikazanu na slici 12.2 stanje nakon 50. i 100. generacija genetskog algoritma prikazano je na slikama 12.5a i 12.5b.

Algoritam deterministčkog grupiranja

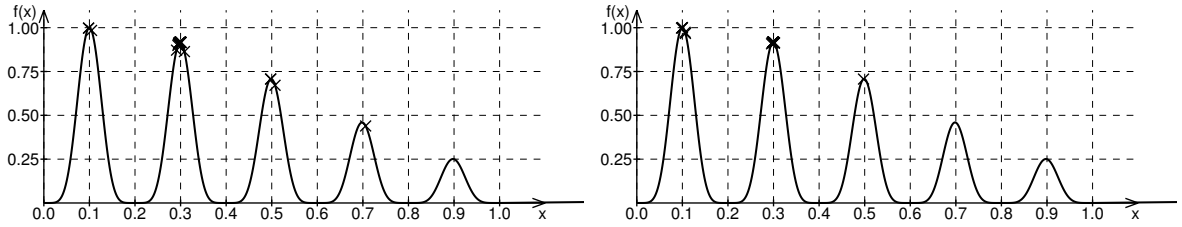
Algoritam deterministčkog grupiranja (engl. *Deterministic crowding*) opisan je u [Mahfoud, 1992]. Algoritam radi na sljedeći način. Neka je veličina populacije $N = 2 \cdot k$ (veličina populacije je parna). Populacija se podijeli u $k = \frac{N}{2}$ parova roditelja. Svaki par križanjem i mutacijom stvara po dva potomka. Za svaki se par roditelja R_1 i R_2 te potomaka koje su oni stvorili D_1 i D_2 gledaju sličnosti djeteta i roditelja. Jedna mogučnost je gledati sličnost između (R_1, D_1) i (R_2, D_2). Druga mogućnost je gledati sličnost između (R_1, D_2) i (R_2, D_1). Odabire se ono uparivanje koje ima veću ukupnu sličnost. Potom se natječu upareni roditelj i dijete; ako dijete ima veći iznos funkcije dobrote u odnosu na roditelja, dijete ulazi u populaciju i iz populacije istiskuje tog roditelja; u suprotnom se dijete odbacuje



(a) Nakon 50 generacija (b) Nakon 100 generacija

Slika 12.5: Dinamika očuvanja raznolikosti kod algoritma grupiranja

i roditelj ostaje u populaciji. Ako s $d(x, y)$ označimo mjeru genotipske ili fenotipske udaljenosti, odlučivanje o načinu uparivanja svodi se provjeru $d(R_1, D_1) + d(R_2, D_2) < d(R_1, D_2) + d(R_2, D_1)$. Ako je lijeva strana manja, gleda se uparivanje (R_1, D_1) i (R_2, D_2); ako je desna strana manja, gleda se uparivanje (R_1, D_2) i (R_2, D_1). Uz početnu populaciju prikazanu na slici 12.2 stanje nakon 50. i 100. generacija genetskog algoritma prikazano je na slikama 12.6a i 12.6b.



(a) Nakon 50 generacija (b) Nakon 100 generacija

Slika 12.6: Dinamika očuvanja raznolikosti kod algoritma deterministčkog grupiranja

Vjerojatnosni algoritam grupiranja

Vjerojatnosni algoritam grupiranja (engl. *Probabilistic crowding*) opisan je u [Mengshoel and Goldberg, 1999]. Algoritam je modifikacija algoritma deterministčkog grupiranja. Jedina razlika je način kako se određuje pobjednik turnira koji se drži između roditelja i djeteta. Kod algoritma deterministčkog grupiranja pobjednik je jedinka s većim iznosom funkcije dobrote; kod vjerojatnosnog algoritma grupiranja pobjednik se određuje uporabom proporcionalne selekcije (gleda se dobrota roditelja i dobrota djeteta i svakom se proporcionalno dodijeli vjerojatnost odabira).

Elitistička rekombinacija

Elitistička rekombinacija (engl. *Elitist recombination*) opisan je u [Thierens, 1997]. Algoritam je također modifikacija algoritma deterministčkog grupiranja. Razlika je što ovaj algoritam od četiri jedinke (dva roditelja i dva djeteta) odabire dvije najbolje jedinke kao jedinke koje ulaze/ostaju u populaciji.

Reprodukcija sa zadržavanjem najboljeg

Reprodukcija sa zadržavanjem najboljeg (engl. *Keep-best reproduction*) opisan je u [Wiese and Goodwin, 1999]. Algoritam je također modifikacija algoritma deterministčkog grupiranja. Algoritam zadržava najboljeg roditelja i najbolje dijete kao jedinke koje preživljavaju/ulaze u populaciju.

Selekcija temeljena na korelaciji u obitelji

Selekcija temeljena na korelaciji u obitelji (engl. *Correlative family-based selection*) opisan je u [Wiese and Goodwin, 1999]. Algoritam je također modifikacija algoritma deterministčkog grupiranja. Kod

ovog algoritma od četiri jedinke deterministički se izabire najbolje jedinka i potom se od preostale tri izabire ona koja je na najvećoj udaljenosti od izabrane najbolje jedinke.

12.2.9 Algoritam s raspodijelom funkcije dobrote

Algoritam s raspodijelom funkcije dobrote (engl. *Sharing function*) opisan je u [Goldberg and Richardson, 1987] i funkcionira drugačije od svih do sada opisanih algoritama. Osnovna ideja algoritma je modificirati dobrotu svake jedinke temeljem gustoće populacije u okolici te jedinke i potom koristiti proporcionalnu selekciju za odabir roditelja.

Neka je s $d(i,j)$ označena genotipska (ili fenotipska) udaljenost jedinki i i j. Definira se funkcija dijeljenja $sh(d(i,j)) \in [0,1]$ čija je vrijednost proporcionalna sličnosti jedinki i i j. Ako su jedinke i i j vrlo slične, vrijednost funkcije dijeljenja će biti visoka (bliska 1); ako su jedinke dovoljno različite, vrijednost funkcije dijeljenja će biti 0. Što znači dovoljno različito definira se pragom σ_{share}. Za jedinke čija je udaljenost manja od praga σ_{share} vrijednost funkcije dijeljenja će biti veća od 0; za jedinke čija je udaljenost veća ili jednaka pragu σ_{share} vrijednost funkcije dijeljenja bit će 0. U [Goldberg and Richardson, 1987] je za izračun funkcije dijeljenja predložen sljedeći izraz:

$$sh(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_{\text{share}}}\right)^{\alpha} & \text{za } d < \sigma_{\text{share}} \\ 0 & \text{inače.} \end{cases} \tag{12.1}$$

Dakako, argument d ove funkcije je udaljenost između dvije promatrane jedinke. Konstanta α omogućava podešavanje raspodjele funkcije dijeljenja s obzirom na omjer $\frac{d}{\sigma_{\text{share}}}$. Za $\alpha = 1$ iznos funkcija dijeljenja linearno pada od 1 do 0 kako se omjer $\frac{d}{\sigma_{\text{share}}}$ mijenja od 0 do 1. Za iznose α koji su veći od 1 funkcija dijeljenja na početku sporije pada i čitavo vrijeme ima vrijednosti koje su iznad linearnog pada. Za vrijednosti α koje su manje od 1 funkcija sve brže i brže pada prema 0.

U populaciji od N jedinki za svaku se jedinku $i \in \{1,\ldots,N\}$ računa *gustoća niše*, nc, koja je definirana tom jedinkom:

$$nc_i = \sum_{j=1}^{N} sh(d(i,j)). \tag{12.2}$$

Za jedinke koje u svojoj okolici do na udaljednosti σ_{share} nemaju niti jednu drugu jedinku, gustoća niše će biti 1 što se vidi ako prethodnu sumu raspišemo:

$$nc_i = sh(d(i,i)) + \sum_{j=1, j\neq i}^{N} sh(d(i,j)).$$

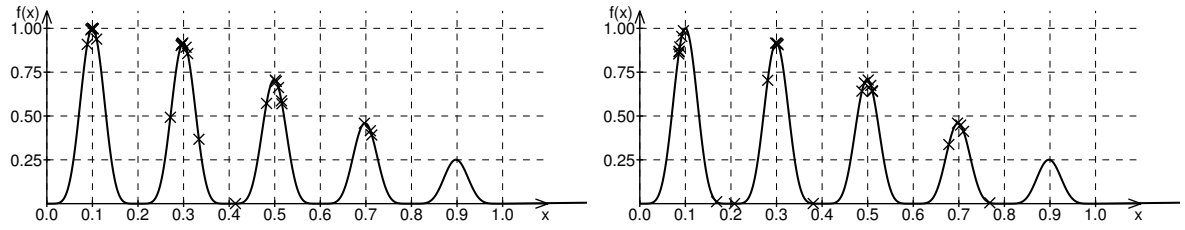
Kako je $d(i,i) = 0$, slijedi da je $sh(d(i,i)) = 1$. Za sve ostale $j \neq i$, s obzirom da je $d(i,j) > \sigma_{\text{share}}$ slijedi $sh(d(i,j)) = 0$, te je čitava suma jednaka upravo prvom članu i iznosi 1. Ako u okolici jedinke ima drugih jedinki, gustoća niše bit će veća od 1, i to to veća što ima više jedinki unutar udaljenosti σ_{share} i što su te jedinke bliže jedinki za koju se računa gustoća niše.

Jednom kad su za sve jedinke izračunate gustoće niše, uz pretpostavku da je s f_i označena dobrota i-te jedinke, svakoj jedinki pridružujemo *dijeljenu vrijednost dobrote* prema izrazu:

$$f_i' = \frac{f_i}{nc_i}. \tag{12.3}$$

Iz izraza je sada jasno vidljivo kako ova tehnika radi: jedinke koje su na području niske gustoće zadržat će punu vrijednost funkcije dobrote; jedinke koje su u guščim područjima imat će umanjenu vrijednost funkcije dobrote – to više što je gustoća područja veća.

Važno je napomenuti da je za dobar rad ovog algoritma nužno koristiti neki oblik proporcionalne selekcije. Uz početnu populaciju prikazanu na slici 12.2 stanje nakon 50. i 100. generacija genetskog algoritma prikazano je na slikama 12.7a i 12.7b.



(a) Nakon 50 generacija (b) Nakon 100 generacija

Slika 12.7: Dinamika očuvanja raznolikosti kod algoritma s raspodjelom funkcije dobrote

Specifičnosti pristupa

Možda bolji podnaslov bi bio – opasnosti pri radu s dijeljenjem funkcije dobrote. Opišimo jedan nezgodan problem koji se može javiti. Pretpostavimo da u populaciji imamo jedinke koje imaju relativno blisku dobrotu; primjerice, neka se populacija sastoji od jedinki od kojih najbolja ima iznos dobrote jednak 5 a najlošija iznos dobrote jednak 3. Pretpostavimo sada da dobrotu 5 imaju dvije jedinke koje su vrlo bliske, tako da je za njih gustoća niše jednaka 2. Neka je ona najlošija jedinka dovoljno izolirana tako da je za nju gustoća niše jednaka 1. Postupkom dijeljenja funkcije dobrote dvije najbolje jedinke dobit će dobrotu $\frac{5}{2} = 2.5$ dok će najlošija jedinka dobiti dobrotu $\frac{3}{1} = 3$, čime će algoritam preferirati najlošija rješenja i gubiti ona dobra. Kako bi se ovo spriječilo, algoritam dijeljenja funkcije dobrote treba primijeniti ne na originalne vrijednosti funkcije dobrote već na translatirane (i po potrebi još obrađene) vrijednosti. Jedna mogućnost je sljedeća. Neka je s δ označena minimalna vrijednost funkcije dobrote u trenutnoj populaciji:

$$\delta = \min(f_1, \ldots, f_N).$$

Svakoj jedinki pridružit ćemo novi iznos funkcije dobrote $f_i^{'}$ na sljedeći način:

$$f_i^{'} = (f_i - \delta)^{\phi}$$

pri čemu je ϕ parametar koji nam omogućava da kontroliramo koliki se naglasak stavlja na veće vrijednosti funkcije dobrote. Na ovaj način najgoroj jedinki ćemo pridružiti vrijednost dobrote 0 tako da se prethodno opisani scenarij preferiranja najgoreg rješenja ne može dogoditi.

Uz opisani pristup, u znanstvenoj je literaturi moguće pronaći i modifikacije koje uvode vremenski promjenjivo potenciranje razlike $(f_i - \delta)$ te dodatne modifikacije kojima se može kontrolirati utjecaj optimuma različitih iznosa dobrota.

Nakon što se na ovaj način odrede vrijednosti dobrota za sve jedinke, algoritam dijeljenja funkcije dobrote se primjenjuje nad njima a ne nad izvornim vrijednostima funkcije dobrote.

Bibliografija

D. Cavicchio. *Adaptive Search Using Simulated Evolution.* PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1970.

W. Cedeño and V. R. Vemuri. Analysis of speciation and niching in the multi-niche crowding ga. *Theor. Comput. Sci.*, 229(1):177–197, 1999.

W. Cedeño, V. R. Vemuri, and T. Slezak. Multi-niche crowding in genetic algorithms and its application to the assembly of dna restriction-fragments. *Evolutionary Computation*, 2(4):321–345, 1994.

L. J. Eshelman and D. J. Schaffer. Preventing premature convergence in genetic algorithms by preventing incest. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122. San Francisco, CA: Morgan Kaufmann, 1991.

D. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic algorithms and their Application*, pages 41–49, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

G. R. Harik. Finding multimodal solutions using restricted tournament selection. In L. J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 24–31, San Mateo, CA, 1995. Morgan Kaufmann. ISBN 1-55860-370-0.

K. D. Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1975.

S. Mahfoud. Crowding and preselection revised. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 27–36, Amsterdam, 1992. Elsevier.

S. W. Mahfoud. *Niching Methods for Genetic Algorithms.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

O. J. Mengshoel and D. E. Goldberg. Probabilistic crowding: Deterministic crowding with probabilisitic replacement. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 409–416, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. ISBN 1-55860-611-4.

B. Sareni and L. Krähenbühl. Fitness sharing and niching methods revisited. *IEEE Trans. Evolutionary Computation*, 2(3):97–106, 1998.

D. Thierens. Selection schemes, elitist recombination, and selection intensity. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 152–159, San Francisco, CA, USA, 1997. Morgan Kaufmann.

D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In D. J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. San Francisco, CA: Morgan Kaufmann, 1989.

K. C. Wiese and S. D. Goodwin. Convergence characteristics of keep-best reproduction. In *Selected Areas in Cryptography*, pages 312–318, 1999.

Dio II

Višekriterijska optimizacija

Poglavlje 13

Pareto optimalnost

Do sada smo se bavili problemima jednokriterijske optimizacije. Stoga ćemo najprije definirati problem višekriterijske optimizacije, usporediti ga s problemom jednokriterijske optimizacije i pogledati koje su posljedice uvođenja više kriterija.

Problem višekriterijske optimizacije. Općeniti problem višekriterijske optimizacije definiran je na sljedeći način.

$$\begin{array}{lll} \text{Minimiziraj / maksimiziraj} & f_m(\vec{x}), & m = 1, 2, \ldots, M \\ \text{uz zadovoljenje ograničenja} & g_j(\vec{x}) \geq 0, & j = 1, 2, \ldots, J \\ & h_k(\vec{x}) = 0, & k = 1, 2, \ldots, K \\ & x_i^L \leq x_i \leq x_i^U, & i = 1, 2, \ldots, n. \end{array}$$

Rješenje $\vec{x}$ je vektor decizijskih varijabli $\vec{x} = \{x_1, x_2, \ldots, x_n\}$. Prvi i drugi skup ograničenja prestavljaju skup nejednakosti odnosno jednakosti koje moraju biti zadovoljene za sva prihvatljiva rješenja. Treći skup ograničenja definira donju i gornju granicu na vrijednosti koje decizijske varijable smiju poprimiti. Ova tri skupa ograničenja definiraju prostor prihvatljivih rješenja. U tom prostoru svako rješenje $\vec{x}$ je jedna točka.

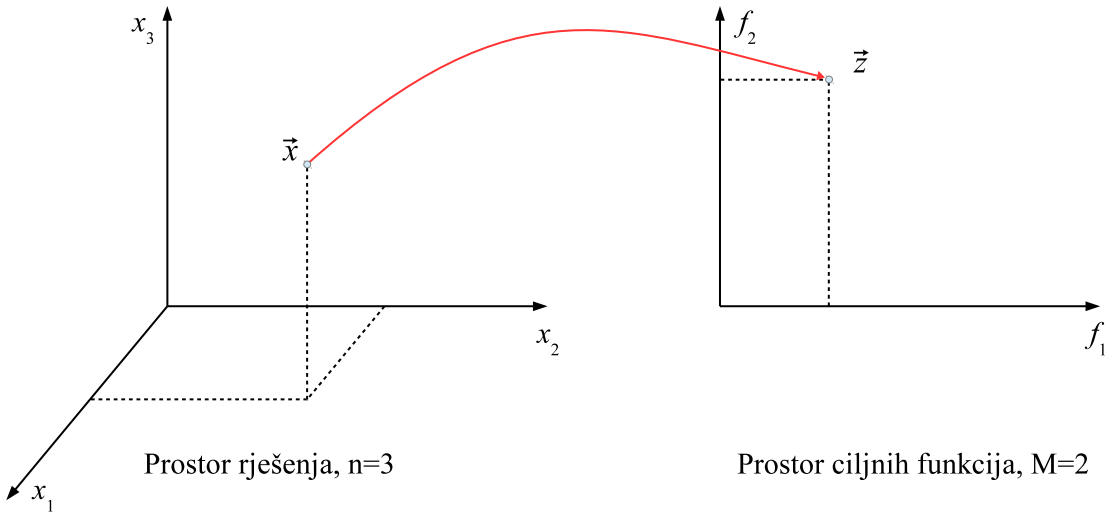
Na prvi pogled, jedina razlika u odnosu na problem jednokriterijske optimizacije je prvi redak: više nemamo jednu funkciju koju je potrebno optimirati, već imamo skup of M funkcija koje je potrebno optimirati. Pri tome problem može biti formuliran na način da je neke od tih M funkcija potrebno minimizirati a neke maksimizirati. Kako se, bez gubitka općenitosti, problem minimizacije funkcije g može svesti na problem maksimizacije funkcije $-g$ (i obratno), problem smo mogli definirati i na način da se traži maksimizacija (ili minimizacija) *svih* M funkcija.

13.1 Višedimenzijski prostor ciljnih funkcija

Jedna od posljedica prethodne definicije višekriterijske optimizacije jest da je prostor ciljnih funkcija (tj. engl. *objective space*) višedimenzijski. Sukladno prethodnoj definiciji, taj je prostor M dimenzijski jer imamo upravo M ciljnih funkcija (odnosno kriterija). Istovremeno, rješenja se pretražuju u prostoru rješenja (engl. *solution space*) koji je n-dimenzijski. Uočimo sada da za svako rješenje $\vec{x}$ iz n-dimenzijskog prostora postoji preslikavanje u točku $\vec{z}$ u M-dimenzijskom prostoru ciljnih funkcija: $(x_1, \ldots, x_n) \rightarrow (f_1, \ldots, f_M)$; ono je definirano upravo s M ciljnih funkcija (vidi sliku 13.1).

Važno je biti svjestan postojanja ovih prostora jer različiti algoritmi višekriterijske optimizacije pojedine postupke provode ili u jednom ili u drugom. Primjerice, ako algoritmu treba informacija koliko sa neka dva rješenja bliska, to se može gledati u prostoru rješenja (pa gledamo genotipsku ili fenotipsku udaljenost dvaju rješenja) ili u prostoru ciljnih funkcija.

Također, iz činjenice da je prostor ciljnih funkcija višedimenzijski, slijedi da ako se rješenje $\vec{x}^{(1)}$ preslika u prostor ciljnih funkcija kao $\vec{z}^{(1)}$ a rješenje $\vec{x}^{(2)}$ preslika u prostor ciljnih funkcija kao $\vec{z}^{(2)}$, generalno govoreći ne znamo kako odati odgovor na pitanje: je li $\vec{x}^{(1)}$ bolji ili lošiji od $\vec{x}^{(2)}$. Problem je u činjenici da su njihove dobrote $\vec{z}^{(1)}$ i $\vec{z}^{(2)}$ vektori – a vektore općenito ne znamo uspoređivati. Problem je što je nad vektorima definirana relacija parcijalnog uređaja – neki vektori jesu usporedivi dok drugi



Slika 13.1: Prostor rješenja i prostor ciljnih funkcija kod višekriterijske optimizacije

nisu. Primjerice, ako radimo minimizaciju prema svim kriterijima, i imamo dva rješenja: $\vec{z}^{(1)} = (5, 3)$ i $\vec{z}^{(2)} = (2, 1)$, evidentno je da je $\vec{z}^{(2)}$ bolje od $\vec{z}^{(1)}$ jer je po svim kriterijima bolje. Međutim, koje je od sljedeća dva rješenja bolje: $\vec{z}^{(1)} = (5, 1)$ ili $\vec{z}^{(2)} = (2, 3)$? Ovdje ne možemo dati odgovor jer rješenja nisu međusobno usporediva.

13.2 Definiranje potpunog uređaja nad rješenjima

Jedna od tehnika rješavanja problema višekriterijske optimizacije temelji se uvođenju direktne usporedivosti dva rješenja. U praksi to se uobičajeno radi ili uvođenjem transformacijom prostora ciljih funkcija u jednodimenzijski prostor, ili uvođenjem prioriteta među kriterijske funkcije.

13.2.1 Svođenje na jednodimenzijski prostor ciljnih funkcija

Svođenje točaka iz M-dimenzijskog prostora u 1-dimenzijski prostor obavlja se uporabom neke od mogućih transformacija. Najjednostavnija tehnika koju ćemo ovdje spomenuti jest definiranje linearne kombinacije, kako prikazuje izraz (13.1).

$$h = \sum_{i=1}^{M} \omega_i \cdot f_i \tag{13.1}$$

Vrijednosti ω_i su pri tome težinski faktori koji omogućavaju podešavanje mjere u kojoj će različite ciljne funkcije imati utjecaja na ukupno izračunatu dobrotu. Treba uočiti da ovo preslikavanje nije reverzibilno, odnosno da se gube informacije: više različitih vektora $\vec{z} = (f_1, \ldots, f_M)$ preslikat će se u istu skalarnu vrijednost h. Time niti optimizacijski algoritam neće biti u stanju razlikovati takva rješenja, što često nije poželjno svojstvo. Treba uočiti još jednu posljedicu ovakvog preslikavanja: linearna kombinacija uprosječuje ukupni iznos dobrote: uz fiksne težine rješenje koje je po prvom kriteriju loše a po drugom odlično, rješenje koje je po prvom kriteriju odlično a po drugom loše te rješenje koje je po prvom i drugom kriteriju prosječno dobro mogu biti preslikani u istu skalarnu vrijednost.

Općenitiji slučaj ovakvog kombiniranja možemo zapisati kako je prikazano izrazom (13.2):

$$h = \sum_{i=1}^{M} \psi_i(f_i) \tag{13.2}$$

gdje su ψ_i proizvoljne funkcije. Odabirom $\psi_i(\phi) = \omega_i \cdot \phi$ dobivamo izraz (13.1). Međutim, porast pojedinih ciljnih funkcija možemo preslikavati i nelinearno, primjerice postavljanjem $\psi_i(\phi) = \omega_i \cdot \phi^{\beta_i}$ gdje su ω_i i β_i koeficijenti koji definiraju nelinearno preslikavanje.

Jednom kada dobrotu svakog rješenja iz M-dimenzijskog prostora možemo možemo preslikati u skalar, automatski dobivamo mogućnost usporedbe dobrote rješenja jer skalare znamo uspoređivati.

13.2.2 Uvođenje prioriteta u prostor ciljnih funkcija

Umjesto svođenjem dobrote na skalare, usporedivost dva rješenja po dobroti možemo provesti uvođenjem prioriteta u ciljne funkcije.

Uspoređivanje prema prioritetima. Pretpostavimo da su kriteriji u problemu višekriterijske optimizacije zadani (numerirani) počev od najvažnijeg pa do najmanje važnog. Neka rješenje $\vec{x}^{(1)}$ ima pridruženu dobrotu $\vec{z}^{(1)} = \{z_1^{(1)}, \ldots, z_M^{(1)}\}$ a rješenje $\vec{x}^{(2)}$ pridruženu dobrotu $\vec{z}^{(2)} = \{z_1^{(2)}, \ldots, z_M^{(2)}\}$.
Rješenja $\vec{x}^{(1)}$ i $\vec{x}^{(2)}$ su jednaka ako $\forall i \in \{1, \ldots, M\}$ vrijedi $z_i^{(1)} = z_i^{(2)}$.
Rješenje $\vec{x}^{(1)}$ je bolje od rješenja $\vec{x}^{(2)}$ ako $\exists j \in \{1, \ldots, M\}, \forall i \in \{1, \ldots, M\} \wedge i < j$ vrijedi $z_i^{(1)} = z_i^{(2)} \wedge z_j^{(1)} > z_j^{(2)}$.
Rješenje $\vec{x}^{(1)}$ je lošije od rješenja $\vec{x}^{(2)}$ ako $\exists j \in \{1, \ldots, M\}, \forall i \in \{1, \ldots, M\} \wedge i < j$ vrijedi $z_i^{(1)} = z_i^{(2)} \wedge z_j^{(1)} < z_j^{(2)}$.

Ovakvom procedurom opet smo definirali postupak temeljem kojeg možemo usporediti svaka dva rješenja i time koristiti uobičajene izvedbe algoritama za jednokriterijsku optimizaciju.

Treba uočiti da oba opisana pristupa imaju svojih mana. Za početak, kombiniranjem više funkcija cilja u skalarnu vrijednost ne možemo razlikovati rješenja koja su na različit način kvalitetna. Evo ilustracije: potrebno je proračunati rutu kojom treba prevoziti teret između dva skladišta koja se nalaze u različitim gradovima; ovisno o tome kojom se vrstom cesta ide, prijevoz tereta će nas koštati različito no i vrijeme transporta će biti različito. Ako teret dominantno usmjeravamo autocestama, imat ćemo veliki trošak (cestarine, brza vožnja troši više goriva i slično) ali će prijevoz trajati kraće; ako biramo lokalne ceste trošak prijevoza će biti manji (nema cestarina, manje su brzine pa time i potrošnja goriva) ali će prijevoz trajati duže. Opisani optimizacijsi problem je očito višekriterijski (želimo smanjiti trajanje prijevoza i troškova) i ima kontradiktorne kriterije (smanjiti vrijeme prijevoza znači birati autoceste i voziti brže što znači veći trošak; smanjiti trošak znači birati lokalne ceste i manje brzine što znači dulje vrijeme prijevoza). Kod takvih problema kao rezultat optimizacijskog postupka želimo ponuditi više mogućih "najboljih" rješenja: ono s jako visokim troškom ali izuzetno kratkim vremenom transporta, ono koje ima nešto niže troškove ali zato nešto dulje vrijeme, ..., pa sve do rješenja koje ima izuzetno niske troškove ali zato dugo vrijeme prijevoza. U tom slučaju krisnik je taj koji će temeljem niza predloženih najboljih rješenja izabrati ono koje mu najviše odgovara.

Prvi korak u razvoju takvih algoritama jest ne koristiti prethodne tehnike za uvođenje usporedivosti rješenja, već se poslužiti pojmom pareto-optimalnosti koji ćemo definirati u nastavku.

13.3 Koncept dominacije

Koncept dominacije definiran je nad vektorima na sljedeći način.

Dominacija. Neka rješenje $\vec{x}^{(1)}$ ima pridruženu dobrotu $\vec{z}^{(1)} = \{z_1^{(1)}, \ldots, z_M^{(1)}\}$ a rješenje $\vec{x}^{(2)}$ pridruženu dobrotu $\vec{z}^{(2)} = \{z_1^{(2)}, \ldots, z_M^{(2)}\}$, pri čemu je $z_j^{(i)} = f_j(\vec{x}^{(i)})$. Rješenje $\vec{x}^{(1)}$ dominira nad rješenjem $\vec{x}^{(2)}$ ako su zadovoljena sljedeća dva uvjeta (oba).

- Rješenje $\vec{x}^{(1)}$ je u svim komponentama bolje ili jednako rješenju $\vec{x}^{(2)}$, tj. vrijedi: $\forall j \in \{1, \ldots, M\} : z_j^{(1)} \geq z_j^{(2)}$.
- Rješenje $\vec{x}^{(1)}$ je u barem jednoj komponenti strogo bolje od rješenja $\vec{x}^{(2)}$, tj. vrijedi: $\exists j \in \{1, \ldots, M\} : \quad z_j^{(1)} > z_j^{(2)}$.

Temeljem ove definicije jasno je da rješenja $\vec{x}^{(1)}$ i $\vec{x}^{(2)}$ mogu biti u relaciji dominacije (prvo rješenje može dominirati nad drugim ili pak drugo rješenje može dominirati nad prvim) ili ne moraju biti. Evo primjera.

$\vec{x}^{(1)}$	$\vec{x}^{(2)}$	$\vec{z}^{(1)}$	$\vec{z}^{(2)}$	dominacija?
$(1,2,3)$	$(1,2,4)$	$(5,6)$	$(3,1)$	$\vec{x}^{(1)}$ dominira nad $\vec{x}^{(2)}$
$(1,2,3)$	$(1,3,4)$	$(5,6)$	$(5,6)$	-
$(1,2,3)$	$(1,4,4)$	$(5,6)$	$(5,7)$	$\vec{x}^{(2)}$ dominira nad $\vec{x}^{(1)}$
$(1,2,3)$	$(2,2,4)$	$(5,6)$	$(6,2)$	-

U optimizacijskim algoritmima pojam dominacije koristi se za procjenu kvalitete rješenja. Ideja je jednostavna: rješenje nad kojim nitko ne dominira je izvrsno rješenje – boljih nema. Stoga će se u algoritmima uobičajeno pratiti za svako rješenje koliko ima rješenja koja ga dominiraju – što je taj broj manji, to je rješenje bolje.

Osvrnimo se ukratko i na svojstva relacije dominacije. Relacija dominacije nije refleksivna jer rješenje ne dominira nad samim sobom. Relacija dominacije nije simetrična (tj. ne vrijedi: ako $\vec{x}^{(1)}$ dominira $\vec{x}^{(2)}$, onda $\vec{x}^{(2)}$ dominira $\vec{x}^{(1)}$). Relacija dominacije je tranzitivna: ako $\vec{x}^{(1)}$ dominira $\vec{x}^{(2)}$ i $\vec{x}^{(2)}$ dominira $\vec{x}^{(3)}$ tada $\vec{x}^{(1)}$ također dominira $\vec{x}^{(3)}$.

U svakom skupu rješenja s kojima optimizacijski algoritam radi u nekom trenutku moguće je za svaki par rješenja iz tog skupa provjeriti da li koje od rješenja dominira ono drugo. Time čitav skup možemo podijeliti u dva podskupa: rješenja nad kojima nitko ne dominira te preostala rješenja koja su dominirana od barem jednog rješenja.

Nedominirani skup. Neka je s $\mathcal{P}$ označen skup rješenja. Nedominiranim skupom $\mathcal{P}'$ zovemo podskup skupa $\mathcal{P}$ koji se sastoji od rješenja nad kojima niti jedno rješenje iz skupa $\mathcal{P}$ ne dominira.

Pogledajmo to opet na primjeru. Neka je skup rješenja $\mathcal{P} = \{\vec{x}^{(1)}, \vec{x}^{(2)}, \vec{x}^{(3)}, \vec{x}^{(4)}, \vec{x}^{(5)}\}$. Neka je $\vec{z}^{(1)} = (9,1)$, $\vec{z}^{(2)} = (2,2)$, $\vec{z}^{(3)} = (7,4)$, $\vec{z}^{(4)} = (6,2)$ te $\vec{z}^{(5)} = (4,6)$. Lako je provjeriti da su rješenja iz skupa $\mathcal{P}$ nad kojima niti jedno rješenje iz skupa $\mathcal{P}$ ne dominira rješenja $\vec{x}^{(1)}$, $\vec{x}^{(3)}$ i $\vec{x}^{(5)}$. Nedominirani skup skupa $\mathcal{P}$ stoga je skup $\mathcal{P}' = \{\vec{x}^{(1)}, \vec{x}^{(3)}, \vec{x}^{(5)}\}$.

Sada možemo definirati i globalni optimum u smislu višekriterijske optimizacije.

Globalni Pareto-optimalni skup. Nedominirani skup skupa svih prihvatljivih rješenja naziva se globalni Pareto-optimalni skup.

Uočite koliko je prethodna definicija elegantna: kao skup $\mathcal{P}$ naprosto se uzme čitav prostor svih prihvatljivih rješenja, i nad njim se pronađe nedominirani skup. Taj skup, iako ga u praksi za većinu složenijih problema nećemo moći u cijelosti izračunati doista sadrži sva najbolja rješenja problema. Zadaća optimizacijskih algoritama bit će iz koraka u korak otkrivati sve bolju i bolju aproksimaciju ovog skupa.

Nedominirani skup zadane populacije rješenje moguće je utvrditi sa složenošću od $O(MN^2)$. Postupak prikazuje pseudokod 13.1.

Pareto fronta. Neka je s $\mathcal{P}$ označen Pareto-optimalni skup rješenja. Skup pripadnih vektora koje u M-dimenzijskom prostoru ciljnih funkcija za svako rješenje iz $\mathcal{P}$ definiraju ciljne funkcije naziva se Pareto fronta.

Kod različitih algoritama višekriterijske optimizacije ponekad se zahtjeva ne samo izračun nedominiranog skupa već sortiranje čitave populacije u različite podskupove prema tome koliko su rješenja "blizu" nedominiranom skupu; svaki taj podskup zvat ćemo jednom *frontom*. Konceptualno, ideja je sljedeća: iz početnog skupa $\mathcal{P}$ odredi se nedominirani skup $\mathcal{P}'$. Taj skup proglasimo prvom frontom. Potom definiramo novi skup $\mathcal{P}_{\in} = \mathcal{P} \setminus \mathcal{P}'$ koji sadrži sve elemente početnog osim elemenata nedominiranog skupa. Sada pronađemo nedominirani skup skupa $\mathcal{P}_{\in}$ i taj skup proglasimo drugom frontom. Postupak iterativno ponavljamo dok preostane elemenata: iz skupa uklonimo upravo pronađenu frontu, izračunamo nedominirani skup, itd. Međutim, opisani postupak je računski vrlo zahtjevan. Stoga je u nastavku dan opis računski prihvatljivijeg postupka.

Pseudokod 13.1 Utvrđivanje nedominiranog skupa.

```
nedominiraniSkup(P)
postavi P'={}
ponavljaj za svaki x iz P
  dominiran = ne
  ponavljaj za svaki y iz P
    ako y dominira x tada
      dominiran = da; prekini petlju
    kraj ako
  kraj
  ako dominiran = ne tada
    P' += x;
  kraj ako
kraj
vrati P'
```

Pseudokod 13.2 Nedominirano sortiranje.

1. Za svaki $i \in \mathcal{P}$ postavi $\eta_i = 0$ i $\mathcal{S}_i = \emptyset$. Za svaki $j \neq i$ i $j \in \mathcal{P}$ provedi korake 2 pa 3.
2. Ako rješenje i dominira nad rješenjem j, dodaj ga: $\mathcal{S}_i = \mathcal{S}_i + j$. Inače, ako rješenje j dominira nad rješenjem i, povećaj brojač $\eta_i = \eta_i + 1$.
3. Ako je $\eta_i = 0$, zadrži rješenje i u prvoj nedominiranoj fronti $\mathcal{P}_1$. Postavi brojač fronti $k = 1$.
4. Sve dok je $\mathcal{P}_k \neq \emptyset$, radi sljedeće korake.
5. Inicijaliziraj $\mathcal{Q} = \emptyset$ koji će čuvati sljedeću nedominiranu frontu. Za svaki $i \in \mathcal{P}_k$ i za svaki $j \in \mathcal{S}_i$ radi:
   (a) Ažuriraj: $\eta_j = \eta_j - 1$.
   (b) Ako je time postao $\eta_j = 0$, dodaj rješenje j u trenutnu frontu: $\mathcal{Q} = \mathcal{Q} + j$.
6. Postavi $k = k + 1$ i $\mathcal{P}_k = \mathcal{Q}$. Vrati se na korak 4.

13.4 Nedominirano sortiranje

Pseudokod 13.2 prikazuje proceduru nedominiranog sortiranja kako je opisana u [Deb, 2009].

Postupak se provodi u dva dijela. Zbor praktičnosti, pretpostavimo da smo sva rješenja numerirali od 1 na dalje, tako sa u postupku možemo raditi sa indeksima i skupovima indeksa.

U prvom dijelu postupka (koraci 1 i 2 u pseudokodu 13.2) za svako se rješenje i računa brojač dominacije η_i – to je broj rješenja koja dominiraju nad rješenjem i. Primjetite da rješenja koja nakon ovog koraka imaju $\eta_i = 0$ (dakle nad kojima nitko ne dominira) čine nedominirani skup. U tom istom koraku, za svako se rješenje i utvrđuje i skup $\mathcal{S}_i$ – to je skup svih rješenja (zapravo indeksa rješenja) nad kojima rješenje i dominira. Ovaj skup nam je važan kako prilikom eliminacije rješenja i ne bismo ponovno morali pretraživati ostatak populacije tražeći sva rješenja nad kojima trenutno rješenje dominira – tu ćemo informaciju pročitati iz ovog skupa. Brojača η_x te skupova $\mathcal{S}_x$ ima onoliko koliko ima rješenja u populaciji koju sortiramo.

Prvi dio postupka završava korakom 3 u kojem se pronalaze sva rješenja čiji je $\eta_i = 0$; ta se ta rješenja postavljaju kao prva fronta (što je ujedno i nedominirani skup početnog skupa).

Drugi dio postupka je petlja koja računa frontu $k + 1$ temeljem fronte k (koraci 4, 5 i 6). Ideja je jednostavna: za svako rješenje zadnje pronađene fronte pogledamo nad kojim su rješenjima ta rješenja dominirala. Kako smo zadnju pronađenu frontu efektivno izbacili iz populacije, tim dominiranim rješenjima umanjimo brojače η_j. Ako je nekom rješenju j time brojač pao na nulu, to znači da u preostaloj populaciji nitko nad njime ne dominira – ta se rješenja dodaju u novu frontu $k + 1$.

