Umjetna inteligencija

Evolucijsko računarstvo

doc. dr. sc. Marko Čupić

IZDAVAČ

JAVNO DOSTUPNO NA WEB STRANICI JAVA.ZEMRIS.FER.HR/NASTAVA/UI

Ovaj materijal nastao je na temelju knjige "Prirodom inspirirani optimizacijski algoritmi" (autor: Marko Čupić), kao popratni materijal na kolegiju Umjetna inteligencija.

Prvo izdanje, lipanj 2016.

Sadržaj

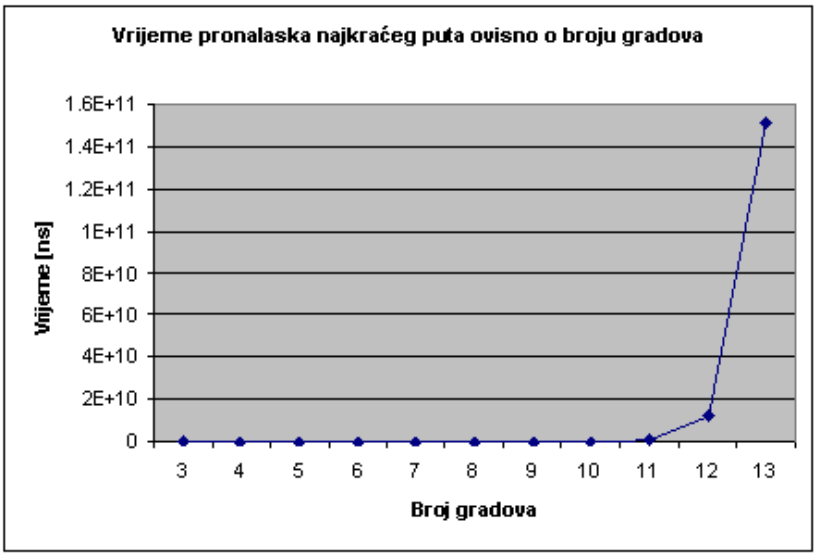
1. Uvod

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 te algoritam iterativnog pretraživanja u dubinu (sve smo ih obradili na ovom kolegiju). 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^* koji smo također obradili.

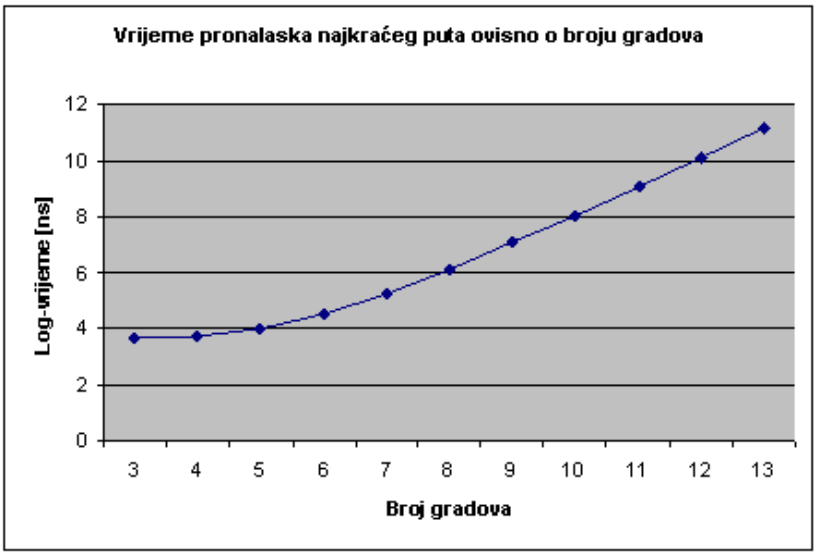
S obzirom da su danas računala ekstremno brza, nije rijetka situacija da se njihovom uporabom u jednoj sekundi mogu istražiti milijuni ili čak milijarde potencijalnih rješenja, i time vrlo brzo pronaći ono najbolje. Ovo, dakako, vrijedi samo ako smo jako sretni, pa rješavamo problem koji se grubom silom, odnosno uporabom upravo spomenutih algoritama, dade riješiti. Nažalost, postoji čitav niz problema koji ne spadaju u ovu kategoriju; spomenimo samo neke od njih.

Problem trgovačkog putnika (engleski termin je *Traveling sales-person problem*, odnosno kraće *TSP*) 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* pripadaju u vrstu problema za koje danas još uvijek nemamo efikasnih algoritama kojima bismo ih mogli riješiti. Uzrok tome je, dakako, ekstremna složenost 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



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

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. Faktorijelna 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, za ovaj konkretan problem pokazano je da nema potrebe ispitivati baš svaku permutaciju rješenja; algoritmom koji se temelji na *dinamičkom programiranju* ovaj problem moguće je rješiti u eksponencijalnoj složenosti (konkretno, u složenosti $O(n^2 \cdot 2^n)$). Iako je ovo bitno prihvatljivije od 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 pokazano da je $\mathscr{NP}$*-potpun* problem.

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

■ **Primjer 1.1 — Raspoređivanje neraspoređenih studenata u grupe za predavanja.** Razmotrimo problem koji se javlja naknadnim upisom studenata na predmete, nakon što je već napravljen raspored (redovno upisanih) 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. ■

■ **Primjer 1.2 — Izrada rasporeda međuispita.** Razmotrimo 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.
■

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

Definicija 1.0.1 — Heuristika. 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*. 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 1.0.2 — Metaheuristika. *Metaheuristika* 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.

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).

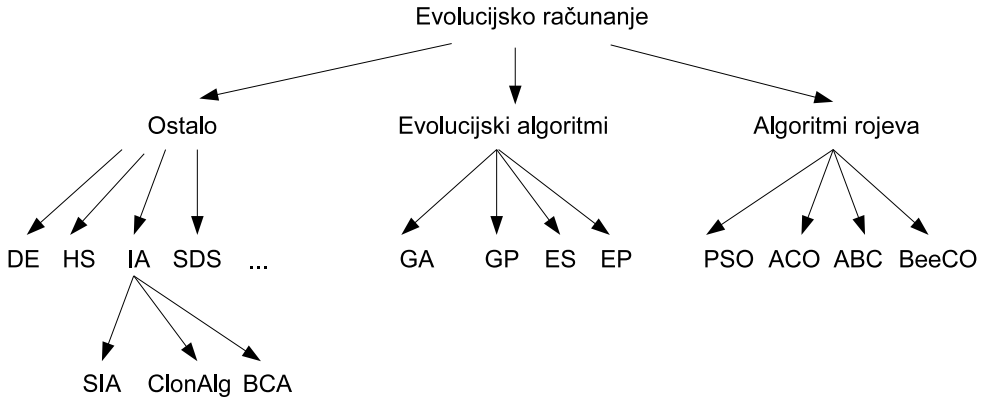
Osvrnimo se ovdje ukratko i na podskup algoritama *evolucijskog računanje* poznat pod nazivom *evolucijski algoritmi*. Danas ga uobičajeno dijelimo na četiri podpodručja: *genetske algoritme*, *genetsko programiranje*, *evolucijske strategije* te *evolucijsko programiranje*. Zajednička im je ideja da 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 dvije takve tehnike: genetski algoritam te optimizaciju temeljenu na mravljem algoritmu.

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. 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.



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

Područje evolucijskog računanja ujedno pripada i u područje metaheuristika – 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.

U algoritme rojeva ubrajamo *mravlje algoritme*, *algoritam roja čestica*, *algoritme pčela* i druge.

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.

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:

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. U ovom tekstu stoga ćemo dati pregled dva algoritma; zainteresiranog se čitatelja upućuje da dalje samostalno istraži druge pristupe, ili da iskoristi ponudu koju FER nudi kroz, primjerice, vještinu *Rješavanje optimizacijskih problema algoritmima evolucijskog računanja u Javi*.

1.3 Optimizacija

Algoritme o kojima govorimo u ovom tekstu koristimo za rješavanje optimizacijskih problema. Prethodno opisani problemi raspoređivanja neraspoređenih studenata u grupe za predavanja te izrade rasporeda međuispita mogu se svesti na optimizacijske probleme na način da se definira funkcija koja mjeri koliko je trenutno rješenje dobro. Potom puštamo algoritam da predlaže nova rješenja odnosno da korigira i kombinira trenutna, a sve kako bi došao do rješenja koja su bolja.

Definicija 1.3.1 — Optimizacija. Optimizaciju ćemo definirati kao postupak pronalaženja najboljeg rješenja problema. Rješenje će pri tome imati definiranu *funkciju dobrote* (engl. *fitness-function*) koju će optimizacijski postupak maksimizirati, ili *funkciju kazne* (cijenu; engl. *cost-function*) koju će optimizacijski postupak minimizirati.

Problemi koje optimiramo kreću se između dvije krajnosti. Mogu biti:

- kombinatorički - koji su definirani nad diskretnom domenom koja može ali čak i ne mora imati uređaj između elemenata (usporedite primjerice domenu cijelih brojeva i činjenicu da je 1 manje od 5 s domenom tropskog voća i činjenicom da ne možemo reći je li banana manja, veća ili jednaka ananasu);
- kontinuirani - koji su definirani nad kontinuiranim domenama poput realnih brojeva, kompleksnih brojeva i slično, i gdje već ta činjenica povlači beskonačan prostor pretraživanja koji grubom silom nikada ne možemo pretražiti.

U praksi, često se susrećemo i s problemima koji su kombinacija ovih krajnosti - neki od elemenata problema su diskretni a neki kontinuirani.

Optimizacijske probleme možemo promotriti i kroz prizmu problema pretraživanja prostora stanja o kojima smo već učili. Prisjetimo se: problem pretraživanja prostora stanja svodi se na problem pronalaska *puta* od početnog stanja s_0 do ciljnog stanja s_f koristeći dozvoljene poteze (odnosno funkciju sljedbenika). Prisjetite se samo problema slagalice.

