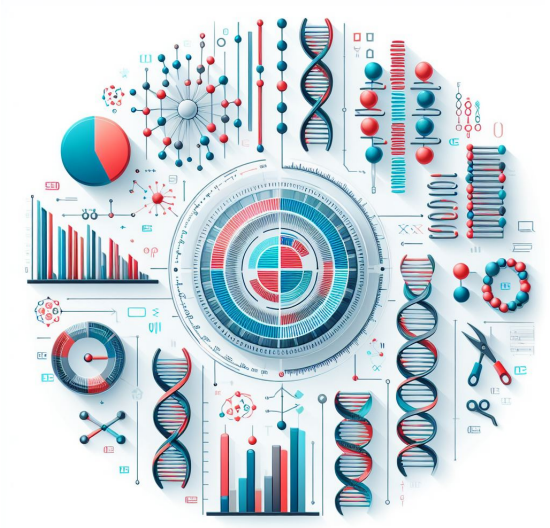
Evolucijsko računarstvo

Željko Kovačević
Domagoj Frank
Ivica Dodig



Odjel računarstva i informatike
Sveučilište Sjever, Koprivnica

Autori:	doc. dr. sc. Željko Kovačević doc. dr. sc. Domagoj Frank izv. prof. dr. sc. Ivica Dodig
Recenzenti:	izv. prof. dr. sc. Marko Horvat, Fakultet elektrotehnike i računarstva u Zagrebu izv. prof. dr. sc. Davor Cafuta, Sveučilište Sjever
Nakladnik:	Sveučilište Sjever
Za nakladnika:	prof. dr. sc. Damir Vusić
Lektura:	Amelia Kovačević, mag. philol. croat.
Tisak:	Sveučilište Sjever, Centar za digitalno nakladništvo
	100 primjeraka
Naklada:	Objavljivanje ovog sveučilišnog udžbenika odobrio je Senat Sveučilišta Sjever na IV. sjednici u akademskoj godini 2025./2026. održanoj 27. studenog 2025. Klasa: 602-04/25-02/16; Ur.broj: 2137-0336-09-25-11
CIP zapis	Dostupan u računalnom katalogu Nacionalne i sveučilišne knjižnice u Zagrebu pod brojem 001283988.
ISBN	978-953-7986-92-6

doc. dr. sc. Željko Kovačević
doc. dr. sc. Domagoj Frank
izv. prof. dr. sc. Ivica Dodig

EVOLUCIJSKO RAČUNARSTVO

Sveučilišni udžbenik Sveučilišta Sjever

Sveučilište Sjever, Koprivnica 2025.

Sadržaj

Predgovor

Ova je knjiga namijenjena čitateljima koji se žele upoznati s osnovama evolucijskih algoritama i razumjeti njihovu primjenu u rješavanju različitih problema optimizacije. Cilj nije samo predstaviti temeljne teorijske koncepte, već ih povezati s praktičnim pristupima koji čitatelju omogućuju neposrednu primjenu stečenog znanja u vlastitim projektima.

Svi praktični primjeri izrađeni su u programskom jeziku C++ korištenjem razvojnog okruženja Microsoft Visual Studio. Za njihovo učinkovito razumijevanje potrebno je barem osnovno poznavanje objektno orijentiranog programiranja. U implementacijama posebna je pažnja posvećena čitljivosti koda, modularnosti i jasnoći prikaza algoritama, s ciljem poticanja čitatelja na eksperimentiranje i samostalno istraživanje.

Knjiga je strukturirana tako da prati prirodan tijek razvoja evolucijskog algoritma - od odabira prikladne reprezentacije rješenja, preko selekcije, križanja i mutacije, do naprednijih pristupa kao što su paralelna evaluacija, lokalno pretraživanje i memetski algoritmi. Svako je poglavlje popraćeno detaljnim objašnjenjima, algoritamskim prikazima i praktičnim primjerima. Dodatni prilozi na kraju knjige omogućuju brz pristup referentnim implementacijama koje se lako mogu prilagoditi konkretnim potrebama.

Knjiga je prvenstveno namijenjena studentima računarstva, informatike i srodnih tehničkih područja, no vjerujemo da će biti korisna i istraživačima, nastavnicima, inženjerima te svima koji žele ovladati osnovama evolucijskih algoritama. Nadamo se da će čitatelje potaknuti ne samo na razumijevanje prikazanih koncepata, već i na njihovu primjenu u stvarnim problemima iz područja struke i znanosti.

Autori,
Sveučilište Sjever, rujan 2025.

POGLAVLJE 1

Uvod u evolucijske algoritme

Evolucijski algoritmi metaheuristički su optimizacijski algoritmi koji simuliraju prirodne evolucijske procese kako bi pronašli najbolje rješenje ili skup rješenja za zadani problem. Koriste se kao razvojni okvir opće namjene za rješavanje širokog raspona problema bez oslanjanja na specifične karakteristike bilo kojeg problema. Njihovu primjenu najčešće vidimo u situacijama gdje je korištenje tradicionalnih metoda i pristupa nepraktično ili računalno preskupo. Primjerice, prilikom pretraživanja ogromnih prostora svih mogućih rješenja loš izbor bilo bi korištenje pristupa *grube sile* (engl. *brute force*) gdje bi se pregledavalo svako moguće rješenje. Ovisno o problemu i s obzirom na trajanje procesa evaluacije pojedinog rješenja, taj pristup obično niti nije opcija jer takva potraga može trajati tisućama godina čak i na najbržim računalima. U ovakvim slučajevima korištenje evolucijskih algoritama znatno je učinkovitiji pristup koji dovodi do puno brže pretrage. Primjerice, u [1] za jedan od najjednostavnijih problema iz područja semantičkog zaključivanja izračunato je da bi pretraga rješenja korištenjem pristupa grube sile u najgorem slučaju trajala 9003 godine, dok je rješenje za isti problem korištenjem genetskog programiranja najčešće bilo pronađeno u svega tridesetak minuta.

Primjena evolucijskih algoritama je široka. Primjerice, koriste se u dizajniranju i projektiranju elektroničkih sklopova [2, 3], obradi slike [4], računalnom prepoznavanju lica u kriminalistici [5] optimizaciji portfelja u ekonomiji i financijama [6], filtriranju i obradi signala [7], raznim aplikacijama za izradu rasporeda i planiranje poslova [8, 9], optimizaciji molekularne strukture u kemiji [10], kriptoanalizi [11], razvoju računalnih igara (izrada likova, okruženja, optimizacija težine igre...) [12] itd.

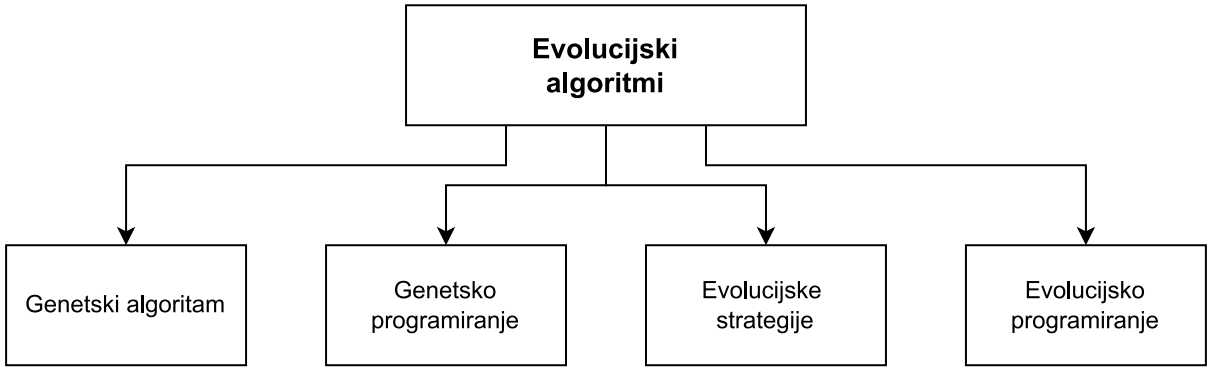
Važno je napomenuti da evolucijski algoritmi ne moraju uvijek nužno pronaći optimalno (potpuno točno) rješenje. Nekada optimalno rješenje možda i ne postoji, a u drugim slučajevima potrebno je algoritam izvršiti više puta kako bi se do njega došlo. Naime, evolucijski su algoritmi stohastički, tj. uključuju slučajnost ili elemente vjerojatnosti u svom radu te zbog toga svako izvršavanje algoritma može imati drukčiji rezultat. Tako u prvom izvršavanju algoritma optimalno rješenje ne mora uopće biti pronađeno. U drugom izvršavanju može biti pronađeno optimalno rješenje, a ovisno o problemu, u trećem

izvršavanju neko drugo optimalno rješenje. Slično vrijedi i za vrijeme pretrage, gdje u jednom izvršavanju algoritma rješenje može biti pronađeno puno brže ili sporije u odnosu na ostale pokušaje. Međutim, čak i u slučajevima kada optimalno rješenje nije pronađeno, evolucijski će algoritmi vrlo često dati "dovoljno dobro" optimizirano rješenje koje se također u velikom broju slučajeva može iskoristiti.

1.1 Tipovi evolucijskih algoritama

Evolucijski algoritmi podijeljeni su na četiri glavna tipa prikazana na Slici 1.1. Genetski algoritam (GA) predstavio je Holland početkom 1970-ih godina [13, 14], te on predstavlja vjerojatno najpoznatiji oblik evolucijskog algoritma koji koristi fiksnu duljinu genetskog niza te mehanizme selekcije, križanja i mutacije kako bi kroz generacije poboljšavao kvalitetu populacije potencijalnih rješenja. Ako se u njemu dodatno koristi tehnika lokalnog pretraživanja, genetski algoritam postaje memetski algoritam [15].

Genetski algoritmi često se koriste u problemima kombinatorne optimizacije, poput generiranja optimalnog rasporeda sati u školama ili fakultetima. Cilj je minimizirati konflikte (primjerice, dva predavanja istog profesora u isto vrijeme) uzimajući u obzir ograničenja poput dostupnosti dvorana, broja sati pojedinog kolegija i zahtjeva profesora. GA najčešće koristi binarnu ili permutacijsku reprezentaciju, gdje je svako rješenje niz koji predstavlja jedan mogući raspored.



Slika 1.1: Tipovi evolucijskih algoritama.