Temeljem opisanih postupaka sada bismo trebali imati dovoljno informacija kako bismo bez problema razumjeli algoritme koji su opisani u sljedećim poglavljima.

Bibliografija

K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley, West Sussex, United Kingdom, 2009.

Poglavlje 14

Algoritam NSGA

14.1 Uvod

Algoritam NSGA (engl. *Non-Dominated Sorting Genetic Algoritm*) opisan je u radu [Srinivas and Deb, 1994]. To je inačica genetskog algoritma koja prestavlja izravnu uporabu koncepta dominacije (odnosno nedominacije i nedominiranog sortiranja) na problem višekriterijske optimizacije.

Ideja algoritma je relativno jednostavna: prilikom vrednovanja rješenja, svakoj se jedinki pridjeljuje skalarna vrijednost koja predstavlja dobrotu, na način da jedinke koje su manjeg ranga (nakon nedominiranog sortiranja pripadaju u frontu što bližu nedominiranom skupu) dobiju veći iznos dobrote; dodatno, unutar iste fronte, rješenja koja se grupiraju (postoji više bliskih rješenja) se kažnjavaju dodjeljivanjem umanjenog iznosa dobrote, čime se pokušava očuvati raznolikost u skupu rješenja.

14.2 Detaljni opis

Algoritam NSGA je inačica generacijskog genetskog algoritma prikladnog za rješavanje problema višekriterijske optimizacije. Algoritam je klasični generacijski – iz populacije roditelja biraju se roditelji koji se kombiniraju primjenom operatora križanja, nastala rješenja se mutiraju i postupak se ponavlja sve dok se ne izgradi populacija djece s istim brojem elemenata kao što je populacija roditelja. U tom trenutku populacija roditelja se briše a populacija djece postaje nova populacija roditelja i postupak se ponavlja.

Algoritam NSGA, zbog specifičnog načina dodjele dobrote jedinkama, zahtjeva da se kao operator selekcije koristi proporcionalna selekcija ili neka njezina izvedenica (primjerice, engl. *Stochastic remainder roulette-wheel selection*, vidi [Goldberg, 1989]). Prvi korak u osnovnoj generacijskoj petlji algoritma jest dodjela dobrote rješenjima iz populacije roditelja, kako bi se mogla koristiti proporcionalna selekcija. Pseudokôd 14.1 prikazuje implementaciju procedure za dodjelu dobrote.

Postupak dodjele dobrote (koraci 1 i 2) započinju definiranjem parametara za uporabu dijeljenja dobrote te nedominiranim soritranjem populacije. Postupkom nedominiranog sortiranja populacija će se raspasti u ρ disjunktnih podskupova – fronti, pri čemu će prva fronta biti upravo nedominirani skup.

Prvoj fronti, s obzirom da sadrži najbolje jedinke pridjeljuje se dobrota iznosa N. Potom se za svako rješenje unutar te fronte računa gustoća niše (broj rješenja koja su u okolici promatranog rješenja skaliran s vrijednosti proporcionalnoj blizini tih rješenja), nakon čega se računa i iznos dijeljene dobrote koji će se koristiti za proporcionalnu selekciju. Doseg dijeljenja određen je parametrom σ_{share}. Ako unutar udaljenosti σ_{share} do nekog rješenja nema drugih rješenja u promatranoj fronti, rješenje će dobiti puni iznos dobrote. Međutim, ako je više rješenja grupirano unutar σ_{share}, dodjelit će im se manji iznos dobrote. Treba napomenuti da se, prema autorima, iznos gustoće dijeljenja računa uporabom normaliziranih udaljenosti d_{ij} u prostoru rješenja, a ne u prostoru ciljnih funkcija:

$$d_{ij} = \sqrt{\sum_{k=1}^{n} \left(\frac{x_k^{(i)} - x_k^{(j)}}{x_k^{\max} - x_k^{\min}} \right)^2}$$

čime se dijeljenje radi na razini genotipa/fenotipa (navedeni izraz pretpostavlja da je prostor rješenja

Pseudokod 14.1 Dodjela dobrote kod NSGA.

1. Odaberi parametar dijeljenja σ_{share} i malu pozitivnu konstantu ϵ. Inicijaliziraj $F_{\min} = N + \epsilon$. Postavi brojač fronti na $j = 1$.
2. Sortiraj populaciju uporabom nedominiranog sortiranja. Time će se populacija raspasti u ρ fronti $(P_1, \ldots, P_\rho)$.
3. Za svako rješenje $q \in P_j$ radi:
   (a) Dodijeli rješenju q iznos dobrote $F_j^{(q)} = F_{\min} - \epsilon$.
   (b) Izračunaj gustoću niše nc_q uvažavajući pri tome samo rješenja iz P_j.
   (c) Izračunaj dijeljenu dobrotu $F_j^{'(q)} = \frac{F_j^{(q)}}{nc_q}$.
4. Utvrdi $F_{\min} = \min(F_j^{'(q)} : q \in P_j)$. Postavi $j = j + 1$.
5. Ako je $j \leq \rho$, nastavi s korakom 3. Inače je postupak gotov.

n-dimenzijski, i prikazuje udaljenost između jedinki i i j). Time se, efektivno, ne potiče raznolikost na razini razdiobe rješenja u prostoru ciljnih funkcija već raznolikost na razini razdiobe rješenja u prostoru rješenja, što jest donekle kontra-intuitivno s obzirom na postavljene ciljeve algoritma. Autori sugeriraju izračun funkcije dijeljenja uz parametar $\alpha = 2$. Podsjećamo, gustoća dijeljenja za rješenje i jednaka je sumi vrijednosti funkcije dijeljenja nad svim parovima rješenja (i, j) (vidi izraz (12.2)).

Nakon što je tako riješena prva fronta, pronalazi se minimalna dodijeljena dobrota bilo kojem od rješenja iz te fronte i definira se novi početni iznos dobrote za sva rješenja iz sljedeće fronte kao utvrđena minimalna dobrota dodatno umanjena za iznos ϵ. Opravdanje za ovakvu proceduru jest činjenica da su rješenja koja se nalaze u drugoj fronti lošija od rješenja koja se nalaze u prvoj fronti. Stoga im želimo dodijeliti iznos dobrote koji je garantirano manji od dobrote bilo kojeg rješenja iz prve fronte. Jednom kad smo utvrdili novu početnu vrijednost dobrote, opisani postupak dodjele dobrote dijeljenjem se provodi unutar druge fronte, pronalazi se minimalna dodijeljena dobrota koja će se nakon umanjivanja za ϵ koristiti kao početna dobrota za rješenja treće fronte, i postupak se ponavlja sve dok se ne riješe sve fronte dobivene nedominiranim sortiranjem.

Prilikom implementacije algoritma treba pripaziti da se uslijed lošeg odabira parametara σ_{share} i ϵ ne dogodi da početni iznos dobrote za neku od fronti postane negativan – da bi proporcionalna selekcija radila korektno, sve dobrote moraju biti nenegativne, a suma im mora biti pozitivna.

Valja također primijetiti da opisani algoritam nije elitistički.

Bibliografija

D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, Reading, MA, 1989.

N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

Poglavlje 15

Algoritam NSGA-II

15.1 Uvod

Algoritam NSGA-II predložili su [Deb et al., 2000]. Za razliku od algoritma NSGA koji nema ugrađen elitizam, osnovna značajka ove izvedbe algoritma jest uporaba elitizma kako bi se osiguralo da ne dolazi do gubitka najboljih rješenja.

15.2 Detaljni opis

Algoritam NSGA-II iz populacije roditelja P_t veličine N stvara populaciju djece Q_t veličine N. Potom ove dvije populacije kombinira u populaciju roditelja i djece $R_t = P_t \cup Q_t$ veličine $2N$. Ta se populacija zatim sortira uporabom nedominiranog sortiranja čime se raspada u ρ fronti $\mathcal{F}_1, \ldots, \mathcal{F}_\rho$. U novu populaciju P_{t+1} kopiraju se prvih μ fronti koje se u cijelosti mogu ubaciti u novu populaciju. Uočimo da će sigurno postojati fronta $\mathcal{F}_{\mu+1}$ koja više neće stati čitava u populaciju P_{t+1}: suma broja rješenja u svim frontama od R_t je $2N$ dok populacija P_{t+1} ima mjesta samo za N rješenja – stoga će ubacivanje cijelih fronti negdje puknuti, i broj te fronte koja se ne može čitava dodati u P_{t+1} označit ćemo s μ.

Pitanje na koje sada treba odgovoriti jest: koja od rješenja iz $\mathcal{F}_\mu$ ubaciti u novu populaciju? Algoritam NSGA-II osim elitizma stavlja naglasak na očuvanje raznolikosti u rješenjima. Stoga se odabir podskupa rješenja koja će iz fronte $\mathcal{F}_\mu$ još biti uzeta u dodana u novu populaciju P_{t+1} temelji na pronalasku podskupa rješenja koja su "dobro" raspršena po čitavoj fronti $\mathcal{F}_\mu$, za što se koristi postupak sortiranja prema grupiranju (engl. *Crowding-sort*). Pseudokod 15.1 prikazuje izvedbu algoritma NSGA-II.

Pseudokod 15.1 Algoritam NSGA-II.

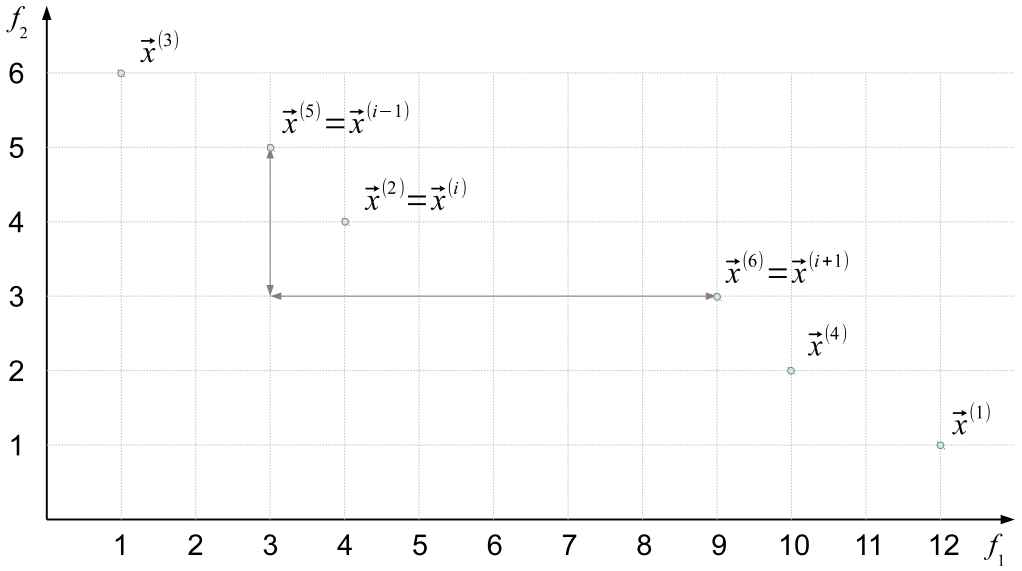
1. Neka je P_t trenutna populacija roditelja a Q_t trenutna populacija djece. Kombiniraj te dvije populacije u populaciju $R_t = P_t \cup Q_t$. Napravi nedominirano sortiranje populacije R_t čime će nastati ρ fronti: $R_t = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \ldots \cup \mathcal{F}_\rho$.

2. Postavi novu populaciju roditelja na praznu: $P_{t+1} = \emptyset$. Postavi brojač $i = 1$. Nadopunjuj novu populaciju frontama tako dugo dok čitave stanu u novu populaciju: dok je $|P_{t+1}| + |\mathcal{F}_i| \leq N$, napravi $P_{t+1} = P_{t+1} \cup \mathcal{F}_i$, $i = i + 1$.

3. Sada je $i = \mu$ – pokazuje na prvu frontu koja nije u cjelosti mogla biti iskopirana u P_{t+1}. Provedi postupak sortiranja fronte $\mathcal{F}_i$ prema grupiranju, te u populaciju P_{t+1} nadodaj $N - |P_{t+1}|$ rješenja iz fronte $\mathcal{F}_i$ koja su najviše raspršena. Time je postupak izgradnje nove populacije roditelja završen.

4. Stvori novu populaciju djece Q_{t+1} iz populacije roditelja P_{t+1} uporabom grupirajuće turnirske selekcije te križanjem i mutiranjem rješenja.

Pseudokod 15.1 se poziva na dva postupka koja još nismo opisali: sortiranje prema grupiranju



Slika 15.1: Izračun udaljenosti grupiranja

(engl. *Crowding-sort*) te grupirajuća turnirska selekcija (engl. *crowded tournament selection*). Oba postupka temelje se na izračunu udaljenosti grupiranja (engl. *crowding distance*), koju ćemo opisati u nastavku.

15.2.1 Udaljenost grupiranja

Kao što smo već pojasnili, kako bi se očuvala raznolikost rješenja, algoritam NSGA-II iz fronte $\mathcal{F}_\mu$ koja nije čitava mogla biti nadodana u novu populaciju roditelja P_{t+1} želi probrati ona rješenja unutar te fronte koja su maksimalno raspršena u prostoru ciljnih funkcija (kojih ima M). Stoga se za svako rješenje i i za svaku ciljnu funkciju f_m gleda suma normiranih udaljenosti do najbližeg rješenja koje ima manji odnosno veći iznos te kriterijske funkcije. Ovo je ilustrirano na slici 15.1. Pretpostavimo da se u fronti $\mathcal{F}_\mu$ nalaze rješenja $\vec{x}^{(1)}$ do $\vec{x}^{(6)}$, te da se radi dvokriterijska optimizacija. Neka su pridružene vrijednosti funkcije dobrote tada $\vec{x}^{(1)} \to (12,1)$, $\vec{x}^{(2)} \to (4,4)$, $\vec{x}^{(3)} \to (1,6)$, $\vec{x}^{(4)} \to (10,2)$, $\vec{x}^{(5)} \to (3,5)$ te $\vec{x}^{(6)} \to (9,3)$.

U navedenom slučaju, za svako će se rješenje izračunati udaljenost grupiranja koja će biti jednaka sumi dviju komponenti (jer su dvije funkcije cilja): udaljenosti uzduž prve funkcije cilja te udaljenosti uzduž druge funkcije cilja. Pretpostavimo najprije da smo sva rješenja sortirali prema prvoj komponenti funkcije cilja (tj. prema f_1). Time smo dobili poredak $\vec{x}^{(3)}$, $\vec{x}^{(5)}$, $\vec{x}^{(2)}$, $\vec{x}^{(6)}$, $\vec{x}^{(4)}$, $\vec{x}^{(1)}$. Promotrimo sada jedno istaknuto rješenje $\vec{x}^{(i)}$ (na slici je to $\vec{x}^{(2)}$). U sortiranom nizu prvo rješenje ispred njega je $\vec{x}^{(5)}$ (usput označeno s $\vec{x}^{(i-1)}$) a prvo rješenje nakon njega je $\vec{x}^{(6)}$ (usput označeno kao $\vec{x}^{(i+1)}$). Doprinost prve ciljne funkcije udaljenosti grupiranja rješenja $\vec{x}^{(i)}$ računa se kao omjer međusobne udaljenosti odnosno razlike u vrijednosti prve ciljne funkcije $(i+1)$-vog i $(i-1)$-vog rješenja (grafički prikazano kao vodoravna strelica) i maksimalne moguće udaljenosti s obzirom na prvu ciljnu funkciju. Doprinost druge ciljne funkcije udaljenosti grupiranja rješenja $\vec{x}^{(i)}$ računa se na sličan način (gleda se okomita strelica). Čitav postupak sažet je u pseudokodu 15.2.

Pogledajmo to na primjeru zadanih 6 rješenja. Sortiranjem prema prvoj funkciji cilja dobije se vektor indeksa $(3,5,2,6,4,1)$. Sortiranjem prema drugoj funkciji cilja dobije se vektor indeksa $(1,4,6,2,5,3)$. To znači da će d_3 (udaljenost grupiranja koja pripada rješenju $\vec{x}^{(3)}$) kao i d_1 (udaljenost grupiranja koja pripada rješenju $\vec{x}^{(1)}$) biti $d_3 = d_1 = \infty$ (jer su to rubna rješenja). Sa slike možemo očitati minimalne i maksimalne vrijednosti svake od dvije funkcije cilja $f_1^{\min} = 1$, $f_1^{\max} = 12$,

Pseudokod 15.2 Izračun udaljenosti grupiranja.

1. Neka je broj rješenja u fronti $\mathcal{F}_\mu$ označen s l, tj. neka je $l = |\mathcal{F}_\mu|$. Za svako rješenje i u tom skupu postavi udaljenost grupiranja na nulu, tj. $d_i = 0$.

2. Ponavljaj za svaku ciljnu funkciju $m \in \{1, 2, \ldots, M\}$: sortiraj rješenja u skupu prema rastućim vrijednostima pridružene m-te funkcije cilja. Označimo s I^m vektor indeksa rješenja koja su na taj način sortirana. Npr. ako je prvi element tog vektora broj 2, to znači da rješenje $\vec{x}^{(2)}$ ima najmanji iznos m-te funkcije cilja.

3. Za svaki $m \in \{1, 2, \ldots, M\}$ dodijeli veliki doprinos udaljenosti grupiranja rubnim rješenjima (prvom i zadnjem, jer oni nemaju lijevog odnosno desnog susjeda): $d_{I_1^m} = d_{I_l^m} = \infty$. Za sva ostala rješenja čiji su indeksi u vektoru I^m na pozicijama $2, \ldots, l-1$ uvećaj doprinos udaljenosti grupiranja prema izrazu:

$$d_{I_j^m} = d_{I_j^m} + \frac{f_m(\vec{x}^{(I_{j+1}^m)}) - f_m(\vec{x}^{(I_{j-1}^m)})}{f_m^{\max} - f_m^{\min}}.$$

$f_2^{\min} = 1$, $f_2^{\max} = 6$. Stoga računom slijedi:

$$\begin{aligned} d_5 &= 0 + \frac{f_1(\vec{x}^{(2)}) - f_1(\vec{x}^{(3)})}{f_1^{\max} - f_1^{\min}} \\ &= 0 + \frac{4-1}{12-1} \\ &= \frac{3}{11} \\ d_5 &= \frac{3}{11} + \frac{f_2(\vec{x}^{(3)}) - f_2(\vec{x}^{(2)})}{f_2^{\max} - f_2^{\min}} \\ &= \frac{3}{11} + \frac{6-4}{6-1} \\ &= \frac{3}{11} + \frac{2}{5} \\ &= \frac{37}{55} \approx 0.673. \end{aligned}$$

Na sličan način slijedi i:

$$\begin{aligned} d_2 &= \frac{6}{11} + \frac{2}{5} = \frac{52}{55} \approx 0.945, \\ d_6 &= \frac{6}{11} + \frac{2}{5} = \frac{52}{55} \approx 0.945, \\ d_4 &= \frac{3}{11} + \frac{2}{5} = \frac{37}{55} \approx 0.673. \end{aligned}$$

15.2.2 Grupirajuća turnirska selekcija

Još je ostalo za pojasniti selekciju koju koristi NSGA-II prilikom odabira roditelja koji će stvarati potomke – ta selekcija nije, kao što je slučaj kod algoritma NSGA, proporcionalna već se temelji na turnirima između jedinki. Kod klasične s-turnirske selekcije iz populacije se izvlači s rješenja i pobjeđuje ono koje ima najveći iznos dobrote. Kod višekriterijske optimizacije, kao što smo se već upoznali, termin "iznos dobrote" je dosta problematičan. Stoga se u kontekstu algoritma NSGA-II definira *grupirajuća turnirska selekcija* koja se zasniva na rangu rješenja (broju fronte kojoj rješenje pripada – manje je bolje) te na udaljenosti grupiranja: poželjnija su rješenja koja imaju veću udaljenost grupiranja – to znači da su ona dobro raspršena, odnosno da u njihovoj blizini nema drugih rješenja. Pseudokod 15.3 objašnjava izvedbu grupirajuće turnirske selekcije prikazan.

Pseudokod 15.3 Grupirajuća turnirska selekcija.

Rješenje i je pobjednik ako vrijedi bilo koji od sljedeća dva uvjeta:

1. rješenje i ima bolji rang od rješenja j, tj. $r_i < r_j$,
2. rješenje i ima jednak rang kao i rješenje j, tj. $r_i = r_j$, ali ima veću udaljenost grupiranja, odnosno $d_i > d_j$.

Bibliografija

K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. M. Guervós, and H.-P. Schwefel, editors, *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, volume 1917 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, September 2000. Springer.

Dio III

Paralelizacija

Poglavlje 16

Paralelizacija algoritama

Evolucijsko računanje pokazalo se je prikladnim za rješavanje čitavog niza teških optimizacijskih problema. Pod pojmom teških podrazumijevamo probleme čija je složenost tolika da bi i na najmodernijim danas dostupnim superračunalima iscrpna pretraga trajala dulje no što smo spremni čekati; u problemima raspoređivanja koji su opisani u ovoj disertaciji čest je slučaj da bi iscrpna pretraga trajala dulje no što je star svemir.

Metaheuristike evolucijskog računanja pomažu nam da unatoč takvoj složenosti dođemo do prihvatljivih rješenja, iako ne nude nikakve garancije optimalnosti rješenja ako im se izvođenje ograniči na konačno vrijeme. Uklonimo li ovu ogradu, za neke od metaheuristika postoje dokazi pronalaska optimalnog rješenja. Primjerice, pogledajmo imunološke algoritme – to su algoritmi koji svoj rad temelje na različitim operatorima mutacije. Ako se pri tome koristi operator kod kojeg je (i) djelovanje upravljano vjerojatnostima te (ii) postoji vjerojatnost različita od nule da se u okviru jedne mutacije promijene svi geni rješenja koje se mutira, takav će algoritam, pustimo li ga da radi do beskonačnosti sigurno u jednom trenutku izgenerirati i ono optimalno rješenje. Ovisno o tome koliko smo sretni ili ne, tada postoji mogućnost i da bi postupak iscrpne pretrage prije došao do tog rješenja, jer će on sigurno završiti u konačnom vremenu (prisjetimo se, rješenja s kojima radimo su diskretna, s fiksnim brojem elemenata i fiksnim brojem izbora).

Odustanemo li od zahtjeva optimalnosti, ipak je činjenica da će rješenja koja algoritam pronalazi biti to bolja što je više različitih rješenja algoritam mogao istražiti. U tom smislu, algoritmima evolucijskog računanja važno je osigurati mogućnost pretraživanja što je moguće većeg područja, a to vremenski košta. Pišemo li algoritam na jednoprocesorskom računalu, pomoći nema. Međutim, imamo li na raspolaganju višeprocesorsko računalo, ili više računala, paralelizacijom je moguće u ograničenom vremenu omogućiti algoritmu šire i kvalitetnije pretraživanje prostora rješenja.

Danas se paralelizacija radi iz nekoliko razloga.

1. *Paralelizacija zbog velikih količina podataka.* Postoje problemi čija su rješenja prevelika da bi stala u memoriju jednog računala. Postoje problemi kod kojih se vrednovanje radi nad velikom količinom podataka koji opet svi ne stanu u memoriju jednog računala. Paralelizacijom je ovakve probleme moguće razdijeliti na više računala i izgradnju čitavog rješenja ili vrednovanje rješenja obavljati dio po dio.

2. *Paralelizacija zbog skraćivanja vremena izvođenja.* Uz konstantan obim posla koji je potrebno napraviti, ako je dijelove algoritma moguće paralelizirati, čitav će postupak biti prije gotov. Ovo može biti interesantno ako postoje stoga vremenska ograničenja koja je potrebno zadovoljiti.

3. *Paralelizacija zbog povećanja obima poslova.* Ako se vrijeme izvođenja algoritma drži konstantnim, paralelizacija dijelova algoritma omogućit će da se u istom vremenu odradi više posla.

Primjena paralelizacije kod algoritama evolucijskog računanja osigurava sljedeće:

- ubrzavanje algoritma kako bi se prije došlo do prihvatljivih rješenja,
- povećanje kvalitete pronađenih rješenja jer algoritam može istražiti veći prostor rješenja te

- povećanje robusnosti algoritma jer će se smanjiti vjerojatnost ostanka u lošijim područjima zbog mogućnosti istraživanja većeg prostora rješenja.

16.1 Vrste paralelizacije

Uobičajno je paralelizaciju promatrati na tri razine:

1. paralelizacija na razini algoritama,
2. paralelizacija na razini populacije te
3. paralelizacija na razini jedne iteracije algoritma.

Paralelizacija na razini algoritama predstavlja svaki oblik paralelizacije kod kojeg se istovremeno izvodi više primjeraka algoritama. Ova vrsta paralelizacije može biti suradna (engl. *cooperative*) i nesuradna (engl. *non-cooperative*). Nesuradna paralelizacija je paralelizacija kod koje različiti primjerci algoritama međusobno ne komuniciraju i ne dijele nikakve informacije. Primjer ovakve paralelizacije jest nezavisno pokretanje više primjeraka algoritama te uzimanje odabir najboljeg rješenja među rješenjima koje su pronašli pojedini primjerci algoritama. Druga mogućnost jest pokretanje više primjeraka algoritama ali s različitim parametrima, u nadi da će uz neke algoritam raditi bolje. Već i ovakav oblik nesuradne paralelizacije donosi povećanu robusnost algoritma te dobivanje boljih rješenja.

Suradna paralelizacija podrazumijeva pokretanje više primjeraka algoritama koji međusobno razmjenjuju informacije. Ovisno o vrsi algoritma, moguće je razmjenjivati različite podatke, poput dijelova populacije (često se dijeli samo najbolje rješenje), ili informacija koje utječu na rad algoritma (trenutna vjerojatnost mutacije, vrijednost feromonskih tragova kod mravljih algoritama i slično). Uvođenje suradne paralelizacije povlači porast broja parametara takvog algoritma: treba odrediti što se šalje, tko kome šalje (topologija) te u kojim se trenutcima informacije šalju. Poznati primjeri ovakve vrste paralelizacije su otočni model te celularni model često korišteni kod genetskih algoritama.

Paralelizacija na razini populacije uključuje postupke paralelizacije koji omogućavaju bržu izmjenu populacija. Primjerice, ako je potrebno izgraditi sljedeću populaciju od n jedinki, posao izgradnje se može paralelizirati na k radnika, pa će vrijeme potrebno za izgradnju populacije tada biti k puta manje u odnosnu na izgradnju jedne po jedne jedinke slijedno. Ovu vrstu paralelizacije moguće je ostvariti na više načina. Jedna mogućnost je paralelizirati čitav postupak izgradnje jedinke. Kod genetskog algoritma to bi značilo da radnik obavlja selekciju, križanje, mutaciju i vrednovanje, i takvu jedinku predaje dalje. Kod mravljih algoritama to bi značilo da jedan radnik obavlja izgradnju čitavog rješenja i provodi njegovo vrednovanje. Slično je moguće ostvariti i kod algoritma diferencijske evolucije i drugih. Druga je mogućnost paralelizirati samo izvođenje vrednovanja, ako je to vremenski zahtjevna operacija. U tom slučaju uobičajeni je scenarij imati k radnika kojima se šalju jedinke, svaki radnik obavlja vrednovanje dobivenih jedinki i vraća rezultat vrednovanja. Uz k radnika uistom će se vremenu moći vrednovati k-puta više jedinki nego na jednom procesoru pa će se time postići ubrzanje.

Paralelizacija na razini jedne iteracije algoritma obuhvaća postupke paralelizacije koji su spušteni na najnižu moguću razinu. Čitav algoritam obavlja se kao da je slijedni, a paralelizacija je spuštena na razinu operatora. Primjerice, moguće je implementirati paralelni operator mutacije, paralelni operator križanja, paralelno vrednovanje rješenja i slično. Razlika u odnosu na paralelizaciju na razini populacije je u tome što kod ovog pristupa nemamo k radnika kod kojih svaki u potpunosti obavlja slijedno implementiran operator (npr. k radnika koji svaki vrednuje svoju jedinku). Umjesto toga, ovdje govorimo o paralelizaciji implementacije operatora, gdje se, primjerice, paralelizira izvedba algoritma vrednovanja jedinke tako da se što prije dođe do rezultata.

Spomenimo još i da suradna paralelizacija na razini algoritama nije isto što i paralelizacija na razini populacije. Naime, iako se možda ne čini očito, nije isto pokrenuti 4 algoritma od kojih svaki ima populaciju od 50 jedinki i povremenu razmjenjuju rješenja te jedan algoritam koji ima 200 jedinki i koristi četiri radnika za paralelizaciju. U oba slučaja, broj obrađenih jedinki u jedinici vremena bit će isti. Međutim, uslijed efekata izoliranog pretraživanja unutar manjih populacija te uslijed povremene razmjene rješenja prvi pristup iskazivat će drugačiju dinamiku promjena u populaciji i često će naći bolja rješenja u odnosu na jednu veliku populaciju.

16.2 Načini i cijena paralelizacije

Kako se može provesti postupak paralelizacije uvelike će ovisiti o željenom broju radnika k (a time će i vrsta radnika biti utvrđena). Ako je prihvatljiv manji broj radnika, pojam radnika tada možemo poistovijetiti s procesorom računala. Tehnologije koje nam pri tome stoje na raspolaganju su uobičajena višedretvenost gdje je algoritam ujedno i jedan proces. S obzirom da sve dretve imaju pristup svim podatcima u adresnom prostoru procesa, dijeljenje poslova moguće je obaviti uz minimalne troškove a jedinke i sve potrebne podatkovne strukture tipično se prenose preko reference (ili pokazivača) čime se stvara minimalni mogući promet na podatkovnoj sabirnici računala. Trošak koji plaćamo kod ovog oblika paralelizacije jest trošak sinkronizacijskih mehanizama kojima ćemo osigurati da su svi podatci u konzistentnom stanju. Ovakav oblik paralelizacije prikladan je za sve vrste paralelizacije i možemo ga koristiti od izvedbe paralelizacije na razini jedne iteracije algoritma pa sve do izvedbe paralelizacije na razini algoritama. Problem ovog pristupa jest u malom broju radnika koje podržava. Danas su među masovno raširenim računalima uobičajena računala dvije do četiri jezgre i tek se pojavljuju računala sa šest jezgri; stoga je to upravo i granica na broj radnika koji se može koristiti u paralelizaciji. Radimo li na četiri-jezgrenom računalu, moći ćemo koristiti najviše četiri radnika.

Razmatramo li druge mogućnosti paralelizacije unutar istog računala, postoji i mogućnost paralelizacije na razini više procesa. Pri tome radnici mogu biti različiti procesi koji međusobno komuniciraju uporabom mehanizama operacijskog sustava za međuprocesnu komunikaciju; to su danas uobičajeno dijeljena memorija, komunikacija cjevovodima ili komunikacija općenitijim mrežnim podsustavom. Cijena koju plaćamo ovakvim pristupom viša je od uporabe radnika unutar jednog procesa. Mehanizmi međuprocesne komunikacija su sporiji, a mogu zahtjevati i dodatni utrošak vremena na poslove serijalizacije, slanja i deserijalizacije jedinki te serijalizacije, slanja i deserijalizacije rezultata (ako se radi o mehanizmima poput cjevovoda ili mrežnog podsustava). Također, ovakav način paralelizacije bit će neprikladan za paralelizaciju na nižim razinama. Broj radnika i dalje je ograničen na broj procesora odnosno jezgri koje se nalaze u računalu, pa nema neke očite prednosti u odnosu na paralelizaciju unutar istog procesa.

Za veći broj radnika danas je uobičajeno posegnuti za paralelizacijom uporabom više računala. Tu možemo razmišljati o specijaliziranim okruženima poput grozda (engl. *cluster*), grida ili pak danas popularnog računarstva u oblacima (engl. *cloud-computing*). Međutim, kako u najvećem broju slučajeva ovakva infrastruktura nije dostupna, najčešće se koriste računala koja su dostupna, poput računala u nekom računalnom laboratoriju u trenutcima kada se ona ne koriste i slično. Kada govorimo o ovom načinu paralelizacije, važno je uočiti da su ovdje troškovi komunikacije te troškovi serijalizacije i deserijalizacije izuzetno veliki, što uvelike ograničava vrste paralelizacije na koje je ovakav scenarij uopće primjenjiv, i to svakako nisu paralelizacije na najnižoj razini. Prednost ovog pristupa je što omogućava izrazito velik broj radnika koje je moguće upogoniti. Tako primjerice imati 80 radnika ili 160 radnika nije ništa neobično. Ovaj način paralelizacije najčešće se izvodi direktno uporabom mrežnog podsustava (primjerice, uporabom protokola TCP) a moguća je i uporaba specijaliziranih biblioteka poput biblioteke MPI koja programeru nudi jednostavan način uporabe ali i povećava troškove komuniciranja i zahtjeva prilagodbu podataka koji se šalju.

Konačno, s današnjim razvojem osobnih računala treba spomenuti još jedan način paralelizacije koji je vezan uz jedno računalo ali nudi relativno velik broj specijaliziranih radnika – radi se o grafičkom procesoru (engl. *Graphics Processing Unit*, GPU) koji je moguće iskoristiti za dobivanje čak i preko 100 procesnih jedinica. Međutim, kako su ti procesori daleko skromniji po mogućnostima od centralnog procesora, nisu prikladni za sve vrste problema. Dodatnu cijenu koju treba uzeti u obzir jest prošak prebacivanje podataka iz radne memorije računala u memoriju grafičke kartice da bi ih GPU uopće vidio te prebacivanje rezultata iz memorije grafičke kartice u radnu memoriju računala da bi ih program koji se izvršava na računalu mogao iskoristiti. Ovisno o vrsti problema koji se rješava, rezultati mogu biti ubrzanje od više stotina puta [Harding and Banzhaf, 2007], ili mogu biti još i lošiji no da se izvode samo na centralnom procesoru. Za programiranje algoritama za izvođenje na grafičkom procesoru danas se najčešće koristi CUDA od tvrtke NVidia.

Na koji način obaviti paralelizaciju, dosta je osjetljivo pitanje. Korist koju možemo dobiti od paralelizacije uobičajeno mjerimo pojmom *ubrzanje*. Promotrimo slučaj u kojem imamo slijedni algoritam koji posao obavlja u vremenu T. Označimo s $r_s = \frac{T_s}{T}$ postotak vremena u kojem se izvodi dio algo-

ritma koji nije moguće paralelizirati. Za dio algoritma koji je trošio preostalo vrijeme $T_p = T - T_s$ (uvedimo $r_p = (1 - r_s)$) pretpostavimo da smo algoritam uspjeli paralelizirati tako da se taj posao obavlja paralelno na k radnika. Time će vrijeme potrebno za izvođenje tog posla pasti na $T_p^{'} = \frac{T_p}{k}$, čime će za čitav posao trebati $T^{'} = T_s + T_p^{'}$. Ubrzanje S definira se kao omjer ova dva vremena:

$$S = \frac{T}{T^{'}} = \frac{T}{T_s + T_p^{'}} = \frac{T}{T \cdot r_s + \frac{T \cdot r_p}{k}} = \frac{T}{T \cdot \left(r_s + \frac{r_p}{k}\right)} = \frac{1}{r_s + \frac{r_p}{k}} = \frac{1}{(1 - r_p) + \frac{r_p}{k}},$$

što direktno odgovara ubrzanju definiranom Amdahlovim zakonom [Amdahl, 1967].

Pogledajmo što ovo ima za posljedicu. Pretpostavimo da imamo generacijski genetski algoritam koji radi s populacijom od 100 jedinki. Algoritam smo pokrenuli na jednoprocesorskom računalu; izmjerili smo da vrijeme potrebno za stvaranje nove populacije iznosi 10 ms, a od tog vremena na vrednovanje jedinki je potrošeno 6 ms. Ako se usredotočimo na paralelizaciju operatora vrednovanja, koliko najviše možemo ubrzati postupak stvaranja nove generacije ako možemo uzeti radnika proizvoljno mnogo? U ovom slučaju imamo $r_s = \frac{4}{10} = 0,4$ i $r_p = \frac{6}{10} = 0,6$. Limes izraza izvedenog za ubrzanje kada pustimo da $k \to \infty$ je

$$S|_{k \to \infty} = \frac{1}{r_s}$$

što je u našem slučaju jednako $\frac{1}{0.4} = 2,5$. Slijedi da, što god da radili, takvom paralelizacijom postupak možemo ubrzati najviše 2,5 puta, odnosno vrijeme skratiti s 10 ms na 4 ms. Dakako, ovo je gornja granica ubrzanja koja pretpostavlja da ne postoji trošak paralelizacije (sinkronizacije, serijalizacije, deserijalizacije, komunikacijski troškovi i slično). Uvedemo li trošak paralelizacije τ po svakom radniku definiran kao postotak s obzirom na vrijeme T, tada ukupno vrijeme uz paralelizaciju na k radnika možemo zapisati kao $T_p^{''} = \frac{T_p}{k} + k \cdot (\tau \cdot T)$. Tada će ubrzanje biti:

$$S = \frac{T}{T^{''}} = \frac{T}{T_s + T_p^{''}} = \frac{T}{T \cdot r_s + \frac{T \cdot r_p}{k} + k \cdot (\tau \cdot T)} = \frac{1}{r_s + \frac{r_p}{k} + k \cdot \tau} = \frac{1}{(1 - r_p) + \frac{r_p}{k} + k \cdot \tau}.$$

Da bi ubrzanje bilo veće od 1, nazivnik razlomka mora biti manji od 1. Međutim, trošak paralelizacije doprinosi iznosu vrijednosti nazivnika, i sasvim je moguće paralelizacijom generirati algoritam koji radi još i sporije no što je radio bez paralelizacije. Da se ovo ne bi dogodilo, član $k \cdot \tau$ mora biti zanemariv u odnosu na $\frac{r_p}{k}$, što pak znači da se paralelizacija isplati samo ako je trošak paralelizacije bitno manji od trajanja posla koji se paralelizira. Upravo iz ovog razloga nikada nećemo raditi paralelizaciju operatora mutacije na način da dijelove jedinke protokolom TCP šaljemo na k radnika i potom prikupljamo natrag rezultat – trošak paralelizacije tu će biti bitno veći od izvođenja operatora mutacije bez paralelizacije i takvom ćemo paralelizacijom usporiti a ne ubrzati algoritam.