Jednu značajnu podvrstu problema pretraživanja prostora stanja čine problemi zadovoljavanja ograničenja.

Definicija 1.3.2 — Problem zadovoljavanja ograničenja. Problem zadovoljavanja ograničenja (engl. *Constraint Satisfaction Problem*) je vrsta problema pretraživanja prostora stanja kod koje put od početnog do konačnog stanja nije bitan. Rješenje je isključivo konačno stanje.

Uobičajena je situacija da kod problema zadovoljavanja ograničenja početna stanja generiramo potpuno slučajno i tijekom postupka pretraživanja ne pamtimo roditeljska stanja - radimo isključivo s jednim ili populacijom "trenutnih" stanja koje mjerimo funkcijom dobrote ili kazne. Postojanje ciljnog stanja također je često nepoznato. Primjerice, vratimo se na problem izrade rasporeda međuispita koji pripada u ovu vrstu problema. Definirajmo ispitni predikat tako da kao ciljno stanje proglasi svaki raspored u kojem niti jedan student nije raspoređen na dva međuispita u isto vrijeme.

Uzmemo li u obzir što je svaki od studenata upisao, koliko termina imamo na raspolaganju, koliko kolegija te koliki su raspoloživi kapaciteti dvorana, u općem slučaju ne možemo znati postoji li uopće ciljno stanje (odnosno raspored bez preklapanja). Stoga je uobičajeno da tijekom provođenja optimizacijskog postupka pratimo kvalitetu rješenja i da isti prekinemo unatoč činjenici da nije pronašao ciljno stanje (ali je neko za koje smo procijenili da je iskoristivo).

Problemi zadovoljavanja ograničenja mogu definirati:

- jedno ili više čvrstih ograničenja (engl. *hard constraints*) te
- jedno ili više mekih ograničenja (engl. *soft constraints*).

Definicija 1.3.3 — Čvrsta ograničenja. Čvrsta ograničenja su ograničenja koja nužno moraju biti ispunjena da bi rješenje bilo prihvatljivo.

Primjerice, kao čvrsto ograničenje mogli bismo definirati da međuispiti dvaju predmeta s prve godine preddiplomskog studija ne mogu ići u istom ili uzastopnim danima. U tam slučaju, svaki predloženi raspored koji ovo ne bi imao zadovoljeno automatski bi bio neprihvatljiv, neovisno o svim njegovim drugim karakteristikama.

Definicija 1.3.4 — Meka ograničenja. Meka ograničenja su ograničenja koja definiraju poželjna (ili nepoželjna) svojstva rješenja.

Primjerice, bilo bi dobro da za svakog studenta vrijedi da mu je razmak između uzastopnih međuispita barem dva dana. Uzmemo li u obzir populaciju od 4000 studenata, meko ograničenje nam omogućava da definiramo relativan odnos kvalitete dvaju rješenja: bolje je ono kod kojeg 3900 studenata nema prekršeno navedeno meko ograničenje od rješenja kod kojega to nema 3000 studenata.

U slučaju da mekih ograničenja ima više, u najjednostavnijem slučaju optimizacijski će ih postupak na određeni način agregirati u jednu kumulativnu mjeru na temelju koje će potom uspoređivati rješenja. Moguća su i druga rješenja koja vode na višekriterijsku odnosno mnogokriterijsku optimizaciju (engl. *multi-objective optimization* vs. *many-objective optimization*).

2. Genetski algoritam

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

Darwin teoriju razvoja vrsta temelji na 5 postavki:

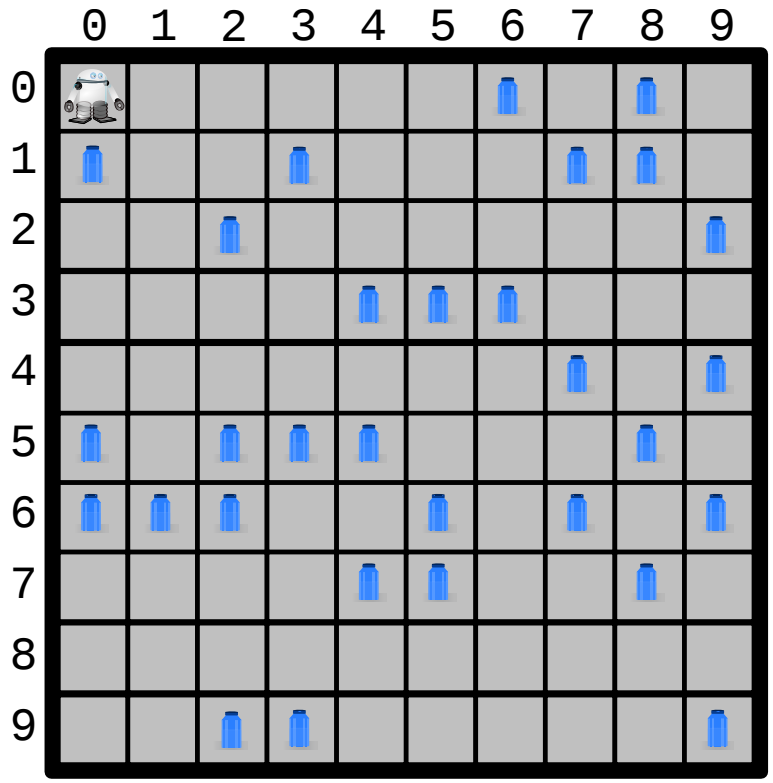
1. potomaka uvijek ima više no što je potrebno,
2. veličina populacije je približno stalna,
3. količina hrane je ograničena,
4. kod vrsta koje se seksualno razmnožavaju, nema identičnih jedinki već postoje varijacije te
5. najveći dio varijacija prenosi se nasljeđem.

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.

Genetski algoritam jedan je od algoritama stvorenih za rješavanje optimizacijskih problema, a koji izravno utjelovljuje navedene postavke. Kroz tekst koji slijedi upoznat ćemo se s idejom genetskog algoritma te uvidjeti da ne postoji "jedan" genetski algoritam već da je navedene postavke moguće implementirati na niz različitih načina čime dobivamo različite inačice genetskog algoritma. Kao motivacijski primjer poslužit će nam zadatak razvoja upravljačkog podsustava jednostavnog robota.

2.1 Motivacijski primjer

U svojoj knjizi *Complexity: A Guided Tour*, američka znanstvenica Melanie Mitchell daje vrlo zgodan primjer koji ćemo ovdje prikazati. Zamislimo zidovima ograđeni teren koji se sastoji od $m \times n$ ploča. Nakon održanog koncerta, na tom su terenu ostale porazbacane prazne boce vode. Pri tome se na svakoj ploči može zateći najviše jedna boca. Kako se koncerti redovno odvijaju,



Slika 2.1: Svijet robota Robbyja

umjesto da plaćamo čovjeka za čišćenje terena, želimo razviti robota koji će za nas obavljati taj posao. Primjer jednog takvog terena dimenzija 10×10 ploča prikazan je na slici 2.1. Robot se nalazi u gornjem lijevom uglu (ploča 0,0) a popunjenost terena bocama je 30% (od 100 ploča, na njih 30 se nalazi boca).

Robot kroz dostupne senzore od okoline može primiti sljedeće informacije:

- što se nalazi na ploči na kojoj je robot,
- što se nalazi jednu ploču iznad ploče na kojoj je robot,
- što se nalazi jednu ploču ispod ploče na kojoj je robot,
- što se nalazi jednu ploču desno od ploče na kojoj je robot te
- što se nalazi jednu ploču lijevo od ploče na kojoj je robot.

Senzor pri tome može dojaviti da je promatrana ploča prazna, da se na njoj nalazi boca ili pak da se radi o zidu. Stanje ploča koje su dijagonalne trenutnoj te stanje ploča koje nisu susjedne ploči na kojoj je robot senzori ne percipiraju. Robot kojim raspolažemo također ne posjeduje nikakvu memoriju - robot ne može zapamtiti da je u nekom ranijem trenutku već vidio na nekoj ploči bocu. U tom smislu, akcije koje robot poduzima u potpunosti su reaktivne: temeljem ulaza koje senzori u određenom trenutku šalju, robot mora odlučiti koju će akciju poduzeti.

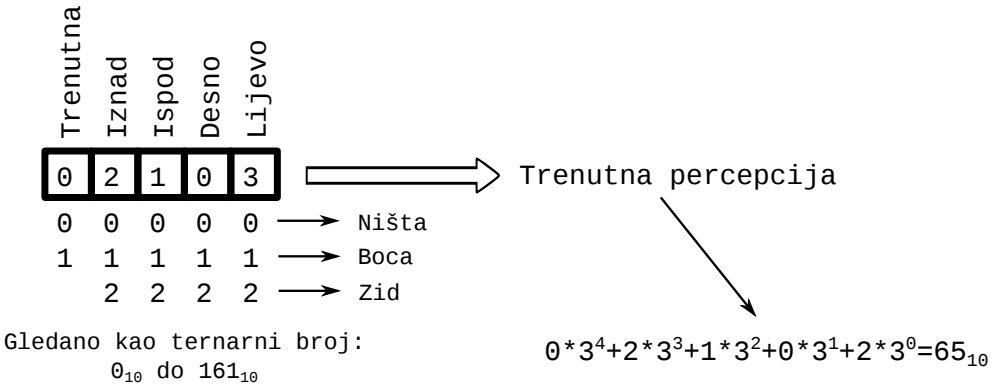
Upravljački podsustav robota omogućava izvođenje sedam različitih akcija, kako slijedi.