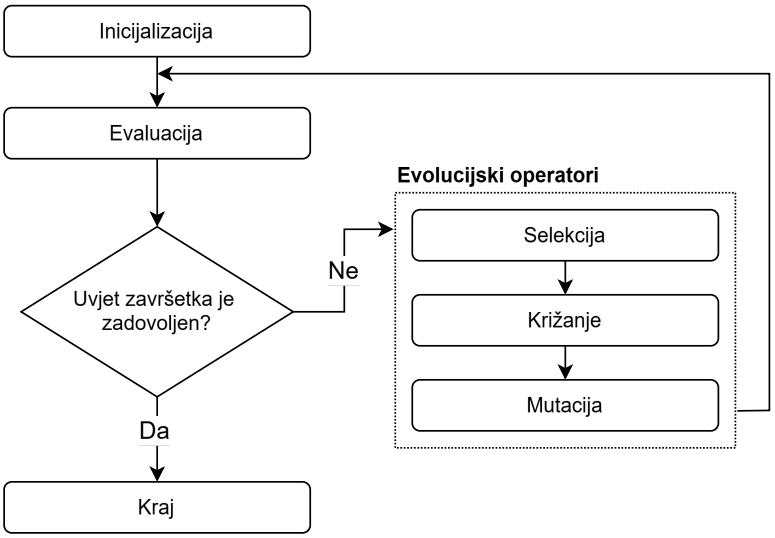
Posebni oblik genetskog algoritma koji se bavi evolucijom računalnih programa i struktura naziva se genetsko programiranje. Predstavio ga je Koza početkom 1990-ih godina [16, 17, 18]. Za razliku od genetskog algoritma gdje se koristi genetski niz fiksne duljine, u genetskom se programiranju vrlo često koriste strukture stabala ili grafova, što omogućuje evoluciju programa promjenjive duljine. Često se koristi za automatsku generaciju matematičkih izraza kada je potrebno pronaći funkciju koja najbolje aproksimira dane podatke. Svako rješenje prikazano je kao stablo, gdje unutarnji čvorovi predstavljaju matematičke operacije (+, -, *, /), a listovi varijable i konstante. Evolucijski proces uključuje selekciju, križanje i mutaciju stabala kako bi se poboljšala točnost aproksimacije. Ova metoda primjenjuje se u znanosti, ekonomiji i bioinformatici za modeliranje odnosa među podacima bez unaprijed definiranog oblika funkcije [19].

Evolucijsko programiranje (EP) predstavili su 1960-ih godina Fogel, Owens i Walsh [20]. Vrlo je slično genetskom programiranju, ali osim stablima i grafovima omogućuje širi spektar reprezentacija rješenja, uključujući i one temeljene na realnim vektorima, matricama ili drugim strukturama. Za razliku od genetskih algoritama, evolucijsko programiranje ne koristi križanje, već se oslanja isključivo na mutaciju, pri čemu koristi Gaussovu mutaciju [21] za prilagodbu numeričkih parametara. Selekcija se obično provodi turnirskim natjecanjem gdje se jedinke međusobno uspoređuju i najbolje opstaju. EP najčešće se primjenjuje u problemima predviđanja vremenskih serija i optimizaciji strategija odlučivanja gdje je cilj evolucijski poboljšavati modele koji predviđaju buduće događaje, primjerice, u financijama ili meteorologiji.

S druge strane, evolucijske strategije (ES) sličnije su genetskom algoritmu. Osmislili su ih Rechenberg i Schwefel [22, 23] te, poput evolucijskog programiranja, ne koriste operator križanja, već se oslanjaju isključivo na mutaciju. Međutim, ključna razlika između EP i ES je u tome što evolucijske strategije koriste samoadaptirajuće mutacije, što znači da algoritam može dinamički prilagođavati intenzitet mutacija tijekom evolucije. Također, selekcija u ES-u se obično provodi prema (μ, λ) ili $(\mu + \lambda)$ strategiji [23], gdje se potomci natječu za mjesto u sljedećoj generaciji, a roditelji mogu (ili ne moraju) preživjeti. Ova svojstva čine ES osobito pogodnima za optimizaciju realnih parametara u inženjerskim problemima, primjerice, pri dizajnu aerodinamičkih struktura ili optimizaciji industrijskih procesa.

1.2 Opći oblik evolucijskog algoritma

Evolucijski algoritmi postoje u mnogo inačica te se implementiraju na razne načine. Nekada implementacija ovisi o tipu algoritma, korištenim mehanizmima (selekcija, križanje i mutacija) ili o samom problemu za koji pokušavamo prilagoditi algoritam. Međutim, sve implementacije temeljno pokušavaju slijediti ista načela vidljiva iz općeg oblika evolucijskog algoritma (Slika 1.2).



Slika 1.2: Opći oblik evolucijskog algoritma.

1.2.1 Inicijalizacija

Evolucijski algoritam započinje inicijalizacijom tijekom koje se generira početna populacija jedinki (rješenja) koja služi kao početna točka pretrage za optimalnim rješenjem. Inicijalna populacija može biti generirana po nekom načelu, ali najčešće je riječ o generiranju slučajne populacije. Uobičajeno je da evolucijski algoritmi, ovisno o problemu, pretražuju velike prostore svih mogućih rješenja što dovodi do vrlo male (gotovo zanemarive) vjerojatnosti da se točno rješenje nalazi upravo u inicijalnoj populaciji. Štoviše, inicijalna populacija najčešće sadrži jedinke niske kvalitete, s idejom da će se kroz generacije njihova kvaliteta poboljšavati sve dok se ne dođe do optimalnog rješenja. Međutim, jedna od ključnih prednosti slučajno generirane populacije raznolikost je generiranih jedinki. Bez dovoljne raznolikosti evolucijski algoritmi obično ne rade učinkovito i teško pronalaze optimalno rješenje.

1.2.2 Veličina i tip populacije

Kako bismo osigurali dovoljno veliku raznolikost, posebice kod rješavanja kompleksnijih problema, koristimo parametar $popSize$ (engl. $population\ size$) kojim se definira veličina populacije, odnosno broj jedinki. Populacija najčešće ima konstantan (uvijek isti) broj pojedinaca, no to ne mora uvijek biti slučaj. U pravilu, veća populacija (raznolikost) često rezultira dužom, ali uspješnijom pretragom za optimalnim rješenjem, dok manja populacija može polučiti bržu pretragu, ali najčešće s manjim uspjehom zbog prerane konvergencije algoritma u nekom lokalnom prostoru gdje se ne nalazi optimalno rješenje [24]. Ne postoji konkretno pravilo kojim se može odrediti najbolja veličina populacije, pa je u tu svrhu potrebno eksperimentiranje s različitim veličinama kako bi se pronašla ona koja najbolje odgovara konkretnom problemu.

Populacije mogu biti generacijske (engl. $generational$) i stacionarne (engl. $steady-state$). Generacijski upravljana populacija ona je gdje evolucijski mehanizmi generiraju novu populaciju. Ona u sljedećem koraku postaje trenutna populacija koja će opet, nakon djelovanja evolucijskih operatora (generacije), rezultirati novom populacijom. U usporedbi s njom, u stacionarnoj populaciji mijenja se tek minimalan broj jedinki (ponekad samo jedan). Zbog sporijih promjena u populaciji postizanje optimalnog rješenja može trajati duže, ali uz manji rizik od gubitka genetske raznolikosti.

1.2.3 Evaluacija i vrijednost dobrote

Nakon kreiranja inicijalne populacije jedinki, pristupamo njihovoj evaluaciji. Zadaća ovog koraka algoritma jest što preciznije odrediti kvalitetu pojedine jedinke korištenjem vrijednosti $dobrote$. Kvalitetna definicija dobrote jedinke za zadani problem od iznimne je važnosti za uspješan rad evolucijskog algoritma jer omogućuje određivanje najkvalitetnijih jedinki u populaciji i njihovo međusobno uspoređivanje.

Dobrota jedinke ovisi i o tipu optimizacije, a može biti jednokriterijska (kvaliteta jedinke ovisi samo o jednom parametru) ili višekriterijska (kvaliteta jedinke ovisi o više parametara). Primjerice, optimizacija duljine žice u električnom krugu gdje je cilj minimizirati

duljinu žice između dviju točaka u električnom krugu primjer je jednokriterijske optimizacije jer ovisi samo o jednom parametru (duljini žice). S druge strane, razvoj računalne aplikacije gdje se u minimalnom vremenu i sa što manje resursa pokušava razviti što kvalitetnija aplikacija predstavlja višekriterijski optimizacijski problem.

Pomoću vrijednosti dobrote zadani problem obično se pokušava predstaviti kao problem minimizacije ili maksimizacije, odnosno gdje je kvalitetnija ona jedinka s manjom vrijednošću dobrote (primjerice, problem generiranja dobrog rasporeda sati nastave gdje je bolji onaj raspored koji ima manje stavki s greškama), te obrnuto u problemu maksimizacije, tj. gdje je bolji onaj raspored koji ima više stavki bez grešaka. Iako se kvaliteta generiranog rasporeda u ovom slučaju može predstaviti dvojako, vjerojatno bi prikladniji bio pristup minimizacije. Naime, tada je vrijednost dobrote optimalnog rješenja uvijek 0, dok bi u slučaju maksimizacije ona ovisila o broju stavki u rasporedu.

Važno je napomenuti da je upravo proces evaluacije najčešće i najsporiji dio evolucijskog algoritma. Ovisno o problemu, proces evaluacije jedinke može trajati nekoliko sekundi, minuta ili čak i duže vrijeme. Da bi se smanjilo vrijeme evaluacije i izbjegla ponovna evaluacija istih jedinki, moguće je korištenje raznih tehnika i pristupa, poput paralelne (višedretvene) evaluacije, tablica raspršivanja, aproksimacije vrijednosti dobrote itd.

1.2.4 Petlja evolucijskih procesa

S obzirom na veličinu prostora za pretraživanje, nakon evaluacije inicijalne populacije najčešće nećemo pronaći optimalno rješenje. Tada se ulazi u petlju ponavljanja selekcije, križanja i mutacije sve dok ne bude zadovoljen uvjet završetka algoritma. Navedeni uvjet najčešće ima dva ključna dijela: ili smo pronašli optimalno rješenje ili je pretraga za optimalnim rješenjem premašila maksimalno predviđeni broj generacija.

Naime, ukoliko optimalno rješenje nije moguće pronaći zbog konvergencije algoritma u nekom lokalnom prostoru (lokalni optimum), algoritam treba zaustaviti nakon maksimalnog broja predviđenih generacija kako bi se spriječila beskonačna petlja. Štoviše, u takvim situacijama gdje se eksperimentalno utvrdi da optimalno rješenje vjerojatno ne postoji, može se promijeniti i prvi dio uvjeta tako da algoritam završava kada se pronađe rješenje koje zadovoljava određeni stupanj optimizacije. Zbog toga je važno razlikovati pojam populacije, koja predstavlja skup jedinki, od pojma generacije, koja predstavlja broj iteracija evolucijskih procesa (selekcija, križanja, mutacija itd.) koje su do sada odrađene u potrazi za optimalnim rješenjem.