Bibliografija

G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. ACM Press, 1967.

S. Harding and W. Banzhaf. Fast genetic programming on GPUs. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101, Valencia, Spain, 2007. Springer. ISBN 3-540-71602-5. doi: doi:10.1007/978-3-540-71605-1_9.

Poglavlje 17

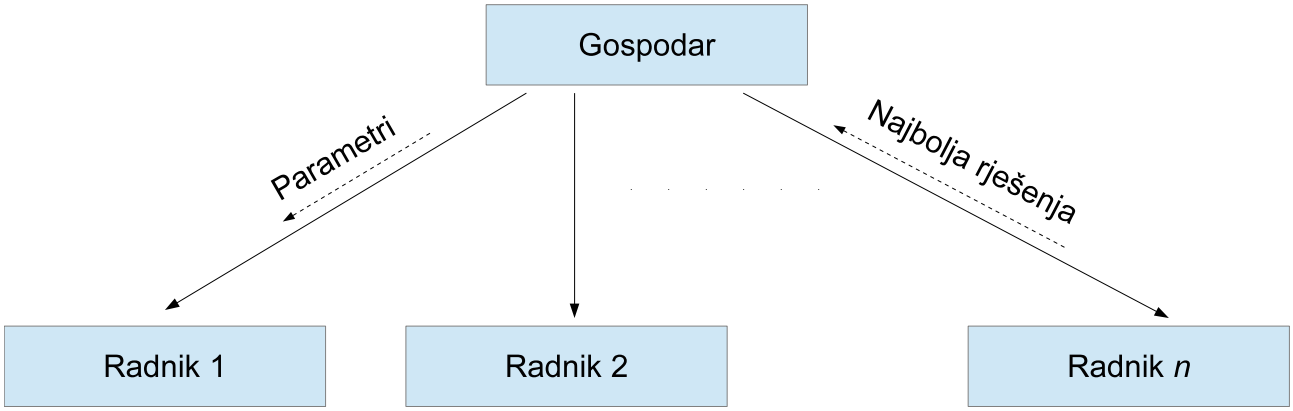
Neki od modela paralelizacije

17.1 Uvod

U prethodnom poglavlju ukratko smo razmotrili razloge za paralelizaciju algoritama te okvirnu podjelu načina na koje se to može obaviti. U ovom poglavlju pozabavit ćemo se s nekoliko klasičnih izvedbi paralelizacije, imajući pri tome u vidu samo implementacije prikladne za izvedbu na uobičajenim računalima – što računalima s više jezgri, što implementacije prikladne za više umreženih računala. Posebne vrste paralelizacije koje bi bile prikladne za specifično sklopovlje ovdje nećemo razmatrati.

17.2 Nesuradna paralelizacija

Jedan od najjednostavnijih oblika paralelizacija jest nesuradna paralelizacija, prikazana na slici 17.1. Pretpostavimo da na raspolaganju imamo klasičnu jednodretvenu implementaciju optimizacijskog algoritma; također pretpostavimo da taj optimizacijski algoritam prima dva argumenta: *opis problema* te *parametre*. Ideja nesuradne paralelizacije jest pokrenuti više algoritama koji rješavaju isti problem s potencijalno različitim parametrima koji utječu na rad algoritma. Promjena parametara ima smisla jer je poznato da zbog *no-free-lunch* teorema ne postoji najbolji optimizacijski algoritam; s druge strane, svaki novi skup parametara algoritma koji je moguće parametrizirati efektivno stvara "novi algoritam". Stoga je za očekivati da će uz konačno vrijeme izvođenja naš optimizacijski algoritam uz različite parametre biti različito uspješan.



Slika 17.1: Nesuradna paralelizacija

Kod nesuradne paralelizacije prikazane na slici 17.1 postoji glavni program (na slici prikazan kao *gospodar*) koji pokreće više primjeraka optimizacijskog algoritma (kao zaseban proces ili u novoj dretvi; na slici su prikazani kao *radnici*). Prilikom stvaranja primjerka optimizacijskog algoritma, algoritmu se predaje opis problema te parametri uz koje će raditi. Jednom kad su algoritmi gotovi, glavnom programu predaju najbolje pronađeno rješenje. Glavni program potom uspoređuje pronađena rješenja i ono najbolje vraća kao rezultat optimizacijskog procesa.

Primjer ovakve paralelizacije izveden kao *Bash*-skripta (Bash je ljuska na operacijskom sustavu Linux) prikazan je pseudokodom 17.1. Skripta pretpostavlja da smo opis problema pohranili u datoteku `problem.txt` te da želimo pokrenuti četiri primjerka algoritma; pri tome su parametri s kojima će raditi svaki od algoritama pripremljeni u datotekama `param1.txt` do `param4.txt`. Optimizacijski algoritam implementirali smo kao Java program `OptimAlgoritam` koji očekuje tri argumenta: opis problema koji rješava, parametre s kojima radi te datoteku u koju će zapisati podatke o najboljem pronađenom rješenju.

Pseudokod 17.1 Nesuradna paralelizacija izvedena kao Bash skripta.

```
#! /bin/bash

#Definiraj koliko radnika treba pokrenuti
RADNIKA=4

# Pokreni četiri optimizacijska algoritma
for i in $(seq 1 ${RADNIKA}); do
 java OptimAlgoritam --out=sol${i}.txt --in=problem.txt --params=param${i}.txt &
done

# Pričekaj da sva četiri završe
wait

# Pronađi najbolje:
java PickBest --inputs=sol1.txt...sol${RADNIKA}.txt --output=final.txt
```

U ovom rješenju prikazana Bash skripta je "gospodar" koji pokreće četiri radnika kao zasebne procese; broj je lagano promijeniti jer je definiran kao varijabla skripte. Zahvaljujući znaku `&` na kraju naredbe za pokretanje radnika, Bash će radnika pokrenuti u pozadini i odmah nastaviti izvođenje sljedeće naredbe. Time će svi radnici biti pokrenuti jedan za drugim bez čekanja. Nakon toga izvođenje ljuske će zapeti na naredbi `wait`. Ta će naredba uzrokovati čekanje dok svi procesi koje je skripta pokrenula ne završe sa izvođenjem; u našem primjeru to će se dogoditi kada svi pokrenuti radnici završe s izvođenjem. Nakon toga pokreće se Java program `PickBest` koji kao argument dobiva raspon datoteka s rješenjima (u svakoj datoteci se nalazi najbolje pronađeno rješenje odgovarajućeg radnika te njegova dobrota). Drugi argument programa je datoteka u koju treba prekopirati najbolje pronađeno rješenje.

Ovakvu paralelizaciju ima smisla raditi i ako se svi radnici pokreću s jednakim parametrima. Naime, prisjetimo se da su optimizacijski algoritmi o kojima je ovdje riječ algoritmi koji prostor pretražuju pod utjecajem slučajnog mehanizma; stoga su osjetljivi na slučajno generirana početna rješenja kao i na odluke koje se temeljem slučajnog mehanizma donose tijekom izvođenja optimizacije. Posljedica je da svaki algoritam može (i najčešće za teže probleme hoće) završiti s nekim drugim, različito kvalitetnim, rješenjem.

Ovaj oblik paralelizacije često se izvodi i skroz ručno, i poznat je još i pod nazivom *trivijalna paralelizacija*.

Umjesto različitih procesa, ovakva se paralelizacija može izvesti i uporabom višedretvenosti. Primjer prikazuje izvorni kod 17.1. Primjer pretpostavlja da su zasebnim razredima modelirani:

- razredom `Problem` problem koji je potrebno riješiti,
- razredom `Parametri` parametri uz koje će optimizacijski algoritam rješavati problem,
- razredom `OptimAlgoritam` optimizacijski algoritam te
- razredom `Solution` jedno rješenje problema koji se rješava.

Ispis 17.1: Nesuradna paralelizacija ostvarena uporabom višedretvenosti u programskom jeziku Java.

```
1  public class Nesuradna {
2
3    public static void main(String[] args) {
4
5      int brojRadnika = 4;
6
7      final Problem problem = ucitajProblem();
8
9      // Priprema i pokretanje radnika:
10     Thread[] radnici = new Thread[brojRadnika];
11     OptimAlgoritam[] algoritmi = new OptimAlgoritam[brojRadnika];
12     for(int i = 0; i < brojRadnika; i++) {
13       Parametri params = pripremiParametre(i);
14       final OptimAlgoritam alg = new OptimAlgoritam(problem, params);
15       algoritmi[i] = alg;
16       radnici[i] = new Thread(new Runnable() {
17         public void run() {
18           alg.optimiraj();
19         }
20       });
21       radnici[i].start();
22     }
23
24     // Cekanje da svi radnici zavrse:
25     for(int i = 0; i < brojRadnika; i++) {
26       try { radnici[i].join(); } catch(Exception ex) {}
27     }
28
29     // Trazenje najboljeg rjesenja:
30     Solution best = algoritmi[0].vratiNajbolje();
31     for(int i = 1; i < brojRadnika; i++) {
32       Solution s = algoritmi[i].vratiNajbolje();
33       if(s.dobrota > best.dobrota) best = s;
34     }
35
36     // Pohrana rjesenja
37     pohrani(best);
38   }
39
40
41   static Parametri pripremiParametre(int index) {
42     // Pripremi parametre za index-ti algoritam
43   }
44
45   static Problem ucitajProblem() {
46     // Vrati problem koji rjesavamo.
47   }
48
49   static void pohrani(Solution sol) {
50     // Pohrani rjesenje, primjerice na disk.
51   }
52 }
```

Napomenimo da ovakvu paralelizaciju, bilo onu prikazanu pseudokodom 17.1 ili onu prikazanu ispisom 17.1 ima smisla raditi s najviše onoliko radnika koliko računalo ima jezgri na raspolaganju. Naime, obje prikazane izvedbe paralelizacije pokreću se na jednom računalu, a kako su optimizacijski algoritmi doslovno programi drobljenja brojeva (nema ulazno/izlaznih operacija i slično), svaki pokrenuti radnik može zauzeti po jednu jezgru i držati je na 100% opterećenja. Ako bismo pokrenuli više radnika no što fizički imamo jezgri, pojavila bi se situacija da se više radnika natječe za istu jezgru – posljedica bi bio neučinkovitiji rad jer bi se pojavili troškovi promjene konteksta koje bi operacijski sustav bio prisiljen raditi kako bi periodički različite radnike puštao na istu jezgru.

Nesuradnu paralelizaciju moguće je ostvariti i s radnicima koji su raspodijeljeni na više računala, uz pretpostavku da su računala umrežena. Više o tome će biti riječi kasnije.

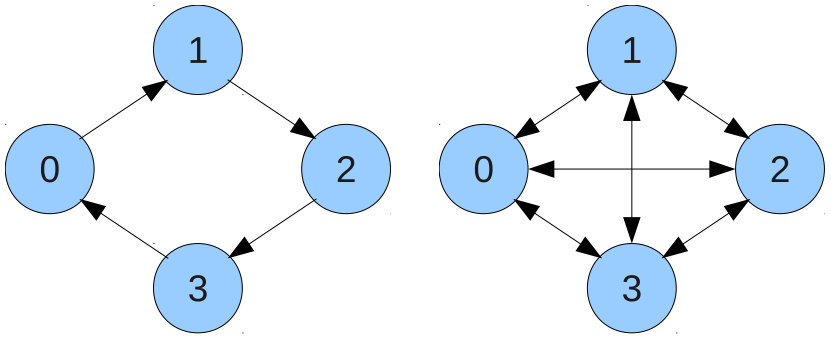
17.3 Suradna paralelizacija

Umjesto nesuradne paralelizacije, ako na raspolaganju imamo više jezgri i/ili računala, relativno jednostavno možemo napraviti suradnu paralelizaciju. Pretpostavimo da imamo više radnika i da svaki radnik izvodi jedan primjerak optimizacijskog algoritma. Nadogradimo sada optimizacijske algoritme tako da im dodamo još samo dvije metode prikazane u nastavku.

```
1  public Solution getSomeSolution() {...}
2  public void useSolution(Solution offering) {...}
```

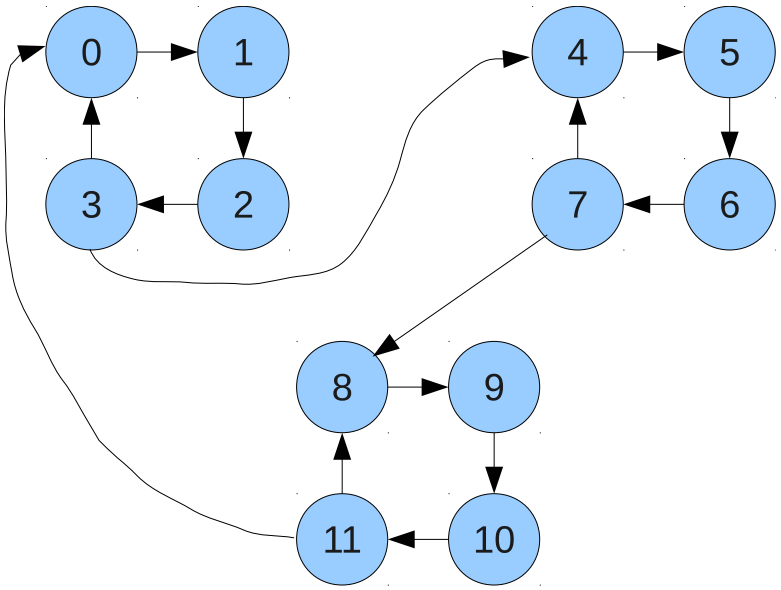
Od metode getSomeSolution() očekujemo da će vratiti neko rješenje iz populacije (možemo se čak dogovoriti da će metoda uvijek vratiti kopiju trenutno najboljeg rješenja koje ima). S druge pak strane, od metode useSolution očekujemo da će rješenje koje joj se preda iskoristiti kako bi poboljšala vlastitu populaciju. Jedna mogućnost jest u vlastitoj populaciji potražiti najgore rješenja, njega izbaciti i na upražnjeno mjesto dodati ponuđeno rješenje.

Ako imamo te dvije metode na raspolaganju, tada se čitav program može složiti na način da gospodar periodički pita svakog od radnika za kopiju najboljeg rješenja i potom te kopije ponudi nekim drugim radnicima. Radimo li višedretvenu aplikaciju, tada je ovo doista trivijalno za izvesti. Odluka tko će kome slati rješenja definirana je *topologijom*. Primjeri nekih od topologija prikazani su na slici 17.2; radnici su predstavljeni kružićima a smjer razmjene rješenja pokazuju strelice.



(a) Topologija prstena

(b) Potpuno povezana topologija



(c) Topologija ugniježđenih prstena

Slika 17.2: Primjeri različitih topologija

Ovdje treba uočiti da postoji još niz pitanja na koja nismo dali odgovor, ali idemo ih barem postaviti.

- *Treba li postojati gospodar?* Ulogu gospodara u prethodnom smo primjeru opravdali potrebom da *netko* periodički proziva radnike, prikuplja jedinke i dostavlja ih drugim radnicima. To međutim

ne mora biti uloga gospodara – radnici i sami mogu biti svjesni trenutne topologije i periodički slati i primati rješenja. Ako se čitav proces izvodi na jednom računalu, tada bi to moglo zakomplicirati program jer umjesto da odluka bude centralizirana na razini gospodara, u opisanom scenariju bismo je morali implementirati kod svih radnika. Međutim, u sustavu koji se sastoji od mnoštva radnika koji se izvode na različitim računalima i komuniciraju putem mreže, centralni gospodar može postati usko grlo – u tom scenariju dodatni trošak implementacije raspodijeljene logike u radnike može se isplatiti. S druge pak strane, nas u konačnici zanima najbolje pronađeno rješenje; ako se postupak izvodi na više računala, nije nam prihvatljivo da moramo trčkarati od računala do računala i na svakom po datotekama tražiti koliko je kvalitetno rješenje pronađeno. Stoga je u takvom sustavu opravdano postojanje i gospodara, ali često samo kao onoga tko na početku pokrene sve radnike i potom samo periodički prikuplja najbolja rješenja koja i sami radnici mogu dojavljivati kada ih pronađu – time se pak izbjegava potreba da gospodar propituje svako malo ima li čega novoga.

- *Koju topologiju uzeti?* Prikladna topologija ovisi o problemu koji se rješava. Koliko će se koristiti radnika često je određeno naprosto količinom računalnih resursa koje imamo na raspolaganju. Kako te radnike povezati – samo eksperimentiranje može dati konačni odgovor. Pri tome rreba voditi računa o fizičkim ograničenjima; primjerice, ako se koristi potpuno povezana topologija (svaki radnik šalje rješenje svim drugim radnicima), sama mreža preko koje se šalju poruke može postati usko grlo, pa time potpuno povezana topologija (kao i slične) mogu postati neprikladne.

- *Koliko često se rješenja razmjenjuju?* Razmjena može biti sinkrona – u tom slučaju određena je parametrom koji se zove *migracijski interval*. Primjerice, migracijski interval može biti 5 sekundi, što znači da će svakih 5 sekundi radnici predati neko svoje rješenje i primiti rješenja od drugih. Razmjena međutim može biti i asinkrona – primjerice, u scenariju u kojem radnik samostalno donosi odluku i šalje novo rješenje svaki puta kada pronađe neko bolje rješenje; tako dugo dok ne pronalazi bolja rješenja, ništa ne šalje dalje. Ako se koristi sinkrona razmjena, treba pripaziti na iznos migracijskog intervala: ako je preveliki, dobit će se efekt nesuradne paralelizacije jer radnici gotovo pa da i ne razmjenjuju rješenja. Ako je premali, dobit će se efekt jedne velike populacije što je opet loše – bolje je pustiti svakog od radnika da neko vrijeme nezavisno istražujeneki podprostor u prostoru rješenja.

- *Što se točno šalje?* Da li samo jedno rješenje, i ako da, koje (je li nužno da to bude najbolje?), ili se možda šalje više rješenja? Ako da, kako se izabiru? Treba uočiti da što se šalju kvalitetnija rješenja, i što je intenzitet te razmjene veći (migracijski interval kraći), u čitavom distribuiranom sustavu efektivno dolazi do porasta selekcijskog pritiska – bolja rješenja brže se šire kroz populacije i doprinose fokusiranju populacije.

- *Na koji se način rješenja ugrađuju?* Ako se radi o genetskom algoritmu, odgovor je jasan – možemo ga naprosto dodati u populaciju. No što ako su algoritmi primjerice mravlji? Mravlji algoritam nema populacije u koju bi dodao trenutno rješenje. Jedna mogućnost je tada koristiti to rješenje za dodatno modificiranje feromonskih tragova. A tada se opet otvara niz pitanja – možda prvo i najvažnije: koliko jako to rješenje treba utjecati na postojeće feromonske tragove?

- *Da li koristiti homogeni ili heterogeni sustav?* Homogeni sustav pretpostavlja da svi radnici izvode identičan algoritam. Međutim, to ne mora biti tako. Svaki radnik bi mogao izvoditi isti algoritam ali uz različito podešene parametre. Štoviše, različiti radnici bi mogli izvoditi i različite optimizacijske algoritme – u tom slučaju govorimo o heterogenom sustavu. Različiti algoritmi na različit način pretražuju prostor rješenja i koriste različite algoritme – što je za jedan algoritam lokalni optimum (zbog reprezentacije i operatora koje koristi), to za neki drugi algoritam ne mora biti lokalni optimum; u slučaju suradnje lako je zamisliti da će se postići bolji rezultati.

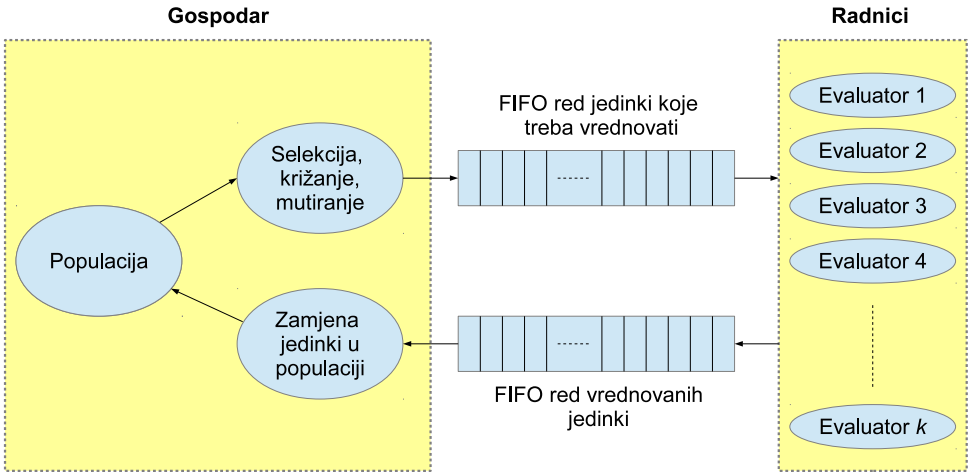
Spomenimo još jedan problem koji se javlja već i kod paralelizacije uporabom višedretvenosti. Pretpostavimo da u sustavu postoji više radnika (svakog izvodi jedna dretva) te gospodar koji se brine za razmjenu rješenja. Gospodar će periodički nad svakim algoritmom pozvati metodu `getSomeSolution()` kako bi dobio neko rješenje. No što bi ta metoda trebala vratiti? Prisjetimo se da je ovaj poziv

napravljen iz dretve koja izvodi gospodara – u tom istom trenutku radnik može biti usred posla stvaranja nove populacije, i trenutna populacija tada možda uopće ne postoji. Gospodar i radnik ovdje će se morati međusobno sinkronizirati – gospodar će trebati pričekati dok radnik s izvođenjem optimizacijskog algoritma ne dođe do točke u kojoj gospodara može smisleno poslužiti. Ovo će pak u gospodarev posao unijeti kašnjenja – od trenutka kada odluči da treba pokupiti rješenja od svih radnika do trenutka kad je to doista uspio i dočekati može proći neprihvatljivo puno vremena. Jedno moguće rješenje jest pisati optimizacijski algoritam tako da uvijek ima referencu na najbolje pronađeno rješenje (koje ažurira svaki puta kada pronađe bolje), te pisati metodu getSomeSolution() tako da naprosto pročita referencu na to rješenje. S obzirom da će tu referencu postavljati jedna dretva a čitati druga, nuno je osigurati da objekt na koji referenca pokazuje bude nepromjenjiv, te da postavljanje reference te čitanje reference bude izvedeno na višedretveno siguran način. Za to se može koristitineki sinkronizacijski mehanizam; međutim, preporučio bih da se za ovo koristi AtomicReference, razred koji je u Javi prisutan od uvođenja paketa java.util.concurrent i koji čitanje i zamjenu reference obavlja uporabom sklopovski podržane instrukcije CAS ("compare & set") štoje često najbrži mogući način.

Implementacija metode useSolution može biti još problematičnija – ovisno o topologiji, svaki radnik s vremena na vrijeme može dobiti jedno ili više rješenja koje treba ugraditi u populaciju. Međutim, trenutci u kojima se ta rješenja mogu ugraditi su još rijeđi i specifični od algoritma do algoritma. Primjerice, kod generacijskog genetskog algoritma praktički jedini smisleni trenutak kada se rješenje može ugraditi u populaciju je na samom kraju generacijskog ciklusa, kad su obavljena sva križanja, mutacije i vrednovanja djece te kada temeljem stare populacije roditelja i nove populacije djece treba odlučiti koje će jedinke ući u novu populaciju roditelja za sljedeću generaciju. Na tom mjestu odabir bi se mogao proširiti tako da se u obzir uzmu i pridošle jedinke drugih algoritama. Ako bi gospodar svo to vrijeme bio zablokiran, njegov bi posao postao opasno ugrožen. Stoga je preporuka svaki od algoritama opskrbiti vlastitim privatnim redom pristiglih rješenja. Svaki puta kada gospodar pripremi jedno ili više rješenja, pozivom metode useSolution rješenja se samo ubace u taj red i gospodar se blokira minimalno (koliko je potrebno da na višedretveno siguran način rješenja ubaci u tu strukturu podataka); s druge strane, optimizacijski algoritam, u trenutcima kada to njemu odgovara treba provjeriti ima li kakvih rješenja u njegovom redu, i ako ima, isprazniti red i obraditi rješenja. Ovdje se kao red mogu koristiti sinkronizirane verzije lista ili baš kolekcije tipa red koje su dostupne u okviru paketa java.util.concurrent.

17.4 Paralelizacija na razini populacije

Paralelizacija na razini populacije kod većine populacijskih optimizacijskih algoritama predstavlja prirodan i često najjednostavniji način paralelizacije, tako dugo dok se čitav program piše za izvođenje na jednom računalu. Naime, u tom slučaju paralelizacija se može obaviti uporabom višedretvenosti što čitav proces čini izuzetno jednostavnim. Prije no što se osvrnemo na ovaj oblik paralelizacije, pogledajmo ovu vrstu paralelizacije u malo širem kontekstu.



Slika 17.3: Paralelizacija na razini populacije

Slika 17.3 prikazuje koncept izvedbe paralelizacije na razini populacije. Uz pretpostavku da je najskuplja operacija optimizacijskog procesa upravo vrednovanje, slika 17.3 prikazuje pristup kod kojeg je gospodar zadužen za održavanje populacije, provođenje odabira roditelja, križanje i mutiranje, dok su zasebni paralelni "entiteti" zaduženi samo za provođenje procesa vrednovanja.

Zamislimo jedan klasični generacijski genetski algoritam. Na početku optimizacijskog postupka gospodar stvara inicijalnu populaciju rješenja. Najčešće, to se obavlja posredstvom slučajnog mehanizma i može biti brzo gotovo. Jednom stvorenu početnu populaciju jedinki potrebno je vrednovati. U tu svrhu gospodar svaku od jedinki prije dodavanja u populaciju umeće u FIFO red nevrednovanih jedinki (FIFO – engl. *First In, First Out*; na slici 17.3 to je gornji red). U sustavu osim radnika postoji i k radnika koji su pokrenuti istovremeno kada je pokrenut i gospodar (u nekim izvedbama jedna od uloga gospodara može biti i pokretanje radnika). Svaki od radnika po pokretanju ostaje zaglavljen u pokušaju da iz reda nevrednovanih jedinki preuzme jedinku i da započne proces vrednovanja. Posao koji obavlja radnik prikazan je pseudokodom 17.2.

Pseudokod 17.2 Posao koji obavlja radnik.

```
ponavljaj dok nije signaliziran kraj
  Solution s = izvadi jedinku iz reda nevrednovanih jedinki;
  Provedi vrednovanje jedinke (s);
  Ubaci jedinku (s) zajedno s mjerom dobrote u red vrednovanih jedinki;
kraj ponavljaj
```

Ubacivanjem svih n jedinki koje trebaju postati početna populacija u red nevrednovanih jedinki, gospodar će iz reda vrednovanih jedinki pokušati pročitati n jedinki i dodati iz u početnu populaciju. Pokušaj čitanja će ga, međutim, zamrznuti jer u tom redu još nema vrednovanih jedinki. Paralelno s time, radnici će početi vaditi jednu po jednu jedinku, provest će vrednovanje i rezultat će ubaciti u red vrednovanih jedinki. Kako u sustavu postoji k radnika koji vrednovanje obavljaju paralelno, te kako je broj jedinki koje treba vrednovati jednak n, očekujemo da ćemo postići ubrzanje od približno $\lceil \frac{n}{k} \rceil$ u odnosu na situaciju u kojoj se vrednovanje svih n jedinki provodi slijedno.

Malo po malo, kako radnici riješe vrednovanje i rezultate ubace u red vrednovanih jedinki, gospodar će se periodički odmrzavati i iz tog reda vaditi jednu po jednu vrednovanu jedniku koju će potom dodati u početnu populaciju. Kada dočeka svih n jedinki, gospodar ulazi u glavnu petlju generacijskog genetskog algoritma:

1. provodi selekciju roditelja,
2. križa roditelje kako bi dobio djecu,
3. provodi mutiranje djece,
4. dodaje m stvorene djece u red nevrednovanih jedinki,
5. iz reda vrednovanih jedinki čita m jedinki koje čine vrednovanu populaciju djece te
6. temeljem n roditelja i m stvorene i vrednovane djece odabire n novih jedinki koje će postati roditelji za sljedeću generaciju.

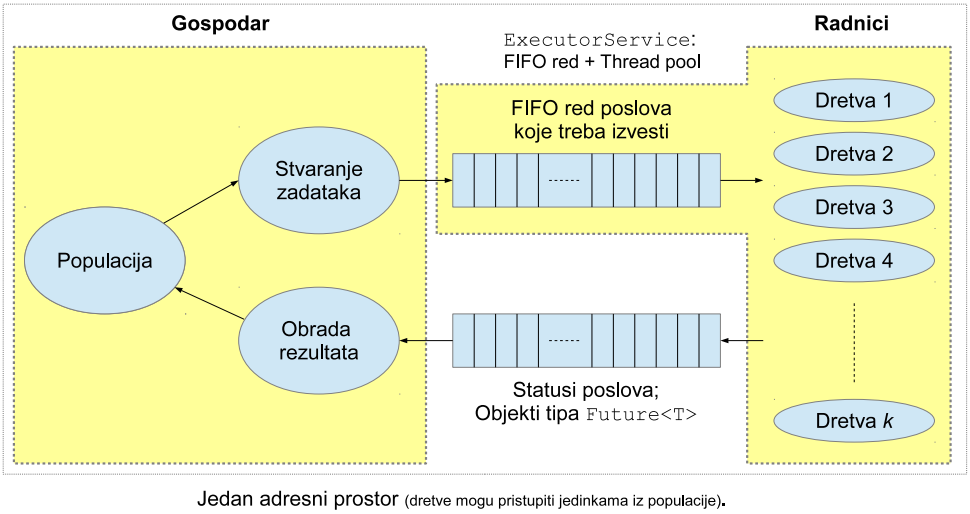
Opisani postupak pri tome je generalno primjenjiv na različite populacijske algoritme a ne samo na genetski algoritam na kojem je paralelizacija ilustrirana; pri tome nazive "križaj/mutiraj" te "zamijeni" samo treba prilagoditi trenutnom algoritmu koji se paralelizira.

Ako pretpostavka da je vrednovanje dominantno najskuplja operacija ne stoji, već ako postupci izbora roditelja te provođenja križanja i mutacije troše nezanemarivo vrijeme u odnosu na postupak vrednovanja, moguće je napraviti postupak paralelizacije kod kojeg sve te operacije provode radnici. Da bi se to moglo provesti efikasno, nužna je pretpostavka da radnici mogu jeftino doći do trenutne populacije i pridruženih dobrota; ta je pretpostavka ispunjena ako se kao paralelizacijski model uzme izrada višedretvene aplikacije na računalu koje ima odgovarajući broj jezgri na raspolaganju; u tom

slučaju, i gospodar i svaki od radnika izvode se u svojim vlastitim dretvama unutar dijeljenog adresnog prostora aplikacije. U tom slučaju svaki od radnika direktno ima pristup svim jedinkama populacije bez ikakvih dodatnih troškova. Kod takve izvedbe paralelizacije red nevrednovanih jedinki možemo općenitije promatrati kao red poslova koje treba provesti. Gospodar će u taj red ubaciti m poslova, gdje svaki posao predstavlja zadatak tipa: *"evo ti populacija, provedi postupak odabira roditelja, križaj ih i stvori dijete, mutiraj ga, vrednuj i ubaci u red vrednovanih jedinki"*. Gospodar potom iz reda vrednovanih jedinki opet čita m jedinki koje su nastale kao rezultat rada radnika, te iz populacije djece i stare populacije roditelja odabire jedinke koje će ući u novu populaciju roditelja, čime se čitav proces može ponavljati potreban broj puta.

U programskom jeziku Java, opisani je model relativno jednostavno implementirati koristeći gotove primitive koje nudi sam jezik. Svaki student koji je odslušao osnove operacijskih sustava trebao bi znati kako uporabom monitora ostvariti višedretveno siguran red, a uz dostupne ugrađene kolekcije i primitive jezika poput ključnih riječi **synchronized**, metode Object.wait(), Object.notify() te Object.notifyAll(), mogućnosti stvaranja dretvi uporabom razreda Thread i modeliranja poslova preko sučelja Runnable to bi trebao biti relativno jednostavan zadatak.

Međutim, kako višedretveno programiranje ipak predstavlja dodatni napor i često je podložno unosu suptilnih pogrešaka, u nastavku ćemo se osvrnuti na mogućnost implementacije ovake paralelizacije uporabom nešto viših primitiva koji su u Javi dostupni od uvođenja paketa `java.util.concurrent`.



Slika 17.4: Paralelizacija na razini populacije uporabom paketa java.tuil.concurrent.

Rješenje koje ćemo analizirati prikazano je na slici 17.4. U okviru paketa `java.util.concurrent` programerima je na raspolaganje dan model paralelizacije koji se temelji na sučelju ExecutorService, usluzi koja sama održava red poslova, omogućava zadavanje jednog ili više poslova koje je potrebno obaviti i pojednostavljuje dohvat rezultata. Paket `java.util.concurrent` programerima na raspolaganje stavlja čitav niz različitih implementacija ovog sučelja: od implementacije koja koristi samo jednu dretvu i sve poslove izvodi slijedno redoslijedom kojim su pristigli pa do implementacija koje same održavaju vlastite "bazene" dretvi koje će izvoditi poslove. Odabir željene implementacije kao i njezino konfiguriranje provodi se preko statičkih metoda razreda Executors. Tako se, primjerice, stvaranje FIFO reda povezanog s fiksnim brojem dretvi koje dohvaćaju poslove i izvode ih postiže pozivom:

```
1 int numberOfThreads = Runtime.getRuntime().availableProcessors();
2
3 ExecutorService executorService = Executors.newFixedThreadPool(
4   numberOfThreads
5 );
```

Koristi se statička metoda newFixedThreadPool koja kao argument prima broj dretvi koje trebaju opsluživati red poslova. U prethodnom isječku taj se broj dinamički utvrđuje i postavlja na raspoloživi broj jezgri računala na kojem se program izvodi. Konkretna implementacija generacijskog genetskog algoritma koja se temelji na opisanom paketu prikazana je na ispisu 17.2 dok su pomoćne metode i

podrazredi prikazani na ispisu 17.3.

U paketu `java.util.concurrent` svaki je posao modeliran sučeljem Callable<T>. Naime, sučelje Runnable koje se tipično koristi kada se direktno radi s dretvama predviđeno je za opisivanje koda koji je potrebno izvesti u drugoj dretvi – koda koji ništa ne proizvodi. Sučelje Callable<T> je parametrizirano i omogućava modeliranje posla koji će proizvesti upravo jedan objekt čiji je tip zadan parametrom T. Sučelje Callable prikazano je u nastavku.

```
1 public interface Callable<T> {
2   public T call();
3 }
```

Sučelje definira samo jednu metodu: call koja ne prima argumente a vraća referencu na objekt koji je proizveden. Dretve koje se nalaze u pridruženom bazenu iz reda će vaditi jedan po jedan posao i pozvati metodu call kako bi obavile zadan posao. Za umetanje poslova u red poslova ExecutorService nudi niz pomoćnih metoda od kojih ćemo istaknuti dvije prikazane u nastavku.

```
1 public <T> Future<T> submit(Callable<T> task);
2 public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);
```

Za svaki posao predan preko te dvije metode ExecutorService vraća objekt tipa Future<T> koji predstavlja omotač oko rezultata koji će predani posao proizvesti. Metoda submit prima samo jedan posao pa vraća jedan omotač, dok metoda invokeAll prima listu poslova pa vraća listu omotača. Sučelje Future<T> nudi niz metoda kojima se može kontrolirati što se događa s poslom (primjerice, je li posao izveden, a omogućava i otkazivanje izvođenja posla); međutim, najčešće korištena metoda ovog sučelja je metoda:

```
1 public <T> get();
```

Pozivom metode get dohvaća se rezultat koji je posao proizveo i vraća se pozivatelju; dakako, ako je do tog trenutka posao doista i izveden i rezultat pripremljen; ako nije, pozivatelj će se zablokirati na tom pozivu sve dok se posao ne izvede i dok se ne pripremi rezultat.

Iz dosadašnjeg opisa trebalo bi biti jasno kako se može izvesti paralelizacija uporabom sučelja ExecutorService:

- stvori se objekt tipa ExecutorService uporabom statičkih metoda razreda Executors,
- rješenje se modelira razredom ili sučeljem (u ispisu 17.2 korišten je razred BitvectorSolution),
- posao se modelira razredom koji implementira sučelje Callable<BitvectorSolution> (drugim rječima, kao posao koji proizvodi objekt tipa BitvectorSolution križanjem, mutacijom i vrednovanjem),
- u red se preda n takvih poslova i dobije se n objekata tipa Future<BitvectorSolution> nad kojima se pozivom metode get dočekaju/dobiju proizvedene jedinke te
- između roditelja i dočekanih jedinki koje čine djecu odabere ne populacija roditelja za sljedeću generaciju.

Primjeri prikazani na ispisima 17.2 i 17.3 predstavljaju najveći dio programa koji uporabom generacijskog genetskog algoritma minimizira zadanu funkciju. Sada je pravi trenutak da malo detaljnije proučite taj kod. Uočite također da je na kraju izvođenja optimizacijskog procesa potrebno nad bazenom dretvi pozvati metodu shutdown, kako bi se sve dretve pogasile i time omogućilo uredno terminiranje procesa.