1. Ne radi ništa.
2. Sagni se i pokupi bocu s ploče na kojoj stojiš.
3. Otiđi na susjednu ploču koja je iznad ploče na kojoj stojiš.
4. Otiđi na susjednu ploču koja je ispod ploče na kojoj stojiš.
5. Otiđi na susjednu ploču koja je desno od ploče na kojoj stojiš.
6. Otiđi na susjednu ploču koja je lijevo od ploče na kojoj stojiš.
7. Otiđi na slučajno odabranu susjednu ploču.

U slučaju da se robot odabranom akcijom sudari sa zidom, trenutna pozicija mu se ne mijenja (ne može se popesti na zid).

Zadatak koji želimo riješiti je sljedeći: za svaku moguću percepciju koju robot prima potrebno je odrediti akciju koju robot treba poduzeti kako bi u ograničenom broju akcija pokupio što veći broj boca. Naravno, idealno bi bilo da robot uspije pokupiti sve boce i tako očisti teren. Za teren dimenzija 10×10 broj akcija ćemo ograničiti na 150.

S obzirom da je za svaku moguću percepciju robota (odnosno različitu situaciju u kojoj se robot može zateći) potrebno odrediti akciju koju će robot poduzeti, idemo najprije odrediti koliko različitih percepcija može dobiti Robby? Kao pomoć razmotrimo sliku 2.2.



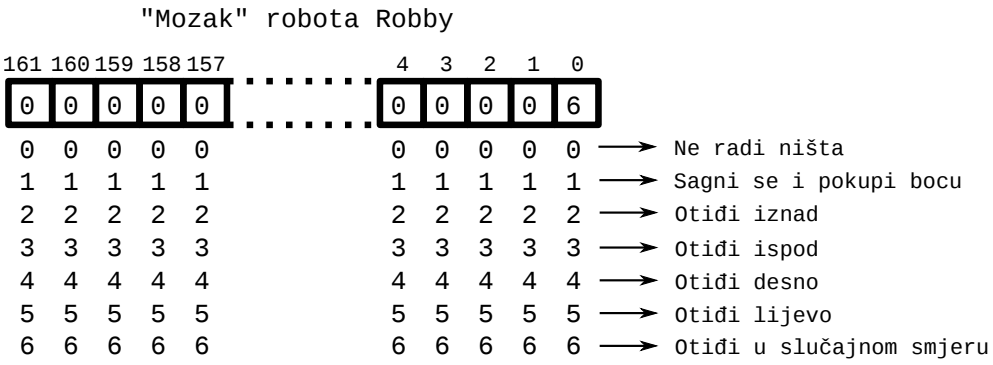
Slika 2.2: Percepcija robota Robbyja

Robby raspolaže s pet senzora koji mu govore što se nalazi na ploči na kojoj je i on sam te na izravno susjednim pločama. Stoga možemo razmišljati na sljedeći način. Na svakoj od tih pet ploča može biti ništa, boca ili zid. Broj mogućih kombinacija tada bi bio $3 \times 3 \times 3 \times 3 \times 3 = 3^5 = 243$. Međutim, stvarni broj mogućih percepcija nešto je manji. Primjerice, nemoguće je da se Robby nalazi na zidu - stoga na trenutnoj ploči imamo samo dvije mogućnosti: ili je prazna ili sadrži bocu. Time je broj mogućih kombinacija $2 \times 3 \times 3 \times 3 \times 3 = 2 \cdot 3^4 = 162$. Mogli bismo isključiti još nekoliko situacija koje se na pravokutnom terenu koji je barem dimenzija 2×2 ne mogu dogoditi: da je Robby okružen s tri ili četiri zida. Međutim, nećemo to učiniti jer uz 162 percepcije imamo vrlo jednostavan način kako svakoj od tih percepcija pridijeliti jedan jedinstven redni broj. Slika 2.2 prikazuje ideju: tri moguće situacije jedne ploče (ništa, boca, zid) kodirat ćemo jednom ternarnom znamenkom. Čitava Robbyjeva percepcija tada je zapisiva jednim 5-znamenkastim ternarnim brojem čija vrijednost (u dekadskom sustavu) ide od 0 do 161.

Dodijelimo li i svakoj akciji redni broj (0: *ne radi ništa*, pa sve do 6: *otiđi na slučajno odabranu susjednu ploču*), postavljeni zadatak tada se svodi na određivanje jednog 162-komponentnog vektora cijelih brojeva, čiji je element na poziciji i broj akcije koju robot treba poduzeti kada kroz senzore dobiva percepciju i. Ovo je ilustrirano na slici 2.3, gdje je akcija za svaku percepciju postavljena na "Ne radi ništa" osim za percepciju 0 (prisjetimo se: situacija u kojoj robot stoji na praznoj ploči i susjedne ploče su također prazne) za koju je podešena akcija 6 (pomakni se na slučajno odabranu susjednu ploču).

Pretpostavimo da problem želimo riješiti na sljedeći način: isprobat ćemo kako se robot ponaša za svaku mogući konfiguraciju mozga i potom ćemo odabrati najbolju (onu uz koju robot prikuplja najveći broj boca uz zadani ograničeni broj koraka). Koliko konfiguracija moramo isprobati? Jedna konfiguracija mozga je 162-komponentni vektor. Svaki njegov element može biti postavljen na jednu od 7 vrijednosti (redni broj akcije). Stoga je broj različitih konfiguracija jednak:

$$7 \cdot 7 \cdot 7 \cdot 7 \cdots 7 \cdot 7 \cdot 7 = 7^{162}.$$



Slika 2.3: Jedna moguća konfiguracija "mozga" robota Robbyja

Ako je za provjeru "kvalitete" jednog mozga potrebna 1 μs, za provjeru kvalitete 7^{162} trebat ćemo 10^{123} godina odnosno 10^{113} starosti svemira. Ovo jednostavno razmatranje upućuje da ćemo do rješenja morati nekim drugim putem.

2.1.1 Evolucija Robbyjevog mozga

Umjesto iscrpne pretrage, primjenit ćemo Darwinovu ideju evolucije. Napisat ćemo program koji će krenuti od populacije slučajno generiranih mozgova (doista, generatorom slučajnih brojeva generirat ćemo svih 162 komponente svakog od mozga iz populacije). Zatim ćemo definirati funkciju dobrote koja će vrednovati kvalitetu mozga. Tom ćemo funkcijom vrednovati svaki od mozgova u populaciji. Potom ćemo ponavljati jednostavan postupak: iz populacije ćemo posredstvom slučajnog mehanizma odabrati tri mozga; kombiniranjem dva bolja (prema dogovorenoj funkciji dobrote) stvorit ćemo novi mozak i taj ćemo umetnuti u populaciju na mjesto trećeg odabranog (najlošijeg od izabrana tri) mozga, i to samo ako novi mozak nije nekvalitetniji od tog trećeg. Pseudokod ovog postupka dan je u nastavku.

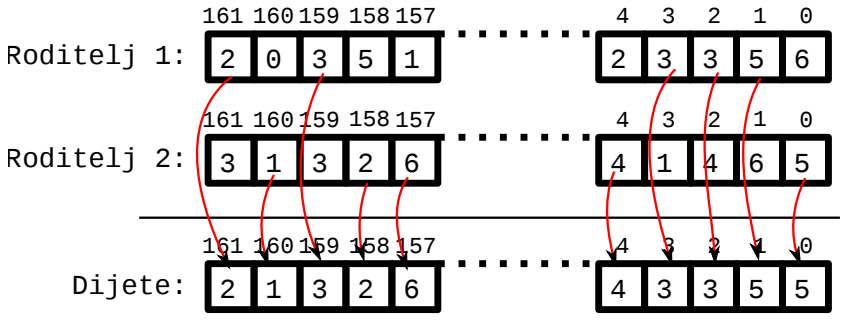
Pseudokod 2.1.1 — Eliminacijski genetski algoritam.

- Generiramo slučajnu populaciju mozgova od VEL_POP jedinki; evaluiramo svaki.
- Ponavljamo dok nije kraj:
  - Biramo slučajno tri jedinke
  - Dijete = Križamo bolje dvije + Mutacija
  - Vrednujemo dijete i ubacimo ga na mjesto treće ako nije lošije od nje

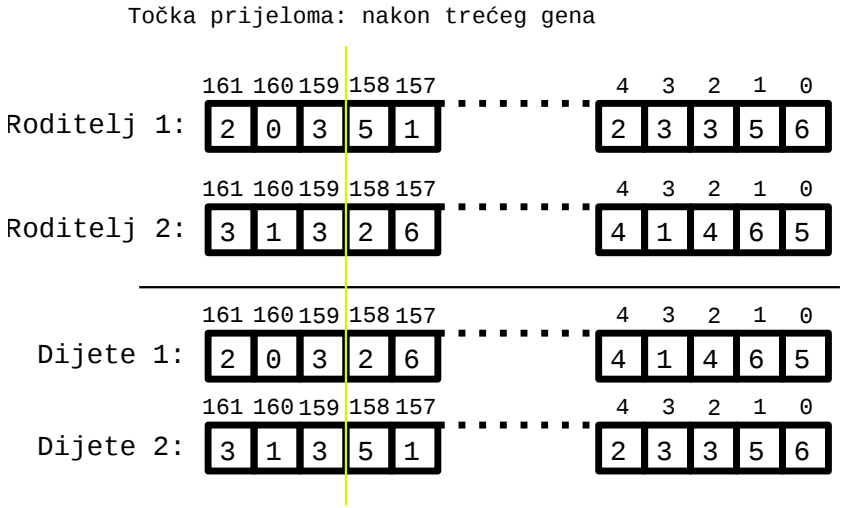
Opisani postupak poznat je kao *Troturnirski genetski algoritam* a koristi se još i naziv *Eliminacijski genetski algoritam* odnosno potpunije *Eliminacijski genetski algoritam s jednostavnom troturnirskom selekcijom*. U jednostavnijoj inačici, stvoreno dijete uvijek se ubacuje na mjesto treće-odabrane jedinke, neovisno o odnosu njihovih dobrota.