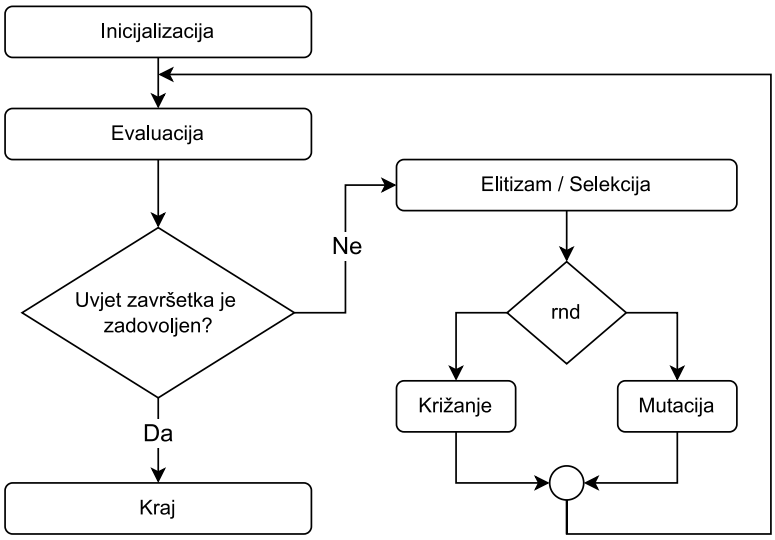
1.2.5 Pseudokod evolucijskog algoritma

Algoritam 1.1 prikazuje osnovne korake evolucijskog algoritma zapisane u obliku pseudokoda. Nakon stvaranja i evaluacije početne populacije, ulazi se u petlju koja počinje selekcijom. Na temelju odabranog mehanizma selekcije stvara se nova populacija P' koja sadrži samo odabrane jedinke iz trenutne populacije koje mogu sudjelovati u procesu križanja. Kao rezultat križanja jedinki iz populacije P' dobiva se populacija P'', a rezultat sljedećeg koraka (mutacije) nad populacijom P'' postaje temelj sljedeće generacije.

Algoritam 1.1 Opći oblik evolucijskog algoritma.

1: $t \leftarrow 0$
2: $P(t) \leftarrow$ INICIJALIZACIJA
3: EVALUACIJA($P(t)$)
4: **while** uvjet završetka nije zadovoljen **do**
5: $P'(t) \leftarrow$ SELEKCIJA($P(t)$)
6: $P''(t) \leftarrow$ KRIŽANJE($P'(t)$)
7: $P(t+1) \leftarrow$ MUTACIJA($P''(t)$)
8: EVALUACIJA($P(t+1)$)
9: $t \leftarrow t+1$
10: **end while**

Kao što je već prethodno napisano, evolucijski algoritmi mogu postojati u mnogo različitih inačica. Još jedna od njih prikazana je Slikom 1.3, gdje se ovisno o elementu vjerojatnosti izvršava samo križanje ili samo mutacija. Tako, primjerice, gen nove jedinke može biti naslijeđen od jednog od roditelja ili može nastati kao slučajna mutacija gena.



Slika 1.3: Nestandardni oblik evolucijskog algoritma.

Pojedini dijelovi evolucijskog algoritma također mogu postojati u više inačica. Tako će u nastavku ovog teksta (Poglavlje 2) biti opisano nekoliko različitih strategija selekcije, križanja i mutacije.

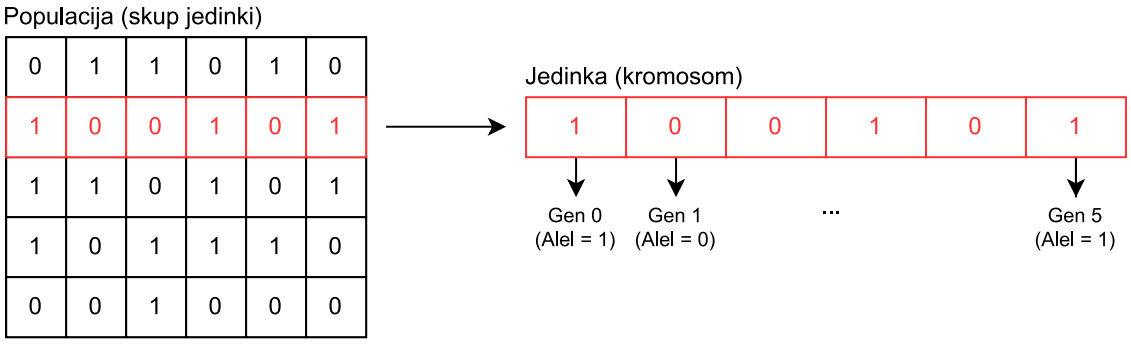
1.3 Reprezentacija rješenja

Pri rješavanju optimizacijskih problema evolucijskim algoritmima od velike važnosti je reprezentacija (način prikaza) rješenja, odnosno jedinki. O reprezentaciji ovisi implementacija pojedinih evolucijskih mehanizama, pa tako i performanse evolucijskih algoritama. Postoji mnogo mogućih načina reprezentacije rješenja, a oni najčešće ovise o tipu korištenih gena, specifičnostima problema i vrsti korištenog evolucijskog algoritma.

Primjerice, postoje binarna, cjelobrojna, znakovna, realna, permutacijska, reprezentacija u obliku stabla i mnoge druge.

1.3.1 Binarna reprezentacija

Binarna reprezentacija jedna je od najčešćih i najjednostavnijih reprezentacija rješenja u genetskim algoritmima. Ona koristi nizove bitova (0 i 1) kako bi predstavila jedinke u populaciji. Svaki bit u nizu predstavlja jedan gen, a svaka vrijednost (varijanta) gena naziva se Alel (Slika 1.4).



Slika 1.4: Binarna reprezentacija rješenja.

Za demonstraciju, prikazana binarna reprezentacija mogla bi se izravno primijeniti na problem pronalaska binarnog ekvivalenta nekog prirodnog broja N korištenjem genetskog algoritma. Svaka jedinka u populaciji sastojala bi se od određenog broja gena, gdje bi vrijednost pojedinog gena (alel) te jedinke mogla biti 0 ili 1, pri čemu 0 predstavlja binarnu nulu, a 1 binarnu jedinicu. Tako bi izdvojena jedinka sa Slike 1.4 (100101) predstavljala broj 37. Ukoliko navedeni problem predstavimo kao problem minimizacije, gdje je kvalitetnije ono rješenje koje je bliže traženom broju N, onda bi dobrota jedinke bila definirana kao $N - jedinka.BinToInt()$.

Predmet	Vrijednost	Težina
A	3 €	1 kg
B	8 €	5 kg
C	5 €	2 kg
D	4 €	4 kg

Slika 1.5: Problem maksimiziranja ruksaka.

Binarnu reprezentaciju može se upotrijebiti i u rješavanju klasičnog problema ruksaka gdje je cilj maksimizirati ukupnu vrijednost predmeta koji se stavljaju u ruksak, uz unaprijed definirano ograničenje kapaciteta ruksaka. Pri tome se u ruksak stavljaju samo cjeloviti predmeti, odnosno nije dopušteno uzimanje dijelova predmeta. Sada se neka od rješenja mogu prikazati na načine prikazane u Tablici 1.1.

Tablica 1.1: Primjeri binarnih reprezentacija rješenja za problem ruksaka.

Reprezentacija	Opis	Vrijednost	Težina
0110	A(0), B(1), C(1), D(0)	13 €	7 kg
1101	A(1), B(1), C(0), D(1)	15 €	10 kg
0101	A(0), B(1), C(0), D(1)	12 €	9 kg
1110	A(1), B(1), C(1), D(0)	16 €	8 kg

Ovaj problem često se naziva problemom "0/1 ruksaka" jer se za svaki predmet odlučuje hoće li se staviti u ruksak (1) ili ne (0). Tako u prvom primjeru rješenja (0110) u ruksak stavljamo samo predmete B i C, čija vrijednost iznosi 13 €, a težina 7 kg. Od prikazanih, najbolje rješenje je 1110 (u ruksaku su predmeti A, B i C) gdje je vrijednost svih predmeta 16 €, uz ukupnu težinu od 8 kg.

1.3.2 Grayev kod

Binarna reprezentacija, iako vrlo česta u genetskim algoritmima, također ima svoje nedostatke. Jedan od njih je velika Hammingova udaljenost [25] binarnih reprezentacija brojeva koji su inače u decimalnom sustavu jako blizu. Primjerice, Hammingova udaljenost između brojeva 127 (binarno: 01111111) i 128 (binarno: 10000000) iznosi 8 jer se ti brojevi razlikuju u svih 8 bitova. U kontekstu genetskih algoritama, ako je zadatak pronaći binarnu reprezentaciju broja 128 postoji visoka vjerojatnost da će algoritam završiti u području lokalnog optimuma ukoliko dosegne vrijednost 127, budući da će u tom slučaju biti potrebno promijeniti svih 8 bitova kako bi se decimalna vrijednost povećala za 1. Da bismo spriječili ovaj problem, binarni kod možemo zamijeniti Grayevim kodom [26] kod kojeg je Hammingova udaljenost za susjedne brojeve u decimalnom sustavu uvijek 1.

Algoritam 1.2 Pretvorba iz binarnog koda u Grayev kod.

```
1: function BINARNIUGRAY(binarni)
2:     n ← length(binarni)
3:     gray ← array[n]
4:     gray[0] ← binarni[0]
5:     for i ← 1 to n − 1 do
6:         gray[i] ← binarni[i − 1] XOR binarni[i]
7:     end for
8:     return gray
9: end function
```

Za pretvorbu binarnog koda u Grayev kod koristimo Algoritam 1.2. Nulti bit binarnog koda postaje nulti bit Grayeva koda, dok ostali bitovi Grayeva koda nastaju kao rezultat XOR (isključivo ili) operacije susjednih bitova binarnog koda. Ovaj postupak možemo ilustrirati na primjeru pretvorbe binarnog broja 1011 (decimalno 11) u Grayev kod.

1. **Kopiranje prvog bita:** Prvi bit Grayeva koda je identičan prvom bitu binarnog koda. Stoga, ako je prvi bit našeg binarnog broja 1, to će biti i prvi bit Grayeva koda.

Binarni kod: `1011`
Grayev kod: `1---`

2. **XOR susjednih bitova:** Svaki sljedeći bit u Grayevom kodu dobiva se primjenom XOR operacije na susjedne bitove u binarnom kodu. Podsjetimo, XOR operacija rezultira `1` ako su bitovi različiti, i `0` ako su isti.