Ispis 17.2: Paralelizacija na razini populacije ostvarena uporabom višedretvenosti u programskom jeziku Java.

```
1  package hr.fer.zemris.optim.paral;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.List;
6  import java.util.Random;
7  import java.util.concurrent.Callable;
8  import java.util.concurrent.ExecutionException;
9  import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.Future;
12 import hr.fer.zemris.optim.paral.EvoUtil.EvoThread;
13 import hr.fer.zemris.optim.sa.BitvectorSolution;
14 import hr.fer.zemris.optim.sa.IFunction;
15 import hr.fer.zemris.optim.sa.NaturalBinaryDecoder;
16
17 /** Program koji ilustrira paralelizaciju na razini populacije **/
18 public class ParalelGA1 {
19
20   /**
21    * Glavni program -- ilustracija paralelizacije GA na razini populacije
22    */
23   public static void main(String[] args)
24     throws InterruptedException, ExecutionException {
25
26     // Pripremi funkciju koja se minimizira:
27     IFunction f = new IFunction() {
28       @Override
29       public double valueAt(double[] point) {
30         return (point[0]-4)*(point[0]-4)+(point[1]+2)*(point[1]+2);
31       }
32     };
33
34     // Veličina populacije, binarni dekoder te vjerojatnost mutacije bita:
35     int popSize = 50;
36     NaturalBinaryDecoder decoder = new NaturalBinaryDecoder(-10, 10, 15, 2);
37     double pm = 2.0 / decoder.getTotalBits();
38
39     // Generator slučajnih brojeva, inicijalizacija populacije:
40     Random rand = new Random();
41     BitvectorSolution[] population =
42       EvoUtil.initializePopulation(popSize, rand, decoder);
43     for(int i = 0; i < population.length; i++) {
44       population[i].fitness = -f.valueAt(decoder.decode(population[i]));
45     }
46
47     // Stvaranje thread-pool-a koji će obavljati zadatke:
48     ExecutorService executorService = Executors.newFixedThreadPool(
49       Runtime.getRuntime().availableProcessors(),
50       new EvoUtil.EvoThreadFactory()
51     );
52
53     // Generacijska petlja:
54     for(int generation = 1; generation <= 500; generation++) {
55       // Stvori listu poslova koje treba napraviti:
56       List<Callable<BitvectorSolution>> jobs = new ArrayList<>();
57       for(int i = 0; i < population.length; i++) {
58         jobs.add(new Job(population, pm, f, decoder));
59       }
60       // Pošalji poslove u thread-pool na izvođenje:
61       List<Future<BitvectorSolution>> results = executorService.invokeAll(jobs);
62       // Dočekaj stvorenu djecu i pohrani ih u populaciju djece:
63       BitvectorSolution[] children = new BitvectorSolution[population.length];
```

```
64        for(int i = 0; i < population.length; i++) {
65          children[i] = results.get(i).get();
66        }
67        // Odaberi temeljem populacije roditelja i djece tko čini novu populaciju:
68        population = selectNewPopulation(population, children);
69      }
70
71      // Pronađi i ispiši najbolje rješenje:
72      BitvectorSolution best = EvoUtil.findBest(population);
73      System.out.println("Najbolje rješenje je: "+Arrays.toString(decoder.decode(best)));
74
75      // Ugasi thread-pool:
76      executorService.shutdown();
77    }
78
79    // Primjer odabira nove populacije: vrati najbolje iz unije
80    // trenutnih roditelja i djece
81    private static BitvectorSolution[] selectNewPopulation(
82      BitvectorSolution[] population, BitvectorSolution[] children) {
83      return EvoUtil.selectBestFromUnion(population, children);
84    }
85
86    /** Razred koji modelira posao odabira dva roditelja, stvaranja djeteta uporabom
87        križanja i mutiranja, vredonovanje nastalog djeteta te vraćanje **/
88    static class Job implements Callable<BitvectorSolution> {
89
90      private BitvectorSolution[] population;
91      private double pm;
92      private IFunction f;
93      private NaturalBinaryDecoder decoder;
94
95      // Konstruktor
96      public Job(BitvectorSolution[] population, double pm, IFunction f,
97        NaturalBinaryDecoder decoder) {
98        super();
99        this.population = population;
100       this.pm = pm;
101       this.f = f;
102       this.decoder = decoder;
103     }
104
105     // Metoda u kojoj se obavlja stvaranje i vrednovanje djeteta.
106     @Override
107     public BitvectorSolution call() throws Exception {
108       Random rand = ((EvoThread)Thread.currentThread()).getRand();
109       BitvectorSolution[] parents =
110         EvoUtil.proportionalRelativeChoose(population, rand, 2);
111       BitvectorSolution child =
112         EvoUtil.uniformCrossover(parents[0], parents[1], rand);
113       EvoUtil.binaryMutate(child, rand, pm);
114       double fValue = f.valueAt(decoder.decode(child));
115       child.fitness = -fValue;
116       return child;
117     }
118   }
119 }
```

Ispis 17.3: Implementacija niza pomoćnih procedura koje će koriste kod populacijskih algoritama napisana u programskom jeziku Java.

```
1  package hr.fer.zemris.optim.paral;
2
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.Random;
6  import java.util.concurrent.ThreadFactory;
7
8  import hr.fer.zemris.optim.sa.BitvectorSolution;
9  import hr.fer.zemris.optim.sa.NaturalBinaryDecoder;
10
11 /**
12  * Razred koji sadrži često korištene metode koje koriste različiti populacijski
13  * algoritmi.
14  */
15 public class EvoUtil {
16
17   // ------------------ PODRŠKA ZA VIŠEDRETVENOST ------------------
18
19   /**
20    * Apstraktna tvornica za stvaranje dretvi koje će se koristiti za izvođenje poslova
21    * vezanih uz optimizacijske algoritme. Metoda garantira da će nove dretve biti
22    * primjerci razreda {@link EvoThread} čime će dretva automatski imati na raspolaganju
23    * svoj privatni generator slučajnih brojeva koji je dostupan pozivom
24    * {@link EvoThread#getRand()}.
25    */
26   public static class EvoThreadFactory implements ThreadFactory {
27     @Override
28     public Thread newThread(Runnable r) {
29       return new EvoThread(r);
30     }
31   }
32
33   /**
34    * Dretva za izvođenje poslova vezanih uz optimizacijske algoritme. Svojim
35    * korisnicima na raspolaganje daje privatni generator slučajnih brojeva
36    * dostupan pozivom {@link EvoThread#getRand()}.
37    */
38   public static class EvoThread extends Thread {
39
40     private Random rand = new Random();
41
42     public EvoThread(Runnable target) {
43       super(target);
44     }
45
46     public Random getRand() {
47       return rand;
48     }
49   }
50
51   // ------------------ INICIJALIZACIJE, POMOĆNE METODE ------------------
52
53   /**
54    * Metoda inicijalizira populaciju zadane veličine koristeći broj bitova kako to
55    * diktira predani dekoder. Jedinke se postavljaju na nasumično odabrane bitove.
56    *
57    * @param popSize koliko veliku populaciju treba stvoriti
58    * @param rand generator slučajnih brojeva
59    * @param decoder dekoder
60    * @return novu populaciju
61    */
62   static BitvectorSolution[] initializePopulation(int popSize, Random rand,
63     NaturalBinaryDecoder decoder) {
```

```
64      BitvectorSolution[] population = new BitvectorSolution[popSize];
65      for(int i = 0; i < population.length; i++) {
66        population[i] = new BitvectorSolution(decoder.getTotalBits());
67        population[i].randomize(rand);
68      }
69      return population;
70    }
71
72    /**
73     * Metoda u polju rješenja pronalazi i vraća rješenje s najvećom dobrotom.
74     *
75     * @param population populacija koju treba pretražiti
76     * @return najbolje rješenje
77     */
78    public static BitvectorSolution findBest(BitvectorSolution[] population) {
79      BitvectorSolution best = population[0];
80      for(int i = 1; i < population.length; i++) {
81        if(population[i].fitness > best.fitness) {
82          best = population[i];
83        }
84      }
85      return best;
86    }
87
88    // ------------------ SELEKCIJE ------------------
89
90    /**
91     * Metoda provodi proporcionalnu selekciju i to inačicu SUS. Dobrote svih jedinki
92     * moraju biti nenegativne a suma im mora biti pozitivna.
93     *
94     * @param population populacija roditelja
95     * @param rand generator slučajnih brojeva
96     * @param howMany koliko roditelja treba odabrati
97     * @return vraća polje s odabranim roditeljima
98     */
99    public static BitvectorSolution[] proportionalChoose(BitvectorSolution[] population,
100     Random rand, int howMany) {
101     BitvectorSolution[] parents = new BitvectorSolution[howMany];
102     double sum = 0;
103     for(int i = 0; i < population.length; i++) {
104       sum += population[i].fitness;
105     }
106     double r = rand.nextDouble();
107     for(int parentIndex = 0; parentIndex < howMany; parentIndex++) {
108       double limit = r + parentIndex/(double)howMany;
109       if(limit>1) limit -= 1;
110       limit *= sum;
111       int chosen = 0;
112       double upperLimit = population[chosen].fitness;
113       while(limit > upperLimit && chosen < population.length-1) {
114         chosen++;
115         upperLimit += population[chosen].fitness;
116       }
117       parents[parentIndex] = population[chosen];
118     }
119     return parents;
120   }
121
122   /**
123    * Metoda provodi proporcionalnu relativnu selekciju i to inačicu SUS (sve dobrote
124    * virtualno translatira za iznos najgore jedinke). Da bi selekcija radila korektno,
125    * ne smije se dogoditi da su dobrote svih jedinki identične jer tada suma
126    * translatiranih dobrota neće biti pozitivna.
127    *
128    * @param population populacija roditelja
129    * @param rand generator slučajnih brojeva
```

```
130     * @param howMany koliko roditelja treba odabrati
131     * @return vraća polje s odabranim roditeljima
132     */
133    public static BitvectorSolution[] proportionalRelativeChoose(
134      BitvectorSolution[] population, Random rand, int howMany) {
135      BitvectorSolution[] parents = new BitvectorSolution[howMany];
136      double sum = 0;
137      double min = population[0].fitness;
138      for(int i = 0; i < population.length; i++) {
139        sum += population[i].fitness;
140        if(min > population[i].fitness) {
141          min = population[i].fitness;
142        }
143      }
144      sum -= population.length * min;
145      double r = rand.nextDouble();
146      for(int parentIndex = 0; parentIndex < howMany; parentIndex++) {
147        double limit = r + parentIndex/(double)howMany;
148        if(limit>1) limit -= 1;
149        limit *= sum;
150        int chosen = 0;
151        double upperLimit = population[chosen].fitness - min;
152        while(limit > upperLimit && chosen < population.length-1) {
153          chosen++;
154          upperLimit += population[chosen].fitness - min;
155        }
156        parents[parentIndex] = population[chosen];
157      }
158      return parents;
159    }
160
161    // ---------------- OPERATORI KRIŽANJA ----------------
162
163    /**
164     * Metoda provodi uniformno križanje dvaju predanih roditelja i vraća novu jedinku
165     * kao rezultat.
166     *
167     * @param parent1 prvi roditelj
168     * @param parent2 drugi roditelj
169     * @param rand generator slučajnih brojeva
170     * @return nova jedinka nastala kao rezultat križanja
171     */
172    public static BitvectorSolution uniformCrossover(
173        BitvectorSolution parent1, BitvectorSolution parent2, Random rand) {
174      BitvectorSolution child = parent1.newLikeThis();
175      for(int i = 0; i < child.bits.length; i++) {
176        if(parent1.bits[i]==parent2.bits[i]) {
177          child.bits[i] = parent1.bits[i];
178        } else {
179          child.bits[i] = rand.nextFloat() < 0.5 ? parent1.bits[i] : parent2.bits[i];
180        }
181      }
182      return child;
183    }
184
185    // ---------------- OPERATORI MUTACIJE ----------------
186
187    /**
188     * Metoda provodi binarnu mutaciju nad predanom jedinkom s vjerojatnošću mutacije
189     * bita određenom parametrom <code>pm</code>.
190     *
191     * @param solution rješenje koje se mutira
192     * @param rand generator slučajnih brojeva
193     * @param pm vjerojatnost mutacije bita
194     */
195    public static void binaryMutate(BitvectorSolution solution, Random rand, double pm) {
```

```
196       float fpm = (float)pm;
197       for(int i = 0; i < solution.bits.length; i++) {
198         if(rand.nextFloat()<fpm) {
199           solution.bits[i] = (byte)(1 - solution.bits[i]);
200         }
201       }
202     }
203
204     // ------------------ OPERATORI SELEKCIJE ZA NOVU POPULACIJU ------------------
205
206     /**
207      * Metoda iz unije populacija roditelja i djece odabire najbolje jednike za novu
208      * populaciju roditelja. Odabrane jedinke odmah upisuje u predanu populaciju
209      * roditelja.
210      *
211      * @param population stara populacija roditelja
212      * @param children populacija djece
213      * @return nova populacija roditelja
214      */
215     public static BitvectorSolution[] selectBestFromUnion(
216       BitvectorSolution[] population, BitvectorSolution[] children) {
217       BitvectorSolution[] union =
218         new BitvectorSolution[population.length + children.length];
219       for(int i = 0; i < population.length; i++) {
220         union[i] = population[i];
221       }
222       for(int i = 0; i < children.length; i++) {
223         union[i+population.length] = children[i];
224       }
225       Arrays.sort(union, new Comparator<BitvectorSolution>() {
226         @Override
227         public int compare(BitvectorSolution o1, BitvectorSolution o2) {
228           double r = o1.fitness - o2.fitness;
229           return r < 0 ? 1 : (r > 0 ? -1 : 0);
230         }
231       });
232       for(int i = 0; i < population.length; i++) {
233         population[i] = union[i];
234       }
235       return population;
236     }
237 }
```

Kada govorimo o stohastičkim algoritmima (a svi optimizacijski algoritmi koje smo do sada spominjali pripadaju u tu kategoriju), važno je naglasiti da svi ti algoritmi trebaju uslugu generiranja slučajnih brojeva, odnosno pseudoslučajnih brojeva. A generatori slučajnih brojeva su, generalno govoreći, vrlo skupa usluga. Stvaranje novog generatora slučajnih brojeva može biti vremenski skupo, jer se generator u konstruktoru pokušava inicijalizirati temeljem različitih informacija koje su dostupne unutar operacijskog sustava kako bi osigurao da prilikom pozivanja daje drugačiji slijed brojeva od nekog drugog generatora koji je stvoren par milisekundi kasnije. Također, sam postupak generiranja slučajnih brojeva relativno je skupa operacija. Kod jednostavnog genetskog algoritma koji koristi binarni prikaz kromosoma lako se može dogoditi da više od 50% ukupnog vremena koje je radio optimizacijski algoritam bude provedeno u kodu koji generira slučajne brojeve.

Stoga je prilikom izrade efikasnih optimizacijskih algoritama nužno pokušati:

1. što manje puta stvarati generatore slučajnih brojeva te
2. što je moguće rjeđe pozivati generatore slučajnih brojeva.

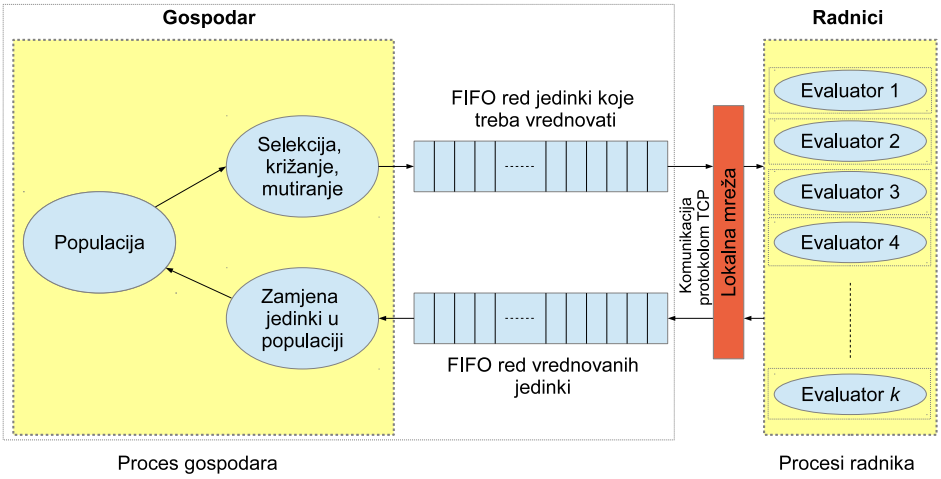
Točku (1) lagano je zadovoljiti ako se radi jednodretveni optimizacijski algoritam; stvori se jedan globalni generator slučajnih brojeva i on se koristi na svim mjestima gdje je to potrebno. U višedretvenom okruženju to međutim nije baš jednostavno ostvarivo: generatori slučajnih brojeva nisu višedretveno sigurni, i ne smije se dozvoliti da više od jedne dretve u bilo kojem trenutku poziva uslugu generiranje slučajnih brojeva. I tu postoje dva rješenja. Prvo rješenje jest napisati omotač za generator slučajnih brojeva koji je sinkroniziran i sve pozive generatora slučajnih brojeva raditi kroz taj omotač. Problem ovog pristupa jest neprihvatljiv sinkronizacijski trošak – kako se usluge generatora slučajnih brojeva često koriste, generator će postati usko grlo i vrlo će efikasno umanjiti paralelizam koji bi se inače mogao razviti. Bolje rješenje jest za svaku dretvu koristiti privatni generator slučajnih brojeva. U Javi je to moguće riješiti na nekoliko način. Ljudi se najčešće dosjete pohraniti generator u kolekciju tipa ThreadLocal pa ih od tamo vaditi. Međutim, taj pristup ima svojih problema, pa je bolje rješenje, gdje je primjenjivo, generator diretno povezati s dretvom. Na sreću, razred Executors nam ovo omogućava na trivijalan način: uz varijantu metode newFixedThreadPool koja prima samo broj dretvi koje će opsluživati red poslova postoji i varijanta koja prima dodatni argument: referencu na objekt koji će stvarati dretve koje su potrebne za opsluživanje reda poslova. U ispisu 17.2 upravo je ta inačica metode iskorištena za stvaranje objekta tipa ExecutorService (retci 48–51). Razred EvoThreadFactory prikazan je u ispisu 17.3 (retci 26–31) i svakim njegovim pozivom metode newThread vraća se novi primjerak razreda EvoThread koji je prikazan na istom ispisu u retcima 38–49. Razred EvoThread direktno nasljeđuje razred Thread te dodatno stvara privatni generator slučajnih brojeva i metodu za njegov dohvat. Ovo nam omogućava da u poslu koji se dodaje u red poslova dohvatimo trenutnu dretvu, ukalupimo je u EvoThread (jer je to sigurno moguće s obzirom da su te dretve stvorene uporabom našeg razreda EvoThreadFactory), i potom pozovemo metodu .getRand() kako bismo došli do privatnog generatora slučajnih brojeva koji opslužuje samo tu dretvu. Ovo je jasno ilustrirano u ispisu 17.2 u retku 108.

Točku (2) nije baš jednostavno zadovoljiti. Ono što se svakako može napraviti jest koristiti operatore koji minimiziraju uporabu generatora slučajnih brojeva; primjerice, koristiti operator selekcije SUS umjesto klasične proporcionalne selekcije. S druge strane, umjesto uporabe generatora prema uniformnoj distribuciji moguće je pripremiti i generatore koji koriste druge distribucije (i koji ih generiraju efikasno) te njih koristiti tamo gdje su primjereni.

17.4.1 Paralelizacija na više računala

Prehodni primjer paralelizacije koji smo detaljno razradili temeljio je na pretpostavci višedretvenosti i jednog adresnog prostora. Zahvaljujući tome, svaka je dretva imala direktan pristup do svih potrebnih podataka. Ponekad, međutim, želimo napraviti paralelizaciju koja uključuje veći broj radnika – 20, 30, 50 ili više. Stolno računalo s toliko jezgri danas još uvijek nemamo. Stoga je u tom slučaju potrebno posegnuti za distribuiranim algoritmom kakav je prikazan na slici 17.5.

U primjeru prikazanom na slici 17.5 možemo identificirati $k+1$ računalo. Proces gospodara izvodi se na jednom računalu, a na još k drugih računala izvode se procesi radnici. Ovdje su uobičajena dva scenarija.



Slika 17.5: Paralelizacija na razini populacije uporabom više računala.

1. *Gospodar se javlja svim radnicima s kojima će surađivati.* U tom slučaju on otvara TCP veze prema radnicima i ponaša se kao TCP klijent. Svaki radnik je modeliran kao TCP poslužitelj koji po pokretanju sluša na određenom pristupu i ostaje zablokiran dok gospodar s njim ne uspostavi TCP spoj. Kod ovog pristupa gospodar unaprijed mora znati na kojim se sve računalima nalaze radnici kako bi ih mogao kontaktirati (primjerice, mora odnekud imati popis njihovih IP adresa te pristupe na kojima oni slušaju).

2. Radnici se nakon pokretanja pokušavaju spojiti na gospodara. U tom slučaju gospodar je modeliran kao TCP poslužitelj koji sluša na određenom pristupu, a radnici su TCP klijenti koji pokušavaju uspostaviti TCP spoj s gospodarem. U ovom scenariju svaki od radnika mora znati koja je IP adresa gospodara i na kojem pristupu gospodar čeka na uspostavu spoja.

Osim opisanih, moguće je zamisliti i varijantu 1. scenarija u kojem gospodar na lokalnu mrežu pošalje upit (pošalje *broadcast*) s informacijom o svojoj IP adresi i pristupom na kojem očekuje da će radnici uspostaviti spoj. Svaki od radnika kad primi takav upis pokuša s navedenom IP adresom i pristupom uspostaviti spoj.

Jednom kad je veza uspostavljena, gospodar i radnik će tipično proći kroz početno dogovaranje i razmjenu informacija. Primjerice, ako zamislimo ovakav sustav koji raspodijeljeno rješava problem trgovačkog putnika, po uspostavi spoja gospodar će svakom od radnika poslati informacije o svim gradovima, kako bi prilikom vrednovanja rješenja radnici mogli računati ukupnu duljinu rute koju predstavlja konkretno rješenje.

S obzirom da radnici nisu u istom adresnom prostoru kao i gospodar, oni nemaju direktan pristup trenutnoj populaciji roditelja. Stoga će se ovakva organizacija sustava najčešće koristiti na način da gospodar odabere roditelje, križa ih i mutira te pošalje u red za vrednovanje. Iz reda za vrednovanje jedinke će u paketima preko mreže biti slane radnicima na vrednovanje koji će obaviti vrednovanje i preko mreže vratiti rezultat vrednovanja. Da bi to bilo moguće, zapis jedinke iz memorije će trebati serijalizirati u prikladni binarni ili tekstovni zapis koji se potom može prebaciti preko mreže do drugog računala gdje će se jedinka deserijalizirati i vrednovati. Rezultat vrednovanja potom će se opet serijalizirati u neki dogovoreni format, prebaciti preko mreže natrag gospodaru koji će ga deserijalizirati. Treba uočiti da uporaba mreže u sustav uvodi dodatna kašnjenja – potrebno je raditi serijalizaciju i deserijalizaciju, a i putovanje podataka preko mreže bitno je sporije no što je slučaj s prijenosom podataka preko memorijske sabirnice. Jedan od načina kako se ublažavaju ovi negativni efekti jest slanje više jedinki odjednom.

Binarnu serijalizaciju objekata u Javi je jednostavno izvesti. Nužno je samo da sve što se pokuša serijalizirati implementira sučelje `java.io.Serializable`. Ako je to ispunjeno, bilo koji objekt se u polje okteta može prebaciti kodom prikazanim u nastavku.

```
1  Serializable objekt = generirajObjektKojiZelimoSerijalizirati();
2  ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

```
3    ObjectOutputStream oos = new ObjectOutputStream(bos);
4    oos.writeObject(objekt);
5    oos.close();
6    byte[] serijaliziraniSadrzaj = bos.toByteArray();
```

Na sličan se način jednom primljeni sadržaj koji je dostupan kao polje okteta može natrag deserijalizirati u objekt.

```
1    byte[] serijaliziraniSadrzaj = procitajSMreze();
2    ByteArrayInputStream bis = new ByteArrayInputStream(serijaliziraniSadrzaj);
3    ObjectInputStream ois = new ObjectInputStream(bis);
4    Serializable objekt = (Serializable)ois.readObject();
5    ois.close();
```

U prethodnim isječcima sučelje Serializable koje se koristi u kod mogli smo zamijeniti s bilo kojim razredom ili sučeljem koje implementira/nasljeđuje ovo sučelje. Od osnovnih tipova podataka, to su svi Javini omotači oko primitivnih vrijednosti (poput razreda Integer i Double), potom razred String, polja i slično. Bilo koji korisnički razred moguće je učiniti serijalizabilnim tako da se u deklaraciji razreda doda da razred implementira sučelje Serializable. Pri tome se mora provjeriti i da svi objekti na koje taj razred ima reference također budu tipa koji je serijalizabilan.

Važno: u današnje doba sigurnosni zaštitni uređaji uobičajen su dio svih operacijskih sustava. Stoga se odluka o vrsti izvedbe mrežne aplikacije (spaja li se gospodar na radnike ili radnici na gospodara) ponekad donosi i na temelju dozvola koje korisnik ima na računalima koja koristi za ovakvu mrežnu aplikaciju. Ako su dozvole problematične, tada se najčešće koristi pristup kod kojeg se radnici spajaju na gospodara. Naime, uobičajena konfiguracija sigurnosnih zaštitnih uređaja je takva da procesima dopušta da se spajaju na druga računala, ali bez posebnih dozvola procesima ne dopušta da prihvaćaju uspostavu veza. Kako je u scenariju gdje je radnici spajaju na gospodara samo gospodar proces koji pokušava prihvaćati veze, sigurnosni zaštitni uređaj je potrebno podesiti samo na njemu. U obrnutom scenariju trebalo bi podešavati sigurnosne zaštitne uređaje na svim radnicima kako bi svaki radnik dobio dozvolu prihvatiti spoj od gospodara.

Bibliografija

Poglavlje 18

Savjeti pri implementaciji algoritama

Prilikom implementacije algoritama evolucijskog računanja potrebno je voditi računa o nizu praktičnih problema. U ovom poglavlju osvrnut ćemo se generatore slučajnih brojeva te njihovu uporabu.

18.1 Generiranje brojeva u skladu s binomnom distribucijom

Poglavlje ćemo započeti razmatranjem jednog od najtipičnijih primjera uporabe generatora slučajnih brojeva u evolucijskim programima: razmotrit ćemo implementaciju operatora mutacije koji djeluje nad binarnim kromosomom. Implementacija je prikazana u izvornom kodu 18.1. Metoda `mutiraj` kao argumente prima referencu na polje bitova koje čini kromosom, vrijerojatnost mutacije jednog bita te referencu na generator slučajnih brojeva koji treba koristiti (u ovom slučaju, primjerak razreda `java.util.Random`.

Ispis 18.1: Tipična izvedba mutacije binarnog kromosoma u programskom jeziku Java.

```
1 public void mutiraj(boolean[] kromosom, double p, Random rand) {
2     final int n = kromosom.length;
3     for(int i = 0; i < n; i++) {
4         if(rand.nextDouble() < p) {
5             kromosom[i] = !kromosom[i];
6         }
7     }
8 }
```

Pokrenete li ovaj program u analizatoru koji može pratiti na koji se dio koda troši najviše vremena, slijedi iznenađenje: dominantni potrošać vremena bit će generator slučajnih brojeva koji, kako bi pri svakom pozivu mogao vratiti jedan "slučajni" decimalni broj obavlja puno složenije izračune no što je naša mutacija bita: jedna logička operacija. Također, uočit ćemo da trajanje izvođenja ovako napisane mutacije ne ovisi bitno o vrijednosti parametra p: gotovo da je nebitno je li ta vrijednost 0.01, 0.05 ili 0.1. Kako bismo ovo ilustrirali, prethodnu metodu zatvorili smo u petlju koja 500000 puta poziva ovu metodu nad kromosomom duljine 100. Pri tome smo metodu pozivali za pet različitih vrijednosti mutacije i svaki eksperiment ponovili 30 puta. Tablica 18.1 prikazuje dobivena prosječna vremena (u milisekundama) izvođenja mutiranja 500000 kromosoma uz zadane vjerojatnosti mutacije bita.

p	**izmjereno vrijeme [ms]**
0.01	4668.233
0.05	4677.267
0.1	4714.300
0.25	4905.233
0.5	5105.833

Tablica 18.1: Ovisnost trajanja operatora mutacije o parametru p

Pretpostavimo li da je vrijeme izvođenja linearno ovisno o parametru p, provođenjem linearne regresije nad podatcima iz tablice 18.1 dolazimo do izraza:

$$T(p) = 942.456 \cdot p + 4642.646$$

iz čega se jasno vidi da pri ovakvoj izvedbi mutacije postoji izuzetno velik fiksni trošak koji ne ovisi o zadanom p (primjerice, kada je $p = 0.01$, varijabilni dio je $942.456 \cdot p = 942.456 \cdot 0.01 = 9.42456$ što čini samo 0.2% ukupno potrošenog vremena.

Dobiveni rezultat lako je objasniti: implementacija koju smo napisali neovisno o parametru p isti broj puta poziva generator slučajnih brojeva i tek povremeno (u vrlo malom broju slučajeva, s obzirom da je p kod evolucijskih algoritama tipično vrlo mali) troši vrijeme na invertiranje bita.

Stoga je razumno postaviti sljedeće pitanje: možemo li napisati djelotvorniju implementaciju operatora mutacije koja će biti "štedljivija" na pozivima generatora slučajnih brojeva? Da bismo odgovorili na to pitanje, razmotrimo najprije malo pobliže kako napisana mutacija djeluje.

Ulaz u algoritam je mjerojatnost mutacije bita p. To je broj koji nam govori kolika je vjerojatnost da operator mutacije promijeni i-ti bit. Vjerojatnost da i-ti bit ostane nepromijenjen tada je $q = 1 - p$. Implementacija koju smo napravili provodi n eksperimenata: za svaki od n bitova iz uniformne se distribucije izvlači decimalni broj iz intervala $[0, 1]$ i provjerava se je li izvučeni broj manji od zadane vjerojatnosti p: ako je, dotični se bit mutira, inače se ne dira. Ako decimalne brojeve izvlačimo iz uniformne distribucije iz intervala $[0, 1]$, izvučeni će brojevi od broja p biti manji upravo u p% slučajeva i tada radimo mutaciju bita – to je u skladu s definicijom parametra p kao vjerojatnosti mutacije bita.

Pokušajmo sada promijeniti način razmišljanja: ako imamo zadanu vjerojatnost mutacije bita p, koliko ćemo bitova u kromosomu duljine n doista i mutirati? Vratimo se opet na ispitivanje i-tog bita. Vjerojatnost da nastupi mutacija tog bita određena je parametrom p i ostvarivanje tog događaja ravna se po Bernoullijevoj distribuciji s vjerojatnošću uspjeha p. Situacija u kojoj se ne mutira niti jedan biti nastaje ako ne mutiramo prvi bit (vjerojatnost $1 - p$), ne mutiramo drugi bit (vjerojatnost $1 - p$), ... sve do zadnjeg bita. Stoga je vjerojatnost situacije u kojoj se ne mutira niti jedan bit određena s

$$p_0 = (1 - p)^n.$$

Razmotrimo sada vjerojatnost situacija u kojoj dolazi do mutacije jednog bita. Ako imamo fiksirani bit, vjerojatnost da ćemo ga mutirati je p; svih preostalih $n - 1$ bitova ne smijemo mutirati pa je vjerojatnost takvog događaja $p \cdot (1 - p)^{n-1}$. Međutim, u situacije u kojima smo mutirali jedan bit ćemo ubrojiti situaciju u kojoj smo mutirali prvi bit, situaciju u kojoj smo mutirali drugi bit, ... sve do situacije u kojoj smo mutirali n-ti bit: kako je takvih situacija upravo n, svaka vjerojatnosti $p \cdot (1 - p)^{n-1}$, ukupna vjerojatnost nasupanja situacije u kojoj je mutiran upravo jedan bit je njihova suma:

$$p_1 = n \cdot p \cdot (1 - p)^{n-1}.$$

Analizu ćemo napraviti još samo jedan korak dalje: razmotrimo sada vjerojatnost situacija u kojoj dolazi do mutacije dva bita. Imamo li fiksirani prvi bit koji želimo mutirati i drugi bit koji želimo mutirati, vjerojatnost nastupanja takve situacije $p^2 \cdot (1 - p)^{n-2}$. Prvu poziciju možemo odabrati na n načina; drugu poziciju biramo od $n - 1$ preostale, što znači da odabir možemo napraviti na $n \cdot (n - 1)$ načina. Kako su nam pri tome jednake situacija u kojoj najprije odaberemo poziciju i a potom j i situacija u kojoj najprije odaberemo poziciju j a potom i, broj različitih situacija dijelimo s 2 (što je broj načina na koje možemo rasporediti te dvije pozicije); stoga je vjerojatnost nastupanja situacije u kojoj su mutirana upravo dva bita jednaka:

$$p_2 = \frac{n \cdot (n - 1)}{2} \cdot p^2 \cdot (1 - p)^{n-2}.$$

Sada je lagano dalje poopćiti razmišljanje: vjerojatnost jedne konkretne situacije u kojoj mutiramo najprije poziciju i, potom poziciju j i potom poziciju k je $p^3 \cdot (1-p)^{n-3}$; te tri pozicije možemo odabrati na $n \cdot (n-1) \cdot (n-2)$ načina ali sve situacije koje odgovaraju bilo kojem redosljedu odabira koji bira isti skup indeksa tretiramo jednako (broj takvih situacija upravo je jednak broju permutacija tročlanog

skupa $\{i, j, k\}$ što je 3!); stoga je vjerojatnost nastupanja situacije u kojoj su mutirana upravo tri bita jednaka:

$$p_3 = \frac{n \cdot (n-1) \cdot (n-2)}{3!} \cdot p^3 \cdot (1-p)^{n-3}$$

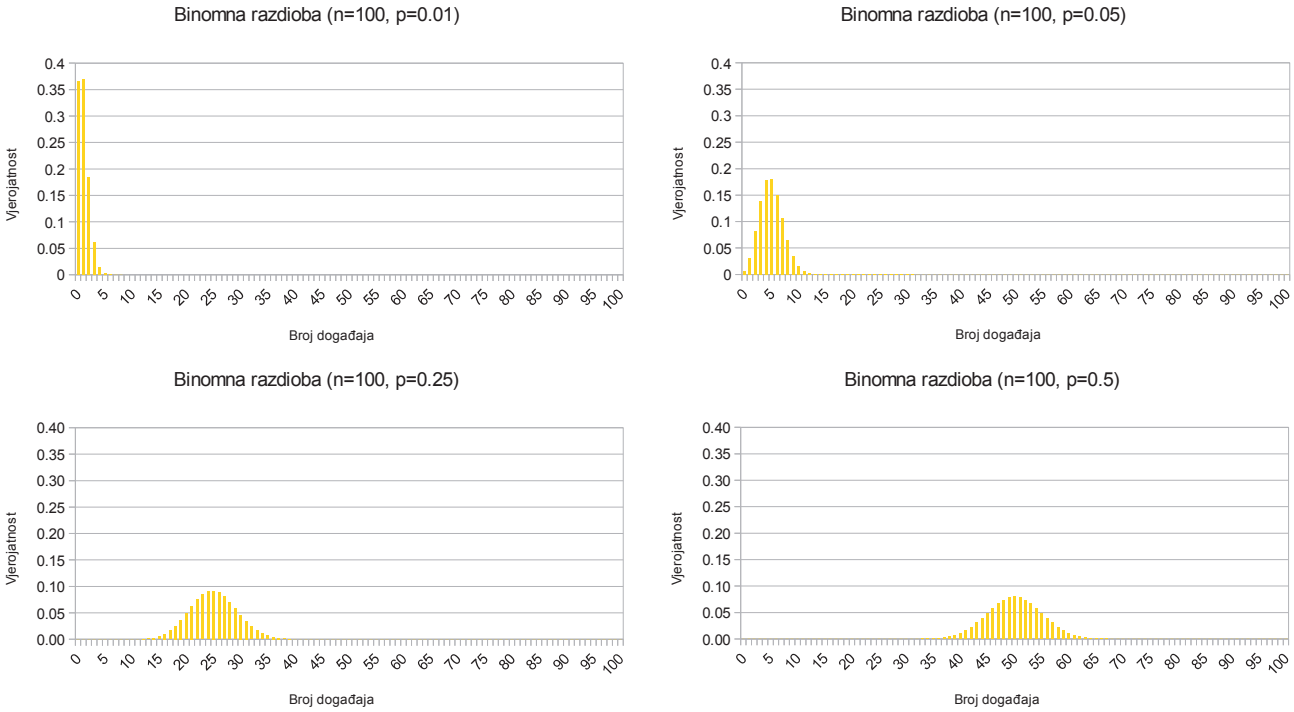
odnosno općenito za i bitova:

$$p_i = \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-i+1)}{i!} \cdot p^i \cdot (1-p)^{n-i}. \tag{18.1}$$

Lagano se možemo uvjeriti da izraz vrijedi za sve prethodno analizirane slučajeve ($i = 0$, $i = 1$, $i = 2$, $i = 3$). U brojniku ovog izraza nalazi se početak izraza koji odgovara $n!$; nedostaje dio $(n-i) \cdot (n-i-1) \cdot \cdots \cdot 1$ što je $(n-i)!$. Pomnožimo li brojnik i nazivnik tog razlomka s $(n-i)!$ dobit ćemo:

$$p_i = \frac{n!}{(n-i)! \cdot i!} \cdot p^i \cdot (1-p)^{n-i} = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}. \tag{18.2}$$

gdje je $\binom{n}{i}$ binomni koeficijent. Izraz (18.2) opisuje vjerojatnost da se u kromosomu mutira upravo i bitova (odnosno da mutacija djeluje na i različitih pozicija). Distribucija kod koje se vjerojatnosti ravnaju prema izrazu (18.2) naziva se *Binomna distribucija*. Slika 18.1 prikazuje raspodijelu vjerojatnosti nastanka mutacije na i pozicija (x-os) kod kromosoma duljine 100 bitova i različite vjerojatnosti mutacije bita p. Drugim riječima, slika prikazuje Binomne distribucije uz parametre (n, p), i to za (100,0.01), (100,0.05), (100,0.25) i (100,0.5).



Slika 18.1: Binomna razdioba.

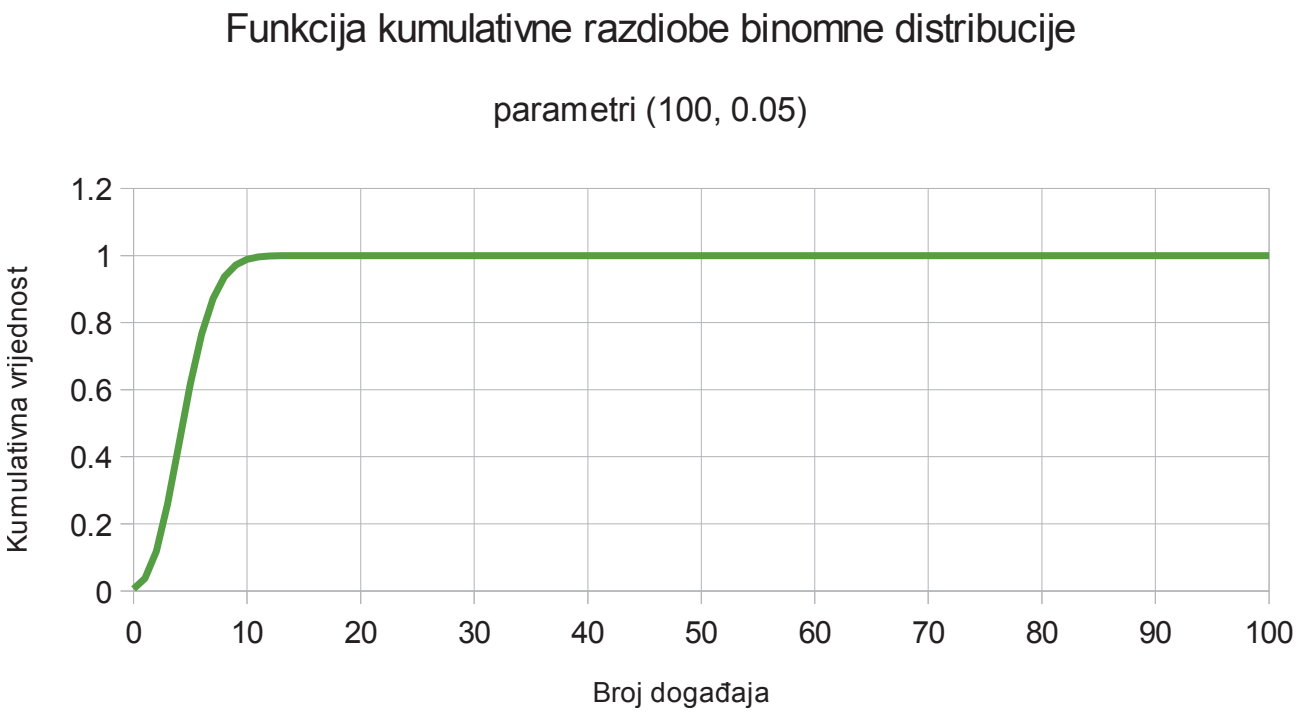
Vjerojatnosti za prvih nekoliko situacija uz binomnu distribuciju s parametrima $(100, 0.01)$ odnosno uz binomnu distribuciju s parametrima $(100, 0.05)$ dane su u tablici u nastavku.

Broj mutacija	0	1	2	3	4	5	...	100
Vjerojatnost za bin(100,0.01)	0.366	0.370	0.185	0.061	0.015	0.003	...	10^{-200}
Vjerojatnost za bin(100,0.05)	0.006	0.031	0.081	0.140	0.178	0.180	...	$7.89 \cdot 10^{-131}$

Za distrbuciju bin(100,0.01), vjerojatnost da nastupi 0 mutacija je preko 36%. Funkcija $cdf(i)$ kumulativne razdiobe definirana je kao vjerojatnost da se dogodi i mutacija ili manje:

$$cdf(i) = \sum_{j=0}^{i} p_j.$$

Grafički prikaz funkcije kumulativne razdiobe za binomnu distribuciju (100,0.05) prikazan je na slici 18.2.



Slika 18.2: Funkcija kumulativne razdiobe za binomnu razdiobu.

Za ovu distribuciju vjerojatnost da se dogodi mutacija na 11 mjesta ili manje već je preko 99% što praktički znači da najčešće u preko 90% vremena nepotrebno pozivamo funkciju generatora slučajnih brojeva. Ovo opažanje može nam poslužiti kao vodilja za djelotvorniju implementaciju. Primjetimo pri tome i da je očekivani broj mutacija u kromosomu određen kao $n \cdot p$ što je nekako intuitivno i sa slike 18.1.

Sada kada razumijemo po kojoj se razdiobi ravna broj pozicija na kojima će djelovati mutacija u kromosomu, pokušajmo napisati algoritam koji koristi upravo tu informaciju: neka funkcija mutacije bude izvedena tako da najprije iz generatora slučajnih brojeva u skladu s binomnom distribucijom izvuče broj pozicija na kojima treba djelovati mutacija i potom tražimo od generatora slučajnih brojeva još samo toliko pozicija na kojima ćemo invertirati bit. Radimo li s kromosomom duljine 100 bita i vjerojatnošću mutacije bita od 0.01, u prosjeku očekujemo da ćemo umjesto 100 poziva generatora slučajnih brojeva imati samo 2 poziva: prvi koji će najčešće reći da treba mutirati samo jednu poziciju i potom drugi kojim ćemo odrediti koja je to pozicija. Stoga bi takav algoritam trebao biti poprilici 50 puta brži od naše početne implementacije.

Opišimo generator slučajnih brojeva prikladan za djelotvorniju izvedbu mutacije sučeljem prikazanim u nastavku.

Ispis 18.2: Sučelje generatora slučajnih brojeva u programskom jeziku Java.