Križanjem dvaju roditelja nastaje dijete koje dio genetskog materijala preuzima od jednog roditelja a dio od drugog. Postoji niz načina na koje se to može ostvariti, a tipični primjeri su *uniformno križanje* te *križanje s jednom točkom prijeloma*. Kod uniformnog križanja za svaki se gen (u našem slučaju to je akcija za pojedinu percepciju) posredstvom slučajnog mehanizma bira hoće li biti preuzet od jednog ili od drugog roditelja. Slika 2.4 ilustrira ovu vrstu križanja.

Križanje s jednom točkom prijeloma bira poziciju gena na kojoj cijepa oba roditelja. Potom stvara dijete tako da od prvog roditelja uzme gene do točke prijeloma a od drugog gene nakon točke prijeloma. Drugo dijete moguće je izgraditi na analogni način: od drugog se roditelja uzimaju geni do točke prijeloma a od prvog nakon točke prijeloma. Slika 2.5 ilustrira ovu vrstu križanja.

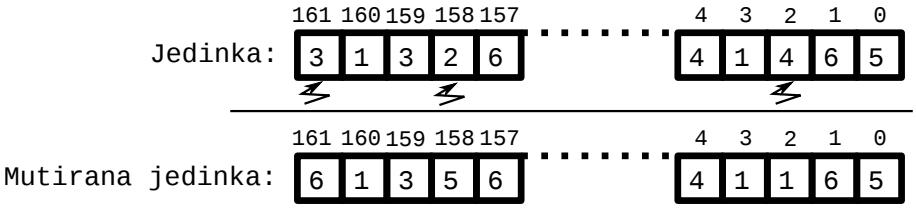


Slika 2.4: Provedba uniformnog križanja



Slika 2.5: Provedba križanja s jednom točkom prijeloma

Mutaciju ćemo provoditi na sljedeći način. Definirat ćemo *vjerojatnost mutacije gena* kao vjerojatnost da ćemo za neki konkretan gen napraviti mutaciju. Potom krećemo od prvog gena na dalje, i za svaki, sa zadanom vjerojatnošću radimo njegovu mutaciju. Primjerice, ako je vjerojatnost mutacije gena jednaka 0.05, tada će svaki od gena ima 5% šanse biti mutiran. Za gene za koje utvrdimo da ih treba mutirati, njihovu vrijednost zamijenimo slučajno odabranom akcijom. Slika 2.6 ilustrira ovako definiranu mutaciju, gdje je pod utjecajem mutacije došlo do promjene u tri gena.



Slika 2.6: Provedba mutacije

Da bismo algoritam mogli isprobati, još je preostalo definirati funkciju dobrote. Postupit ćemo na sljedeći način.

- Posredstvom slučajnog mehanizma stvorit ćemo N_S različitih terena, pri čemu će svi imati

popunjenost bocama od pb.

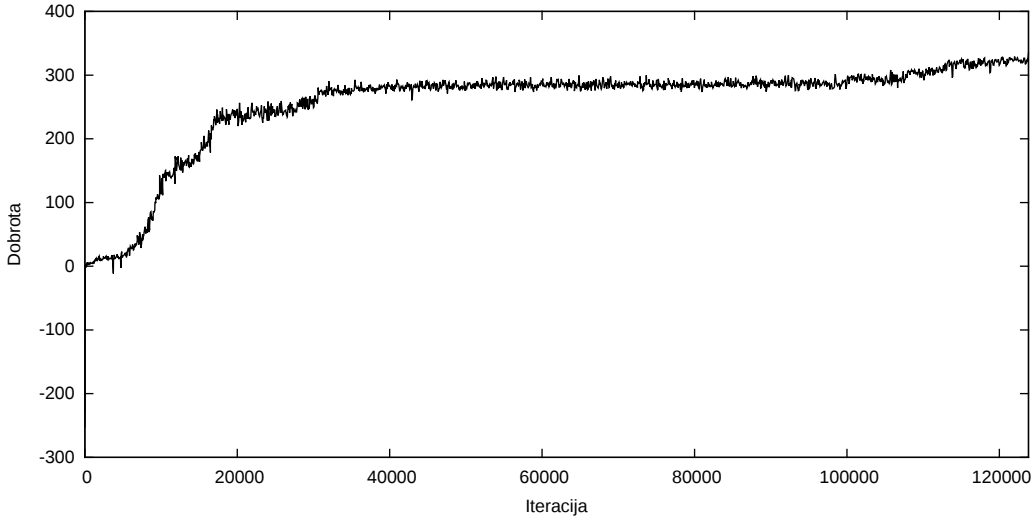
- Rad robota simulirat ćemo u svakom od N_S terena kroz zadani broj koraka N_K i pri tome ćemo robotu davati nagradne ili kaznene bodove. Dobrotu ponašanja u jednom terenu definirat ćemo kao ukupnu sumu bodova dobivenih na tom terenu. Konačnu dobrotu mozga postavit ćemo na prosječnu dobrotu ponašanja u svih N_S terena.
- Bodovanje radimo na sljedeći način.
  - $+10$ bodova dodijelit ćemo robotu svaki puta kada pokupi bocu.
  - -5 bodova dodijelit ćemo robotu svaki puta kada se sagne da pokupi bocu na ploči na kojoj nema boce.
  - -10 bodova dodijelit ćemo robotu svaki puta kada se zabije u zid.

Ponašanje robota na više terena provjeravamo kako se ne bi dogodilo da se mozak robota prilagodi jednoj specifičnoj konfiguraciji terena. Međutim, kako je stvaranje novih terena dosta skupo, postupak možemo ubrzati tako da jednom stvoreni skup terena koristimo određen broj iteracija (za ocjenu dobrote više od jednog mozga) pa ih tek tada nanovo generiramo.

Uvjet zaustavljanja algoritma može biti pronalazak dovoljno dobrog rješenja ili pak premašivanje zadanog broja iteracija.

Rad algoritma provjeren je uz sljedeće parametre. Postotak popunjenosti bocama postavljen je na 0.35, dimenzije terena su 10×10, broj koraka (odnosno akcija) koje robot smije napraviti je 150. Razvijamo mozak uporabom populacije od 50 jedinki, vjerojatnost mutacije gena je 0.02, vrednovanje robota provodimo na skupu od 20 svjetova i taj skup nanovo generiramo nakon svakih 100 vrednovanih robota. Uvjet zaustavljanja je pronalazak mozga s kojim robot u prosjeku uspije dobiti 330 bodova (uz dane parametre, teorijski maksimum je 350).

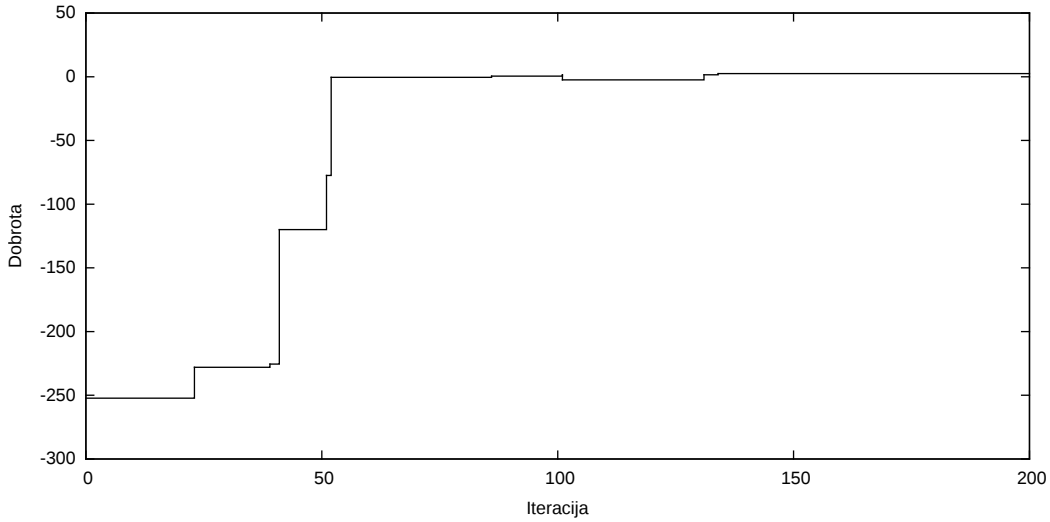
Kretanje funkcije dobrote za jedno konkretno pokretanje algoritma prikazano je na slici 2.7. Šumovitost prikazane krivulje rezultat je periodičke promjene skupa terena na kojima se roboti vrednuju. Uočite kako s vremenom dobrota rješenja postaje sve bolja i bolja.



Slika 2.7: Evolucija rješenja

Postupak je zaustavljen nakon iteracije 123826 u kojoj je pronađen mozak prosječne dobrote 334.5. Slika 2.8 detaljnije prikazuje zbivanja na samom početku evolucijskog procesa.

Na samom početku rada algoritma (iteracija 0), najbolje rješenje u populaciji imalo je prosječnu dobrotu -252.25. Slijedi da se radi o robotu koji se često zabija u zidove te se saginje da skuplja boce na poljima na kojima boca nema. U ranim trenutcima evolucije, taj se problem rješava pa se preferiraju mozgovi koji to ne rade. Oko iteracije 50 dolazi se do konfiguracije mozga čija je



Slika 2.8: Evolucija rješenja - početak

prosječna dobrota oko nule: radi se ili o mozgu koji je naučio akcije koje kažnjavamo kompenzirati povremenim prikupljanjem boca, ili pak o mozgu koji je naučio da je najbolje ne raditi ništa.