   - XOR prvog i drugog bita binarnog koda (`1 XOR 0`) daje `1`.
   - XOR drugog i trećeg bita (`0 XOR 1`) daje `1`.
   - XOR trećeg i četvrtog bita (`1 XOR 1`) daje `0`.

Na kraju postupka binarni broj `1011` postaje `1110` u Grayevu kodu. Usporedbom brojeva 127 i 128 u binarnom i Grayevom kodu (Tablica 1.2) uočavamo da Hammingova udaljenost u Grayevom kodu iznosi samo 1, što je očekivano. Ova značajka Grayeva koda smanjuje rizik od zapinjanja u lokalnim optimumima, posebice u situacijama poput prijelaza s broja 127 na 128, gdje je sada potrebno promijeniti samo jedan bit. Nasuprot tome, primjena originalne binarne reprezentacije u istom slučaju zahtijeva promjenu svih osam bitova, što znatno komplicira proces pronalaska optimalnog rješenja.

Tablica 1.2: Prikaz decimalnih brojeva u obliku binarnog i Grayeva koda.

Decimalno	**Binarni kod**	**Grayev kod**
127	01111111	01000000
128	10000000	11000000

Grayev kod moguće je pretvoriti nazad u originalni binarni kod korištenjem Algoritma 1.3. Za primjer demonstrirajmo pretvorbu prethodno dobivenog Grayeva koda 1110 nazad u binarni kod.

Algoritam 1.3 Pretvorba iz Grayeva koda u binarni kod.

```
1: function GRAYUBINARNI(gray)
2:     n ← length(gray)
3:     binarni ← array[n]
4:     binarni[0] ← gray[0]
5:     for i ← 1 to n − 1 do
6:         binarni[i] ← binarni[i − 1] XOR gray[i]
7:     end for
8:     return binarni
9: end function
```

Slijedimo ove korake:

1. Prvi bit binarnog koda isti je kao prvi bit Grayeva koda. Dakle, prvi bit je `1`.

   Grayev kod: `1110`
   Binarni kod: `1___`

2. Za svaki sljedeći bit izvršimo XOR operaciju između prethodno dobivenog bita u binarnom kodu i sljedećeg bita u Grayevom kodu.

- Drugi bit Grayeva koda je `1`. XOR operacija između `1` (prvi bit binarnog koda) i `1` (drugi bit Grayeva koda) daje `0`.
- Treći bit Grayeva koda je `1`. XOR operacija između `0` (drugi bit binarnog koda) i `1` (treći bit Grayeva koda) daje `1`.
- Četvrti bit Grayeva koda je `0`. XOR operacija između `1` (treći bit binarnog koda) i `0` (četvrti bit Grayeva koda) daje `1`.

Kao rezultat dobivamo originalni binarni kod: `1011`.

1.3.3 Cjelobrojna, znakovna i realna reprezentacija

Dugo vremena binarna reprezentacija bila je dominantna u genetskim algoritmima, ali kasnije su istraživači uvidjeli da različite vrste problema zahtijevaju različite reprezentacije kako bi se postigle što bolje performanse. Stoga će u nastavku biti prikazani primjeri optimizacijskih problema gdje su prikladnije reprezentacije temeljene na drugim tipovima podataka, pristupima i strukturama.

Primjerice, pretpostavimo da imamo problem raspoređivanja gdje treba dodijeliti N zadataka (označenih brojevima od 1 do N) na M radnika (označenih brojevima od 1 do M) tako da su ispunjeni neki zadani uvjeti. Jedan od načina da predstavimo rješenje ovog problema korištenje je cjelobrojne reprezentacije gdje svaki element u nizu označava kojem je radniku dodijeljen određeni zadatak. Primjer rješenja mogao bi biti niz [2, 1, 1, 3, 2]. Ovaj niz znači da je zadatak 1 dodijeljen radniku 2, zadatak 2 radniku 1, zadatak 3 također radniku 1, zadatak 4 radniku 3, i zadatak 5 radniku 2. U ovom primjeru brojevi u nizu predstavljaju identifikatore radnika, a redoslijed brojeva u nizu odgovara redoslijedu zadataka. Budući da svakom radniku može biti dodijeljeno više zadataka, isti broj (identifikator radnika) može se pojaviti više puta u nizu.

Na sličnim problemima možemo demonstrirati i znakovnu reprezentaciju gdje je moguće koristiti set pojedinačnih znakova ili set znakovnih nizova. Za demonstraciju takvog tipa rješenja možemo zamisliti problem optimizacije menija. Meni se sastoji od nekoliko jela, a svako jelo pripada određenoj kategoriji (predjelo, glavno jelo, desert). Ukoliko je cilj kreirati meni koji se sastoji od jednog predjela, jednog glavnog jela i jednog deserta, uzimajući u obzir različite kriterije poput popularnosti jela, troškova sastojaka, i vremena pripreme, rješenje se može reprezentirati kao niz znakova gdje svaki znak predstavlja odabrano jelo iz jedne od kategorija (Tablica 1.3).

Tablica 1.3: Problem optimizacije menija.

Kategorija	**Opcije**
Predjela	Salata (S), Juha (J)
Glavna jela	Odrezak (O), Riba (R)
Deserti	Kolač (K), Sladoled (L)

Primjerice, rješenje (znakovni niz) "JOL" bi značilo da je meni sastavljen od "Juha" kao predjelo, "Odrezak" kao glavno jelo, i "Sladoled" kao desert. Isto rješenje bi moglo biti

predstavljeno i kao niz ["Juha", "Odrezak", "Sladoled"].

U sljedećem primjeru pretpostavimo da imamo tvrtku koja ima tri projekta: A, B i C. Svaki projekt ima svoj očekivani prihod i trošak (Tablica 1.4), a cilj je pronaći optimalnu raspodjelu resursa kako bi se maksimizirala ukupna zarada tvrtke. U ovom slučaju koristit ćemo realnu reprezentaciju rješenja kako bismo predstavili udio resursa dodijeljen svakom projektu. Svaki gen u jedinki predstavlja postotni udio dodijeljenih resursa, a ukupna suma gena u jedinki mora biti jednaka 1. Primjeri rješenja prikazani su u Tablici 1.5.

Tablica 1.4: Problem optimalne raspodjele resursa na projektima.

Projekt	**Prihod**	**Trošak**
A	100.000 €	40.000 €
B	150.000 €	60.000 €
C	120.000 €	50.000 €

Tablica 1.5: Primjeri rješenja za problem optimalne raspodjele resursa na projektima.

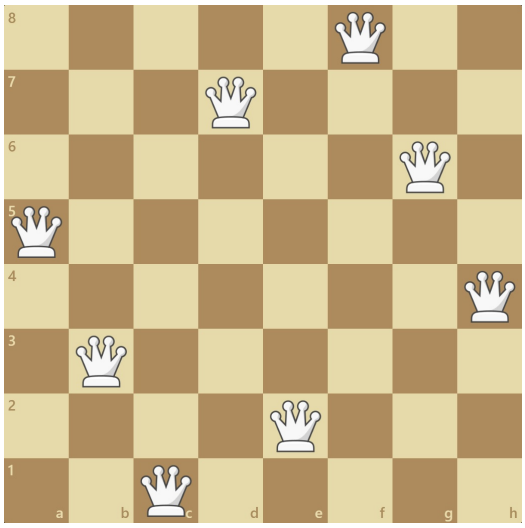
Jedinka	**Opis**
(1.0, 0.0, 0.0)	Svi resursi dodijeljeni su projektu A.
(0.3, 0.4, 0.3)	Resursi su podjednako raspoređeni između sva tri projekta.
(0.1, 0.5, 0.4)	Najveći udio resursa dodijeljen je projektu B.

U ovom primjeru problema također je moguće koristiti cjelobrojnu reprezentaciju rješenja ako gene želimo prikazati kao postotke u rasponu [0, 100].

1.3.4 Permutacijska reprezentacija

Permutacijska reprezentacija rješenja vrlo se često može pogrešno zamijeniti s cjelobrojnom, znakovnom ili realnom reprezentacijom. Ipak, za razliku od njih, u permutacijskoj reprezentaciji rješenje se predstavlja nizom u kojem ne postoje duplikati. Primjerice, u prethodno objašnjenim primjerima raspoređivanja zadataka među radnicima te u primjeru optimizacije raspodjele resursa može se dogoditi da isti radnik bude zadužen za dva posla (njegov ID se dva puta nalazi u nizu), odnosno da više projekata ima isti udio dodijeljenih resursa. Međutim, u određenim tipovima problema, svaki je element niza (gen) različit te je cilj za svakog od njih pronaći što bolje mjesto u nizu putem permutacija. Rješenja u takvim tipovima problema predstavljamo permutacijskom reprezentacijom.
Jedan od takvih problema klasični je problem osam kraljica u kojemu je cilj pronaći položaj svih osam kraljica na šahovskoj ploči tako da nijedna kraljica ne napada neku drugu [27]. Slika 1.6 prikazuje jedno od mogućih rješenja, dok ukupan broj točnih rješenja na ploči dimenzija 8×8 iznosi 92.

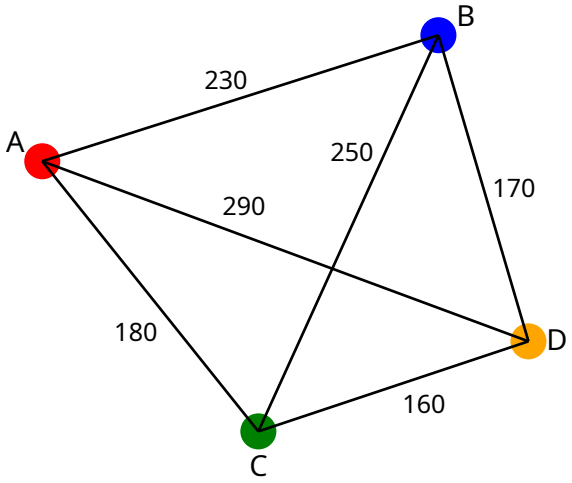
Da bismo predstavili neko od mogućih rješenja ovog problema korištenjem permutacijske reprezentacije, za svaku od kraljica možemo pretpostaviti da se nalazi u zasebnom stupcu ($A - H$). Zatim, korištenjem cijelih brojeva u rješenju za svaku od kraljica navodimo



Slika 1.6: Problem osam kraljica (jedno od rješenja).

broj retka u kojem se ona nalazi. Tako bi rješenje prikazano na Slici 1.6 bilo predstavljeno na sljedeći način: $(5, 3, 1, 7, 2, 8, 6, 4)$. Prva kraljica smještena je na polje $A5$, druga na polje $B3$, treća na polje $C1$, itd. Rješenje je moguće predstaviti i na obrnut način, odnosno s pretpostavkom da se svaka od kraljica nalazi u zasebnom retku $(1 - 8)$. U ovom slučaju, rješenje prikazano na Slici 1.6 bilo bi predstavljeno kao sljedeći niz: (C, E, B, H, A, G, D, F). Prva kraljica smještena je na polje $C1$, druga na polje $E2$, treća na polje $B3$ itd.