```
1 public interface IBinomialRNG {
2     public int nextBinomialInt();
3     public int nextPosition();
4 }
```

Ideja je da jednom pozovemo funkciju nextBinomialInt() i potom onoliko puta koliko odgovara dobivenom rezultatu pozivamo metodu nextPosition() koja svakim pozivom vraća novu različitu slučajnu poziciju. Ovakva implementacija prikazana je u izvornom kodu 18.3. Pretpostavka je da generator slučajnih brojeva koji se predaje kao posljednji argument doista pri pozivu metode nextBinomialInt() vraća broj pozicija na kojima treba napraviti mutaciju prema binomnoj distribuciji s parametrima (n, p).

Ispis 18.3: Poboljšana izvedba mutacije binarnog kromosoma u programskom jeziku Java.

```
public void mutiraj(boolean[] kromosom, double p, IBinomialRNG rng) {
    int brojOkretanja = rng.nextBinomialInt();
    for(int i = 0; i < brojOkretanja; i++) {
        int pos = rng.nextPosition();
        kromosom[pos] = !kromosom[pos];
    }
}
```

Pitanje na koje još treba odgovoriti je kako napraviti potreban generator slučajnih brojeva. Uobičajene implementacije generatora slučajnih brojeva koje su dostupne u većini modernih programskih jezika nude generiranje slučajnih brojeva u skladu s uniformnom distribucijom te eventualno prema normalnoj distribuciji uz parametre (0, 1). Stoga ćemo u nastavku razmotriti nekoliko implementacija. Najprije ćemo dati apstraktan razred koji će sadržavati implementaciju sučelja IBinomialRNG (osim metode nextPosition(). Implementacija je dana u nastavku.

Ispis 18.4: Baza za izgradnju generatora slučajnih brojeva prema binomnoj distribuciji u programskom jeziku Java.

```
package hr.fer.zemris.optjava.rng;

import java.util.Random;

public abstract class AbstractBinomialRNG implements IBinomialRNG {

    protected Random rand = new Random();

    protected int n;
    protected double p;

    protected double[] cdf;

    protected int left;
    protected int[] elems;

    public AbstractBinomialRNG(int n, double p) {
        super();
        this.n = n;
        this.p = p;

        // Priprema binomnih koeficijenata:
        double[] binCoefs = new double[n+1];
        double max = 0;
        if(n==0) {
            binCoefs[0] = 1;
        } else if(n==1) {
            binCoefs[0] = binCoefs[1] = 1;
        } else {
            binCoefs[0] = binCoefs[1] = 1;
            for(int i = 2; i <= n; i++) {
                binCoefs[i] = 1;
                for(int j = i-1; j>0; j--) {
                    binCoefs[j] = binCoefs[j]+binCoefs[j-1];
                    if(binCoefs[j]>max) {
                        max = binCoefs[j];
                        System.out.println("Novi max: "+max);
                    }
                }
            }
        }

        // Izračun funkcije kumulativne razdiobe:
        cdf = new double[n+1];
        double sum = 0;
        for(int i = 0; i <= n; i++) {
```

```
47              sum += binomProb(n, p, i, binCoefs);
48              cdf[i] = sum;
49          }
50
51          // Priprema pozicija koje ćemo vraćati:
52          elems = new int[n];
53          for(int i = 0; i < n; i++) {
54              elems[i] = i;
55          }
56
57          System.out.println("Ukupna suma: "+sum);
58      }
59
60      public abstract int nextBinomialInt();
61
62      @Override
63      public int nextPosition() {
64          int pos = rand.nextInt(left);
65          left--;
66          if(pos!=left) {
67              int tmp = elems[pos];
68              elems[pos] = elems[left];
69              elems[left] = tmp;
70          }
71          return elems[left];
72      }
73
74      private static double binomProb(int n, double p, int i, double[] binomCoef) {
75          return binomCoef[i]*Math.pow(p, i)*Math.pow(1-p, n-i);
76      }
77
78  }
```

Zadaća razreda AbstractBinomialRNG je za zadane parametre (n, p) pripremiti funkciju kumulativne razdiobe. Pojedine vjerojatnosti pri tome se računaju uporabom izraza (18.2). Zaseban dio koda u konstruktoru zadužen je za tabeliranje binomnih koeficijenata koristeći postupak izgradnje Pascalovog trokuta. Ovi koeficijenti mogu se računati na više načina, no treba obratiti pažnju da maksimalni koeficijent raste ekstremno brzo u ovisnosti o n: trend ilustrira sljedeća tablica.

n	Najveći binomni koeficijent
1	1
2	2
5	10
10	252
20	184756
50	$1.26410606437752 \cdot 10^{14}$
100	$1.0089134454556424 \cdot 10^{29}$
200	$9.054851465610329 \cdot 10^{58}$
500	$1.1674431578827774 \cdot 10^{149}$
1000	$2.702882409454366 \cdot 10^{299}$

Stoga već u primjeru s kojim radimo (kromosom od 100 bitova) koristeći **double** kao tip podataka ne možemo pamtiti točan iznos koeficijenta već samo njegovu aproksimaciju zbog čega su i sve izračunate vjerojatnosti aproksimacije. Polje elems služi za brzo izvlačenje slučajnih pozicija na kojima će se dogoditi mutacija uz garanciju da neće biti ponavljanja pozicija unutar jednog izvlačenja. Ovaj generator interno će koristiti Javin generator slučajnih brojeva koji omogućava izvlačenje brojeva prema uniformnoj distribuciji i potom će to iskoristiti za generiranje slučajnog broja prema binomnoj distribuciji. Ostaje još za odgovoriti na pitanje: kako to ostvariti. Generiranje brojeva prema binomnoj distribuciji moguće je ostvariti na više načina, a ovdje ćemo dati tri primjera koja se temelje na uporabi funkcije kumulativne razdiobe – analogija će direktna sa slučajnom proporcionalnom selekcijom (engl. *roulette-wheel selection*): generirat ćemo slučajni broj prema uniformnoj distribuciji iz intervala [0, 1]

i potom ćemo potražiti onaj broj mutacija u čije područje upada generirani slučajni broj: tu možemo direktno iskoristiti funkciju kumulativne razdiobe i vratiti najmanji k za koji je $cdf(k) > r$ gdje je r slučajni broj iz intervala $[0, 1]$ generiran prema uniformnoj distribuciji (sjetite se, funkcija kumulativne razdiobe je monotono rastuća funkcija).

Postavlja se samo pitanje: kako napraviti ovo pretraživanje. U nastavku su dane tri implementacije.

1. Razred BinomialRNGV1 pretraživanje implementira koristeći binarno raspolavljanje intervala.
2. Razred BinomialRNGV2 pretraživanje implementira slijedno od početka.
3. Razred BinomialRNGV3 pretraživanje implementira slijedno ali od pozicije koja je očekivana pa od te pozicije ili kreće prema manjim indeksima, ili prema većima.

Ispis 18.5: Tri implementacije generatora slučajnih brojeva prema binomnoj distribuciji u programskom jeziku Java.

```
1  public class BinomialRNGV1 extends AbstractBinomialRNG {
2
3      public BinomialRNGV1(int n, double p) {
4          super(n, p);
5      }
6
7      @Override
8      public int nextBinomialInt() {
9          left = n;
10         double r = rand.nextDouble();
11         if(r >= cdf[n]) return n;
12         int left = 0;
13         int right = n+1;
14         while(left < right) {
15             int current = (left+right)/2;
16             if(cdf[current] > r) {
17                 right = current-1;
18             } else {
19                 left = current+1;
20             }
21         }
22         return cdf[left]>r ? left : left+1;
23     }
24 }
25
26 public class BinomialRNGV2 extends AbstractBinomialRNG {
27
28     public BinomialRNGV2(int n, double p) {
29         super(n, p);
30     }
31
32     @Override
33     public int nextBinomialInt() {
34         left = n;
35         double r = rand.nextDouble();
36         if(r >= cdf[n]) return n;
37
38         for(int i = 0; i <= n; i++) {
39             if(cdf[i] > r) return i;
40         }
41
42         return n;
43     }
44 }
45
46 public class BinomialRNGV3 extends AbstractBinomialRNG {
47
48     private double mean;
```