Kako vrijeme napreduje, mozak koji se razvija ipak postaje sve bolji i bolji. Jedan od trikova koji evolucija relativno rano razvija jest ponašanje koje tjera robota da, ako oko njega nema boca, konzistentno krene u jednom smjeru do prvog zida i potom krene kružiti ili u smjeru kazaljke na satu, ili u suprotnom smjeru (ali konzistentno). Zahvaljujući takvom ponašanju, robot će često pronalaziti nove boce (a jednom kad ih nađe, skuplja ih sve dok mu se ne dogodi situacija da oko sebe više nema ničega, nakon čega ponavlja postupak - odlazi do zida i nastavlja kruženje).

Međutim, samo to nije još dovoljno da bismo postigli konačnu traženu dobrotu. S vremenom, evolucija pronalazi još jedno interesantno rješenje: ako se robot nađe u situaciji da su oko njega s obje strane boce te ako je i on sam na polju na kojem je boca, tu bocu nije dobro pokupiti; umjesto toga, robot će krenuti u jednom smjeru dok ne dođe do kraja niza boca i potom se krenuti vraćati i tek ih tada sve redom skupljati. Razmislite zašto je ovakva strategija bitna!

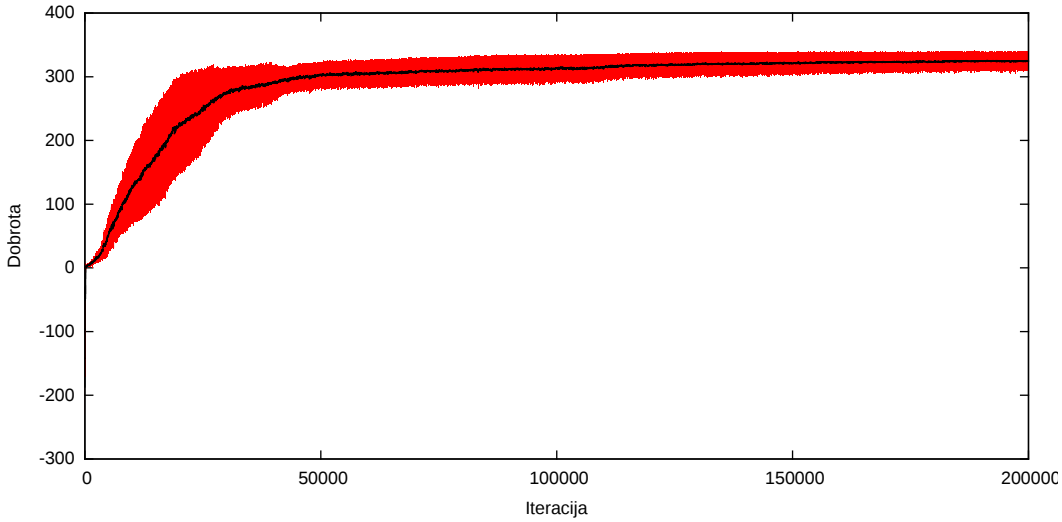
Slika 2.9 prikazuje još jednu karakteristiku navedenog algoritma. Postupak optimizacije pokrenuli smo 20 puta te smo za svako od pokretanja vodili detaljni dnevnik iteracija u kojima su pronađena nova bolja rješenja. Ova slika prikazuje kretanje prosječne dobrote a crvena područja oko krivulje označavaju pojas širine jednog standardnog odstupanja (engl. *standard deviation*) izračunatih na temelju dobrote svakog od 20 eksperimenata u promatranoj iteraciji.

2.2 Primjena na numeričku optimizaciju

U prethodnom poglavlju razmotrili smo zadatak evolucije upravljačkog podsustava robota, što je primjer kombinatoričke optimizacije. Od mogućih 7^{162} različitih konfiguracija zadatak je bio pronaći neku koja je dovoljno dobra za daljnju uporabu. Ista se načela mogu primijeniti i na zadatke numeričke optimizacije. Numeričke optimizacije tipično su problemi pretraživanja n-dimenzijskog podprostora kartezijevih produkata skupova realnih brojeva.

U nastavku ćemo razmotriti dva od mnoštva sličnih primjera koje nije moguće jednostavno (ili uopće) rješavati analitički. Razmotrit ćemo i način na koji bismo iste mogli krenuli rješavati genetskim algoritmom.

■ **Primjer 2.1** Zadana je funkcija $f(x,y,z) = \sin(z^2 \cdot \frac{1}{|x|+0.02} \cdot e^{0.2 \cdot y}) + \cos(0.02 \cdot \frac{x \cdot z}{|y|+0.02} - 3 \cdot x)$.

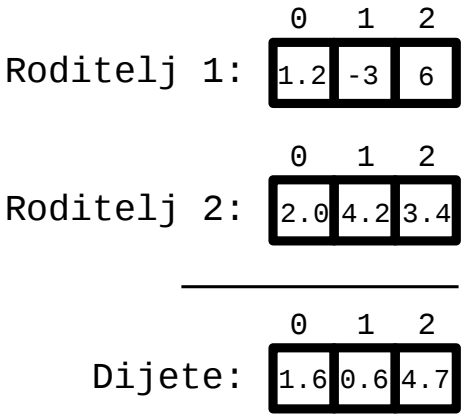


Slika 2.9: Evolucija rješenja - prosječno ponašanje

Pronaći za koju vrijednost argumenata x, y i z ova funkcija poprima minimalnu vrijednost na podprostoru određenom s $x \in [-10, 10]$, $y \in [-10, 10]$ i $z \in [-10, 10]$. ∎

U ovom primjeru *jedinka* (odnosno sinonimi: *rješenje*, *kromosom*) bi predstavljala jednu točku u trodimenzijskom prostoru realnih brojeva. U konkretnom programskom jeziku, mogli bismo je, primjerice, pamtiti kao polje od tri decimalna broja. Postavljeni zadatak ima prirodnu definiciju *funkcije kazne*: za svaku jedinku njezinu ćemo kaznu definirati kao iznos funkcije u točki koju predstavlja jedinka.

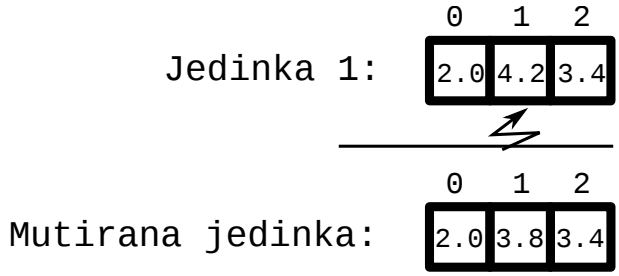
Do funkcionalnog genetskog algoritma još nas dijeli samo zadaća definiranja operatora križanja i mutacije. Križanje dviju jedinki koje su prikazane kao polje decimalnih brojeva možemo ostvariti na mnoštvo načina. Primjerice, u djetetu svaki od elemenata postavimo na aritmetičku sredinu pripadnih elemenata iz oba roditelja. Primjer takvog križanja ilustriran je na slici 2.10. Postoji i niz drugih mogućnosti; primjerice, križanje s jednom točkom prijeloma koje smo već opisali.



Slika 2.10: Definicija operatora križanja: aritmetička sredina po komponentama

Mutaciju također možemo raditi na više načina. Primjerice, možemo definirati vjerojatnost mutacije varijable p_b, pa u skladu s tom vjerojatnošću za svako od elemenata polja provjeriti hoćemo li ga mutirati ili ne. Ako je odgovor da, možemo mu dodati neki slučajno odabrani broj

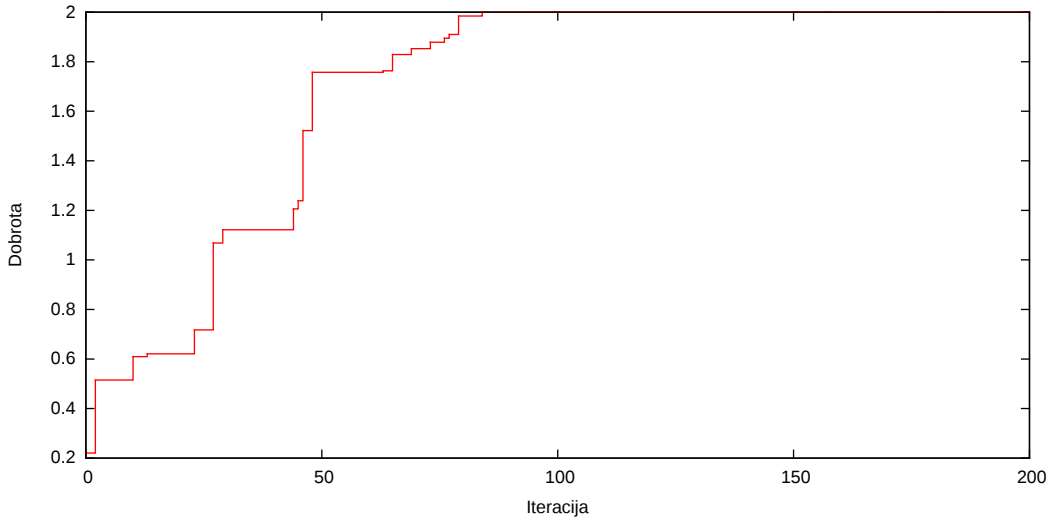
izvučen iz normalne distribucije centrirane oko 0 i prikladne standardne devijacije. Primjer ovakve mutacije prikazan je na slici 2.11. Pitali smo se želimo li mutirati prvi element, odgovor je bio ne; za drugi element odgovor je bio da; za treći element odgovor je bio ne. Za potrebe mutacije drugog elementa generirali smo slučajni broj iz normalne distribucije (u prikazanom primjeru na slici broj je bio -0.4) i dodali ga drugom elementu roditelja čime smo dobili vrijednost $4.2 + (-0.4) = 3.8$ koju smo upisali kao drugi element polja djeteta. Uočite da ovako definirana mutacija dozvoljava da se ponekad mutira samo jedan element, ponekad dva elementa, ponekad tri, itd., ali u prosjeku, broj mutiranih elemenata bit će proporcionalan zadanom vjerojatnoću p_b.