Još jedan klasični problem u kojemu je moguće koristiti permutacijsku reprezentaciju rješenja problem je trgovačkog putnika (Slika 1.7). Ovaj problem svodi se na pronalazak najkraće rute među zadanim točkama (primjerice, gradovima), pri čemu svaku od točki treba posjetiti samo jedanput. Na kraju, potrebno je vratiti se u početnu točku.



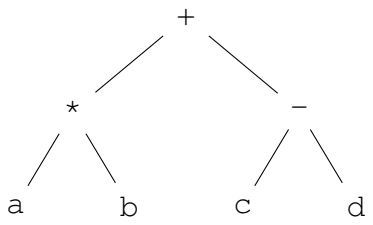
Slika 1.7: Primjer problema trgovačkog putnika (četiri grada).

Ukoliko obilazak počinje od točke A, pojedina rješenja možemo predstaviti nizovima poput (A, B, C, D), (A, B, D, C), (A, D, B, C) itd. Budući da permutacijska reprezentacija

rješenja ne sadrži duplikate, u svakom od ovih rješenja točka A podrazumijeva se kao zadnja točka u nizu.

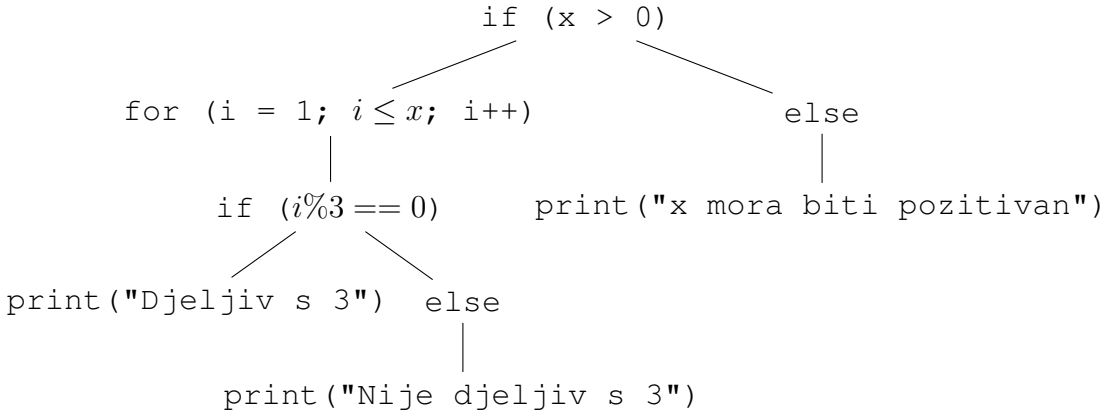
1.3.5 Reprezentacija stablom

Za razliku od prethodnih reprezentacija koje se najčešće koriste u genetskim algoritmima, reprezentacija rješenja stablom uobičajeno se koristi u genetskom programiranju (GP). U ovom pristupu, razvijeni matematički izrazi i programi prikazuju se kao stabla gdje unutarnji čvorovi predstavljaju operatore (kao što su +, -, *, /) ili funkcije, dok listovi stabla (terminalni čvorovi) predstavljaju konstante ili varijable. Evaluacija takvog stabla obično se vrši rekurzivnim obilaskom ($preorder$, $inorder$ ili $postorder$) ovisno o specifičnoj primjeni i vrsti operacija koje se izvode na stablu.



Slika 1.8: Primjer stablaste reprezentacije rješenja (matematičkog izraza) u GP-u.

Na Slici 1.8 prikazano je jedno GP rješenje razvijeno kao matematički izraz u obliku stabla. U ovom slučaju najprikladnije bi bilo obići stablo koristeći metodu $inorder$. Tom metodom prvo se posjećuje lijevo podstablo, zatim korijenski čvor i na kraju desno podstablo. Kao rezultat takvog prelaska, za navedeno stablo dobivamo izraz $(a * b) + (c - d)$.



Slika 1.9: Primjer stablaste reprezentacije programa u GP-u s uvjetom i petljom.

Slično, na Slici 1.9, razvijeno stablo (GP rješenje) predstavlja primjer C programa koje bi najprikladnije bilo obići korištenjem metode $preorder$. Ova metoda prvo posjećuje korijenski čvor, a nakon toga lijevo i desno podstablo gdje se rekurzivno ponavlja isti postupak. U navedenom primjeru stabla, $preorder$ prelazak će rezultirati programom prikazanim u Primjeru 1.1.

Primjer 1.1: Reprezentacija GP rješenja sa Slike 1.9 u programskom jeziku C, dobiveno $preorder$ prelaskom stabla.

```
if (x >= 10) {
    for (int i = 0; i < x; i++) {
        if (i % 2 == 0)
            printf("Paran");
        else
            printf("Neparan");
    }
} else
      printf("Min: 10");
```

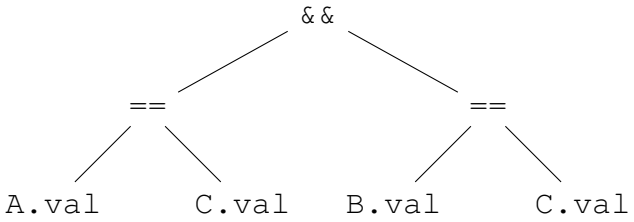
Iako se u GP-u stabla najčešće koriste za reprezentaciju rješenja (jedinki) gdje geni predstavljaju čvorove tih stabala (operatore ili funkcije), ponekad se sami geni predstavljaju stablima. Takav je pristup korišten u [1], gdje pojedini geni GP rješenja predstavljaju zasebne matematičke izraze (Primjer 1.2).

Primjer 1.2: Reprezentacija GP rješenja gdje su geni predstavljeni stablima.

```
language AnBnCn {
  lexicon {
    Comment /\*[^\*]+\*/
    TokenA a
    TokenB b
    TokenC c
    ignore [\ \0x0D\0x0A\0x09]+ | #Comment
  }
  attributes int *.val;
             boolean *.ok;
  rule S {
    S ::= A B C compute {
      S.ok = (A.val == C.val) && (B.val == C.val); // gen
    };
  }
  rule A {
    A ::= a A compute {
     A.val = (A[1].val + 1) + (1 + 1); // gen
    };
    A ::= a compute {
      A.val = 1; // gen
    };
  }
  rule B {
    B ::= b B compute {
     B.val = (1 + B[1].val) + (1 + 1); // gen
    };
    B ::= b compute {
      B.val = 1; // gen
    };
  }
  rule C {
    C ::= c C compute {
     C.val = (1 + C[1].val) + (1 + 1); // gen
    };
    C ::= c compute {
      C.val = 1; // gen
    };
  }
}
```

Prikazani primjer predstavlja izvorni kod prevoditelja za domenskospecifični jezik *AnBnCn* [1] pisan u LISA *(Language Implementation System based on Attribute grammars)* prevoditelj-prevoditelj alatu [28]. Dok cijeli izvorni kod predstavlja jedno rješenje (jedinku), poje-

dine semantičke funkcije unutar produkcija predstavljaju gene. Svaka semantička funkcija (gen) zapravo je matematički izraz generiran iz stabla izraza. Tako Slika 1.10 prikazuje stablastu strukturu gena, odnosno izraza *(A.val == C.val) && (B.val == C.val)*.

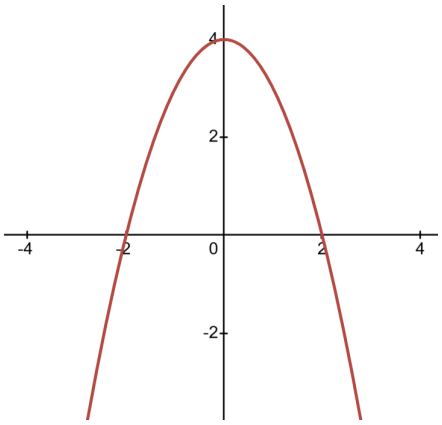


Slika 1.10: Stablasta reprezentacija izraza (A.val == C.val) && (B.val == C.val).

1.4 Prostor pretraživanja

Prostor pretraživanja u evolucijskim algoritmima odnosi se na skup svih mogućih rješenja koja se mogu istražiti tijekom procesa optimizacije, odnosno na raspon mogućih rješenja koje algoritam može razmotriti kako bi pronašao najbolje rješenje za zadani problem. Ovisno o problemu, često se vizualizira kao višedimenzionalni krajolik u kojem svaka točka predstavlja jedno moguće rješenje, a njezina "visina" ili vrijednost odgovara kvaliteti ili prikladnosti tog rješenja prema vrijednosti dobrote [29].

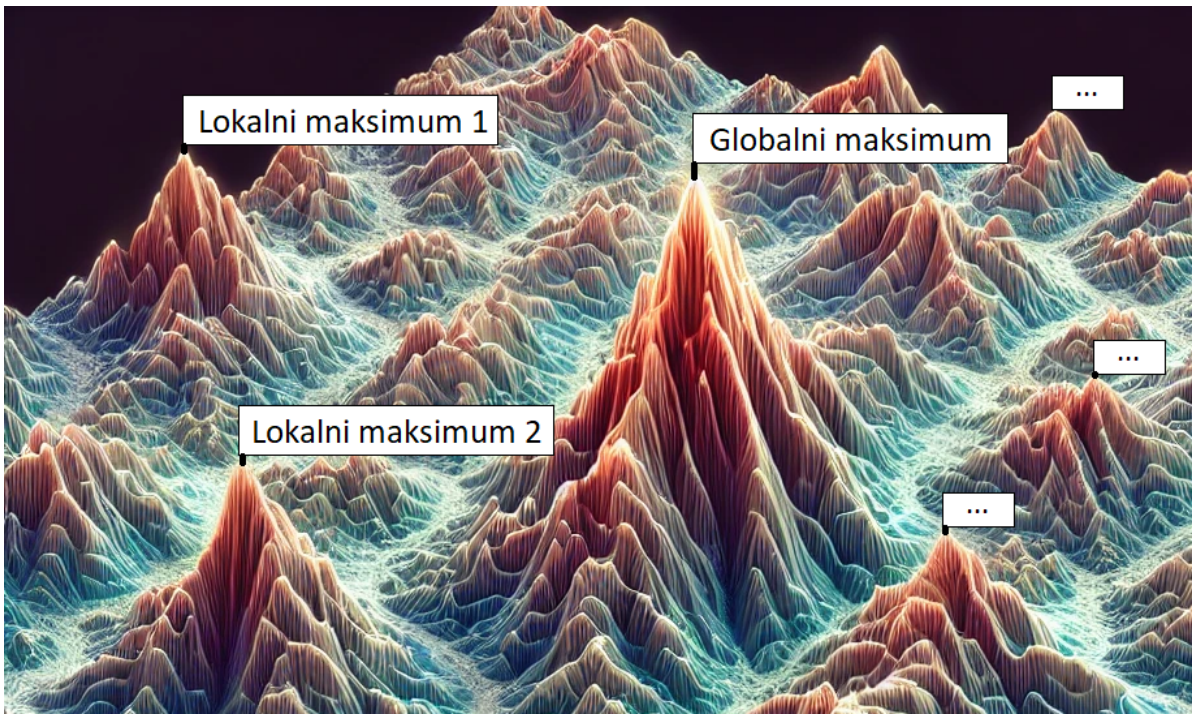
Dimenzionalnost prostora pretraživanja ovisi o kompleksnosti zadanog problema. Što je problem kompleksniji (ima više parametara o kojima ovisi rješenje) time se povećavaju i dimenzije prostora. Primjerice, za pronalaženje maksimuma funkcije $f(x) = -x^2 + 4$, gdje se traži vrijednost x koja daje najveću vrijednost funkcije, rješenje može biti predstavljeno parabolom (Slika 1.11). Ovo je primjer problema koji ima 1D prostor pretraživanja jer za jedan parametar (x) tražimo najveću vrijednost za y.