```
49
50      public BinomialRNGV3(int n, double p) {
51          super(n, p);
52          mean = n*p;
53      }
54
55      @Override
56      public int nextBinomialInt() {
57          left = n;
58          double r = rand.nextDouble();
59          if(r >= cdf[n]) return n;
60
61          int startIndex = (int)(mean+0.5);
62
63          if(r < cdf[startIndex]) {
64              for(int i = startIndex-1; i >= 0; i--) {
65                  if(cdf[i] <= r) return i+1;
66              }
67              return 0;
68          } else {
69              for(int i = startIndex+1; i <= n; i++) {
70                  if(cdf[i] > r) return i;
71              }
72              return n;
73          }
74      }
75
76  }
```

S obzirom da je složenost binarnog pretraživanja $O(log(n))$ a slijednog $O(n)$, te uzevši u obzir činjenicu da treća implementacija kreće od najvjerojatnije pozicije, intuitivno bismo mogli pretpostaviti da će prva implementacija biti najbrža, treća nešto sporija a druga najsporija. Vrlo informativan eksperiment pokazat će jesmo li bili u pravu: za svaki od četiri slučaja (početna implementacija te implementacije koje koriste jednu od tri naprednije implementacije) napravljeno je 30 mjerenja pri čemu je kromosom duljine 100 bita mutiran 500000 puta uz različite vjerojatnosti mutacije bita. Rezultat je tabeliran u nastavku te grafički prikazan na slici 18.3. Vremena prikazana u tablici i na grafu su dana u milisekundama i prikazuju prosječno ukupno trajanje izvođenja 500000 mutacija kromosoma.

p	T_0 (početna)	T_1 (binarno pretraživanje)	T_2 (slijedno)	T_3 (slijedno, od očekivanja)
0.01	4668.23	134.17	100.3	101.4
0.05	4677.27	305.47	264.1	270.07
0.1	4714.3	528.47	489.77	490.57
0.25	4905.23	1091.13	1109.83	1116.1
0.5	5105.83	2092.3	2177.3	2066.57

Prvo što uočavamo: tri napisane poboljšane inačice brže su od početne implementacije. Primjerice, za $p = 0.01$, druga inačica (slijedno pretraživanje) brža je od početne implementacije za $4668.23/100.3 = 46.5$ puta.

Do daljnjih zanimljivih spoznaja možemo doći ako napravimo linearnu regresiju trajanja svake od implementiranih mutacija u ovisnosti o parametru p. Rezultati su sljedeći.

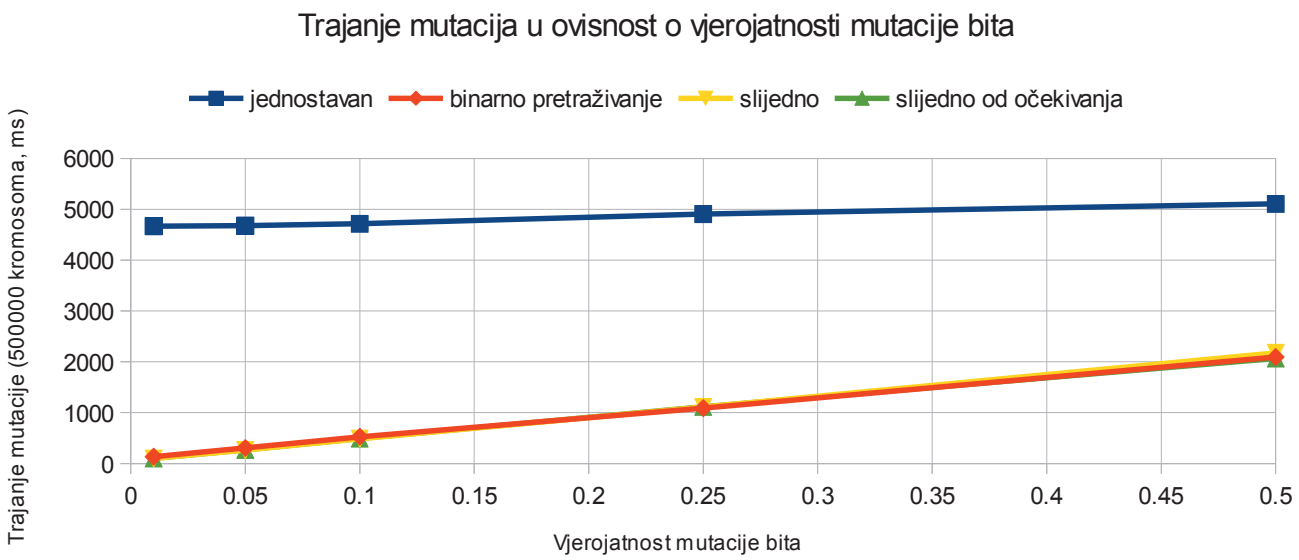
$$T_0(p) = 942.456 \cdot p + 4642.646$$

$$T_1(p) = 3967.996 \cdot p + 108.132$$

$$T_2(p) = 4236.120 \cdot p + 57.286$$

$$T_3(p) = 4011.456 \cdot p + 78.855$$

Kod početne implementacije ukupno vrijeme vrlo malo ovisi o parametru p: najviše vremena je fiksni trošak pozivanja generatora slučajnih brojeva. Kod poboljšanih implementacija situacija se stubokom mijenja: fiksni trošak je vrlo mali i vrijeme izvođenja praktički je određeno parametrom p što je i



Slika 18.3: Usporedba implementacija mutacije u ovisnosti o parametru p.

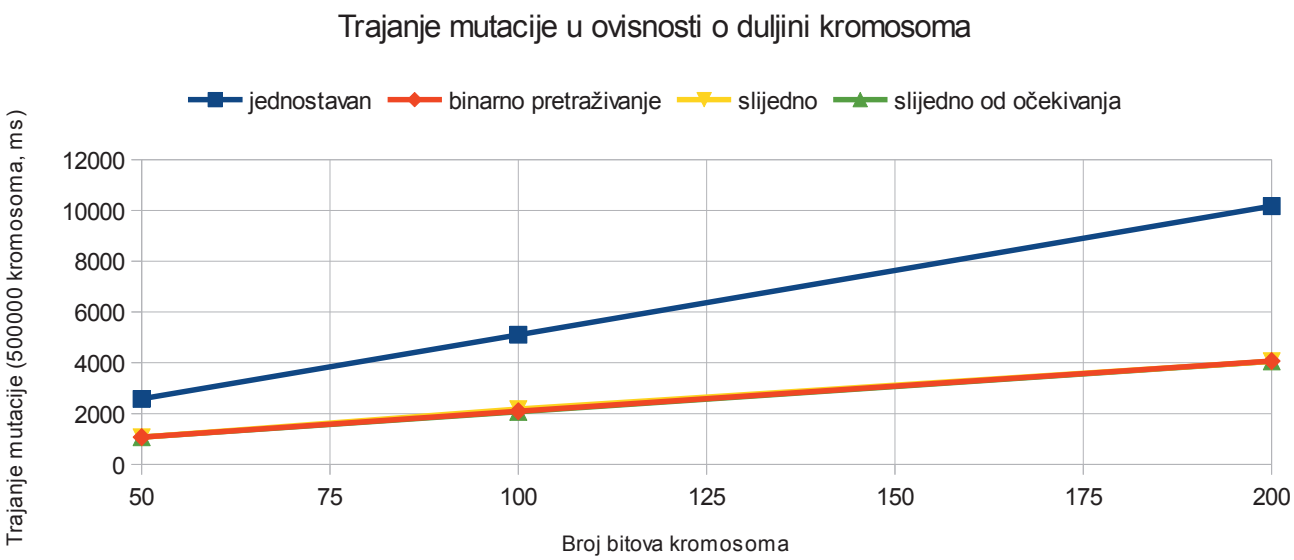
razumljivo: trošimo vrijeme da doznamo na koliko pozicija trebamo napraviti mutaciju (to ide u fiksni trošak) i potom toliko puta generiramo slučajni broj koji određuje koje će to biti pozicije; osim toga, ništa dodatno nemamo u kodu što bi trošilo vrijeme. posljedica je da smo smanjili fiksne troškove kao i ukupan broj poziva generatora slučajnih brojeva (koji je u pozadini svega i dalje Javin standardni generator).

Gledajući sliku 18.3, čini se da su sve tri poboljšane inačice jednako brze. Pažljivijom analizom podataka u tablici dolazimo do vrlo interesantnih zaključaka: za male p najbrža je implementacija koja radi slijedno pretraživanje od početka; nešto malo za njom zaostaje implementacija koja radi slijedno pretraživanje od očekivane pozicije a najgore rezultate postiže implementacija koja koristi binarno pretraživanje. Koji su razlozi za to? Implementacija koja koristi binarno pretraživanje uvijek koristi fikan broj koraka pretrage (određen s $logn$), polju koje tabelira vrijednosti *cdf* pristupa nasumično te malo izvodi kod koji povećava varijablu $left$ a malo kod koji umanjuje varijablu $right$: posljedica je da upravljačka jedinica procesora koja spekulativno pokušava pogoditi u koju će granu otići izvođenje koda često griješi (krivo pretpostavi) a zbog nasumičnog dohvaćanja vrijednosti *cdf*-a dolazi do promašaja u priručnoj memoriji: posljedica su loše performanse koje ovdje vidimo. Slijedna implementacija ima svojstvo predvidivosti: malo je grananja i lako je korektno predvidjeti granu u koju će kod otići; također, kako je p mali, *cdf* raste vrlo brzo i u prosjeku ćemo u vrlo malo koraka doći do tražene vrijednosti. Konačno, polju koje čuva vrijednosti *cdf*-a pristupa se slijedno što je puno povoljnije za priručnu memoriju procesora. Posljedica je da ovakva implementacija nadmašuje preostale dvije (i daleko nadmašuje početnu). Treća implementacija za male je vrijednosti p bliska drugoj ali ponešto lošija. Međutim, za veće vrijednosti p-a treća implementacija, činjenica da pretragu pokreće od očekivanog broja mutacija te činjenica da je binomna distribucija relativno uska rezultira najboljim performansama.

Koju od ovih implementacija koristiti u praksi? Najprije moramo uzeti u obzir vrijednosti parametra p koje se u evolucijskim algoritmima koriste: kako se radi o vjerojatnosti mutacije bita, njezin je iznos uobičajeno vrlo mali i iznosi svega nekoliko posto. Stoga, uspredimo li početnu implementaciju i preostale tri, odgovor je svakako: bilo što samo ne početna implementacija. Usporedimo li pak tri poboljšane implementacije, mala prednost svakako je na implementaciji koja radi slijedno pretraživanje.

Kako se implementacije ponašaju ako se varira n ispitano je novim nizom eksperimenata uz sve ostale parametre fiksne ($p = 0.5$, broj kromosoma nad kojima su rađene mutacije je 500000). Rezultati su dani u tablici u nastavku (grafički prikaz dan je na slici 18.4).

n	T_0 (početna)	T_1 (binarno pretraživanje)	T_2 (slijedno)	T_3 (slijedno, od očekivanja)
50	2579.03	1078.03	1069.57	1078.1
100	5105.83	2092.3	2177.3	2066.57
200	10175.03	4072.1	4058.8	4059.4



Slika 18.4: Usporedba implementacija mutacije u ovisnosti o parametru n.

Linearna regresija trajanja svake od implementiranih mutacija u ovisnosti o parametru n daje sljedeće rezultate.

$$T_0(p) = 50.648 \cdot n + 44.433$$

$$T_1(p) = 19.937 \cdot n + 88.133$$

$$T_2(p) = 19.769 \cdot n + 128.817$$

$$T_3(p) = 19.883 \cdot n + 81.683$$

Uz fiksirani p opet se vidi da tri poboljšane implementacije imaju blaži porast vremena od početne implementacije; najmanji koeficijent pri tome ima upravo implementacija koja radi slijednu pretragu.

Jednom kada se krene raditi s velikim brojem bitova, izračun točnih vjerojatnosti binomne distribucije postat će nemoguć, što zbog vrijednosti binomnih koeficijenata koje će postati enormne, što zbog izračuna potencija brojeva manjih od 1 na vrlo visoke potencije što će generirati ekstremno male vrijednosti; utjecaj numeričkih pogrešaka u tom će slučaju postati jako izražen. U tom slučaju, binomnu distribuciju potrebno je nekako aproksimirati. U slučajevima kada je p malen a n velik, dobra aproksimacija binomne distribucije je Poissonova distribucija kod koje je vjerojatnost da je $X = k$ definirana izrazom:

$$P(X = k; \lambda) = \frac{\lambda^k \cdot e^{-\lambda}}{k!}$$

pri čemu je λ parametar distribucije. Na prvi pogled čini se da se pri izračunu mogu pojaviti visoke potencije i velike faktorije (npr. 100!). Međutim, uočimo li da je:

$$P(X = k+1; \lambda) = \frac{\lambda^{k+1} \cdot e^{-\lambda}}{(k+1)!},$$

slijedi da je omjer ovih vjerojatnosti:

$$\frac{P(X = k+1; \lambda)}{P(X = k; \lambda)} = \frac{\frac{\lambda^{k+1} \cdot e^{-\lambda}}{(k+1)!}}{\frac{\lambda^k \cdot e^{-\lambda}}{k!}} = \frac{\lambda}{k+1}$$

pa je:

$$P(X = k+1; \lambda) = P(X = k; \lambda) \cdot \frac{\lambda}{k+1}.$$

Drugim riječima, vjerojatnosti možemo računati prema izrazima:

$$p_0 = e^{-\lambda}, \quad p_1 = p_0 \cdot \frac{\lambda}{1}, \quad p_2 = p_1 \cdot \frac{\lambda}{2}, \quad p_3 = p_2 \cdot \frac{\lambda}{3}, \quad p_4 = p_3 \cdot \frac{\lambda}{4}, \quad \cdots$$

Ove vrijednosti možemo generirati slijedno i temeljem njih računati i funkciju kumulativne razdiobe. Jednom kada imamo izračunatu ovu funkciju, dalje možemo koristiti već napisane implementacije koje brojeve generiraju koristeći uniformni generator slučajnih brojeva i tabeliranu funkciju kumulativne razdiobe. Iskustvena pravila kažu da će Poissonova distribucija biti dobra aproksimacija binomne distribucije s parametrima (n,p) ako se kao parametar Poissonove distribucije uzme vrijednost $\lambda = n \cdot p$ i ako je zadovoljeno da je $n \geq 20$ i $p \leq 0.05$ ili ako je $n \geq 100$ i $np \leq 10$.

Alternativno, binomnu distribuciju moguće je aproksimirati i normalnom distribucijom uz srednju vrijednost $n \cdot p$ i standardnu devijaciju $\sigma = \sqrt{n \cdot p \cdot (1-p)}$ u slučajevima kada je n velik i kada p nije blizak 0 ili 1; naime, na rubovima je binomna distribucija izrazito asimetrična dok je normalna distribucija simetrična – stoga se aproksimacija popravlja približavanjem parametra p prema vrijednosti 0.5 za koju je i binomna distribucija simetrična. U praksi, evolucijski algoritmi rade s malim vrijednostima parametra p i dodatno, funkciju kumulativne razdiobe kod normalne distribucije nije moguće analitički računati. Stoga je ovo nepovoljniji izbor za programiranje.

18.2 Generatori slučajnih brojeva i paralelizacija

Generatori slučajnih brojeva inherentno su višedretveno nesigurni jer sadrže stanje koje svakim pozivom ažuriraju. Neki programski jezici stoga nude višedretveno sigurne generatore slučajnih brojeva koji višedretvenu sigurnost postižu ili uporabom sinkronizacijskih mehanizama ili uporabom posebnih strojnih instrukcija za atomičko ažuriranje podataka (*compare-and-set*, *test-and-set* i sličnih). U takvim slučajevima uporaba jednog generatora u višedretvenom okruženju radit će korektno ali uz velik dodatni trošak. Uzevši u obzir da programi evolucijskog računanja obilno koriste generatore slučajnih brojeva, ovi troškovi nikako nisu prihvatljivi.

Loše rješenje opisanog problema bilo bi svaki puta kada metoda treba pristup generatoru slučajnih brojeva na tom mjestu stvoriti novi generator (odnosno jednom po ulasku u metodu). Nažalost, stvaranje generatora povlači njegovu inicijalizaciju što može biti vremenski vrlo skupo. Sjetite se samo inicijalizacije generatora koji smo napisali za binomnu distribuciju i koji u konstruktoru tabelira funkciju $cdf(k)$ u kvadratnoj složenosti.

Stoga se u višedretvenom okruženju preporuča koristiti jedan generator po dretvi – moderni programski jezici za ovo nude koncept *varijabli lokalnih za dretve* te različite implementacije tog koncepta. U programkom jeziku Java tako nam na raspolaganju stoji kolekcija `ThreadLocal<T>`. Alternativno, kako su u Javi dretve punokrvni razredi, možemo napisati novi razred koji predstavlja dretvu i koji kao privatnu člansku varijablu čuva svoj primjerak generatora slučajnih brojeva koji stvara u konstruktoru. Kod koji treba generator može dohvatiti informaciju o trenutnoj dretvi i iz nje dohvatiti pohranjeni generator.

Bibliografija

Dodatak A

Zadatci

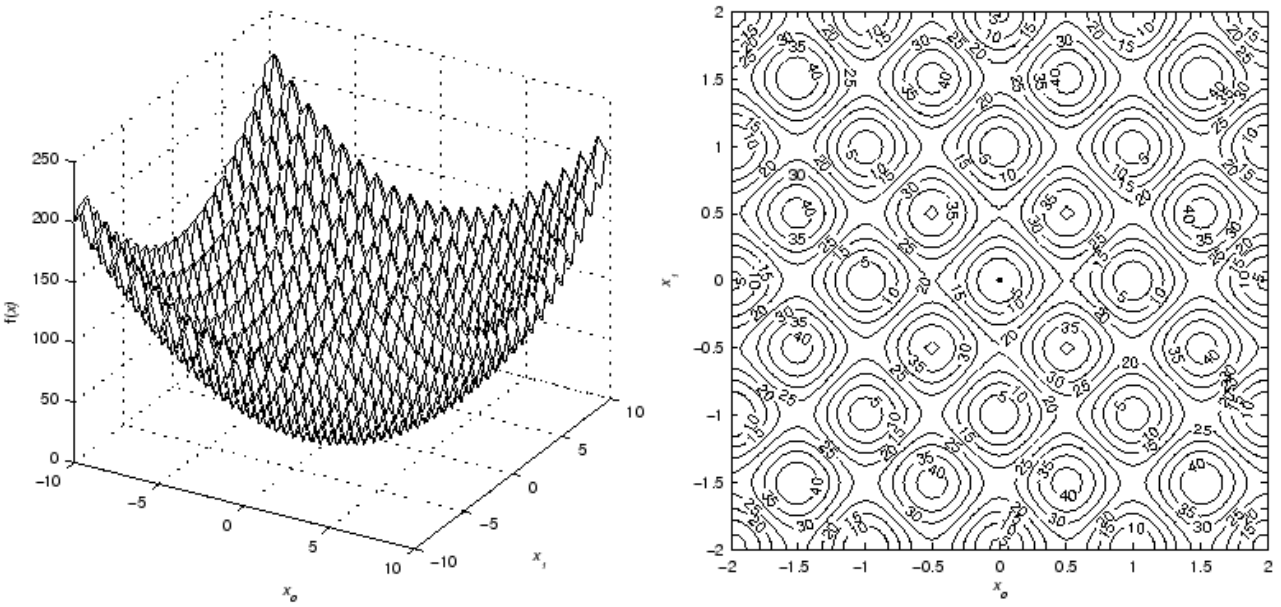
U ovom poglavlju opisani su konkretni programski zadatci čijim će se rješavanjem steći puno bolji dojam o ponašanju pojedinih evolucijskih algoritama.

A.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:

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

Funkcija je za slučaj dvije dimenzije prikazana je na slici A.1.



Slika A.1: Rastriginova funkcija.

Globalni optimum ove funkcije je poznat. To je $\vec{x}^* = (0, 0, \ldots, 0)$ 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.

A.1.1 Prikladni algoritmi za rješavanje problema

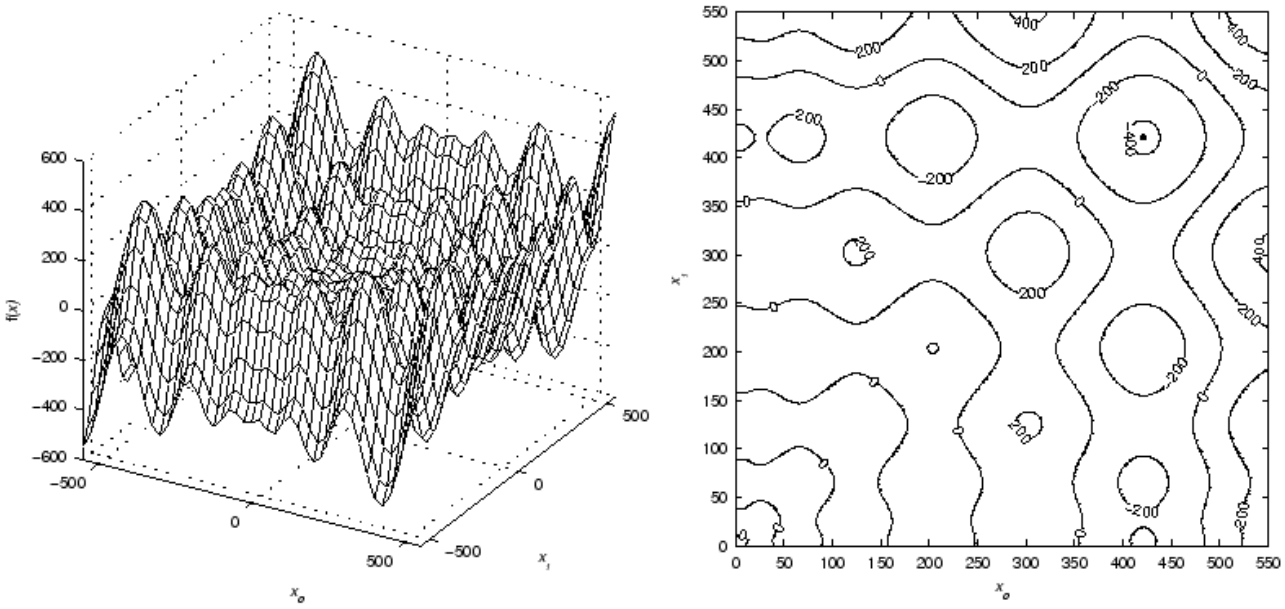
Ovaj zadatak možete rješavati algoritmom simuliranog kaljenja, genetskim algoritmom, algoritmom roja čestica, algoritmom diferencijske evolucije te umjetnim imunološkim algoritmom.

A.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:

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

Funkcija je za slučaj dvije dimenzije prikazana je na slici A.2. Optimum ove funkcije ovisi o intervalu



Slika A.2: Normalizirana Schwefelova funkcija.

vrijednosti koje varijable mogu poprimiti. Ako se ispituje interval ± 512, to je $x_i^* \approx 420.968746$, $i = 1, 2, \ldots, D$ u kojem je $f(\vec{x}^*) = -418.982887$. 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 $\vec{x}$.

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

A.2.1 Prikladni algoritmi za rješavanje problema

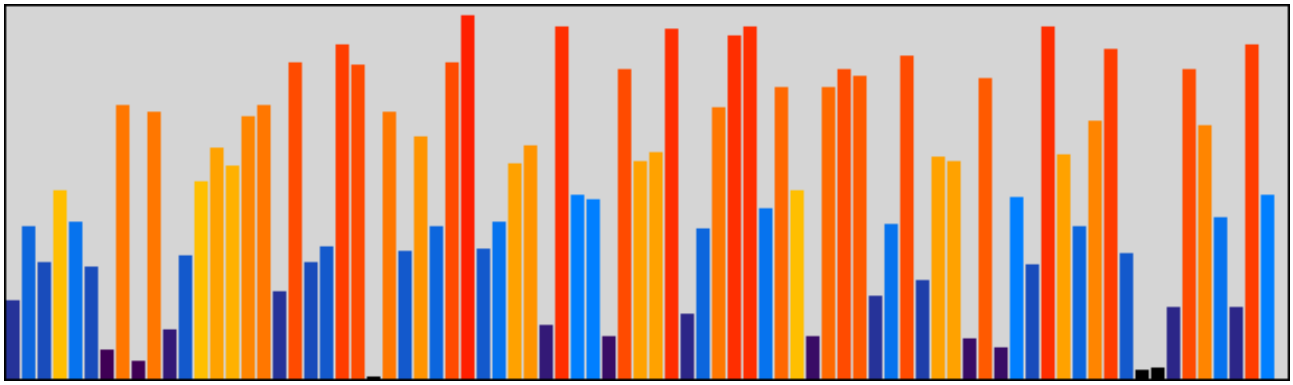
Ovaj zadatak možete rješavati algoritmom simuliranog kaljenja, genetskim algoritmom, algoritmom roja čestica, algoritmom diferencijske evolucije te umjetnim imunološkim algoritmom.

A.3 Problem popunjavanja kutija

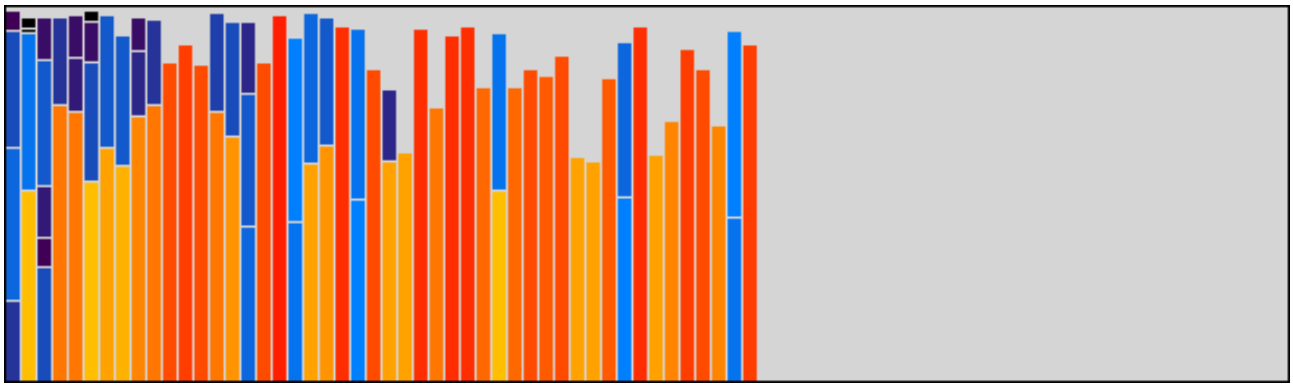
Ovo je jedan od problema koji se često javljaju u privredi. Ilustrirajmo ga na primjeru jednodimenzijskih objekata. Neki proizvođač proizvodi štapove različitih duljina. Iz tvornice, štapove je potrebno transportirati do distribucijskog centra u spremnicima fiksne visine (radi jednostavnosti pretpostavimo

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 primjeru ilustriranom slikom A.3. Slika prikazuje niz štapova koje treba pakirati u raspoloživi spremnik; pri tome su svi štapovi poslagani jedan do drugoga. Ovisno o visini, štapovi su prikazani različitim bojama.



(a) Moguće ali loše pakiranje štapova



(b) Puno bolje pakiranje štapova

Slika A.3: Problem popunjavanja kutija

Primjer jednog mogućeg pakiranja prikazan je na slici A.3a. Odmah se vidi da takvo popunjavanje nije optimalno. Na slici A.3b prikazano je drugačije popunjavanje; duljina potrebnog spremnika je manja čime je to rješenje bolje.

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

A.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 prikažete odgovarajućom strukturom podataka.

A.3.2 Prikladni algoritmi za rješavanje problema

Ovaj zadatak možete rješavati bilo kojim od obrađenih algoritma. Međutim, kako je zadatak izraženo kombinatoričke prirode, možda je dobro fokusirati se na algoritme koji su prirodni za rješavanje kombinatoričkih problema.

A.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.

A.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 s postojećim obavezama studenata. 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 tim (od svojih timova). 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`.

A.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 prikažete odgovarajućom strukturom podataka.

A.4.3 Prikladni algoritmi za rješavanje problema

Ovaj problem možete rješavati svim algoritmima opisanim u ovoj skripti.

A.5 Izrada prezentacijskih grupa za seminare (1)

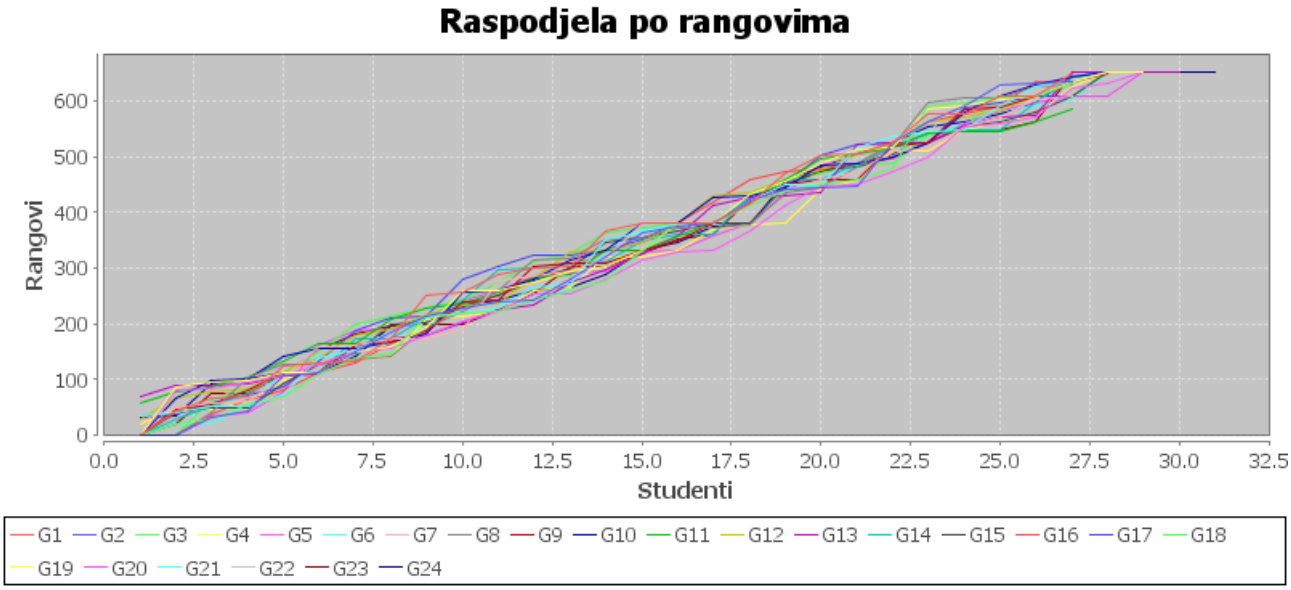
Na preddiplomskom predmetu *Seminar* studente vode voditelji koji tipično imaju dodijeljena do četiri studenta. Jednog voditelja s njegovim studentima nazvat ćemo jednom *mikro-grupom*. Prezentacije seminarskih radova odvijaju se u većim prezentacijskim grupama, koje tipično broje do 30 studenata. Kako 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.

A.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 ± 1 student), s time da je potrebno minimizirati ukupno odstupanje veličine grupa (npr. bolje je rješenje s 22 grupe od 28 studenata i 2 grupe od 29 studenata, nego rješenja koje ima 20 grupa od 28 studenata, 2 grupe od 27 studenata i 2 grupe 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 poprilici 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 studenta 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 prikazanom na slici A.4. 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).



Slika A.4: Raspodjela rangova po prezentacijskim grupama.

S obzirom da svaka prezentacijska grupa u ovom zadatku smije imati 28 ± 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 te
- osmislite kako ćete vrednovati kvalitetu razdiobe rangova unutar jednog rješenja (dakle temeljem svih 24 grupa).

A.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 ± 1,
- meko ograničenje je da želimo minimizirati razliku u veličini pojedinih grupa te
- meko ograničenje je 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`. 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.

A.5.3 Prikladni algoritmi za rješavanje problema

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

A.6 Izrada prezentacijskih grupa za seminare (2)

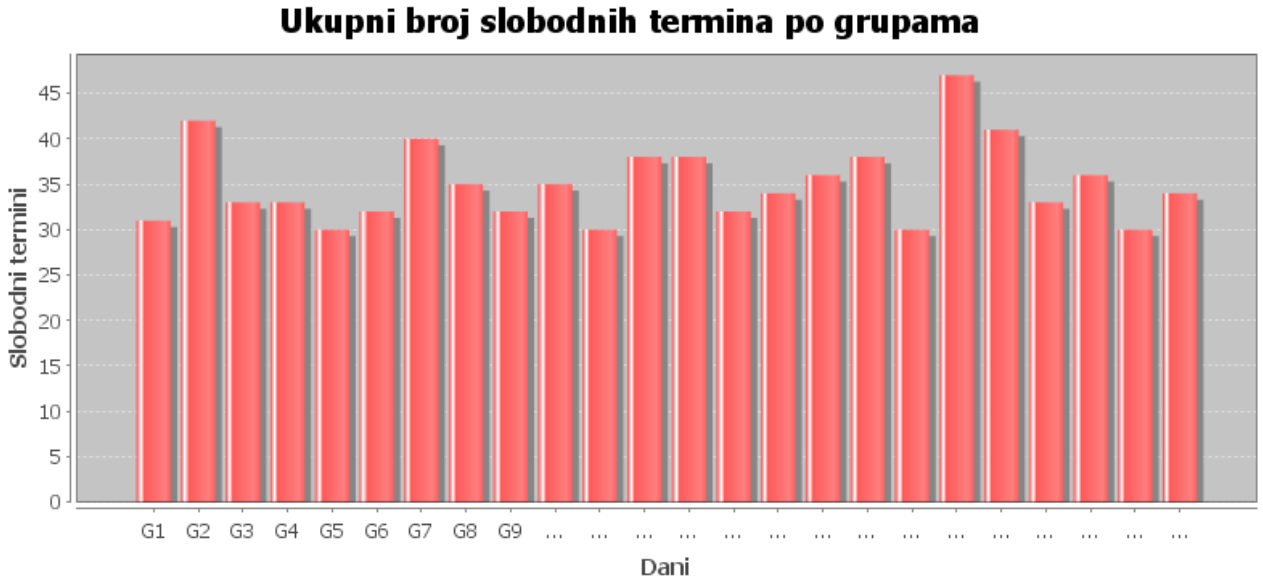
Na preddiplomskom predmetu *Seminar* studente vode voditelji koji tipično imaju dodijeljena do četiri studenta. Jednog voditelja s njegovim studentima nazvat ćemo jednom *mikro-grupom*. Prezentacije seminarskih radova odvijaju se u većim prezentacijskim grupama, koje tipično broje do 30 studenata. Kako 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.

A.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±1 student), s time da je potrebno minimizirati ukupno odstupanje veličine grupa (npr. bolje je rješenje s 22 grupe od 28 studenata i 2 grupe od 29 studenata, nego rješenje koje ima 20 grupa od 28 studenata, 2 grupe od 27 studenata te 2 grupe 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 birajući među mnoštvom slobodnih termina koje biste Vi trebali osigurati. 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 A.5, 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).



Slika A.5: Broj slobodnih termina po prezentacijskim grupama.

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.

A.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 ± 1 te
- 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`. 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.

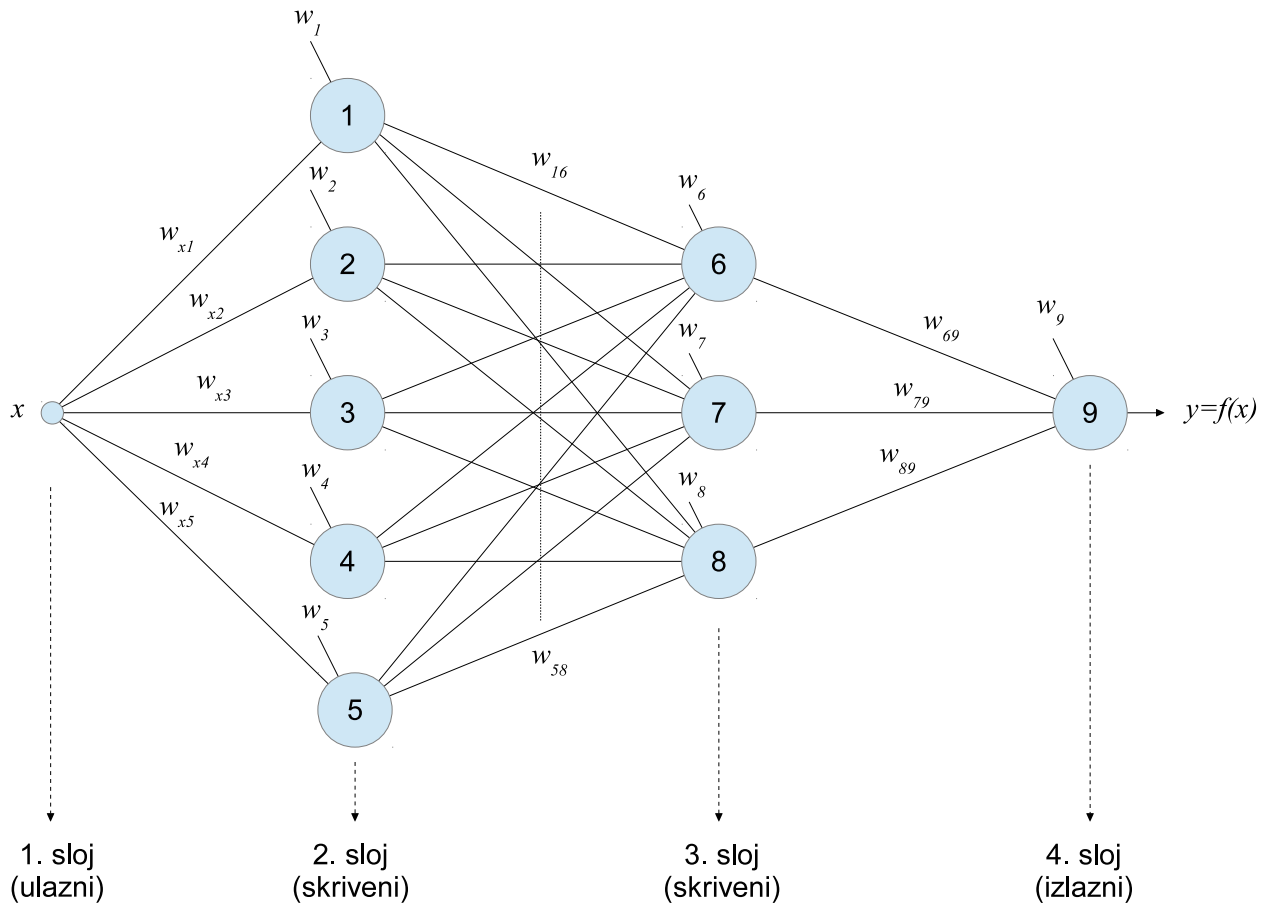
A.6.3 Prikladni algoritmi za rješavanje problema

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

A.7 Učenje umjetne neuronske mreže

Umjetne neuronske mreže prirodom su inspirirani formalizam koji nudi mogućnost izgradnje klasifikacijskih odnosno aproksimacijskih sustava koji danas imaju niz primjena u praksi. Umjetne neuronske mreže predstavnik su konektivističkog pristupa unutar područja umjetne inteligencije koji se temelji na upogonjavanju niza vrlo jednostavnih procesnih elemenata koji sami po sebi ne ispoljavaju nikakva inteligentna obilježja, ali kada ih se međusobno velik broj, dobiva se sustav koji iskazuje vrlo interesantna svojstva. Primjer jedne neuronske mreže prikazan je na slici A.6.

Mreža prikazana na toj slici ima jedan ulaz (označen s x) te jedan izlaz (označen s y). Svaki sloj sastoji se od jednog ili više neurona (odnosno procesnih elemenata). U prvom sloju nalaze se neuroni koji ne obavljaju nikakvu funkciju – oni naprosto u mrežu prosljeđuju ulaz koji dobivaju. U drugom sloju nalazi se pet neurona (označenih s plavim kružićima), u trećem sloju tri neurona te u četvrtom sloju jedan neuron. Zadaća prikazane neuronske mreže jest raditi preslikavanje $x \rightarrow f(x)$. Broj neurona



Slika A.6: Unaprijedna četiri-slojna potpuno povezana neuronska mreža.

u ulaznom sloju određen je dimenzionalnošću domene; kako je funkcija od jedne varijable, postoji samo jedan neuron. Broj neurona u izlazu određen je brojem funkcija koje mreža mora aproksimirati. Kako je naš zadatak aproksimacija funkcije f, postoji samo jedan neuron u izlaznom sloju. Broj skrivenih slojeva koje ćemo koristiti kao i broj neurona koji ćemo staviti u svaki sloj ovisi o kompleksnosti preslikavanja koje mreža mora raditi. Pri tome postoji matematički dokaz da su neuronske mreže univerzalni aproksimatori – za svako preslikavanje koje je potrebno raditi postoji neuronska mreža koja ga može raditi do na proizvoljno malu pogrešku.

Kod mreže prikazane na slici A.6 svaki neuron mreže (osim neurona ulaznog sloja) pobuđivan je od neurona prethodnog sloja (koji neuron pobuđuje kojeg prikazano je linijama); tako primjerice neuron 2 kao pobudu dobiva ulaz x, dok neuron 7 kao pobudu dobiva izlaze neurona 1, 2, 3, 4 i 5. Svaki neuron dobivenu pobudu transformira u svoj izlaz. Ta se transformacija može računati na različite načine i karakteristika je odabrane neuronske mreže. Označimo izlaz proizvoljnog neuron i oznakom y_i. U prikazanoj mreži tada ćemo imati izlaze y_1 do y_9, pri čemu se željeno preslikavanje $x \to f(x)$ dobiva upravo na izlazu y_9.

Na koji način neuron transformira dobivenu pobudu? Uobičajeno transformaciju raditi u dva koraka. Najprije se računa težinska suma pobude koja se dovodi na neuron s time da se toj težinskoj sumi nadoda još iznos slobodne težine neurona kako bi se omogućilo da za pobudu 0 neuron na izlazu može dati vrijednost koja je različita od nule. Svaki neuron pri tome ima svoju vlastitu slobodnu težinu. Na slici A.6 imamo 9 neurona i 9 slobodnih težina; slobodna težina neurona i označena je s w_i. Težine kojima se množi pobuda kojom neuron i pobuđuje neuron j na slici je označena s w_{ij}; pri tome na slici zbog preglednosti nisu označene sve težine između prvog i drugog skrivenog sloja. Označimo težinsku sumu neurona i oznakom net_i. Napišimo te sume za nekoliko neurona; primjerice, za neurone

2, 8 i 9.

$$net_2 = x \cdot w_{x2} + w_2$$
$$net_8 = y_1 \cdot w_{18} + y_2 \cdot w_{28} + y_3 \cdot w_{38} + y_4 \cdot w_{48} + y_5 \cdot w_{58} + w_8$$
$$net_9 = y_6 \cdot w_{69} + y_7 \cdot w_{79} + y_8 \cdot w_{89} + w_9$$

Drugi korak u izračunu izlaza jest vrijednost net_i propustiti kroz prijenosnu funkciju. U ovom zadatku prijenosne funkcije neurona u skrivenih slojevima bit će *sigmoidalne prijenosne funkcije* koje su definirane na sljedeći način:

$$f(net) = \frac{1}{1 + e^{-net}}.$$

To znači da će za izlaze neurona 2 i 8 za koje smo prethodno izračunali net_2 i net_8 vrijedi:

$$y_2 = \frac{1}{1 + e^{-net_2}}, y_8 \qquad\qquad = \frac{1}{1 + e^{-net_8}}.$$

Izlazni sloj neuronske mreže prikazan u ovom zadatku treba koristiti *prijenosnu funkciju identiteta*:

$$f(net) = net$$

čime za izlaz neurona 9 možemo pisati:

$$y_9 = net_9.$$

Uočimo da se izlazi moraju računati s lijeva na desno. Najprije je potrebno izračunati sve net_i pa y_i za neurone drugog sloja; tek tada računamo sve net_i pa y_i za neurone trećeg sloja jer se oni računaju temeljem prethodno izračunatih izlaza drugog sloja, i tako redom sve do zadnjeg sloja.

A.7.1 Zadatak

Zadane su funkcije određene izrazima (A.1) i (A.2).

$$f(x) = \sin(x), \quad x \in [0, 2\pi] \tag{A.1}$$

$$g(x) = \sin(x) + \sin(4x + \frac{\pi}{7}) \cdot \frac{1}{2} \quad x \in [0, 2\pi] \tag{A.2}$$

Naučite neuronsku mrežu da aproksimira funkciju f. Naučite neuronsku mrežu da aproksimira funkciju g.

Koju će funkciju neuronska mreža aproksimirati ovisi o vrijednostima težinskih faktora (naime, se ostalo smo fiksirali – struktura mreže je zadana, prijenosne funkcije su zadane). Stoga se postupak učenja neuronske mreže može svesti na postupak traženja vrijednosti težinskih faktora uz koje će neuronska mreža raditi minimalnu pogrešku. U uvodnom dijelu zadatka je rečeno da su neuronske mreže univerzalni aproksimatori – da se za zadano preslikavanje može pronaći neuronska mreža koja će to preslikavanje obavljati s proizvoljno malom pogreškom. Međutim, u ovom zadatku struktura mreže je zadana i jedino što se može mijenjati jesu vrijednosti težinskih faktora. Ova mreža neće moći naučiti zadana preslikavanja savršeno; pokušajte međutim učenjem dobiti nešto što je dovoljno dobro.

A.7.2 Priprema podataka

Za obje zadane funkcije pripremite skup za učenje: domenu svake funkcije uzorkujte uniformno i pripremite 40 parova za učenje oblika $(x, f(x))$. Te podatke pospremite u tekstualnu datoteku. Neka optimizacijski algoritam uzorke za učenje pročita iz zadane tekstualne datoteke.

A.7.3 Pretvorba u optimizacijski problem

Označimo s $\vec{w}$ vektor svih težina neuronske mreže. Za mrežu prikazanu na slici A.6 taj će vektor imati 32 komponente jer u mreži postoje 32 težine. Prepostavimo da imate N uzoraka za učenje $(x_i, f(x_i))$, gdje je $i \in \{1, 2, \ldots, N\}$. Uz zadane težine $\vec{w}$ za svaki uzorak x_i može se izračunati izlaz koji mreža daje kada joj se na ulaz narine taj uzorak; označimo taj izlaz s o_i. Neuronska mreža radi dakle preslikavanje $x_i \to o_i$ a htjeli bismo dobiti preslikavanje $x_i \to f(x_i)$. Stoga je pogreška mreže za i-ti uzorak jednaka:

$$e_i = |f(x_i) - o_i|.$$

Ukupna pogreška mreže tada je suma pogrešaka za sve uzorke:

$$E(\vec{w}) = \sum_{i=1}^{N} e_i = \sum_{i=1}^{N} |f(x_i) - o_i|.$$

Ova pogreška ovisi samo o korištenim težinama mreže $\vec{w}$ jer je sve ostalo fiksno. Iznos funkcije $E(\vec{w})$ tada se može promatrati kao funkcija kazne za rješenje $\vec{w}$, što se dalje može koristiti kako bi se pronašao optimalni vektor $\vec{w}$ koji minimizira tu funkciju kazne.

A.7.4 Naputak

U nastavku je dan *prijedlog* kako oblikovati kod neuronske mreže. Programsko ostvarenje neuronske mreže riješite tako da sve težine fizički čuvate u jednom polju **double**-ova. Ako neurone modelirate kao zasebne objekte (primjerke razreda neuron), svaki od neurona neka pamti indekse lokacija na kojima se u vektoru (tj. polju) težina nalaze njegove težine. Na taj način nikad u neurone nećete morati kopirati trenutne vrijednosti težina već je dovoljno da neuron dobije referencu u trenutni vektor težina (možda čak direktno kao parametar funkcije koja treba izračunati izlaz). Na isti način riješite i pamćenje ulaza i izlaza svih neurona: koristite jedno polje **double**-ova za ulaze i izlaze istovremeno; naime, uočite da je izlaz jednog neurona istovremeno i ulaz nekog drugog neurona (odnosno čak i za više neurona). Tada je dovoljno da imate jedno polje i da svaki neuron zapamti indeks lokacije na koju treba upisati izračunati izlaz, odnosno da zapamti indekse lokacija s kojih čita ulaze.

A.7.5 Prikladni algoritmi za rješavanje problema

Ovaj zadatak možete rješavati algoritmom simuliranog kaljenja, genetskim algoritmom, algoritmom roja čestica, algoritmom diferencijske evolucije te umjetnim imunološkim algoritmom.

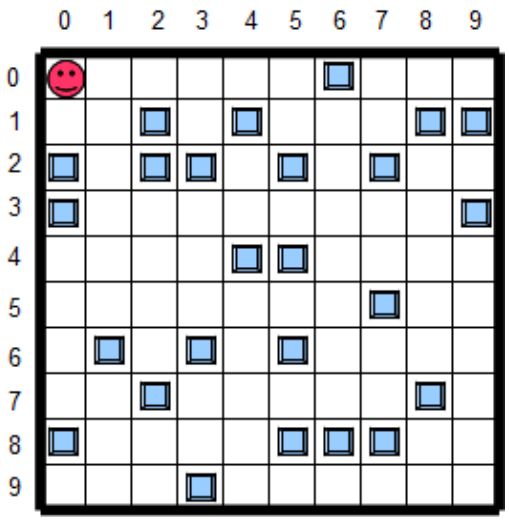
A.8 Učenje robota Robby

U svojoj vrlo interesantnoj knjizi [Mitchell, 2009], Melanie Mitchell za ilustraciju rada genetskog algoritma navodi primjer učenja robota Robby – vrlo jednostavnog mehaničkog sustava s ograničenom senzorikom i relativno kompleksnim zadatkom. Prenosimo taj zadatak u donekle izmijenjenom obliku ovdje.

Na raspolaganju Vam je robot čija je namjena prikupljati odbačene boce. Robot radi u ograđenoj prostoriji dimenzija 10×10 oko koje se nalaze zidovi, kako je prikazano na slici A.7. Početna pozicija robota je uvijek u gornjem lijevom uglu prostorije. Plavi kvadratići označavaju odbačene boce koje treba pokupiti.

A.8.1 Opis

Prostorija je podijeljena u ukupno 100 ćelija. Ćelija može biti prazna ili se na njoj može nalaziti jedna odbačena boca. Početna pozicija robota je uvijek gornji lijevi ugao, tj. pozicija $(0, 0)$. Robot ima relativno ograničeno vidno polje: vidi što se nalazi u ćeliji na kojoj se upravo nalazi te što se nalazi direktno na susjednim ćelijama (sjeverna, južna, istočna i zapadna). Robot pri tome nije svjestan apsolutnih koordinata ćelije na kojoj se nalazi – on ne zna je li trenutno na poziciji $(5, 7)$ ili $(2, 4)$.



Slika A.7: Svijet robota Robby.

Sve informacije koje su mu dostupne su relativne: što je na *trenutnoj* ćeliji te što je sjeverno, južno, istočno i zapadno od trenutne ćelije.

Temeljem tih informacija, robotu stoji na raspolaganju sedam akcija koje može poduzeti u trenutnom koraku. Akcije su sljedeće:

- ništa,
- sagni se i pokupi bocu s ćelije na kojoj stojiš,
- pomakni se na susjednu sjevernu ćeliju,
- pomakni se na susjednu južnu ćeliju,
- pomakni se na susjednu istočnu ćeliju,
- pomakni se na susjednu zapadnu ćeliju te
- pomakni se u slučajno odabranom smjeru.

Uočite da dolaskom na ćeliju koja sadrži bocu robot tu bocu ne skuplja automatski. Bocu će pokupiti samo ako na toj ćeliji pokrene odgovarajuću akciju, i to će ga koštati jednog potrošenog koraka. Za čišćenje čitave sobe robot smije napraviti najviše 200 akcija; pri tome nije važno gdje će se na kraju zateći. Također, sklopovsko ostvarenje robota je vrlo primitivno tako da robot ništa ne može zapamtiti (primjerice, da je prije dvije akcije bio na poziciji (x, y) na kojoj je bila i boca); jedina informacija koja mu stoji na raspolaganju dok stoji na ćeliji (i, j) je ono što s te ćelije vidi i temeljem te informacije treba donijeti odluku što dalje.

Vaš je zadatak genetskim algoritmom razviti "mozak" robota: temeljem onoga što robot vidi treba odlučiti koju akciju treba poduzeti a sve u cilju razvijanja uspješnog prikupljanja odbačenih boca. U tom smislu mozak robota možete modelirati kao jednu veliku preglednu tablicu koja za svaku moguću situaciju govori koju akciju treba poduzeti (primjerice, jedna moguća situacija bi bila: "na lokaciji na kojoj sam, južno od mene, istočno od mene i zapadno od mene nema boca, a sjeverno od mene nalazi se boca", a povezana akcija tada bi bila: "pomakni se na sjevernu ćeliju") S obzirom da robot vidi samo trenutnu ćeliju te susjedne ćelije na istoku, zapadu, sjeveru i jugu, broj mogućih situacije je konačan (i ne pretjerano veliki); sve situacije moguće je pobrojati (naputak: ternarni brojevni sustav, razmislite), i svaka situacija tada može jednoznačno odrediti lokaciju u preglednoj tablici s koje treba pročitati akciju koju robot treba izvesti. Zadatak optimizacijskog algoritma tada je pronaći optimalni sadržaj pregledne tablice. Evo kako treba vrednovati akcije robota.

- Ako se robot sagne i pokupi bocu, dobiva 3 boda.

- Ako se robot sagne da pokupi bocu na ćeliji koja ne sadrži bocu, kažnjava ga se oduzimanjem 5 bodova.
- Ako se robot prilikom pomicanja s pozicije (x, y) zabije u zid, vraća ga se na poziciju (x, y) i oduzima mu se 10 bodova.

Uzeviši u obzir ograničenje da robot može napraviti najviše 200 akcija, uporabom optimizacijskog algoritma razvijte strategiju uz koju će robot prikupiti maksimalnu količinu bodova.

A.8.2 Dodatni naputci

- Razmislite kako ćete uopće predstaviti "strategiju" koju robot koristi. Uz prikaz za koji se odlučite definirajte kako djeluje kombiniranja a kako modificiranja rješenja (ovisno o tome što Vaš algoritam radi).
- Da biste vrednovali kvalitetu strategije, stvorite 10 slučajnih razmještaja boca po sobi; za svaki razmještaj pustite robota da počisti sobu i pogledajte koliko je bodova prikupio u 200 dozvoljenih koraka. Potom uzmite prosječan broj bodova ostvarenih na tih 10 čišćenja.
- Ako koristite generacijske izvedbe optimizacijskih algoritama, za svaku generaciju generirajte novih 10 slučajnih razmještaja na kojima ćete vrednovati rješenja. Neovisno o vrsti optimizacijskog algoritma, pobrinite se da ne koristite vječno 10 inicijalno stvorenih razmještaja, jer ćete time učiti robota da se dobro ponaša baš na tim 10 razmještaja, što nije ideja.

Parametar problema koji se početno zadaje je postotak ćelija koje sadrže boce; primjerice, svijet prikazan na slici generiran je uz $p = 25\%$ pa sadrži točno 25 boca (jer je broj mogućih ćelija jednak 100).

Programsko rješenje treba sadržavati i grafičko korisničko sučelje (GUI) koje će omogućiti prikaz:

- kretanja dobrote najboljeg rješenja kroz vrijeme (odnosno kroz generacije),
- kretanja dobrote prosječnog rješenja kroz vrijeme (odnosno kroz generacije),
- animirani prikaz postupka čišćenja za najbolje pronađeno rješenje na kraju rada algoritma.

A.8.3 Prikladni algoritmi za rješavanje problema

Ovaj zadatak možete rješavati bilo kojim od obrađenih algoritma. Međutim, kako je zadatak izraženo kombinatoričke prirode, možda je dobro fokusirati se na algoritme koji su prirodni za rješavanje kombinatoričkih problema.

A.8.4 Problem pronalaska najveće klike

Neka je zadan neusmjereni graf $G = (V, E)$ gdje je V skup vrhova a E skup bridova. Klika Q grafa G je podskup vrhova V za koje vrijedi da su svaka dva vrha iz tog podskupa u grafu G susjedna (tj. povezana bridom):

$$\forall (i, j) \in Q \times Q, (i, j) \in E$$

Najveća klika je klika maksimalne kardinalnosti.

U praksi, problem pronalaska najveće klike se pojavljuje na raznim mjestima, a najjednostavnije ga je ilustrirati preko društvenih mreža poput Facebooka i LinkedIna; obje društvene mreže vode evidenciju o korisnicima te poznanstvima/prijateljstvima (ili kako se već u kojoj mreži zove relacija koja se prati). Ovo se može prikazati kao graf kod kojeg su korisnici vrhovi a poznanstva/prijateljstva/itd. bridovi; ako korisnik a poznaje korisnika b, postojat će brid između korisnika a i b. Da bismo dobili neusmjereni graf, pretpostavimo da prijateljstvo nije jednosmjerno (tj. a i b ili jesu prijatelji, ili nisu; nije moguće da je a prijatelj od b a b nije prijatelj od a). Poželimo li sada otkriti koji je najveći skup korisnika u kojem je svatko poznanik/prijatelj sa svim drugima u tom skupu, imamo problem pronalaska najveće klike.

Problem pronalaska maksimalne klike je $\mathcal{NP}$-potpun problem što je 1972 u radu [Karp, 1972] Richard Karp. problem je u znanstvenoj zajednici dosta interesantan te je do sada održano i nekoliko natjecanja u izradi algoritama za rješavanje ovog problema. Na stranici `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm` može se skinuti nekoliko različitih grafova za koje se znaju maksimalne klike i koji se mogu koristiti kao ulaz za rješavanje ovog problema.

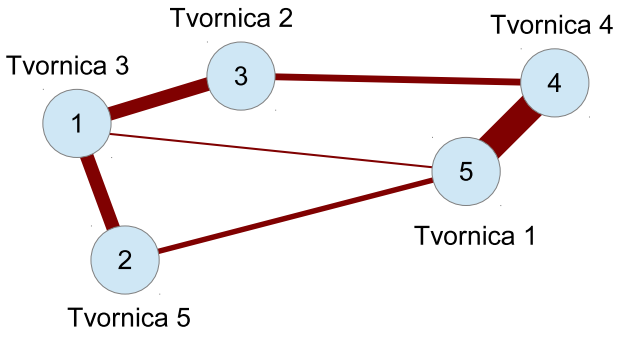
A.8.5 Zadatak

Napišite program koji iz datoteke čita definiciju grafa i pronalazi maksimalnu kliku.

A.9 Problem kvadratne dodjele

Problem kvadratne dodjele (engl. *Quadratic Assignment Problem*) još je jedan iz porodice $\mathcal{NP}$-potpunih problema. Problem možemo opisati na sljedeći način.

Na raspolaganju je n lokacija koje ćemo označiti skupom $L = \{L_1, \ldots, L_n\}$. Udaljenosti između pojedinih lokacija definirane su kvadratnom matricom reda n gdje element matrice d_{ij} predstavlja udaljenost između lokacije i i lokacije j. Na tih n lokacija potrebno je rasporediti n tvornica, koje ćemo označiti skupom $F = \{F_1, \ldots, F_n\}$, pri čemu na jednu lokaciju možemo staviti točno jednu tvornicu. Svaki mogući razmještaj tvornica po lokacijama tada je jedna permutacija niza $(1, 2, \ldots, n)$. Označimo s p jednu takvu permutaciju. Oznakom $p(i)$ označit ćemo i-ti element te permutacije, i on govori na koju je lokaciju (od njih n) smještena i-ta tvornica. Ovo je ilustrirano na slici A.8 gdje je prikazana razdioba uz $n = 5$. Prikazanoj raspodjeli tvornica po lokacijama odgovara permutacija $p = (5, 3, 1, 4, 2)$; npr. $p(5) = 2$ što nam govori da je peta tvornica smještena na drugu lokaciju.



Slika A.8: Problem kvadratne dodjele.

Tvornice proizvode niz proizvoda od čega se dio proizvoda direktno šalje u drugu tvornicu kao ulazna sirovina za daljnju proizvodnju. Koliko sirovine tvornice međusobno razmjenjuju određeno je matricom C, pri čemu element matrice c_{ij} predstavlja količinu proizvoda koju tvornica i šalje tvornici j.

Transport ovih sirovina predstavlja trošak za čitav proizvodni sustav. Stoga je u okviru problema kvadratne dodjele potrebno pronaći takvu dodjelu tvornica lokacijama (tj. permutaciju p iz skupa svih permutacije Π) uz koju je ukupni trošak transporta minimalan:

$$\min_{p \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} d_{p(i)p(j)}.$$

Naime, trošak prijevoza je to veći što se više sirovine prevozi i što su lokacije međusobno udaljenije. Gledano za transport količine proizvoda c_{ij} koji se prevozi od tvornice i do tvornice j koje su smještene na lokacijama $p(i)$ i $p(j)$ i stoga udaljene $d_{p(i)p(j)}$, trošak prijevoza iznosi $c_{ij} d_{p(i)p(j)}$. Da bi se dobio ukupni trošak, ovo se sumira po svim parovima tvornica. Uočimo da, generalno govoreći, promet ne mora biti simetričan: c_{ij} može biti različito od c_{ji}.

A.9.1 Zadatak

Na adresi `http://www.seas.upenn.edu/qaplib/` dostupna je baza konkretnih primjeraka problema kvadratne dodjele. Skinite podatke za zadatak `Els19` koji opisuje prijevoz pacijenata između bolnica te zadatke `Nug12` i `Nug25`. Napišite optimizacijski algoritam koji rješava te probleme.

Razmislite o prikazu rješenja koji ćete koristiti kao i o načinu kako ćete izvesti modificiranja i kombiniranja rješenja.

Bibliografija

R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. ISBN 0-306-30707-3.

M. Mitchell. *Complexity: A Guided Tour*. Oxford: University Press, 2009.

Dodatak B

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 adresi: `http://java.zemris.fer.hr/nastava/ui/evoAlg.zip`.

B.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.

Ispis B.1: Razred GeneticAlgorithm.