Slika 2.11: Definicija operatora mutacije: nadodajemo broj iz normalne razdiobe oko nule

Uz ovaj dogovor imamo sve što je potrebno kako bismo upogonili prethodno opisani eliminacijski genetski algoritam. Pri provođenju turnira potrebno je samo obratiti pažnju da su sada dva bolja roditelja ona koja imaju *manju* dodijeljenu kaznu.

Želimo li raditi s *funkcijom dobrote* (za koju vrijedi: veće je bolje), istu bismo mogli definirati kao negativnu vrijednost iznosa funkcije u točki koju predstavlja jedinka. u tom slučaju, od tri izabrane jedinke, roditelji postaju dvije jedinke koje imaju veću dobrotu. Slika 2.12 prikazuje postupak traženja minimuma ove funkcije pri čemu radimo s funkcijom dobrote. Uočite, kako se radi o zbroju sinusa i kosinusa, teorijski minimum je -2 (odnosno maksimalna dobrota je 2).



Slika 2.12: Evolucija rješenja kod traženja minimuma funkcije

Prikazana evolucija rješenja ostvarena je populacijom od samo 5 jedinki, a pronađeno rješenje je $x = 9.113271967556997$, $y = 1.1167555646506055$, $z = -5.869755348529594$. Opetovanim pokretanjem algoritma lako je utvrditi i da rješenje nije jedinstveno. Križanje i mutacija izvedeni

su u skladu s prethodno danim opisom i slikama 2.10 i 2.11. Mutacija je brojeve kojima korigira vrijednost zapisanu u roditelju dobivala izvlačenjem iz normalne distribucije s parametrima $(0, 0.2)$ pri čemu je prvi broj srednja vrijednost distribucije a drugi standardno odstupanje.

■ **Primjer 2.2** Kroz određeno vrijeme promatran je rad nekog sustava. Sustav ima 3 ulaza (označimo ih x, y i z) i jedan izlaz (označimo ga s P). Ulazima sustava upravljaju naponski. Na izlazu sustav generira laserski impuls određene snage. Poznato je da je ovisnost snage tog impulsa o naponima koji se dovode na ulaz oblika $P(x,y,z) = \frac{\alpha \cdot e^{\beta \cdot x}}{y} + \gamma \cdot z^2$. Međutim, konkretne vrijednosti parametara α, β i γ nisu poznate. Tijekom praćenja rada sustava napravljen je i zabilježen niz mjerenja oblika: $(x,y,z) \to P$; primjerice, kada je x bio 0.1 V i y bio 0.3 V i z bio -0.2 V, snaga lasera na izlazu je bila 0.48 W. Uz pretpostavku da smo prikupili stotinjak takvih mjerenja, potrebno je odrediti parametre α, β i γ uz koje će zadani model ovisnosti snage o ulaznim naponima najbolje odgovarati opaženom ponašanju sustava. ■

Kako bismo postupili u ovom slučaju? Kao prikaz rješenja opet možemo koristiti polje decimalnih brojeva. Kako imamo tri parametra koji se traže (α, β i γ), polje će imati tri elementa. Operatore križanja i mutacije možemo napraviti kao u prethodnom primjeru. Jedino pitanje koje preostaje jest kako sada definirati funkciju dobrote ili funkciju kazne. No to nije problem. Uočimo: uz zadane vrijednosti parametara model ovisnosti je fiksiran: imamo funkciju koja uz zadane x, y i z može izračunati koju snagu model predviđa. Ako su parametri dobro pogođeni, razlika između predviđene snage i snage koja je stvarno bila izmjerena bit će mala. Stoga ćemo najprije definirati funkciju pogreške kao srednje kvadratno odstupanje stvarno izmjerene snage i snage koju predviđa model za svaki od izmjerenih podataka. Ta funkcija pogreške izravno ovisi o traženim parametrima α, β i γ i nju ćemo poistovjetiti s funkcijom kazne. Genetskim algoritmom potom ćemo evoluirati prijedloge ovih parametara uz zadaću minimizacije funkcije kazne.

2.3 Dvije vrste genetskih algoritama

Kada razmatramo različite izvedbe genetskih algoritama, pronaći ćemo dvije krajnosti. Prvu čini implementacija algoritma koju smo uveli kao *eliminacijski genetski algoritam* a koji je u engleskoj terminologiji poznat kao *Steady-state Genetic Algorithm*. Radi se o izvedbi algoritma koja u svakom koraku radi minimalnu izmjenu u populaciji: mijenja se samo jedna jedinka. Isto tako, već u sljedećem koraku, ta novododana jedinka može biti roditelj na temelju kojeg se stvara sljedeće dijete.

Bitno drugačiju izvedbu čini takozvani *Generacijski genetski algoritam* čiji je pseudokod prikazan u nastavku. Radi se o izvedbi kod koje je na početku svake iteracije populacija roditelja fiksirana. Iz te populacije biraju se roditelji koji križanjem i mutacijom stvaraju djecu koja se pohranjuju u pomoćnu privremenu populaciju. Postupak se radi sve do trenutka dok u populaciji djece nema jednak broj jedinki koliko ih je i u roditeljskoj populaciji. Kada se to dogodi, roditeljska se populacija briše a stvorena populacija djece promovira se u roditelje. Taj trenutak označava smjenu generacije nakon čega se postupak ciklički ponavlja.

Pseudokod 2.3.1 — Generacijski genetski algoritam.

- Generiraj slučajnu populaciju mozgova od VEL_POP jedinki; evaluiraj svaki.
- Ponavljaj dok nije kraj:
  - Inicijaliziraj pomoćnu populaciju na praznu.
  - Ponavljaj dok veličina pomoćne populacije ne postane jednaka veličini populacije roditelja
    - Odaberi dva roditelja iz populacije roditelja

        * Dijete = Križaj roditelje + Mutacija
        * Vrednuj dijete
        * Ubaci ga u pomoćnu populaciju
    - Obriši populaciju roditelja
    - Promoviraj pomoćnu populaciju u populaciju roditelja

Kod generacijskog genetskog algoritma eventualno dobro dijete ne može se odmah iskoristiti; ono mora pričekati dok se ne završi čitava generacija. Prikazani algoritam također nije elitistički.

Definicija 2.3.1 — Elitizam. Elitizam je svojstvo algoritma da ne može izgubiti najbolje pronađeno rješenje.

Kod elitističkih algoritama graf koji prikazuje kretanje funkcije dobrote kroz evoluciju je monoton; pogledajte ponovno sliku 2.12 koja to ilustrira. Kod algoritama koji nisu elitistički taj će graf biti nazubljen: prosječne vrijednosti funkcije dobrote asimptotski će rasti ali je sasvim moguće da u nekon koraku algoritma dobrota najbolje jedinke bude manja od dobrote najbolje jedinke koju je algoritam imao u nekom prethodnom koraku.

Vježba 2.1 Generacijski genetski algoritam prikazan u prethodnom pseudokodu nije elitistički. Možete li objasniti zašto? ■

Vježba 2.2 Eliminacijski genetski algoritam kojim smo razvijali upravljački podsustav Robbyja je elitistički. Možete li objasniti zašto? ■

Generacijski genetski algoritam jednostavno se može pretvoriti u elitističku inačicu. Primjerice, na početku svake generacije u pomoćnu se populaciju djece može ubaciti kopija najbolje jedinke iz roditeljske populacije.

2.4 Utjecaj operatora

Neovisno o tome koju inačicu genetskog algoritma koristimo, operatori križanja i mutacije imaju važnu ulogu.

Operator križanja na temelju roditelja stvara djecu. Kako su ta djeca slična roditeljima, ovaj operator ima ulogu fine pretrage prostora rješenja u okolici rješenja koja predstavljaju roditelji. Operator križanja na čitavu populaciju često djeluje kontrakcijski. Uzmite za primjer križanje koje smo prikazali za jedinke koje reprezentiramo poljem decimalnih brojeva i koje djecu stvara izračunom aritmetičke sredine. Kada bismo koristili samo taj operator, svaka bi generacija bila sve zgusnutija i zgusnutija. Ako zamislite populaciju kao skup točaka u R-dimenzijskom prostoru koje zauzimaju određen volumen, jasno je vidljivo da će volumen populacije djece koja je dobivena opisanim križanjem biti manji. Ponovimo li to kroz više iteracija, populacija postaje sve gušća i gušća i gubi genetski materijal.

Operator križanja ima suprotnu ulogu: on nad jedinkom radi jednu ili više slučajnih izmjena koje jedinu mogu drastično izmijeniti. Ovaj operator u postupku pretraživanja povećava volumen populacije. Njegova druga važna uloga jest izbacivanje populacije iz lokalnih optimuma.

Da bi evolucijski postupak pretraživanja bio djelotvoran, nužno je postići dobar balans između sila koje uvode ovi operatori. Ako je kontrakcijsko djelovanje križanja prejako, postupak će se vrlo brzo zaglaviti u lokalnom optimumu. Ako je djelovanje mutacije prejako, ono će konzistentno uništavati sve pozitivno što je postupak pretraživanja do tada pronašao i pretragu će svesti na slučajnu. U idealnoj situaciji, ove su dvije sile međusobno u balansu čime se osigurava da postupak pretraživanja može napredovati prema kvalitetnijim rješenjima.

Opisana dinamika u stvarnosti je još malo kompliciranija: u obzir treba uzeti i *selekcijski pritisak* koji je posljedica načina na koji dobrota jedinke utječe na vjerojatnost njezinog razmnožavanja i preživljavanja. Na ovom mjestu nećemo se baviti formalnim definicijama ovog pojma već ćemo razmotriti dva primjera izvedbe operatora selekcije.