Slika 1.11: Primjer jednodimenzionalnog prostora pretraživanja funkcije $f(x) = -x^2 + 4$.

S druge strane, u 2D prostoru pretraživanja možemo razmatrati funkciju poput $f(x, y) = -x^2 - y^2 + 4$ koja predstavlja površinu u 3D prostoru. Ovdje rješenje može biti u obliku točke (x, y) koja maksimizira funkciju tako da za odgovarajuće vrijednosti x i y dobivamo

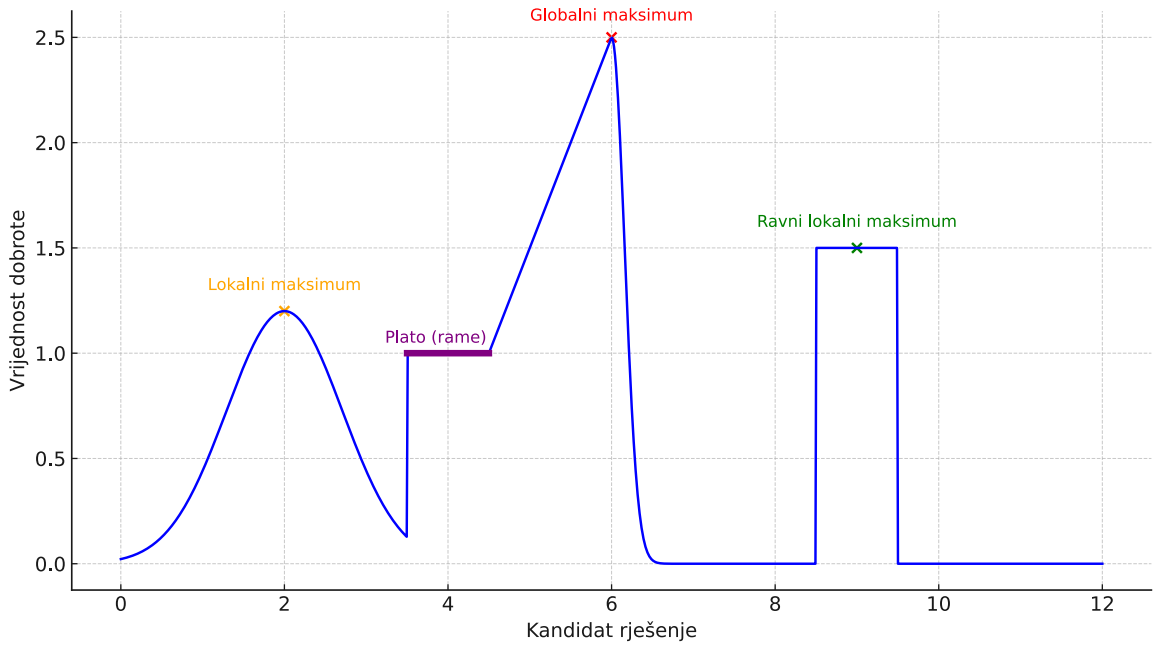
najveću vrijednost za z. Prikazani krajolik na Slici 1.12 predstavlja jedan takav primjer prostora za pretraživanje.



Slika 1.12: Primjer dvodimenzionalnog prostora pretraživanja.

Slika 1.12 također prikazuje i karakteristične točke u prostoru pretraživanja koje nazivamo ekstremima. Ovisno o prirodi problema, oni mogu biti lokalni i globalni maksimumi ili minimumi, a njihova prisutnost i broj ovise o zadanom problemu i definiciji dobrote. U nekom problemu može postojati nula ili više lokalnih ekstrema, kao i globalnih ekstrema. Lokalni ekstrem točka je u prostoru gdje dobrota poprima bolju vrijednost od svih ostalih susjednih točaka unutar određene okoline. Međutim, lokalni ekstrem ne predstavlja nužno i optimalno rješenje, već najčešće tek jedno od rješenja u prostoru pretraživanja koje je najbliže optimalnom rješenju za zadani problem. Optimalno rješenje predstavljeno je globalnim ekstremom (primjerice, globalnim maksimumom). Ako optimalno rješenje ne postoji, neće postojati niti jedan globalni ekstrem, dok u nekim slučajevima može postojati više optimalnih rješenja, odnosno više globalnih maksimuma ili minimuma [30, 31]. Poželjno je imati što manji broj lokalnih ekstrema kako evolucijski algoritam tijekom pretrage ne bi zaglavio u jednom od tih područja te time izbjegao pronaći globalni ekstrem. To je jedan od razloga zašto je evolucijske algoritme ponekad potrebno pokrenuti više puta ako od prvog pokušaja ne pronađu optimalno rješenje.

Prostor pretraživanja vrlo se često predstavlja i 2D grafom (Slika 1.13). Međutim, sada je uz lokalni i globalni maksimum prikazan plato (rame) te ravni lokalni maksimum. Plato je područje u prostoru pretraživanja u kojem vrijednost dobrote ostaje konstantna unutar određenog raspona. To znači da, bez obzira na to koju točku unutar tog područja uzmemo, vrijednost dobrote bit će jednaka. Međutim, ključno je da plato nije ekstrem – izlazak iz tog područja može voditi prema višim ili nižim vrijednostima. Dakle, u nekom



Slika 1.13: Prostor pretraživanja u evolucijskim algoritmima (2D graf).

smjeru izvan platoa moguće je pronaći bolje rješenje. S druge strane, ravni lokalni ekstrem (primjerice, ravni lokalni maksimum) također predstavlja područje s konstantnom vrijednosti dobrote, ali ta vrijednost ujedno je i najviša (ili najniža) u lokalnom okruženju. To znači da sve susjedne točke izvan tog područja imaju jednaku ili nižu (odnosno lošiju) vrijednost dobrote. U tom slučaju svi smjerovi iz ravnog lokalnog maksimuma vode prema lošijim rješenjima, pa je to zaista lokalni ekstrem.

Prostor pretraživanja moguće je istraživati i eksploatirati. Istraživanje (engl. *exploration*) odnosi se na pretraživanje novih, još neistraženih područja prostora kako bi se povećala raznolikost rješenja i izbjeglo zaglavljivanje u lokalnim ekstremima. Nasuprot tome, eksploatacija (engl. *exploitation*) odnosi se na usmjereno pretraživanje oko trenutno najboljih rješenja s ciljem njihovog daljnjeg poboljšanja. Ravnoteža između istraživanja i eksploatacije ključna je za učinkovito pretraživanje prostora jer omogućuje algoritmu da kontinuirano pronalazi nova potencijalna rješenja, ali i da optimalno iskorištava već pronađena dobra rješenja.

Također je važno napomenuti da prostor pretraživanja može biti statičan (uvijek istog oblika) ili dinamičan (podložan promjenama). Dinamički prostor pretraživanja mijenja svoj oblik ili svojstva tijekom vremena zbog promjena u uvjetima ili u funkciji dobrote. Primjerice, ako funkcija dobrote predstavlja model prometnog toka, tada se prostor pretraživanja mijenja tijekom dana ovisno o intenzitetu prometa. To predstavlja dodatni izazov jer evolucijski algoritam mora kontinuirano prilagođavati svoje strategije pretrage novim uvjetima, za razliku od statičnog prostora gdje uvjeti ostaju nepromijenjeni tijekom cijelog procesa optimizacije.

POGLAVLJE 2

Selekcija

Evolucijski algoritmi oponašaju prirodne evolucijske procese da bi pronašli optimalna rješenja za zadani problem. Kroz generacije, populacija rješenja postupno se prilagođava i poboljšava, sve dok se ne postigne zadovoljavajući kriterij završetka algoritma (Poglavlje 1.2.4). Ova prilagodba temelji se na ključnim evolucijskim procesima: selekciji, križanju (rekombinaciji) i mutaciji. Njihova pravilna primjena omogućuje učinkovito istraživanje i eksploataciju prostora mogućih rješenja, što značajno doprinosi ukupnoj učinkovitosti evolucijskog algoritma.

Ovisno o zadanom problemu, evolucijski algoritam neće nužno uvijek koristiti sve evolucijske procese. Primjerice, u evolucijskim strategijama križanje se najčešće izostavlja. Čak i kada se koriste svi evolucijski procesi, njihova strategija i način rada ne moraju nužno biti isti u svakoj implementaciji evolucijskog algoritma. Iako svaki proces prati osnovni koncept, postoji više različitih mehanizama selekcije, križanja i mutacije. Nadalje, navedene evolucijske procese moguće je poboljšati tehnikama paralelizacije i memoizacije. Ove tehnike omogućuju paralelnu evaluaciju jedinki u populaciji te pamćenje rezultata (vrijednosti dobrote) već evaluiranih rješenja, čime se može ubrzati rad evolucijskog algoritma.

Nakon generiranja i evaluacije početne populacije, evolucijski algoritam ulazi u petlju evolucijskih procesa (Poglavlje 1.2.4). Ovisno o implementaciji, prvi korak u toj petlji najčešće je selekcija. Cilj selekcije, koja ovisi o odabranom mehanizmu, jest odabrati što kvalitetnije jedinke u trenutnoj populaciji koje će sudjelovati u narednom procesu križanja, gdje će se rekombinacijom njihovih gena nastojati dobiti još kvalitetnije jedinke (rješenja). Bez posebnih mehanizama selekcije, evolucijski algoritam neće biti naročito učinkovit jer neće usmjeravati pretragu prema kvalitetnijim rješenjima. Iz istog razloga, mehanizmi poput potpuno slučajne selekcije vrlo se rijetko koriste.