```
1  package hr.fer.zemris.ga;
2
3  import hr.fer.zemris.numeric.IFunkcija;
4
5  import java.util.Arrays;
6  import java.util.Random;
7
8  /**
9   * Primjer implementacije genetskog algoritma za optimizaciju funkcije jedne
10  * varijable.
11  *
12  * Program koristi binarni prikaz kromosoma.
13  *
14  * @author marcupic
15  */
16 public class GeneticAlgorithm {
17
18   /**
19    * Glavni program.
20    *
21    * @param args argumenti komandne linije - ne koriste se.
22    */
23   public static void main(String[] args) {
24
25     // Definiranje osnovnih parametara algoritma
26     int VEL_POP = 50;
27     double VJER_KRIZ = 0.7;
28     double VJER_MUT = 0.005;
29
30     // Generator slučajnih brojeva
31     Random rand = new Random();
32
33     // Stvaranje dekodera binarnog kromosoma:
```

```
34      // jedna varijabla, 10 bitova, raspon pretraživanja [-5, 5]
35      KromosomDekoder dekoder = new KromosomDekoder(1, 10, -5, 5);
36
37      // Stvaranje dviju populacija n-ta, i (n+1)-va
38      // Zbog optimizacije, kasnije ćemo samo mijenjati ta polja
39      Kromosom[] populacija = stvoriPopulaciju(VEL_POP, dekoder, rand);
40      Kromosom[] novaGeneracija = stvoriPopulaciju(VEL_POP, dekoder, null);
41
42      // Definiranje funkcije koju optimiramo
43      IFunkcija funkcija = new IFunkcija() {
44        @Override
45        public double izracunaj(double[] varijable) {
46          int n = varijable.length;
47          double vrijednost = 10*n;
48          for(int i = 0; i < n; i++) {
49            vrijednost += varijable[i]*varijable[i]
50              - 10*Math.cos(2*Math.PI*varijable[i]);
51          }
52          return vrijednost;
53        }
54      };
55
56      // Početna evaluacija populacije
57      evaluirajPopulaciju(populacija, funkcija);
58
59      // Ponovi kroz 1000 generacija
60      for(int generacija = 0; generacija < 1000; generacija++) {
61
62        // Sortiraj populaciju po dobroti; najbolja jedinka bit će prva
63        Arrays.sort(populacija);
64
65        // U novu populaciju prekopiraj dvije najbolje (elitizam!)
66        kopiraj(populacija[0], novaGeneracija[0]);
67        kopiraj(populacija[1], novaGeneracija[1]);
68
69        // Stvori preostale jedinke nove generacije
70        for(int i=1; i < VEL_POP/2; i++) {
71          Kromosom roditelj1 = odaberiRoditelja(populacija, rand);
72          Kromosom roditelj2 = odaberiRoditelja(populacija, rand);
73          Kromosom dijete1 = novaGeneracija[2*i];
74          Kromosom dijete2 = novaGeneracija[2*i+1];
75          krizaj1TockaPrijeloma(
76            VJER_KRIZ, roditelj1, roditelj2, dijete1, dijete2, rand);
77          mutiraj(VJER_MUT, dijete1, rand);
78          mutiraj(VJER_MUT, dijete2, rand);
79        }
80
81        // Zamijeni staru i novu populaciju
82        Kromosom[] pomocni = populacija;
83        populacija = novaGeneracija;
84        novaGeneracija = pomocni;
85
86        // Vrednuj populaciju
87        evaluirajPopulaciju(populacija, funkcija);
88
89        // pronađi najbolje rješenje
90        Kromosom najbolji = null;
91        for(int i = 0; i < populacija.length; i++) {
92          if(i==0 || najbolji.fitnes>populacija[i].fitnes) {
93            najbolji = populacija[i];
94          }
95        }
96        // I ispiši ga...
97        System.out.println("Trenutno rjesenje: f("
98            + Arrays.toString(najbolji.varijable)
99            +") = "+funkcija.izracunaj(najbolji.varijable));
```

```
100       }
101     }
102
103     /**
104      * Pomoćna funkcija koja obavlja kopiranje jednog kromosoma u drugi.
105      * @param original
106      * @param kopija
107      */
108     private static void kopiraj(Kromosom original, Kromosom kopija) {
109       for(int i = 0; i < original.bitovi.length; i++) {
110         kopija.bitovi[i] = original.bitovi[i];
111       }
112     }
113
114     /**
115      * Stvaranje nove populacije.
116      *
117      * @param brojJedinki broj jedinki koji treba stvoriti
118      * @param dekoder koji se dekoder koristi
119      * @param rand generator slučajnih brojeva
120      * @return novu populaciju
121      */
122     public static Kromosom[] stvoriPopulaciju(int brojJedinki, KromosomDekoder dekoder,
123                                               Random rand) {
124       Kromosom[] populacija = new Kromosom[brojJedinki];
125       for(int i = 0; i < populacija.length; i++) {
126         if(rand==null) {
127           populacija[i] = new Kromosom(dekoder);
128         } else {
129           populacija[i] = new Kromosom(dekoder, rand);
130         }
131       }
132       return populacija;
133     }
134
135
136     /**
137      * Metoda vrednuje predanu populaciju.
138      *
139      * @param populacija populacija
140      * @param funkcija funkcija koja se optimira
141      */
142     private static void evaluirajPopulaciju(Kromosom[] populacija, IFunkcija funkcija) {
143       for(int i = 0; i < populacija.length; i++) {
144         evaluirajJedinku(populacija[i], funkcija);
145       }
146     }
147
148     /**
149      * Metoda vrednuje predani kromosom.
150      *
151      * @param kromosom kromosom
152      * @param funkcija funkcija koja se optimira
153      */
154     private static void evaluirajJedinku(Kromosom kromosom, IFunkcija funkcija) {
155       kromosom.dekoder.dekodirajKromosom(kromosom);
156       kromosom.fitnes = funkcija.izracunaj(kromosom.varijable);
157     }
158
159     /**
160      * Metoda za odabir jednog roditelja, gdje je vjerojatnost odabira
161      * proporcionalna dobroti.
162      *
163      * @param populacija populacija iz koje se bira
164      * @param rand generator slučajnih brojeva
165      * @return odabranog roditelja
```

```
166     */
167    private static Kromosom odaberiRoditelja(Kromosom[] populacija, Random rand) {
168      double sumaDobrota = 0;
169      double najvecaVrijednost = 0;
170      for(int i = 0; i < populacija.length; i++) {
171        sumaDobrota += populacija[i].fitnes;
172        if(i==0 || najvecaVrijednost<populacija[i].fitnes) {
173          najvecaVrijednost = populacija[i].fitnes;
174        }
175      }
176      sumaDobrota = populacija.length * najvecaVrijednost - sumaDobrota;
177      double slucajniBroj = rand.nextDouble() * sumaDobrota;
178      double akumuliranaSuma = 0;
179      for(int i = 0; i < populacija.length; i++) {
180        akumuliranaSuma += najvecaVrijednost - populacija[i].fitnes;
181        if(slucajniBroj<akumuliranaSuma) return populacija[i];
182      }
183      return populacija[populacija.length-1];
184    }
185
186    /**
187     * Metoda obavlja križanje s jednom točkom prijeloma. Križanje se obavlja s
188     * zadanom vjerojatnošću. Ako se ne dogodi križanje, kao djeca se vraćaju
189     * roditelji.
190     *
191     * @param vjerKriz vjerojatnost križanja (decimalni broj između 0 i 1)
192     * @param roditelj1 prvi roditelj
193     * @param roditelj2 drugi roditelj
194     * @param dijete1 prvo dijete
195     * @param dijete2 drugo dijete
196     * @param rand generator slučajnih brojeva
197     */
198    private static void krizaj1TockaPrijeloma(double vjerKriz, Kromosom roditelj1,
199        Kromosom roditelj2, Kromosom dijete1, Kromosom dijete2, Random rand) {
200      if(rand.nextFloat() <= vjerKriz) {
201        int tockaPrijeloma = rand.nextInt(roditelj1.dekoder.ukupnoBitova-1)+1;
202        for(int i = 0; i < tockaPrijeloma; i++) {
203          dijete1.bitovi[i] = roditelj1.bitovi[i];
204          dijete2.bitovi[i] = roditelj2.bitovi[i];
205        }
206        for(int i = tockaPrijeloma; i < roditelj1.dekoder.ukupnoBitova; i++) {
207          dijete1.bitovi[i] = roditelj2.bitovi[i];
208          dijete2.bitovi[i] = roditelj1.bitovi[i];
209        }
210      } else {
211        for(int i = 0; i < roditelj1.dekoder.ukupnoBitova; i++) {
212          dijete1.bitovi[i] = roditelj1.bitovi[i];
213          dijete2.bitovi[i] = roditelj2.bitovi[i];
214        }
215      }
216    }
217
218    /**
219     * Operator mutacije. Bitovi se okreću zadanom vjerojatnošću.
220     *
221     * @param vjerMut vjerojatnost mutacije bita (broj od 0 do 1)
222     * @param dijete dijete koje se mutira
223     * @param rand generator slučajnih brojeva
224     */
225    private static void mutiraj(double vjerMut, Kromosom dijete, Random rand) {
226      for(int i = 0; i < dijete.dekoder.ukupnoBitova; i++) {
227        if(rand.nextFloat() <= vjerMut) {
228          dijete.bitovi[i] = (byte)(1-dijete.bitovi[i]);
229        }
230      }
231    }
```

```
232
233 }
```

Ispis B.2: Razred Kromosom.

```
1  package hr.fer.zemris.ga;
2
3  import java.util.Random;
4
5  /**
6   * Binarni kromosom - jedno rješenje genetskog algoritma. Pretpostavlja
7   * se da je posrijedi rješavanje problema koji se sastoji od više realnih
8   * varijabli što se reflektira u građi samog kromosoma.
9   *
10  * Važno: ugrađena funkcija za usporedbu kromosoma pretpostavlja
11  * da se radi o minimizacijskom problemu.
12  *
13  * @author marcupic
14  */
15 public class Kromosom implements Comparable<Kromosom> {
16   // Bitovi kromosoma
17   byte[] bitovi;
18   // Vrijednost funkcije dobrote kromosoma (zapravo, to je vrijednost
19   // funkcije u promatranoj točki)
20   double fitnes;
21   // Vrijednosti realnih varijabli koje kromosom predstavlja
22   double[] varijable;
23   // Dekoder koji zna konvertirati binarni prikaz u realne varijable
24   KromosomDekoder dekoder;
25
26   /**
27    * Konstruktor koji stvara novi kromosom ali ga ne inicijalizira.
28    *
29    * @param dekoder dekoder kromosoma
30    */
31   public Kromosom(KromosomDekoder dekoder) {
32     this.dekoder = dekoder;
33     this.bitovi = new byte[dekoder.ukupnoBitova];
34     this.fitnes = 0;
35     this.varijable = new double[dekoder.brojVarijabli];
36   }
37
38   /**
39    * Konstruktor koji stvara novi kromosom i inicijalizira
40    * ga na slučajni uzorak bitova.
41    *
42    * @param dekoder dekoder kromosoma
43    * @param rand generator slučajnih brojeva
44    */
45   public Kromosom(KromosomDekoder dekoder, Random rand) {
46     this.dekoder = dekoder;
47     this.bitovi = new byte[dekoder.ukupnoBitova];
48     this.fitnes = 0;
49     this.varijable = new double[dekoder.brojVarijabli];
50     for(int i = 0; i < dekoder.ukupnoBitova; i++) {
51       this.bitovi[i] = rand.nextBoolean() ? (byte)1 : (byte)0;
52     }
53   }
54
55   /**
56    * Funkcija za definiranje prirodnog poretka kromosoma. Pretpostavka
57    * je da se radi minimizacijski problem pa je manji (bolji) onaj kromosom
58    * koji ima manju vrijednost {@linkplain #fitnes} koja zapravo čuva vrijednost
59    * funkcije u promatranoj točki.
60    *
61    * @see java.lang.Comparable#compareTo(java.lang.Object)
```

```
62     */
63    @Override
64    public int compareTo(Kromosom o) {
65      if(this.fitnes < o.fitnes) {
66        return -1;
67      }
68      if(this.fitnes > o.fitnes) {
69        return 1;
70      }
71      return 0;
72    }
73  }
```

Ispis B.3: Razred KromosomDekoder.

```
1   package hr.fer.zemris.ga;
2
3   /**
4    * Razred koji predstavlja dekoder binarnog kromosoma. Temeljem informacija
5    * o minimalnim i maksimalnim vrijednostima varijabli te broju varijabli konvertira
6    * niz bitova kromosoma u vrijednosti varijabli.
7    *
8    * @author marcupic
9    */
10  public class KromosomDekoder {
11    // Donje granice varijabli
12    double[] xMin;
13    // Gornje granice varijabli
14    double[] xMax;
15    // Broj bitova koji se troši na svaku varijablu
16    int[] bitova;
17    // Koji je najveći binarni broj pridijeljen svakoj varijabli
18    int[] najveciBinarniBroj;
19    // Koliko ukupno bitova ima kromosom
20    int ukupnoBitova;
21    // Koliko varijabli predstavlja kromosom
22    int brojVarijabli;
23
24    /**
25     * Konstruktor dekodera kromosoma.
26     *
27     * @param brojVarijabli broj varijabli
28     * @param brojBitovaPoVarijabli broj bitova koji će biti korišten za svaku varijablu
29     * @param xMin donja granica (pretpostavka je da sve varijable imaju istu granicu)
30     * @param xMax gornja granica (pretpostavka je da sve varijable imaju istu granicu)
31     */
32    public KromosomDekoder(int brojVarijabli, int brojBitovaPoVarijabli,
33          double xMin, double xMax) {
34      this.brojVarijabli = brojVarijabli;
35      this.xMin = new double[brojVarijabli];
36      this.xMax = new double[brojVarijabli];
37      this.bitova = new int[brojVarijabli];
38      this.najveciBinarniBroj = new int[brojVarijabli];
39      for(int i = 0; i < brojVarijabli; i++) {
40        this.xMin[i] = xMin;
41        this.xMax[i] = xMax;
42        this.bitova[i] = brojBitovaPoVarijabli;
43        this.najveciBinarniBroj[i] = (1 << brojBitovaPoVarijabli) - 1;
44      }
45      this.ukupnoBitova = brojBitovaPoVarijabli * brojVarijabli;
46    }
47
48    /**
49     * Funkcija obavlja dekodiranje predanog kromosoma. Temeljem bitova u kromosomu
50     * obavlja izračun stvarnih vrijednosti koje ti bitovi predstavljaju, i u kromosomu
51     * popunjava polje {@linkplain Kromosom#varijable}.<br>
```

```
52      * Napomena: ova funkcija ne poziva automatski i izračun dobrote kromosoma u
53      * zadanoj točki; to treba obaviti naknadno.
54      *
55      * @param k kromosom koji treba dekodirati
56      */
57     public void dekodirajKromosom(Kromosom k) {
58       int indeksBita = 0;
59       for(int brojVarijable = 0; brojVarijable < brojVarijabli; brojVarijable++) {
60         int prviBit = indeksBita;
61         int zadnjiBit = prviBit + bitova[brojVarijable] - 1;
62         indeksBita += bitova[brojVarijable];
63         int binarniBroj = 0;
64         for(int i = prviBit; i <= zadnjiBit; i++) {
65           binarniBroj = binarniBroj * 2;
66           if(k.bitovi[i]==1) {
67             binarniBroj = binarniBroj + 1;
68           }
69         }
70         double vrijednostVarijable = (double)binarniBroj /
71                       (double)najveciBinarniBroj[brojVarijable] *
72                       (xMax[brojVarijable]-xMin[brojVarijable]) + xMin[brojVarijable];
73         k.varijable[brojVarijable] = vrijednostVarijable;
74       }
75     }
76 }
```

B.2 Mravlji algoritmi

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

Ispis B.4: Razred SimpleACO.

```
1  package hr.fer.zemris.aco;
2
3  import hr.fer.zemris.graphics.tsp.PrepareTSP;
4  import hr.fer.zemris.tsp.City;
5  import hr.fer.zemris.tsp.TSPUtil;
6  import hr.fer.zemris.tsp.TSPSolution;
7  import hr.fer.zemris.util.ArraysUtil;
8
9  import java.io.IOException;
10 import java.util.List;
11 import java.util.Random;
12
13 /**
14  * Jednostavni mravlji algoritam koji ne koristi
15  * heurističku informaciju.
16  *
17  * @author marcupic
18  */
19 public class SimpleACO {
20
21   // Polje gradova
22   private City[] cities;
23
24   // Generator slučajnih brojeva
25   private Random rand;
26
27   // Polje indeksa radova (uvijek oblika 0, 1, 2, 3, ...).
28   private int[] indexes;
29
30   // Feromonski tragovi - simetrična matrica
31   private double[][] trails;
32
33   // Udaljenosti između gradova - simetrična matrica
34   private double[][] distances;
35
36   // Populacija mrava koji rješavaju problem
37   private TSPSolution[] ants;
38
39   // Pomoćno polje indeksa mravu dostupnih gradova
40   private int[] reachable;
41
42   // Pomoćno polje vjerojatnosti odabira grada
43   private double[] probabilities;
44
45   // Konstanta isparavanja
46   private double ro;
47
48   // Pomoćno rješenje koje pamti najbolju pronađenu turu - ikada.
49   private TSPSolution best;
50   private boolean haveBest = false;
51
52   /**
53    * Konstruktor.
54    *
55    * @param cities lista gradova
56    */
57   public SimpleACO(List<City> cities) {
58     this.cities = new City[cities.size()];
59     cities.toArray(this.cities);
60     ro = 0.2;
61     rand = new Random();
```

```
62      indexes = new int[this.cities.length];
63      ArraysUtil.linearFillArray(indexes);
64      probabilities = new double[this.cities.length];
65      reachable = new int[this.cities.length];
66      distances = new double[this.cities.length][this.cities.length];
67      trails = new double[this.cities.length][this.cities.length];
68      double initTrail = 1.0/5000.0;
69      int m = 30;
70      for(int i = 0; i < this.cities.length; i++) {
71        City a = this.cities[i];
72        distances[i][i] = 0;
73        trails[i][i] = initTrail;
74        for(int j = i+1; j < this.cities.length; j++) {
75          City b = this.cities[j];
76          double dist = Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
77          distances[i][j] = dist;
78          distances[j][i] = dist;
79          trails[i][j] = initTrail;
80          trails[j][i] = initTrail;
81        }
82      }
83      ants = new TSPSolution[m];
84      for(int i = 0; i < ants.length; i++) {
85        ants[i] = new TSPSolution();
86        ants[i].cityIndexes = new int[this.cities.length];
87      }
88      best = new TSPSolution();
89      best.cityIndexes = new int[this.cities.length];
90    }
91
92    /**
93     * Glavna metoda algoritma.
94     */
95    public void go() {
96      int iter = 0;
97      int iterLimit = 500;
98
99      // ponavljaj dozvoljeni broj puta
100     while(iter < iterLimit) {
101       iter++;
102       // Za svakog mrava iz populacije
103       for(int antIndex = 0; antIndex < ants.length; antIndex++) {
104         // S kojim mravom radim?
105         TSPSolution ant = ants[antIndex];
106         doWalk(ant);
107       }
108       updateTrails();
109       evaporateTrails();
110       checkBestSolution();
111     }
112     System.out.println("Best length: "+best.tourLength);
113     System.out.println(best);
114     PrepareTSP.visualize(TSPUtil.reorderCities(cities, best.cityIndexes));
115   }
116
117   /**
118    * Metoda koja obavlja hod jednog mrava.
119    *
120    * @param ant mrav
121    */
122   private void doWalk(TSPSolution ant) {
123     // Svi su gradovi dostupni
124     System.arraycopy(indexes, 0, reachable, 0, indexes.length);
125     // Permutirajmo redosljed gradova tako da krenemo iz slučajnog
126     ArraysUtil.shuffleArray(reachable, rand);
127     // Neka je prvi grad fiksiran
```

```
128     ant.cityIndexes[0] = reachable[0];
129     // trebamo utvrditi kamo iz drugoga pa na dalje:
130     for(int step = 1; step < cities.length-1; step++) {
131       int previousCityIndex = ant.cityIndexes[step-1];
132       // Koji grad biram u koraku "step"?
133       // Mogu ici u sve gradove od step do cities.length-1
134       double probSum = 0.0;
135       for(int candidate = step; candidate < cities.length; candidate++) {
136         int cityIndex = reachable[candidate];
137         probabilities[cityIndex] = trails[previousCityIndex][cityIndex];
138         probSum += probabilities[cityIndex];
139       }
140       // Normalizacija vjerojatnosti:
141       for(int candidate = step; candidate < cities.length; candidate++) {
142         int cityIndex = reachable[candidate];
143         probabilities[cityIndex] = probabilities[cityIndex] / probSum;
144       }
145       // Odluka kuda dalje?
146       double number = rand.nextDouble();
147       probSum = 0.0;
148       int selectedCandidate = -1;
149       for(int candidate = step; candidate < cities.length; candidate++) {
150         int cityIndex = reachable[candidate];
151         probSum += probabilities[cityIndex];
152         if(number <= probSum) {
153           selectedCandidate = candidate;
154           break;
155         }
156       }
157       if(selectedCandidate==-1) {
158         selectedCandidate = cities.length-1;
159       }
160       int tmp = reachable[step];
161       reachable[step] = reachable[selectedCandidate];
162       reachable[selectedCandidate] = tmp;
163       ant.cityIndexes[step] = reachable[step];
164     }
165     ant.cityIndexes[ant.cityIndexes.length-1] = reachable[ant.cityIndexes.length-1];
166     TSPUtil.evaluate(ant, distances);
167   }
168
169   /**
170    * Metoda koja obavlja ažuriranje feromonskih tragova
171    */
172   private void updateTrails() {
173     // Koliko mravaca radi ažuriranje feromona?
174     int updates = ants.length;
175     // Ako zelim samo da najbolji rade update...
176     if(true) {
177       updates = 5;
178       //ili updates = ants.length / 10;
179       TSPUtil.partialSort(ants, updates);
180     }
181     // Azuriranje feromonskog traga:
182     for(int antIndex = 0; antIndex < updates; antIndex++) {
183       // S kojim mravom radim?
184       TSPSolution ant = ants[antIndex];
185       double delta = 1.0 / ant.tourLength;
186       for(int i = 0; i < ant.cityIndexes.length-1; i++) {
187         int a = ant.cityIndexes[i];
188         int b = ant.cityIndexes[i+1];
189         trails[a][b] += delta;
190         trails[b][a] = trails[a][b];
191       }
192     }
193   }
```

```
194
195   /**
196    * Metoda koja obavlja isparavanje feromonskih tragova.
197    */
198   private void evaporateTrails() {
199     // Isparavanje feromonskog traga
200     for(int i = 0; i < this.cities.length; i++) {
201       for(int j = i+1; j < this.cities.length; j++) {
202         trails[i][j] = trails[i][j]*(1-ro);
203         trails[j][i] = trails[i][j];
204       }
205     }
206   }
207
208   /**
209    * Metoda provjerava je li pronađeno bolje rješenje od
210    * prethodno najboljeg.
211    */
212   private void checkBestSolution() {
213     // Nadi najbolju rutu
214     if(!haveBest) {
215       haveBest = true;
216       TSPSolution ant = ants[0];
217       System.arraycopy(
218                 ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
219       best.tourLength = ant.tourLength;
220     }
221     double currentBest = best.tourLength;
222     int bestIndex = -1;
223     for(int antIndex = 0; antIndex < ants.length; antIndex++) {
224       TSPSolution ant = ants[antIndex];
225       if(ant.tourLength < currentBest) {
226         currentBest = ant.tourLength;
227         bestIndex = antIndex;
228       }
229     }
230     if(bestIndex!=-1) {
231       TSPSolution ant = ants[bestIndex];
232       System.arraycopy(
233                 ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
234       best.tourLength = ant.tourLength;
235     }
236   }
237
238   /**
239    * Ulazna točka u program.
240    *
241    * @param args argumenti komandne linije
242    */
243   public static void main(String[] args) throws IOException {
244     String fileName = args.length<1 ?
245         "data/gradovi01.txt"
246         : args[0];
247     List<City> cities = TSPUtil.loadCities(fileName);
248     if(cities==null) return;
249     new SimpleACO(cities).go();
250   }
251
252 }
```

Ispis B.5: Razred AntSystem.

```
1  package hr.fer.zemris.aco;
2
3  import hr.fer.zemris.graphics.tsp.PrepareTSP;
4  import hr.fer.zemris.tsp.City;
```

```
5  import hr.fer.zemris.tsp.TSPUtil;
6  import hr.fer.zemris.tsp.TSPSolution;
7  import hr.fer.zemris.util.ArraysUtil;
8
9  import java.io.IOException;
10 import java.util.List;
11 import java.util.Random;
12
13 /**
14  * Razred modelira rad algoritma AntSystem na problemu trgovačkog
15  * putnika.
16  *
17  * @author marcupic
18  */
19 public class AntSystem {
20
21   // Polje gradova
22   private City[] cities;
23
24   // Generator slučajnih brojeva
25   private Random rand;
26
27   // Polje indeksa radova (uvijek oblika 0, 1, 2, 3, ...).
28   private int[] indexes;
29
30   // Feromonski tragovi - simetrična matrica
31   private double[][] trails;
32
33   // Udaljenosti između gradova - simetrična matrica
34   private double[][] distances;
35
36   // Heurističke vrijednosti
37   private double[][] heuristics;
38
39   // Populacija mrava koji rješavaju problem
40   private TSPSolution[] ants;
41
42   // Pomoćno polje indeksa mravu dostupnih gradova
43   private int[] reachable;
44
45   // Pomoćno polje vjerojatnosti odabira grada
46   private double[] probabilities;
47
48   // Konstanta isparavanja
49   private double ro;
50
51   // Konstanta alfa
52   private double alpha;
53
54   // Konstanta beta
55   private double beta;
56
57   // Pomoćno rješenje koje pamti najbolju pronađenu turu - ikada.
58   private TSPSolution best;
59   private boolean haveBest = false;
60
61   /**
62    * Konstruktor.
63    *
64    * @param cities lista gradova
65    */
66   public AntSystem(List<City> cities) {
67     this.cities = new City[cities.size()];
68     cities.toArray(this.cities);
69     ro = 0.2;
70     rand = new Random();
```

```
71      indexes = new int[this.cities.length];
72      ArraysUtil.linearFillArray(indexes);
73      probabilities = new double[this.cities.length];
74      reachable = new int[this.cities.length];
75      distances = new double[this.cities.length][this.cities.length];
76      heuristics = new double[this.cities.length][this.cities.length];
77      trails = new double[this.cities.length][this.cities.length];
78      double initTrail = 1.0/5000.0;
79      int m = 30;
80      alpha = 3;
81      beta = 2;
82      for(int i = 0; i < this.cities.length; i++) {
83        City a = this.cities[i];
84        distances[i][i] = 0;
85        trails[i][i] = initTrail;
86        for(int j = i+1; j < this.cities.length; j++) {
87          City b = this.cities[j];
88          double dist = Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
89          distances[i][j] = dist;
90          distances[j][i] = dist;
91          trails[i][j] = initTrail;
92          trails[j][i] = initTrail;
93          heuristics[i][j] = Math.pow(1.0 / dist, beta);
94          heuristics[j][i] = heuristics[i][j];
95        }
96      }
97      ants = new TSPSolution[m];
98      for(int i = 0; i < ants.length; i++) {
99        ants[i] = new TSPSolution();
100       ants[i].cityIndexes = new int[this.cities.length];
101     }
102     best = new TSPSolution();
103     best.cityIndexes = new int[this.cities.length];
104   }
105
106   /**
107    * Glavna metoda algoritma.
108    */
109   public void go() {
110     System.out.println("Zapocinjem s populacijom:");
111     System.out.println("========================");
112     int iter = 0;
113     int iterLimit = 500;
114
115     // ponavljaj dozvoljeni broj puta
116     while(iter < iterLimit) {
117       iter++;
118       for(int antIndex = 0; antIndex < ants.length; antIndex++) {
119         // S kojim mravom radim?
120         TSPSolution ant = ants[antIndex];
121         doWalk(ant);
122       }
123       updateTrails();
124       evaporateTrails();
125       checkBestSolution();
126     }
127     System.out.println("Best length: "+best.tourLength);
128     System.out.println(best);
129     PrepareTSP.visualize(TSPUtil.reorderCities(cities, best.cityIndexes));
130   }
131
132   /**
133    * Metoda koja obavlja hod jednog mrava.
134    *
135    * @param ant mrav
136    */
```

```
137   private void doWalk(TSPSolution ant) {
138     // Svi su gradovi dostupni
139     System.arraycopy(indexes, 0, reachable, 0, indexes.length);
140     // Permutirajmo redosljed gradova tako da krenemo iz slučajnog
141     ArraysUtil.shuffleArray(reachable, rand);
142     // Neka je prvi grad fiksiran
143     ant.cityIndexes[0] = reachable[0];
144     // trebamo utvrditi kamo iz drugoga pa na dalje:
145     for(int step = 1; step < cities.length-1; step++) {
146       int previousCityIndex = ant.cityIndexes[step-1];
147       // Koji grad biram u koraku "step"?
148       // Mogu ici u sve gradove od step do cities.length-1
149       double probSum = 0.0;
150       for(int candidate = step; candidate < cities.length; candidate++) {
151         int cityIndex = reachable[candidate];
152         probabilities[cityIndex] =
153           Math.pow(trails[previousCityIndex][cityIndex],alpha) *
154           heuristics[previousCityIndex][cityIndex];
155         probSum += probabilities[cityIndex];
156       }
157       // Normalizacija vjerojatnosti:
158       for(int candidate = step; candidate < cities.length; candidate++) {
159         int cityIndex = reachable[candidate];
160         probabilities[cityIndex] = probabilities[cityIndex] / probSum;
161       }
162       // Odluka kuda dalje?
163       double number = rand.nextDouble();
164       probSum = 0.0;
165       int selectedCandidate = -1;
166       for(int candidate = step; candidate < cities.length; candidate++) {
167         int cityIndex = reachable[candidate];
168         probSum += probabilities[cityIndex];
169         if(number <= probSum) {
170           selectedCandidate = candidate;
171           break;
172         }
173       }
174       if(selectedCandidate==-1) {
175         selectedCandidate = cities.length-1;
176       }
177       int tmp = reachable[step];
178       reachable[step] = reachable[selectedCandidate];
179       reachable[selectedCandidate] = tmp;
180       ant.cityIndexes[step] = reachable[step];
181     }
182     ant.cityIndexes[ant.cityIndexes.length-1] = reachable[ant.cityIndexes.length-1];
183     TSPUtil.evaluate(ant, distances);
184   }
185
186   /**
187    * Metoda koja obavlja ažuriranje feromonskih tragova
188    */
189   private void updateTrails() {
190     // Koliko mrava radi ažuriranje?
191     int updates = ants.length;
192     // Ako zelim samo da najbolji rade ažuriranje...
193     if(false) {
194       updates = 5;
195       //ili updates = ants.length / 10;
196       TSPUtil.partialSort(ants, updates);
197     }
198     // Azuriranje feromonskog traga:
199     for(int antIndex = 0; antIndex < updates; antIndex++) {
200       // S kojim mravom radim?
201       TSPSolution ant = ants[antIndex];
202       double delta = 1.0 / ant.tourLength;
```

```
203         for(int i = 0; i < ant.cityIndexes.length-1; i++) {
204           int a = ant.cityIndexes[i];
205           int b = ant.cityIndexes[i+1];
206           trails[a][b] += delta;
207           trails[b][a] = trails[a][b];
208         }
209       }
210     }
211
212     /**
213      * Metoda koja obavlja isparavanje feromonskih tragova.
214      */
215     private void evaporateTrails() {
216       // Isparavanje feromonskog traga
217       for(int i = 0; i < this.cities.length; i++) {
218         for(int j = i+1; j < this.cities.length; j++) {
219           trails[i][j] = trails[i][j]*(1-ro);
220           trails[j][i] = trails[i][j];
221         }
222       }
223     }
224
225     /**
226      * Metoda provjerava je li pronađeno bolje rješenje od
227      * prethodno najboljeg.
228      */
229     private void checkBestSolution() {
230       // Nadi najbolju rutu
231       if(!haveBest) {
232         haveBest = true;
233         TSPSolution ant = ants[0];
234         System.arraycopy(
235                   ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
236         best.tourLength = ant.tourLength;
237       }
238       double currentBest = best.tourLength;
239       int bestIndex = -1;
240       for(int antIndex = 0; antIndex < ants.length; antIndex++) {
241         TSPSolution ant = ants[antIndex];
242         if(ant.tourLength < currentBest) {
243           currentBest = ant.tourLength;
244           bestIndex = antIndex;
245         }
246       }
247       if(bestIndex!=-1) {
248         TSPSolution ant = ants[bestIndex];
249         System.arraycopy(
250                   ant.cityIndexes, 0, best.cityIndexes, 0, ant.cityIndexes.length);
251         best.tourLength = ant.tourLength;
252       }
253     }
254
255
256     /**
257      * Ulazna točka u program.
258      *
259      * @param args argumenti komandne linije
260      */
261     public static void main(String[] args) throws IOException {
262       String fileName = args.length<1 ?
263           "data/gradovi01.txt"
264           : args[0];
265       List<City> cities = TSPUtil.loadCities(fileName);
266       if(cities==null) return;
267       new AntSystem(cities).go();
268     }
```

```
269
270 }
```

B.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`.

Ispis B.6: Razred ParticleSwarmOptimization.

```
1  package hr.fer.zemris.pso;
2
3  import hr.fer.zemris.numeric.IFunkcija;
4
5  import java.text.DecimalFormat;
6  import java.util.Random;
7
8  /**
9   * Primjer uporabe algoritma roja čestica za optimizaciju.
10  *
11  * @author marcupic
12  */
13 public class ParticleSwarmOptimization {
14
15   // Funkcija koju optimiramo
16   private IFunkcija funkcija;
17
18   // Mimimumi varijabli u prostoru pretraživanja
19   private double[] varMin;
20
21   // Maksimumi varijabli u prostoru pretraživanja
22   private double[] varMax;
23
24   // Maksimalna promjene varijable odjednom, za svaku dimenziju
25   private double[] velBounds;
26
27   // Postotak raspona prostora pretraživanja koji se koristi za
28   // izračun ograničenja pomaka u jednom koraku
29   double velBoundsPercentage;
30
31   // Globalni brojač iteracija
32   private int iteracija;
33
34   // Težina koju korisimo na početku
35   private double linWeightStart;
36
37   // Težina koju korisimo na kraju smanjivanja
38   private double linWeightEnd;
39
40   // Iteracija u kojoj težina pada na krajnju i dalje se ne mijenja
41   private int linWeightTreshold;
42
43   // Broj čestica s kojima radimo
44   int VEL_POP;
45
46   // Generator slučajnih brojeva
47   Random rand;
48
49   // Broj dimenzija funkcije
50   int dims;
51
52   // Konstanta c1
53   double c1;
54
55   // Konstanta c2
56   double c2;
57
```

```
58    // Čestice
59    Particle[] particles;
60
61    // Susjedstvo
62    Neighborhood neighborhood;
63
64    // Pomoć u formatiranu brojeva
65    DecimalFormat df = new DecimalFormat("0.00000");
66
67    /**
68     * Glavni program.
69     *
70     * @param args argumenti komandne linije - ne koriste se.
71     */
72    public static void main(String[] args) throws Exception {
73      final ParticleSwarmOptimization t = new ParticleSwarmOptimization();
74      t.go();
75    }
76
77    /**
78     * Konstruktor.
79     */
80    public ParticleSwarmOptimization() {
81
82      // Definiranje parametara algoritma
83      VEL_POP = 20;
84      dims = 2;
85      c1 = 2;
86      c2 = 2.5;
87
88      iteracija = 0;
89      linWeightStart = 0.9;
90      linWeightEnd = 0.4;
91      linWeightTreshold = 50;
92
93      // Definiranje minimalnih i maksimalnih vrijednosti po dimenzijama,
94      // te definiranje maksimalne promjene varijable u jednom koraku.
95      varMin = new double[dims];
96      varMax = new double[dims];
97      velBounds = new double[dims];
98      velBoundsPercentage = 0.05;  // dozvoli pomak od 5% raspona
99      for(int d = 0; d < dims; d++) {
100       varMin[d] = -5;
101       varMax[d] =  5;
102       velBounds[d] = (varMax[d]-varMin[d])*velBoundsPercentage;
103     }
104
105     // Definiranje funkcije koju optimiramo.
106     funkcija = new IFunkcija() {
107       @Override
108       public double izracunaj(double[] varijable) {
109         int n = varijable.length;
110         double vrijednost = 10*n;
111         for(int i = 0; i < n; i++) {
112           vrijednost += varijable[i]*varijable[i]
113                         - 10*Math.cos(2*Math.PI*varijable[i]);
114         }
115         return vrijednost;
116       }
117     };
118
119     // Generator slučajnih brojeva.
120     rand = new Random();
121
122     // Inicijalizacija
123     particles = new Particle[VEL_POP];
```

```
124       for(int i = 0; i < VEL_POP; i++) {
125         particles[i] = new Particle(dims);
126         for(int d = 0; d < dims; d++) {
127           particles[i].vars[d] = rand.nextDouble()*(varMax[d]-varMin[d])+varMin[d];
128           particles[i].oldVars[d] = particles[i].vars[d];
129           particles[i].bestVars[d] = particles[i].vars[d];
130           particles[i].velocity[d] = rand.nextDouble() *
131                                         (2*velBounds[d])-velBounds[d];
132         }
133         particles[i].bestValue = funkcija.izracunaj(particles[i].vars);
134       }
135
136       // Definiranje susjedstva koje koristimo - lokalno veličine 5
137       neighborhood = new LocalNeighborhood(VEL_POP, dims, 5, true);
138
139       // Za odabir globalnog susjedstva može se iskoristiti sljedeće:
140       //neighborhood = new GlobalNeighborhood(VEL_POP, dims, true);
141
142       // Ispiši inicijalnu statistiku.
143       updateStatistics();
144     }
145
146     /**
147      * Glavna metoda optimizacijskog algoritma.
148      */
149     public void go() {
150       // Ponavljaj zadani broj puta
151       for(int iter = 0; iter < 100; iter++) {
152         nextIteration();
153       }
154     }
155
156     /**
157      * Jedna iteracija algoritma PSO.
158      */
159     protected void nextIteration() {
160       iteracija++;
161
162       // Koju težinu koristimo? Težina linearno pada s iteracijama do
163       // neke male zadane, i dalje ostaje konstantna.
164       double w;
165       if(iteracija>linWeightTreshold) {
166         w = linWeightEnd;
167       } else {
168         w = linWeightStart + (linWeightEnd-linWeightStart) *
169                                 (iteracija-1.0)/linWeightTreshold;
170       }
171
172       // Ažurirajmo "znanje" susjedstva
173       neighborhood.scan(particles);
174
175       // Ažuriraj pozicije i brzine svake čestice
176       for(int i = 0; i < particles.length; i++) {
177         double[] socialBest = neighborhood.findBest(i);
178         for(int d = 0; d < dims; d++) {
179           particles[i].oldVars[d] = particles[i].vars[d];
180           particles[i].velocity[d] =
181             w * particles[i].velocity[d]
182             + c1*rand.nextDouble()*(particles[i].bestVars[d]-particles[i].vars[d])
183             + c2*rand.nextDouble()*(socialBest[d]-particles[i].vars[d])
184             ;
185           if(particles[i].velocity[d] < -velBounds[d]) {
186             particles[i].velocity[d] = -velBounds[d];
187           } else if(particles[i].velocity[d] > velBounds[d]) {
188             particles[i].velocity[d] = velBounds[d];
189           }
```

```
190           particles[i].vars[d] = particles[i].vars[d] + particles[i].velocity[d];
191         }
192       }
193
194       // Izračunaj vrijednost funkcije u novim pozicijama svih čestica,
195       // i po potrebi ažuriraj najbolje rješenje čestice
196       for(int i = 0; i < particles.length; i++) {
197         particles[i].value = funkcija.izracunaj(particles[i].vars);
198         if(particles[i].value < particles[i].bestValue) {
199           particles[i].bestValue = particles[i].value;
200           for(int d = 0; d < dims; d++) {
201             particles[i].bestVars[d] = particles[i].vars[d];
202           }
203         }
204       }
205
206       // Ispiši statistiku na ekran
207       updateStatistics();
208     }
209
210     /**
211      * Pomoćna metode koja na ekran ispisuje statističke podatke o populaciji,
212      * te najbolje pronađeno rješenje.
213      */
214     private void updateStatistics() {
215       int bestindex = 0;
216       double bestValue = particles[bestindex].bestValue;
217       double sum = bestValue;
218       for(int i = 1; i < particles.length; i++) {
219         if(particles[i].bestValue<bestValue) {
220           bestValue = particles[i].bestValue;
221           bestindex = i;
222         }
223         sum += particles[i].bestValue;
224       }
225       StringBuilder sb = new StringBuilder();
226       sb.append("Iter: ");
227       sb.append(iteracija);
228       sb.append(", Average: ");
229       sb.append(df.format(sum/particles.length));
230       sb.append(", (");
231       for(int d = 0; d < dims; d++) {
232         if(d>0) {
233           sb.append(", ");
234         }
235         sb.append(df.format(particles[bestindex].bestVars[d]));
236       }
237       sb.append(")=");
238       sb.append(df.format(bestValue));
239       System.out.println(sb.toString());
240     }
241
242 }
```

Ispis B.7: Razred Particle.