2.4.1 k-turnirska selekcija

Selekcija poznata pod nazivom k-turnirska selekcija omogućava kontroliranje selekcijskog pritiska jednostavnim podešavanjem jednog parametra: k iz naziva selekcije. Pseudokod ove selekcije prikazan je u nastavku.

Pseudokod 2.4.1 — k-turnirska selekcija.
Ulaz: populacija P jedinki veličine VEL_POP; $k \leq$VEL_POP. Izlaz: jedna jedinka

- Inicijaliziraj pomoćnu populaciju P' na praznu
- Ponavljaj k puta:
  - Iz populacije P slučajnim postupkom uz jednoliku distribuciju vjerojatnosti odaberi jednu jedinku koja još nije u populaciji P'
  - Ubaci je u populaciju P'
- Vrati najbolju (ili najgoru - ovisno što tražimo) jedinku iz populacije P'

Opisani postupak iz populacije vadi uzorak od k jedinki i vraća onu koja je najpoželjnija. Ako ovom selekcijom tražimo kandidata za roditelja, najpoželjnija će jedinka biti ona koja od jedinki iz uzorka ima najveću dobrotu (ili najmanju kaznu). Tražimo li jedinu za eliminaciju, najpoželjnija će jedinka biti ona koja od jedinki iz uzorka ima najmanju dobrotu (ili najveću kaznu). Trebamo li dva roditelja za križanje, uporabom ovakvog operatora selekcije, roditelje ćemo dobiti tako da dva puta pozovemo operator.

```
roditelj1 = $k$-turnir(P)
roditelj2 = $k$-turnir(P)
```

Što je parametar k veći, veći se naglasak daje na najbolja rješenja. Postavimo li da je $k =$VEL_POP, algoritam će kao roditelja uvijek birati najbolju jedinku populacije, a to će pak za posljedicu najčešće imati strelovitu konvergenciju algoritma u loš lokalni optimum. S druge pak strane, ako k postavimo na 1, roditelji se biraju potpuno slučajno i njihova dobrota uopće nije bitna; u ovom slučaju selekcijski pritisak uopće ne postoji i algoritam će nasumice i neusmjereno generirati nova rješenja. U praksi se ovakva inačica operatora koristi uz manje vrijednosti k; primjerice, 2 ili 3, čime se osigurava da postoji barem minimalna kompeticija između jedinki za roditeljstvo.

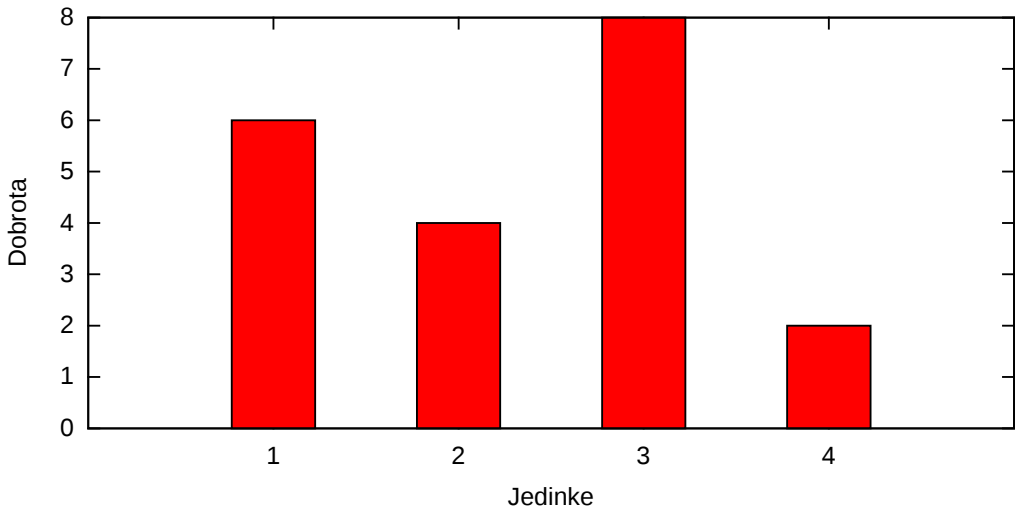
Kod eliminacijske izvedbe genetskog algoritma, ovaj se operator može koristiti za odabir jedinke koja će biti zamijenjena (eliminirana) stvorenim djetetom. Evo primjera takvog jednostavnog algoritma.

```
ponavljaj dok nije kraj
  roditelj1 = k-turnir(P, biraj najbolju)
  roditelj2 = k-turnir(P, biraj najbolju)
  dijete    = stvori(roditelj1, roditelj2)
  vrednuj(dijete)
  jedinka3  = k-turnir(P, biraj najlošiju)
  u populaciji P ukloni jedinku3 i zamijeni je sa stvorenim dijetetom
kraj
```

R U praksi često dolazi do konfuzije oko termina troturnirska selekcija. Ponekad se naprosto misli na operator koji turnirom uz $k = 3$ obavlja selekciju, a ponekad se misli na čitav eliminacijski genetski algoritam kakav smo opisali u poglavlju s robotom Robby. Pogledajte prethodni primjer. Ako u njemu postavimo $k = 3$, radimo tri turnira (dva za izbor roditelja i jedan za izbor jedinke koju ćemo eliminirati) odnosno biramo ukupno devet jedinki. Čitav se postupak može pojednostavniti tako da napravimo samo jedan turnir nad tri jedinke i dvije bolje odaberemo za roditelje a treću (najlošiju) zamijenimo nastalim djetetom. Gdje god ćemo govoriti o takvoj implementaciji algoritma, koristit ćemo termin *eliminacijski genetski algoritam s jednostavnom troturnirskom selekcijom*. Napomenimo i da dinamika takvog algoritma u odnosu na prethodno opisani algoritam koji dobivamo uz $k = 3$ nije jednaka.

2.4.2 Proporcionalna selekcija

Termin *Proporcionalna selekcija* (engl. *Roulette-Wheel Selection*) koristi se za operator koji jedinke bira iz populacije na temelju vjerojatnosti koje su im dodijeljene proporcionalno njihovim dobrotama. Pretpostavimo da imamo populaciju od 4 jedinke, i neka su im dobrote redom 6, 4, 8 i 2 (vidi sliku 2.13).



Slika 2.13: Razdioba dobrota jedinki iz populacije

Svakoj jedinki htjeli bismo pridijeliti vjerojatnost odabira koja je proporcionalna njezinoj dobroti. Označimo s fit_i dobrotu jedinke i a s $prob_i$ vjerojatnost da i-ta jedinka bude izabrana. Kako želimo da ta vjerojatnost bude proporcionalna dobroti, neka k predstavlja (za sada nepoznati) koeficijent proporcionalnosti, tako da možemo pisati:

$$prob_i = k \cdot fit_i \quad \forall i \in \{1,2,3,4\}.$$

Da bi $prob_i$ bile vjerojatnosti, njihova suma mora biti 1. Slijedi da možemo pisati:

$$prob_1 = k \cdot fit_1$$
$$prob_2 = k \cdot fit_2$$
$$prob_3 = k \cdot fit_3$$
$$prob_4 = k \cdot fit_4$$
$$prob_1 + prob_2 + prob_3 + prob_4 = 1$$

što je sustav 5 jednadžbi s 5 nepoznanica. Uvrštavanjem prve četiri u petu izravno dobivamo rješenje za faktor proporcionalnosti k:

$$k = \sum_{i=1}^{4} fit_i.$$

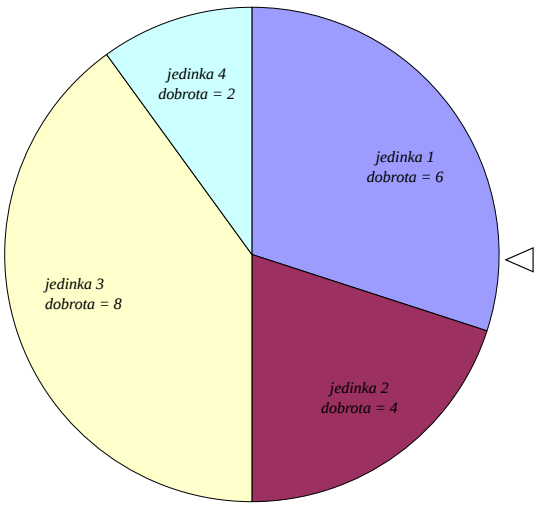
Uvrštavanjem u prve četiri jednadžbe dobivamo:

$$prob_1 = \frac{fit_1}{\sum_{i=1}^{4} fit_i}$$
$$prob_2 = \frac{fit_2}{\sum_{i=1}^{4} fit_i}$$
$$prob_3 = \frac{fit_3}{\sum_{i=1}^{4} fit_i}$$
$$prob_4 = \frac{fit_4}{\sum_{i=1}^{4} fit_i}$$

odnosno opći izraz:

$$prob_i = \frac{fit_i}{\sum_{j=1}^{n} fit_j}. \tag{2.1}$$

Pogledajmo to na našem konkretnom slučaju. Ukupna suma dobrota iznosi $6+4+8+2=20$, pa ćemo vjerojatnosti odabira jedinki redom postaviti na $6/20 = 0.3$, $4/20 = 0.2$, $8/20 = 0.4$ i $2/20 = 0.1$. Uz ovakvu distribuciju vjerojatnosti vidimo da će, primjerice, treća jedinka imati 40% šanse da postane roditelj. Ovo je lijepo prikazano i na slici 2.14 gdje su sve jedinke predstavljene kao kružni isječci pri čemu je širina oboda isječka proporcionalna dobroti jedinke koju isječak predstavlja.



Slika 2.14: Razdioba dobrota jedinki iz populacije prikazana kao kolo ruleta

Na slici je s desne strane prikazan i mali trokutić koji služi selekciji jedinki. Zamislimo da je prikazani krug u centru fiksiran i da ga možemo potezom ruke zavrtiti. Uslijed trenja kolo će u jednom trenutku stati, i jedinka koja se zatekne ispod trokutića bit će odabrana. Uočite: vjerojatnost

da to bude jedinka i proporcionalna je duljini oboda kružnog isječka koji je predstavlja. Engleski naziv ovakve selekcije (*Roulette Wheel*) upravo dolazi od ovakve konceptualne implementacije.