Evolucijski algoritmi koriste različite mehanizme selekcije da bi odabrali jedinke za korake križanja i mutacije. Među najpoznatijima su proporcionalna (engl. *roulette-wheel*) [32], stohastička univerzalna [33], rangirajuća [34] i turnirska [35] selekcija. Proporcionalna

selekcija oslanja se na načelo proporcionalnosti, gdje jedinke s boljim vrijednostima dobrote imaju veće izglede da budu odabrane. Stohastička univerzalna selekcija dodatno poboljšava ovu metodu smanjujući slučajnost u procesu odabira. Rangirajuća selekcija, s druge strane, rangira jedinke prema kvaliteti i na temelju tog poretka ih odabire, čime izbjegava dominaciju izuzetno dobrih rješenja. Konačno, turnirska selekcija slučajnim odabirom formira manje skupine jedinki i iz svake skupine bira najbolju.

2.1 Proporcionalna selekcija

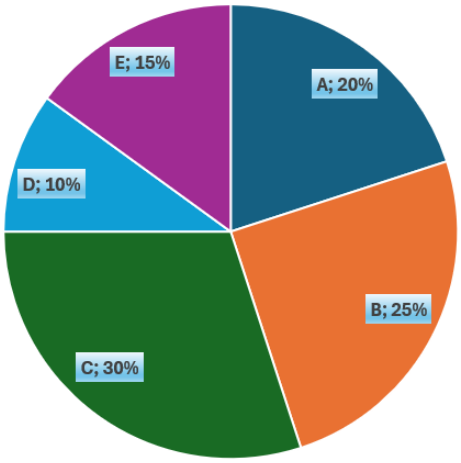
Proporcionalna (engl. $roulette - wheel$) selekcija [32] temelji se na igri ruleta u kasinu. Nakon što se zbroje vrijednosti dobrote svih jedinki u populaciji, izračunava se vjerojatnost odabira svake od njih, pri čemu je vjerojatnost odabira pojedine jedinke proporcionalna vrijednosti njezine dobrote. Što je vrijednost dobrote veća, to je veća i vjerojatnost odabira takve jedinke. Iako ovaj mehanizam selekcije daje prednost najkvalitetnijim jedinkama, uvijek ostavlja mogućnost odabira i slabijih jedinki, čime doprinosi očuvanju genetske raznolikosti u populaciji.

Tablica 2.1: Vjerojatnost odabira jedinki kod proporcionalne selekcije.

Jedinka	Vrijednost dobrote	Vjerojatnost odabira (%)
A	20	20
B	25	25
C	30	30
D	10	10
E	15	15
Zbroj	100	100

Za demonstraciju možemo razmotriti primjere jedinki prikazanih u Tablici 2.1. Svaka od navedenih jedinki ima svoju vrijednost dobrote, a ukupna vrijednost iznosi 100 ($20 + 25 + 30 + 10 + 15$). Na temelju ovog zbroja može se izračunati vjerojatnost odabira svake jedinke kao udio vrijednosti dobrote u ukupnom zbroju. Primjerice, vjerojatnost odabira jedinke A iznosi 20% ($20/100 \cdot 100$), dok za jedinku B iznosi 25% ($25/100 \cdot 100$) itd. Izračunate vjerojatnosti moguće je prikazati u obliku kola ruleta da bismo i vizualno dobili dojam o vjerojatnosti odabira pojedine jedinke u populaciji (Slika 2.1).

Nakon što se izračunaju vjerojatnosti odabira za svaku jedinku, sljedeći korak u proporcionalnoj selekciji simulacija je vrtnje kola ruleta. Jedinka A pokriva prvih 20% kola (interval 0%–20%), jedinka B sljedećih 25% (interval 21%–45%), jedinka C 30% (interval 46%–75%), jedinka D 10% (interval 76%–85%), i jedinka E posljednjih 15% (interval 86%–100%). Nakon toga generira se slučajni broj između 0 i 1 da bi se odredilo u kojem segmentu se rulet zaustavlja, odnosno koja jedinka je odabrana. Ako, primjerice, bude generirana vrijednost 0.4, odabrana će biti jedinka B jer pripada intervalu 21%–45%. Taj se postupak ponavlja sve dok ne bude odabran dovoljan broj jedinki za formiranje nove populacije (Algoritam 2.1).



Slika 2.1: Vjerojatnost odabira jedinki kod proporcionalne selekcije.

Algoritam 2.1 Proporcionalna selekcija.

function PROPORCIONALNASELEKCIJA(P)
    $S \leftarrow 0$
    for svaka jedinka $p_i \in P$ **do**
        $S \leftarrow S + f_i$ ▷ Zbrajanje vrijednosti dobrote svih jedinki.
    end for
    $C_1 \leftarrow \frac{f_1}{S}$ ▷ Kumulativna vjerojatnost prve jedinke.
    for $i \leftarrow 2$ **to** n **do**
        $C_i \leftarrow C_{i-1} + \frac{f_i}{S}$ ▷ Kumulativna vjerojatnost za ostale jedinke.
    end for
    for $i \leftarrow 1$ **to** $P.velicina$ **do**
        $r \leftarrow \text{Random}(0, 1)$ ▷ Generiraj slučajni broj između 0 i 1.
        for svaka jedinka $p_i \in P$ **do**
            if $C_i \geq r$ **then** ▷ Broj pada unutar kumulativne vjerojatnosti jedinke p_i?
                $P' \leftarrow P' \cup \{p_i\}$ ▷ Dodaj odabranu jedinku u novu populaciju.
                break ▷ Prekini petlju nakon odabira jedinke.
            end if
        end for
    end for
    return P' ▷ Vrati novu populaciju.
end function

Prikazani algoritam možemo implementirati i testirati korištenjem programskog jezika C++. U tu svrhu prvo ćemo kreirati klasu $Jedinka$ čiji će objekti predstavljati jedinke u populaciji, gdje svaka od njih ima svoju jedinstvenu oznaku i vrijednost dobrote:

```
class Jedinka {
public:
    char oznaka; // identifikator jedinke
    double dobrota;  // Vrijednost dobrote
    Jedinka(char _oznaka, double _dobrota) : oznaka(_oznaka), dobrota(_dobrota) {};
};
```

Sukladno postojećoj klasi i pseudokodu u Algoritmu 2.1 implementiramo C++ funkciju

za proporcionalnu selekciju (Primjer 2.1).

Primjer 2.1: Implementacija proporcionalne selekcije u programskom jeziku C++.

```
std::vector<Jedinka> proporcionalnaSelekcija(const std::vector<Jedinka>& P) {
    // Ukupna suma vrijednosti dobrote svih jedinki
    double S = 0;
    for (const auto& p : P)
        S += p.dobrota;

    // Kumulativne vjerojatnosti
    std::vector<double> C;
    double C_i = P[0].dobrota / S;
    C.push_back(C_i);

    for (size_t i = 1; i < P.size(); ++i) {
        C_i += P[i].dobrota / S;
        C.push_back(C_i);
    }

    // Nova populacija
    std::vector<Jedinka> P_;

    // Generator slucajnih brojeva
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    // Petlja za odabir jedinki
    for (size_t i = 0; i < P.size(); ++i) {
        double r = dis(gen);  // Slucajni broj [0, 1]
        for (size_t j = 0; j < P.size(); ++j) {
            if (C[j] >= r) {
                P_.push_back(P[j]);  // Dodaj odabranu jedinku u novu populaciju
                break;
            }
        }
    }
    return P_;  // Vrati novu populaciju
}
```

Prikazana implementacija funkcije kao parametar prima trenutnu populaciju jedinki. Zatim vraća novu populaciju koja se sastoji od odabranih (selektiranih) jedinki. Nova i stara populacija trebaju imati istu veličinu, zbog čega se proces selekcije ponavlja *P.size()* puta. Funkciju možemo testirati na sljedeći način:

```
std::vector<Jedinka> P = { {'A', 20}, {'B', 25}, {'C', 30} , {'D', 20}, {'E', 15} };
std::vector<Jedinka> novaPopulacija = proporcionalnaSelekcija(P);

std::cout << "Nova populacija (jedinka, dobrota):\n";
for (const auto& jedinka : novaPopulacija)
    std::cout << jedinka.oznaka << ", " << jedinka.dobrota << std::endl;
```

Puni primjer rada proporcionalne selekcije moguće je pronaći u Prilogu 9.1, a kao jedan od rezultata izvršavanja prethodnog programskog koda možemo dobiti sljedeći izlaz:

```
Nova populacija (jedinka, dobrota):
C, 30
A, 20
C, 30
```

```
C, 30
A, 20
```

Iako proporcionalna selekcija preferira odabir kvalitetnijih jedinki u populaciji, uvijek postoji vjerojatnost da najbolje jedinke ipak ne budu izabrane za postupak križanja te da u konačnici budu izgubljene. Da bi se to spriječilo potrebno je primijeniti elitizam (Poglavlje 5.1) na kraju evolucijskog procesa.

2.2 Stohastička univerzalna selekcija

Proporcionalna selekcija jednostavna je i brza. Međutim, zbog veće vjerojatnosti odabira kvalitetnijih jedinki, manja raznolikost populacije često predstavlja njen glavni nedostatak. Da bismo povećali raznolikost selektirane populacije, možemo koristiti stohastičku univerzalnu selekciju (SUS) [33].

Algoritam 2.2 Stohastička univerzalna selekcija (SUS).

function SUS(P)
  ukupnaDobrota $\leftarrow 0$ ▷ Inicijalizacija ukupne vrijednosti dobrote.
  for svaka jedinka $p_i \in P$ **do**
    ukupnaDobrota $\leftarrow$ ukupnaDobrota $+ f_i$ ▷ Sumiraj vrijednosti dobrote.
  end for
  $D \leftarrow \frac{\text{ukupnaDobrota}}{|P|}$ ▷ Izračunaj razmak između pokazivača.
  $r \leftarrow \text{Random}(0, D)$ ▷ Generiraj početni pokazivač u intervalu $[0, D)$.
  kumulativnaDobrota $\leftarrow 0$
  $i \leftarrow 1$
  for $k \leftarrow 1$ **to** $P.velicina$ **do**
    while $r >$ kumulativnaDobrota **do**
      kumulativnaDobrota $\leftarrow$ kumulativnaDobrota $+ f_i$
      $i \leftarrow i + 1$
    end while
    $P' \leftarrow P' \cup \{p_{i-1}\}$ ▷ Dodaj odabranu jedinku u novu populaciju.
    $r \leftarrow r + D$ ▷ Pomakni pokazivač za razmak D.
  end for
  return P' ▷ Vrati novu populaciju.
end function

Za razliku od proporcionalne selekcije, gdje se rulet vrti svaki put za pojedinačni odabir jedinke, kod SUS-a rulet se vrti samo jednom za cijelu populaciju. Naime, SUS koristi ravnomjerno raspoređene pokazivače za odabir jedinki na ruletu. Budući da pokazivači pokrivaju cijelu populaciju, svaka jedinka ima priliku biti izabrana proporcionalno svojoj dobroti. Ovaj pristup omogućuje ravnomjerniji odabir i smanjuje mogućnost dominacije nekoliko kvalitetnih jedinki, čime SUS pridonosi većoj raznolikosti selektirane populacije u usporedbi s proporcionalnom selekcijom.