```
1  package hr.fer.zemris.pso;
2
3  /**
4   * Čestica algoritma PSO.
5   *
6   * @author marcupic
7   */
8  public class Particle {
9
10   // Najbolje rješenje
```

```
11    double[] bestVars;
12    // Dobrota najboljeg rješenja
13    double bestValue;
14    // Prethodno rješenje
15    double[] oldVars;
16    // Trenutno rješenje
17    double[] vars;
18    // Vektor brzine
19    double[] velocity;
20    // Vrijednost funkcije u trenutnom rješenju
21    double value;
22
23    /**
24     * Konstruktor čestice. Prima broj dimenzija.
25     *
26     * @param dimensions broj dimenzija prostora
27     */
28    public Particle(int dimensions) {
29      vars = new double[dimensions];
30      oldVars = new double[dimensions];
31      velocity = new double[dimensions];
32      bestVars = new double[dimensions];
33    }
34  }
```

Ispis B.8: Sučelje Neighborhood.

```
1  package hr.fer.zemris.pso;
2
3  /**
4   * Sučelje koje apstrahira pojam susjedstva.
5   * Različite implementacije ponudit će konkretne
6   * definicije susjedstva.
7   *
8   * @author marcupic
9   *
10  */
11 public interface Neighborhood {
12
13    /**
14     * Metoda koja se mora pozvati nad populacijom
15     * kako bi se napunili podaci o rješenjima iz
16     * susjedstva. Ovo mora biti napravljeno prije
17     * uporabe funkcije {@linkplain #findBest(int)}
18     * i svakako prije bilo kakvih izmjena u česticama.
19     *
20     * @param particles populacija čestica
21     */
22    void scan(Particle[] particles);
23
24    /**
25     * Metoda koja za česticu određenu indeksom vraća
26     * poziciju najboljeg rješenja pronađenog u njezinom
27     * susjedstvu.
28     *
29     * @param forIndex indeks čestice
30     * @return najbolje rješenje u susjedstvu te čestice
31     */
32    double[] findBest(int forIndex);
33 }
```

Ispis B.9: Razred GlobalNeighborhood.

```
1  package hr.fer.zemris.pso;
2
3  /**
```

```
4   * Razred implementira pojam globalnog susjedstva. Kod ove
5   * vrste susjedstva, svaka čestica svjesna je najboljeg rješenja
6   * koje je pronašla cjelokupna populacija.
7   *
8   * @author marcupic
9   */
10  public class GlobalNeighborhood implements Neighborhood {
11
12    // Broj čestica
13    int particlesCount;
14
15    // Dimezija prostora
16    int dims;
17
18    // Najbolje globalno rješenje
19    double[] best;
20
21    // Radi li se minimizacija (true) ili maksimizacija (false)
22    boolean minimize;
23
24    /**
25     * Konstruktor susjedstava.
26     *
27     * @param particlesCount broj čestica
28     * @param dims dimenzija
29     * @param minimize true ako se radi minimizacija, false inače
30     */
31    public GlobalNeighborhood(int particlesCount, int dims, boolean minimize) {
32      this.particlesCount = particlesCount;
33      this.dims = dims;
34      this.minimize = minimize;
35      best = new double[dims];
36    }
37
38    /**
39     * Pronalazi globalno najbolje rješenje populacije.
40     *
41     * @see hr.fer.zemris.pso.Neighborhood#scan(hr.fer.zemris.pso.Particle[])
42     */
43    @Override
44    public void scan(Particle[] particles) {
45      int bestindex = 0;
46      double bestValue = particles[bestindex].bestValue;
47      for(int i = 1; i < particles.length; i++) {
48        if((minimize && particles[i].bestValue<bestValue) ||
49            (!minimize && particles[i].bestValue>bestValue)) {
50          bestValue = particles[i].bestValue;
51          bestindex = i;
52        }
53      }
54      for(int d = 0; d < dims; d++) {
55        best[d] = particles[bestindex].bestVars[d];
56      }
57    }
58
59    /**
60     * Vraća najbolje rješenje za zadanu česticu.
61     * @see hr.fer.zemris.pso.Neighborhood#findBest(int)
62     */
63    @Override
64    public double[] findBest(int forIndex) {
65      return best;
66    }
67  }
```

Ispis B.10: Razred LocalNeighborhood.

```
1  package hr.fer.zemris.pso;
2
3  /**
4   * Razred implementira pojam lokalnog susjedstva određene širine.
5   *
6   * @author marcupic
7   */
8  public class LocalNeighborhood implements Neighborhood {
9
10   // Broj čestica
11   int particlesCount;
12
13   // Dimezija prostora
14   int dims;
15
16   // Veličina susjedstva
17   int nSize;
18
19   // Najbolja rješenje za susjedstvo svake čestice
20   double[][][] best;
21
22   // Radi li se minimizacija (true) ili maksimizacija (false)
23   boolean minimize;
24
25   /**
26    * Konstruktor.
27    *
28    * @param particlesCount broj čestica
29    * @param dims broj dimenzija
30    * @param nSize veličina susjedstva
31    * @param minimize true ako se radi minimizacija, false inače
32    */
33   public LocalNeighborhood(int particlesCount, int dims, int nSize,
34                            boolean minimize) {
35     this.particlesCount = particlesCount;
36     this.dims = dims;
37     this.minimize = minimize;
38     this.nSize = nSize;
39     best = new double[particlesCount][dims];
40   }
41
42   /**
43    * Pronalazi najbolja rješenja susjedstva za sve jedinke.
44    *
45    * @see hr.fer.zemris.pso.Neighborhood#scan(hr.fer.zemris.pso.Particle[])
46    */
47   @Override
48   public void scan(Particle[] particles) {
49     for(int index = 0; index < particles.length; index++) {
50       int startFrom = index - nSize/2;
51       int endAt = index + nSize/2;
52       if(startFrom < 0) startFrom = 0;
53       if(endAt >= particles.length) endAt = particles.length-1;
54
55       int bestindex = startFrom;
56       double bestValue = particles[bestindex].bestValue;
57       for(int i = startFrom+1; i <= endAt; i++) {
58         if((minimize && particles[i].bestValue<bestValue) ||
59             (!minimize && particles[i].bestValue>bestValue)) {
60           bestValue = particles[i].bestValue;
61           bestindex = i;
62         }
63       }
64       for(int d = 0; d < dims; d++) {
65         best[index][d] = particles[bestindex].bestVars[d];
66       }
```

```
67        }
68      }
69
70      /**
71       * Vraća najbolje rješenje za zadanu česticu.
72       * @see hr.fer.zemris.pso.Neighborhood#findBest(int)
73       */
74      @Override
75      public double[] findBest(int index) {
76        return best[index];
77      }
78    }
```

B.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.

Ispis B.11: Razred SimpleIA.

```
1  package hr.fer.zemris.ais;
2
3  import hr.fer.zemris.graphics.tsp.PrepareTSP;
4  import hr.fer.zemris.tsp.City;
5  import hr.fer.zemris.tsp.TSPSolution;
6  import hr.fer.zemris.tsp.TSPSolutionPool;
7  import hr.fer.zemris.tsp.TSPUtil;
8
9  import java.io.IOException;
10 import java.util.List;
11 import java.util.Random;
12
13 /**
14  * Razred prikazuje implementaciju algoritma SimpleIA (jednostavan
15  * imunološki algoritam) na problemu trgovačkog putnika.
16  *
17  * @author marcupic
18  */
19 public class SimpleIA {
20
21   // Polje gradova
22   private City[] cities;
23
24   // veličina populacije (broj antitijela)
25   private int paramD;
26
27   // broj klonova svakog rješenja (antitijela)
28   private int paramDup;
29
30   // priručna memorija s rješenjima
31   private TSPSolutionPool pool;
32
33   // Populacija rješenja
34   private TSPSolution[] population;
35
36   // Populacija klonova
37   private TSPSolution[] clonedPopulation;
38
39   // Unija obiju populacija
40   private TSPSolution[] unionPopulation;
41
42   // Generator slučajnih brojeva
43   private Random rand;
44
45   /**
46    * Konstruktor.
47    *
48    * @param cities lista gradova
49    */
50   public SimpleIA(List<City> cities) {
51     this.cities = new City[cities.size()];
52     cities.toArray(this.cities);
53     paramD = 50;
54     paramDup = 30;
55     this.pool = new TSPSolutionPool(this.cities.length);
56     rand = new Random();
57     population = new TSPSolution[paramD];
58     clonedPopulation = new TSPSolution[paramD*paramDup];
```

```
59      unionPopulation = new TSPSolution[paramD*paramDup + paramD];
60      initialize();
61      TSPUtil.evaluate(population, this.cities);
62    }
63
64    /**
65     * Inicijalizacija rješenja (antitijela).
66     */
67    private void initialize() {
68      for(int i = 0; i < paramD; i++) {
69        TSPSolution s = pool.get();
70        population[i] = s;
71        TSPUtil.randomInitializeSolution(s, rand);
72      }
73    }
74
75    /**
76     * Glavna petlja optimizacijskog algoritma.
77     */
78    public void go() {
79
80      // Postavi parametre
81      int iter = 0;
82      int iterLimit = 2000;
83
84      // Ponavljaj zadani broj puta
85      while(iter < iterLimit) {
86        iter++;
87        cloning();
88        hyperMutation();
89        TSPUtil.evaluate(clonedPopulation, cities);
90        select();
91      }
92      // Najbolje rješenje je prvo zbog sortiranja!
93      System.out.println("Best length: "+population[0].tourLength);
94      System.out.println(population[0]);
95      PrepareTSP.visualize(TSPUtil.reorderCities(cities, population[0].cityIndexes));
96    }
97
98    /**
99     * Operator kloniranja. Za svaku jedinku iz glavne populacije stvara
100    * zadani broj klonova i time generira populaciju klonova.
101    */
102   private void cloning() {
103     int index = 0;
104     for(int i = 0; i < population.length; i++) {
105       TSPSolution s = population[i];
106       for(int j = 0; j < paramDup; j++) {
107         TSPSolution c = pool.get();
108         System.arraycopy(
109             s.cityIndexes, 0, c.cityIndexes, 0, s.cityIndexes.length);
110         clonedPopulation[index] = c;
111         index++;
112       }
113     }
114   }
115
116   /**
117    * Operator hipermutacije. Svaku jedinku iz populacije klonova mutira
118    * tako da zamijeni redosljed dva slučajno odabrana grada.
119    */
120   private void hyperMutation() {
121     for(int index = 0; index < clonedPopulation.length; index++) {
122       TSPSolution c = clonedPopulation[index];
123       int a = rand.nextInt(c.cityIndexes.length);
124       int b = rand.nextInt(c.cityIndexes.length);
```

```
125         if(a==b) {
126           if(b==c.cityIndexes.length-1) {
127             b--;
128           } else {
129             b++;
130           }
131         }
132         int tmp = c.cityIndexes[a];
133         c.cityIndexes[a] = c.cityIndexes[b];
134         c.cityIndexes[b] = tmp;
135       }
136     }
137
138     /**
139      * Operator selekcije. U uniju dodaje izvornu populaciju i klonove,
140      * traži najbolja rješenja za novu populaciju a ostalo otpušta u
141      * priručnu memoriju.
142      */
143     private void select() {
144       int index = 0;
145       for(int i = 0; i < population.length; i++) {
146         unionPopulation[index++] = population[i];
147       }
148       for(int i = 0; i < clonedPopulation.length; i++) {
149         unionPopulation[index++] = clonedPopulation[i];
150       }
151       TSPUtil.partialSort(unionPopulation, population.length);
152       for(int i = 0; i < population.length; i++) {
153         population[i] = unionPopulation[i];
154       }
155       for(int i = population.length; i < unionPopulation.length; i++) {
156         pool.release(unionPopulation[i]);
157       }
158     }
159
160     /**
161      * Ulazna točka u program.
162      *
163      * @param args argumenti komandne linije
164      */
165     public static void main(String[] args) throws IOException {
166       String fileName = args.length<1 ?
167           "data/gradovi03.txt"
168           : args[0];
169       List<City> cities = TSPUtil.loadCities(fileName);
170       if(cities==null) return;
171       new SimpleIA(cities).go();
172     }
173 }
```

Ispis B.12: Razred ClonAlg.

```
1  package hr.fer.zemris.ais;
2
3  import hr.fer.zemris.graphics.tsp.PrepareTSP;
4  import hr.fer.zemris.tsp.City;
5  import hr.fer.zemris.tsp.TSPSolution;
6  import hr.fer.zemris.tsp.TSPSolutionPool;
7  import hr.fer.zemris.tsp.TSPUtil;
8
9  import java.io.IOException;
10 import java.util.Arrays;
11 import java.util.List;
12 import java.util.Random;
13
14 /**
```

```
15  * Razred prikazuje implementaciju algoritma ClonAlg
16  * na problemu trgovačkog putnika.
17  *
18  * @author marcupic
19  */
20 public class ClonAlg {
21
22   // Polje gradova
23   private City[] cities;
24
25   // Parametar koji određuje veličinu populacije klonova
26   private int paramBeta;
27
28   // broj novih rješenja (antitijela) koje ćemo dodavati
29   private int paramD;
30
31   // Broj rješenja u populaciji (broj antitijela)
32   private int paramN;
33
34   // priručna memorija s rješenjima
35   private TSPSolutionPool pool;
36
37   // Populacija rješenja
38   private TSPSolution[] population;
39
40   // Populacija klonova
41   private TSPSolution[] clonedPopulation;
42
43   // Generator slučajnih brojeva
44   private Random rand;
45
46   // Veličina populacije klonova
47   private int clonedPopulationSize;
48
49   // Rangovi u populaciji klonova
50   private int[] clonedPopulationRanks;
51
52   /**
53    * Konstruktor.
54    *
55    * @param cities lista gradova
56    */
57   public ClonAlg(List<City> cities) {
58     this.cities = new City[cities.size()];
59     cities.toArray(this.cities);
60     paramN = 100;
61     paramD = 10;
62     paramBeta = 10;
63     this.pool = new TSPSolutionPool(this.cities.length);
64     rand = new Random();
65     population = new TSPSolution[paramN];
66     clonedPopulationSize = 0;
67     for(int i = 1; i <= paramN; i++) {
68       clonedPopulationSize += (int)((paramBeta*paramN)/((double)i)+0.5);
69     }
70     clonedPopulation = new TSPSolution[clonedPopulationSize];
71     clonedPopulationRanks = new int[clonedPopulationSize];
72     initialize();
73   }
74
75   /**
76    * Inicijalizacija rješenja (antitijela).
77    */
78   private void initialize() {
79     for(int i = 0; i < paramN; i++) {
80       TSPSolution s = pool.get();
```

```
81        population[i] = s;
82        TSPUtil.randomInitializeSolution(s, rand);
83      }
84    }
85
86    /**
87     * Glavna petlja optimizacijskog algoritma.
88     */
89    public void go() {
90
91      // Postavi parametre
92      int iter = 0;
93      int iterLimit = 2000;
94
95      // Ponavljaj zadani broj puta
96      while(iter < iterLimit) {
97        iter++;
98        TSPUtil.evaluate(population, cities);
99        cloning();
100       hyperMutation();
101       TSPUtil.evaluate(clonedPopulation, cities);
102       select();
103       birthAndReplace();
104     }
105     // Najbolje rješenje je prvo zbog sortiranja!
106     System.out.println("Best length: "+population[0].tourLength);
107     System.out.println(population[0]);
108     PrepareTSP.visualize(TSPUtil.reorderCities(cities, population[0].cityIndexes));
109   }
110
111   /**
112    * Operator kloniranja koji broj klonova koje će stvoriti utvrđuje proporcionalno
113    * dobroti samog rješenja i time generira populaciju klonova.
114    */
115   private void cloning() {
116     Arrays.sort(population, TSPUtil.solComparator);
117     int index = 0;
118     for(int i = 1; i <= population.length; i++) {
119       TSPSolution s = population[i-1];
120       int copies = (int)((paramBeta*paramN)/((double)i)+0.5);
121       for(int j = 0; j < copies; j++) {
122         TSPSolution c = pool.get();
123         System.arraycopy(
124                   s.cityIndexes, 0, c.cityIndexes, 0, s.cityIndexes.length);
125         clonedPopulation[index] = c;
126         clonedPopulationRanks[index] = i;
127         index++;
128       }
129     }
130   }
131
132   /**
133    * Operator hipermutacije koji broj mutacija u jediniki ima obrnuto proporcionalan
134    * dobroti jedinke. Što je jedinka bolja, to će se manje mutirati. Sama mutacija
135    * provodi se tako da se zamijeni redosljed dva slučajno odabrana grada.
136    */
137   private void hyperMutation() {
138     // Kada dodem na zadnju jedinku, zelim obaviti 1+c.cityIndexes.length/4 mutacija
139     double tau = 3.476 * (population.length-1);
140     // Idem od 1 kako bih najbolje rjesenje, i to njegovu prvu kopiju, ostavio
141       // netaknuto
142     for(int index = 1; index < clonedPopulation.length; index++) {
143       TSPSolution c = clonedPopulation[index];
144       int rank = clonedPopulationRanks[index]-1;
145       int mutations = (int)(1+c.cityIndexes.length*0.25*
146                                         (1-Math.exp(-rank/tau))+0.5);
```

```
147         for(int attempt = 0; attempt < mutations; attempt++) {
148           int a = rand.nextInt(c.cityIndexes.length);
149           int b = rand.nextInt(c.cityIndexes.length);
150           if(a==b) {
151             if(b==c.cityIndexes.length-1) {
152               b--;
153             } else {
154               b++;
155             }
156           }
157           int tmp = c.cityIndexes[a];
158           c.cityIndexes[a] = c.cityIndexes[b];
159           c.cityIndexes[b] = tmp;
160         }
161       }
162     }
163
164     /**
165      * Operator selekcije. Radi nad populacijom klonova i iz nje odabire
166      * najbolja rješenja za novu populaciju. Preostala neiskorištena rješenja
167      * vraćaju se priručnoj memoriji.
168      */
169     private void select() {
170       Arrays.sort(clonedPopulation, TSPUtil.solComparator);
171       for(int i = 0; i < population.length; i++) {
172         pool.release(population[i]);
173         population[i] = clonedPopulation[i];
174       }
175       for(int i = population.length; i < clonedPopulation.length; i++) {
176         pool.release(clonedPopulation[i]);
177       }
178     }
179
180     /**
181      * Operator rađanja - D najgorih rješenja nanovo slučajno generira.
182      */
183     private void birthAndReplace() {
184       int offset = population.length-paramD;
185       for(int i=0; i < paramD; i++) {
186         TSPUtil.randomInitializeSolution(population[offset+i], rand);
187       }
188     }
189
190
191     /**
192      * Ulazna točka u program.
193      *
194      * @param args argumenti komandne linije
195      */
196     public static void main(String[] args) throws IOException {
197       String fileName = args.length<1 ?
198           "data/gradovi01.txt"
199           : args[0];
200       List<City> cities = TSPUtil.loadCities(fileName);
201       if(cities==null) return;
202       new ClonAlg(cities).go();
203     }
204 }
```

B.5 Pomoćni razredi

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

Ispis B.13: Sučelje IFunkcija.

```
1  package hr.fer.zemris.numeric;
2
3  /**
4   * Sučelje koje opisuje funkciju koja se
5   * optimira.
6   *
7   * @author marcupic
8   */
9  public interface IFunkcija {
10
11   /**
12    * Metoda na temelju varijabli računa vrijednost
13    * funkcije.
14    *
15    * @param varijable varijable
16    * @return vrijednost funkcije
17    */
18   public double izracunaj(double[] varijable);
19 }
```

Ispis B.14: Razred City.

```
1  package hr.fer.zemris.tsp;
2
3  /**
4   * Razred predstavlja jedan grad. Grad je određen svojim imenom te
5   * koordinatama X i Y.<br>
6   * <i>Važno:</i> Varijable {@linkplain #x}, {@linkplain #y}
7   * i {@linkplain #name} su javne kako bi se omogućio minimalni
8   * "overhead" prilikom izvođenja evolucijskih algoritama. Ovo ima
9   * kao ružnu posljedicu da se neopreznim programiranjem vrijednosti
10  * mogu mijenjati od bilo kuda, što može dovesti do pogrešnog rada
11  * programa!
12  *
13  * @author marcupic
14  */
15 public class City {
16   // Koordinata X
17   public int x;
18   // Koordinata Y
19   public int y;
20   // Naziv grada
21   public String name;
22
23   /**
24    * Konstruktor. Ime se postavlja na null.
25    * @param x x koordinata
26    * @param y y koordinata
27    */
28   public City(int x, int y) {
29     this(null, x, y);
30   }
31
32   /**
33    * Konstruktor.
34    * @param name ime grada
35    * @param x x koordinata
36    * @param y y koordinata
37    */
```

```
38    public City(String name, int x, int y) {
39      super();
40      this.name = name;
41      this.x = x;
42      this.y = y;
43    }
44
45    @Override
46    public String toString() {
47      if(name!=null) {
48        return name+" ("+x+","+y+")";
49      } else {
50        return "City at ("+x+","+y+")";
51      }
52    }
53  }
```

Ispis B.15: Razred TSPSolution.

```
1   package hr.fer.zemris.tsp;
2
3   import java.util.Arrays;
4
5   /**
6    * Razred koji predstavlja jedno rješenje problema TSP.
7    * Razred automatski nudi mogućnosti <i>pool</i>anja
8    * pomoću razreda {@linkplain TSPSolutionPool}.<br>
9    * <i>Važno:</i> Varijable {@linkplain #cityIndexes},
10   * {@linkplain #tourLength} i {@linkplain #next} su javne kako bi
11   * se omogućio minimalni "overhead" prilikom izvođenja evolucijskih
12   * algoritama. Ovo ima kao ružnu posljedicu da se neopreznim
13   * programiranjem vrijednosti mogu mijenjati od bilo kuda, što može
14   * dovesti do pogrešnog rada programa!
15   *
16   * @author marcupic
17   */
18  public class TSPSolution {
19    // Indeksi kojima treba obići gradove
20    public int[] cityIndexes;
21    // Ukupna duljina ture
22    public double tourLength;
23    // Sljedeće rješenje; koristi se uz razred TSPSolutionPool
24    public TSPSolution next;
25
26    /**
27     * Konstruktor.
28     */
29    public TSPSolution() {
30    }
31
32    /**
33     * Konstruktor.
34     */
35    public TSPSolution(TSPSolution next) {
36      this.next = next;
37    }
38
39    @Override
40    public String toString() {
41      return Arrays.toString(cityIndexes)+", len="+tourLength;
42    }
43  }
```

Ispis B.16: Razred TSPSolutionPool.

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

```
2
3  /**
4   * Pomoćni razred koji služi za iznajmljivanje rješenja.
5   * Uporaba ovog razreda preporuča se u slučaju kada algoritam
6   * u petlji privremeno stvara veliku količinu novih rješenje
7   * i potom ih odbacuje. Neprestana uporaba memorijskog alokatora
8   * u takvom bi slučaju bila izuzetno neefikasna.
9   * Umjesto toga, ovaj razred objekte iznajmljuje i stvara ih
10  * po potrebi. Jednom kada je objekt stvoren, operacija
11  * dohvata i vraćanja je o(1).
12  *
13  * @author marcupic
14  *
15  */
16 public class TSPSolutionPool {
17   int citiesNumber;
18   TSPSolution first;
19
20   /**
21    * Konstruktor.
22    *
23    * @param citiesNumber broj gradova koji rješenja sadrže
24    */
25   public TSPSolutionPool(int citiesNumber) {
26     super();
27     this.citiesNumber = citiesNumber;
28   }
29
30   /**
31    * Metoda iznajmljuje jedno rješenje.
32    *
33    * @return rješenje
34    */
35   public TSPSolution get() {
36     if(first!=null) {
37       TSPSolution s = first;
38       first = (TSPSolution)first.next;
39       return s;
40     }
41     TSPSolution s = new TSPSolution();
42     s.cityIndexes = new int[citiesNumber];
43     return s;
44   }
45
46   /**
47    * Metoda preuzima vraćeno rješenje. To rješenje
48    * kasnije se može iznajmiti, i originalni vlasnik
49    * ga više NE SMIJE koristiti.
50    *
51    * @param sol rješenje koje se vraća
52    */
53   public void release(TSPSolution sol) {
54     sol.next = first;
55     first = sol;
56   }
57 }
```

Ispis B.17: Razred TSPUtil.

```
1  package hr.fer.zemris.tsp;
2
3  import hr.fer.zemris.util.ArraysUtil;
4
5  import java.io.BufferedReader;
6  import java.io.FileReader;
7  import java.io.IOException;
```

```
8  import java.util.ArrayList;
9  import java.util.Comparator;
10 import java.util.List;
11 import java.util.Random;
12
13 /**
14  * Pomoćni razred koji sadrži metode vezane uz TSP.
15  *
16  * @author marcupic
17  */
18 public class TSPUtil {
19
20   /**
21    * Metoda učitava listu gradova iz datoteke. Datoteka je tekstualna.
22    * U svakom retku nalazi se x i y koordinata grada razdvojene znakom
23    * tab.
24    *
25    * @param fileName naziv datoteke
26    * @return listu gradova ili null ako dođe do pogreške
27    * @throws IOException ako se dogodi pogreška u radu s datotekom
28    */
29   public static List<City> loadCities(String fileName) throws IOException {
30     BufferedReader br = null;
31     try {
32       br = new BufferedReader(new FileReader(fileName));
33       List<City> cities = new ArrayList<City>();
34       while(true) {
35         String line = br.readLine();
36         if(line==null) break;
37         line = line.trim();
38         if(line.isEmpty()) continue;
39         String[] elems = line.split("\\t");
40         cities.add(new City(
41                         Integer.parseInt(elems[0]),Integer.parseInt(elems[1])));
42       }
43       br.close();
44       return cities;
45     } catch(IOException ex) {
46       System.out.println("Pogreška prilikom rada s datotekom "+fileName);
47       if(br!=null) try { br.close(); } catch(Exception ignorable) {}
48       return null;
49     }
50   }
51
52   /**
53    * Metoda koja služi parcijalnom sortiranju predanog polja rješenja TSP-a.
54    * Zadatak metode je na početak polja staviti <code>number</code> najboljih
55    * rješenja (to su ona s najmanjom duljinom ture); poredak preostalog dijela
56    * polja nije bitan.
57    *
58    * @param population rješenja koja treba parcijalno sortirati
59    * @param number broj najboljih rješenja koja treba staviti na početak polja
60    */
61   public static void partialSort(TSPSolution[] population, int number) {
62     for(int i = 0; i < number; i++) {
63       int best = i;
64       for(int j = i+1; j < population.length; j++) {
65         if(population[best].tourLength > population[j].tourLength) {
66           best = j;
67         }
68       }
69       if(best != i) {
70         TSPSolution tmp = population[i];
71         population[i] = population[best];
72         population[best] = tmp;
73       }
```

```
74      }
75    }
76
77    /**
78     * Metoda za predano rješenje računa njegovu duljinu temeljem predane
79     * matrice udaljenosti. U toj matrici, na mjestu [i,j] nalazi se udaljenost
80     * od grada <code>i</code> do grada <code>j</code>.
81     *
82     * @param sol rješenje za koje treba izračunati duljinu ture
83     * @param distanceMatrix matrica udaljenosti gradova
84     */
85    public static void evaluate(TSPSolution sol, double[][] distanceMatrix) {
86      int pocetni = sol.cityIndexes[0];
87      double distance = 0;
88      for(int i = 1; i < sol.cityIndexes.length; i++) {
89        distance += distanceMatrix[pocetni][sol.cityIndexes[i]];
90        pocetni = sol.cityIndexes[i];
91      }
92      distance += distanceMatrix[pocetni][sol.cityIndexes[0]];
93      sol.tourLength = distance;
94    }
95
96    /**
97     * Metoda za predano rješenje računa njegovu duljinu temeljem predanog
98     * polja gradova i indeksa koji se nalaze u samom rješenju.
99     *
100    * @param sol rješenje
101    * @param cities polje gradova
102    */
103   public static void evaluate(TSPSolution sol, City[] cities) {
104     int pocetni = sol.cityIndexes[0];
105     double distance = 0;
106     for(int i = 1; i < sol.cityIndexes.length; i++) {
107       City a = cities[pocetni];
108       City b = cities[sol.cityIndexes[i]];
109       distance += Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
110       pocetni = sol.cityIndexes[i];
111     }
112     City a = cities[pocetni];
113     City b = cities[sol.cityIndexes[0]];
114     distance += Math.sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
115     sol.tourLength = distance;
116   }
117
118   /**
119    * Pomoćna metoda koja računa dobrotu svih tura u predanom polju.
120    *
121    * @param population polje rješenja
122    * @param cities polje gradova
123    */
124   public static void evaluate(TSPSolution[] population, City[] cities) {
125     for(int i = 0; i < population.length; i++) {
126       TSPSolution s = population[i];
127       evaluate(s, cities);
128     }
129   }
130
131   /**
132    * Metoda vraća novo polje gradova ne temelju izvornog polja gradova
133    * i predanog redosljeda određenog indeksima.
134    *
135    * @param cities originalno polje gradova
136    * @param indexes željeni poredak gradova
137    * @return novo polje složeno prema indeksima
138    */
139   public static City[] reorderCities(City[] cities, int[] indexes) {
```

```
140       City[] array = new City[indexes.length];
141       for(int i = 0; i < indexes.length; i++) {
142         City c = cities[indexes[i]];
143         array[i] = new City(c.x, c.y);
144       }
145       return array;
146     }
147
148     /**
149      * Pomoćna metoda koja stvara nasumični poredak gradova.
150      *
151      * @param sol rješenje
152      * @param rand generator slučajnih brojeva
153      */
154     public static void randomInitializeSolution(TSPSolution sol, Random rand) {
155       ArraysUtil.linearFillArray(sol.cityIndexes);
156       ArraysUtil.shuffleArray(sol.cityIndexes, rand);
157     }
158
159     /**
160      * Komparator dvaju rješenja. Rješenje je manje ako je duljina ture manja.
161      */
162     public static Comparator<TSPSolution> solComparator = new Comparator<TSPSolution>() {
163       @Override
164       public int compare(TSPSolution o1, TSPSolution o2) {
165         double razlika = o1.tourLength - o2.tourLength;
166         return razlika < 0 ? -1 : (razlika > 0 ? 1 : 0);
167       }
168     };
169
170 }
```

Ispis B.18: Razred ArraysUtil.

```
1  package hr.fer.zemris.util;
2
3  import java.util.Random;
4
5  /**
6   * Pomoćni razred s metodama za rad nad poljima.
7   *
8   * @author marcupic
9   */
10 public class ArraysUtil {
11
12   /**
13    * Metoda koja popunjava polje integera počev od 0 na dalje.
14    * Primjerice, ako je polje duljine 3, sadržaj će postati:
15    * 0, 1, 2.
16    *
17    * @param array polje koje treba popuniti
18    */
19   public static void linearFillArray(int[] array) {
20     for(int i = 0; i < array.length; i++) {
21       array[i] = i;
22     }
23   }
24
25   /**
26    * Metoda koja permutira redosljed elemenata u predanom
27    * polju posredstvom slučajnog mehanizma.
28    *
29    * @param array polje koje treba permutirati
30    * @param rand generator slučajnih brojeva
31    */
32   public static void shuffleArray(int[] array, Random rand) {
```

```
33      for(int i = array.length; i>1; i--) {
34        int b = rand.nextInt(i);
35        if(b!=i-1) {
36          int e = array[i-1];
37          array[i-1] = array[b];
38          array[b] = e;
39        }
40      }
41    }
42
43  }
```

Ispis B.19: Razred PrepareTSP.

```
1  package hr.fer.zemris.graphics.tsp;
2
3  import hr.fer.zemris.tsp.City;
4
5  import java.awt.BorderLayout;
6  import java.awt.Color;
7  import java.awt.Graphics;
8  import java.awt.GridLayout;
9  import java.util.ArrayList;
10 import java.util.List;
11
12 import javax.swing.JComponent;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15 import javax.swing.JPanel;
16 import javax.swing.SwingUtilities;
17 import javax.swing.WindowConstants;
18
19 /**
20  * Razred koji obavlja vizualizaciju pronađene rute kod problema TSP.
21  *
22  * @author marcupic
23  */
24 public class PrepareTSP extends JFrame {
25
26   private static final long serialVersionUID = 1L;
27
28   // Lista gradova; poredak već predstavlja rutu
29   private List<City> cities = new ArrayList<City>();
30   // Pomoćna labela za prikaz duljine rute
31   private JLabel duljinaLabel;
32   // Pomoćna labela za prikaz brja gradova
33   private JLabel brojLabel;
34
35   // Pomoćna komponenta koja obavlja iscrtavanje slike gradova
36   private VisualizeComponent komponenta;
37
38   /**
39    * Konstruktor koji prima polje gradova. Redosljed elemenata automatski određuje
40    * i samu turu.
41    *
42    * @param presetCities polje gradova
43    */
44   public PrepareTSP(City[] presetCities) {
45     setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
46     setSize(500, 500);
47     setLocation(20, 20);
48
49     if(presetCities!=null) {
50       for(City c : presetCities) {
51         cities.add(c);
52       }
```

```
53      }
54      komponenta = new VisualizeComponent();
55      this.getContentPane().setLayout(new BorderLayout());
56      this.getContentPane().add(komponenta, BorderLayout.CENTER);
57
58      JPanel p = new JPanel(new GridLayout(1,2));
59      duljinaLabel = new JLabel();
60      brojLabel = new JLabel();
61      p.add(brojLabel);
62      p.add(duljinaLabel);
63      this.getContentPane().add(p, BorderLayout.PAGE_END);
64
65      azurirajDuljinu();
66      setVisible(true);
67    }
68
69    /**
70     * Pomoćna funkcija koja računa duljinu ture i ažurira podatke u labelama
71     * {@linkplain #duljinaLabel} i {@linkplain #brojLabel}.
72     */
73    private void azurirajDuljinu() {
74      double d = 0;
75      if(!cities.isEmpty()) {
76        City cc = null;
77        City oldCc = null;
78        for(int i = 0; i < cities.size(); i++) {
79          cc = cities.get(i);
80          if(oldCc!=null) {
81            d += Math.sqrt((oldCc.x-cc.x)*(oldCc.x-cc.x) +
82                           (oldCc.y-cc.y)*(oldCc.y-cc.y));
83          }
84          oldCc = cc;
85        }
86        cc = cities.get(0);
87        if(cc!=oldCc) {
88          d += Math.sqrt((oldCc.x-cc.x)*(oldCc.x-cc.x) +
89                         (oldCc.y-cc.y)*(oldCc.y-cc.y));
90        }
91      }
92      duljinaLabel.setText("Duljina: "+((double)((int)(d*1000+0.5))/1000.0));
93      brojLabel.setText("Broj gradova: "+cities.size());
94
95    }
96
97    /**
98     * Pomoćna metoda koju smije pozvati proizvoljna dretva a služi za inicijalizaciju
99     * prikaza ture. Metoda prima polje gradova i generira prikaz. Polje se pri tome
100    * kopira pa je dretva-pozivatelj slobodna kasnije obavljati modifikacije nad
101    * poljem; to više neće imati nikakvog utjecaja na prikaz.
102    *
103    * @param cities polje gradova čijim je poretkom u polju ujedno određena i tura
104    */
105   public static void visualize(final City[] cities) {
106     try {
107       SwingUtilities.invokeAndWait(new Runnable() {
108         @Override
109         public void run() {
110           new PrepareTSP(cities);
111         }
112       });
113     } catch (Exception ignorable) {
114       System.out.println("Prikaz nije moguć.");
115     }
116   }
117
118   /**
```

```
119     * Pomoćna komponenta koja crta turu.
120     *
121     * @author marcupic
122     */
123    private class VisualizeComponent extends JComponent {
124
125      private static final long serialVersionUID = 1L;
126
127      @Override
128      protected void paintComponent(Graphics g) {
129        super.paintComponent(g);
130        if(cities.isEmpty()) return;
131        City cc = null;
132        City oldCc = null;
133        g.setColor(Color.BLACK);
134        for(int i = 0; i < cities.size(); i++) {
135          cc = cities.get(i);
136          if(oldCc!=null) {
137            g.drawLine(oldCc.x, oldCc.y, cc.x, cc.y);
138          }
139          oldCc = cc;
140        }
141        cc = cities.get(0);
142        if(cc!=oldCc) {
143          g.drawLine(oldCc.x, oldCc.y, cc.x, cc.y);
144        }
145        g.setColor(Color.BLACK);
146        for(int i = 0; i < cities.size(); i++) {
147          cc = cities.get(i);
148          g.fillRect(cc.x-2, cc.y-2, 6, 6);
149        }
150      }
151    }
152  }
```

Indeks