Programka implementacija ovakve selekcije prikazana je u pseudokodu u nastavku. Funkcija `random(...)` korištena u pseudokodu ovdje prima dva argumenta: donju granicu i gornju granicu te prema uniformnoj distribuciji vraća slučajno generirani decimalni broj iz tog raspona.

Pseudokod 2.4.2 — Implementacija proporcionalne selekcije.

Ulaz: P - populacija
Izlaz: odabrana jedinka

```
suma = 0
za svaku jedinku J iz P: suma += J.dobrota

trokutic = random(0, suma)

podrucje = 0
za svaku jedinku J iz P:
  podrucje += J.dobrota
  ako je trokutic <= podrucje
    vrati J
  kraj ako
kraj za
```

Da bismo proporcionalnu selekciju mogli koristiti, uočite da jedinke moraju imati dobrote koje su nenegativni brojevi. Ovo slijedi iz početne pretpostavke da želimo da je vjerojatnost odabira jedinke proporcionalna njezinoj dobroti. Ako je dobrota -5, što bi to značilo za vjerojatnost?

U slučaju da želimo koristiti proporcionalnu selekciju a sve jedinke nemaju nenegativne dobrote, moguće je prije selekcije napraviti jednostavnu transformaciju: potrebno je pronaći jedinku čija je dobrota minimalna (označimo tu dobrotu s fit_{min} i potom njoj i svim ostalim jedinkama u populaciji dobrotu korigirati na dobrotu jedinke umanjenu za identificiranu minimalnu dobrotu:

$$fit'_i \leftarrow fit_i - fit_{min}.$$

Dakako, nećemo zbog selekcije mijenjati stvarne vrijednosti dobrote jedinkama: u prethodnom izrazu dobrote prema kojima se radi selekcija označili smo s fit'_i i njih će pripremiti sam operator selekcije prije no što krene u vjerojatnosni odabir.

Ovaj operator podložan je još jednom problemu poznatom kao *problem skale*. Ako su vrijednosti funkcije dobrote jako visoke, vjerojatnosti odabira svih jedinki bit će podjednake, neovisno o dobroti samih jedinki. Primjerice, neka su dobrote triju jedinki koje čine populaciju redom 1000001, 1000002 i 1000003. Iako je treća jedinka bolja od prethodne dvije, vjerojatnost odabira svih triju jedinki je 33.3% uz razliku na toliko dalekoj decimali da u praksi to neće imati nikakvog utjecaja. Jedno moguće rješenje ovog problema opet je translacija svih dobrota prema nuli za dobrotu najgore funkcije, ili pak rangiranje jedinki po dobroti i dodjela vjerojatnosti proporcionalno rangu jedinke (najbolji rang, najveća vjerojatnost). Mana tog pristupa je povećana računska složenost jer je jedinke potrebno sortirati.

2.5 Binarna reprezentacija

Kratak osvrt na genetski algoritam koji ovdje dajemo ne bi bio potpun a da ne spomenemo kako je jedan od koraka primjene genetskog algoritma na optimizacijski problem upravo odabir načina

na koji će rješenje ("kromosom") biti predstavljeno. Tek kada je dogovoren način predstavljanja rješenja, definiraju se operatori križanja i mutacije koji djeluju nad tom reprezentacijom. U uvodnom dijelu ovog poglavlja vidjeli smo već dvije reprezentacije: polje cijelih brojeva te polje decimalnih brojeva.

Praktički najranija reprezentacija rješenja u genetskom algoritmu bila je binarna reprezentacija koja ima izravnu analogiju s genetikom: svaki bit je jedan gen, nakupina bitova čini kromosom. Križanje se provodi cijepanjem kromosoma i njihovom rekombinacijom dok mutacije invertiraju vrijednosti bitova.

Ovakav prikaz moguće je koristiti i kada se radi numerička optimizacija nad realnim područjem. Sve što je potrebno jest definirati postupak *dekodiranja* vrijednosti na temelju binarnog zapisa. Evo kako to možemo napraviti.

Pretpostavimo da tražimo optimum funkcije $f(x)$. Moguće vrijednosti varijable x kodirat ćemo binarno uporabom k-bitova. Potrebno je definirati donju i gornju granicu za varijablu x unutar koje radimo pretragu: $x_{min} \leq x \leq x_{max}$, čime je definirana i širina prostora pretrage: $x_{max} - x_{min}$. Primjetimo sada da k-bitni binarni vektor ima 2^k različitih vrijednosti. Svaku od tih vrijednosti znamo pretvoriti u cijeli broj: binarni broj $a_n a_{n-1} \ldots a_1 a_0$ pretvaramo oslanjajući se na bazu:

$$a = \sum_{i=0}^{n} a_i \cdot 2^i.$$

Broj a bit će minimalno 0 a maksimalno $2^k - 1$. Te brojeve iskoristit ćemo kako bismo jednoliko uzorkovali prostor realnih brojeva iz intervala $[x_{min}, x_{max}]$. Broj 0 pri tome ćemo preslikati u x_{min}, broj $2^k - 1$ u x_{max} a sve brojeve između linearno u taj interval. Izraz prema kojem to radimo je:

$$x = \frac{a}{2^k - 1} \cdot (x_{max} - x_{min}) + x_{min}. \tag{2.2}$$

Pogledajmo to na primjeru.

■ **Primjer 2.3** 3-bitnim binarnim kromosom želimo pretražiti područje $[-30, +40]$. Odredimo koje su sve moguće vrijednosti realne varijable koje se mogu dobiti.

Trobitni kromosom ($k = 3$) može poprimiti vrijednosti od 000 do 111, što predstavlja cijele brojeve od 0 do 7. Kako je u našem primjeru $x_{min} = -30$ a $x_{max} = +40$, širina područja je 70, pa uporabom izraza (2.2) slijedi:

$$x = \frac{a}{7} \cdot 70 - 30 = 10 \cdot a - 30$$

što daje redom:

$$\begin{aligned}
&\texttt{000} \rightarrow && a = 0 \rightarrow && x = 10 \cdot 0 - 30 = -30 \\
&\texttt{001} \rightarrow && a = 1 \rightarrow && x = 10 \cdot 1 - 30 = -20 \\
&\texttt{010} \rightarrow && a = 2 \rightarrow && x = 10 \cdot 2 - 30 = -10 \\
&\texttt{011} \rightarrow && a = 3 \rightarrow && x = 10 \cdot 3 - 30 = 0 \\
&\texttt{100} \rightarrow && a = 4 \rightarrow && x = 10 \cdot 4 - 30 = +10 \\
&\texttt{101} \rightarrow && a = 5 \rightarrow && x = 10 \cdot 5 - 30 = +20 \\
&\texttt{110} \rightarrow && a = 6 \rightarrow && x = 10 \cdot 6 - 30 = +30 \\
&\texttt{111} \rightarrow && a = 7 \rightarrow && x = 10 \cdot 7 - 30 = +40
\end{aligned}$$

■

U prethodnom primjeru kromosom je imao samo 3 bita odnosno 8 mogućih vrijednosti. Kako je širina prostora pretrage bila poprilično velika, pretraga je morala raditi velike korake.

Definicija 2.5.1 — Preciznost pretrage. Preciznost pretrage definirat ćemo kao minimalni korak (kvant) kojim se radi pretraga. Kod prethodno opisane binarne reprezentacije preciznost Δ određena je izrazom:

$$\Delta = \frac{1}{2^k - 1} \cdot (x_{max} - x_{min}). \tag{2.3}$$

U slučaju iz prethodnog primjera, preciznost pretraživanja bila je $\frac{1}{2^3-1} \cdot (40 - (-30)) = 10$. Poznavanjem izraza (2.3) lako je uz zadane granice područja i zadanu preciznost doći do potrebnog broja bitova.

Prokomentirajmo ovdje još jedno svojstvo reprezentacije rješenja: sama reprezentacija za optimizacijski algoritam može unijeti lokalni optimum koji u funkciji koja se optimira ne postoji. Primjerice, pretpostavimo da je genetski algoritam došao do sljedećeg rješenja koje je vrlo blizu optimumu: `0111111`. Stvarni optimum postiže se za `1000000`. Da bi iz trenutnog rješenja postupak stigao u optimalno, morao bi doslovno promijeniti sve bitove što je vrlo malo vjerojatan scenarij - stoga će na tom mjestu optimizacijski postupak najčešće zaglaviti. Ovakav problem moguće je riješiti promjenom načina dekodiranja rješenja (primjerice, uporabom Grayevog koda za pretvaranje binarnog uzorka u cijeli broj).

Za kraj, spomenimo da binarna reprezentacija iskazuje još jedno zanimljivo svojstvo: pri djelovanju mutacije, promjena na različitim bitovima rješenje modificira u različitoj mjeri: promjena prvog bita radi promjenu od čak pola širine prostora pretraživanja dok promjena zadnjeg bita radi promjenu od samo jednog kvanta. Kod ove reprezentacije jasno je zašto za mutaciju kažemo da joj je uloga izbacivanje iz lokalnih optimuma - promjenom jednog jedinog bita ovdje mutacija može napraviti dovoljno veliku izmjenu da se rješenje preseli u skroz novo područje i dalje nastavi optimizaciju.

U posljednjih desetak godina izravna uporaba decimalnih brojeva (ili polja decimalnih brojeva) pokazala se praktičnijom u odnosu na binarnu reprezentaciju.

3. Mravlji algoritmi

...

4. Kamo dalje?

...

Bibliografija

Knjige

Članci

Indeks

N

O

P

T