Primjer 2.2: Implementacija stohastičke univerzalne selekcije u programskom jeziku C++.

```
1 std::vector<Jedinka> SUS(const std::vector<Jedinka>& populacija) {
```

```
    // Korak 1: Izracunaj ukupnu vrijednost dobrote
    double ukupnaDobrota = 0.0;
    for (const auto& jedinka : populacija) {
        ukupnaDobrota += jedinka.dobrota;
    }

    // Korak 2: Odredi razmak izmedu pokazivaca
    double D = ukupnaDobrota / populacija.size();

    // Korak 3: Generiraj pocetni slucajni broj u intervalu [0, D)
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(0, D);
    double r = dist(gen);

    // Korak 4: Kreiraj novu populaciju
    std::vector<Jedinka> novaPopulacija;
    double kumulativnaDobrota = 0.0;
    size_t i = 0;

    for (size_t k = 0; k < populacija.size(); ++k) {
        while (r > kumulativnaDobrota) {
            kumulativnaDobrota += populacija[i].dobrota;
            ++i;
        }

        // Dodaj trenutnu jedinku u novu populaciju
        novaPopulacija.push_back(populacija[i - 1]);
        r += D;
    }
    return novaPopulacija;
}
```

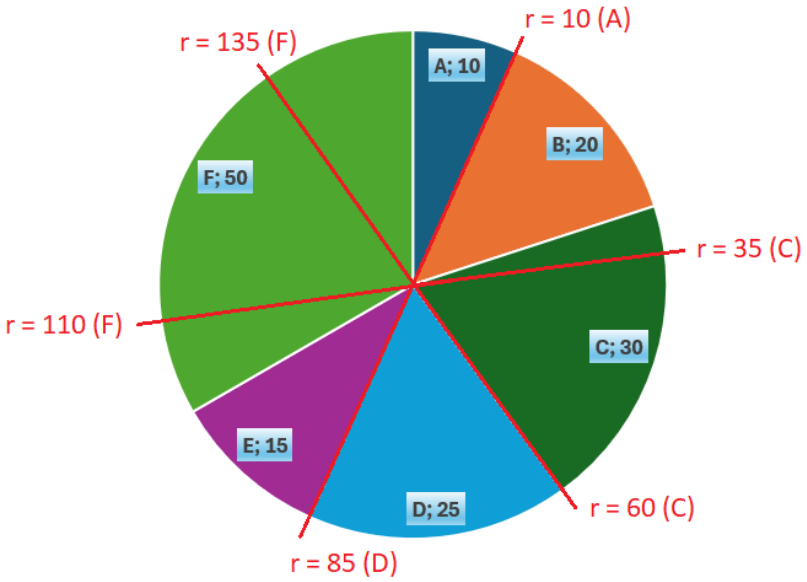
Algoritam 2.2 prikazuje pseudokod SUS algoritma, a Primjer 2.2 prikazuje njegovu implementaciju u programskom jeziku C++. Puni programski kod nalazi se u Prilogu 9.2.

Za demonstraciju pretpostavimo da imamo šest jedinki sa sljedećim vrijednostima dobrote: $A; 10$, $B; 20$, $C; 30$, $D; 25$, $E; 15$ i $F; 50$. Prvo računamo ukupnu vrijednost dobrote: 150 te razmak među pokazivačima kao

$$D = \frac{\text{ukupna dobrota}}{\text{veličina populacije}} = \frac{150}{6} = 25$$

U trećem koraku generiramo slučajni broj r u intervalu $[0, D)$ koji predstavlja početni pokazivač na kolu ruleta. Ako pretpostavimo da je $r = 10$, možemo vizualizirati kolo ruleta koje će biti korišteno pri odabiru jedinki (Slika 2.2). Kolo ima šest ravnomjerno raspoređenih pokazivača s razmakom $D = 25$ te je iz njega odmah moguće vidjeti koje će jedinke biti odabrane. To su jedinke $A; 10$ ($r = 10$, početni pokazivač), $C; 30$ ($r = 35$), $C; 30$ ($r = 60$), $D; 25$ ($r = 85$), $F; 50$ ($r = 110$), $F; 50$ ($r = 135$). Ovako ravnomjerno raspoređeni pokazivači omogućuju odabir dovoljnog broja jedinki iz samo jedne vrtnje ruleta, pri čemu se dobiva i veća raznolikost rezultantne populacije.

U ovom primjeru nije eksplicitno izračunata vjerojatnost odabira svake jedinke, ali ona je implicitno uključena u način rada SUS-a kroz proporcionalnu raspodjelu dobrote jedinki na ruletu. Primjerice, jedinka $E; 15$ nije bila izabrana u ovom primjeru upravo zbog male vrijednosti dobrote, odnosno zbog manje zastupljenosti na kolu ruleta. Isti pristup moguće je primijeniti i u proporcionalnoj selekciji jer u obje metode ukupna dobrota populacije definira dužinu ruleta, a vjerojatnost odabira ovisi o vrijednosti dobrote pojedine jedinke.



Slika 2.2: Odabir jedinki kod stohastičke univerzalne selekcije.

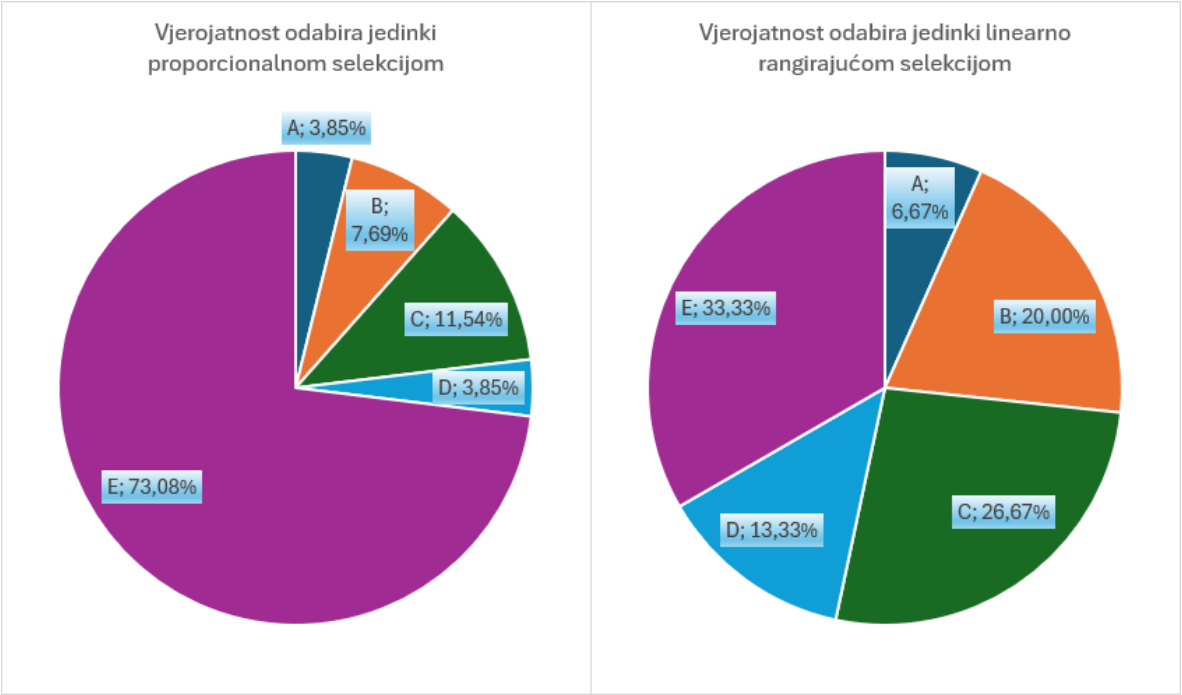
2.3 Rangirajuća selekcija

U oba prethodna algoritma, jedinke s ekstremno visokim vrijednostima dobrote u populaciji imaju znatno veću vjerojatnost odabira. Primjerice, ako imamo jedinke s vrijednostima dobrote $A; 5$, $B; 10$, $C; 15$, $D; 5$ i $E; 95$ može se zaključiti da će jedinka E dominirati rezultantnom populacijom, što može narušiti njezinu raznolikost i dovesti do prerane konvergencije u lokalni optimum. Da bismo to spriječili i slabijim jedinkama dali veću vjerojatnost odabira, možemo koristiti rangirajuću selekciju koja se temelji na relativnom rangu jedinki, a ne na njihovoj apsolutnoj vrijednosti dobrote. Koliku će vjerojatnost odabira imati pojedina jedinka na temelju svog ranga ovisi o tipu rangirajuće selekcije. Tako postoji linearno rangirajuća selekcija (Algoritam 2.3) kod koje se, nakon uzlaznog sortiranja jedinki prema vrijednosti dobrote, vjerojatnost odabira računa proporcionalno njihovom rangu. Pri tome slabije jedinke dobivaju niži, a kvalitetnije jedinke viši rang.

$$Vjerojatnost[rang] = \frac{rang}{sumaRangova}$$

Kvalitetnije jedinke i dalje imaju veću vjerojatnost odabira jer će biti višeg ranga. Međutim, ta je vjerojatnost znatno manja nego u slučaju proporcionalne ili stohastičke univerzalne selekcije, što može rezultirati većom raznolikošću rezultantne populacije. Osim linearne, rangirajuća selekcija može biti i eksponencijalna, a ona daje još veće prednosti kvalitetnijim jedinkama. Rangirajući pristup selekciji može se također kombinirati s turnirskom selekcijom.

Slika 2.3 prikazuje primjer u kojem se, u obliku kola rulet, prikazuje vjerojatnost odabira prethodno spomenutih pet jedinki $A - E$. Dok dominantna jedinka E ima vjerojatnost odabira $73,08\%$ u proporcionalnoj selekciji $(95/130)$, u linearno rangirajućoj selekciji ta ista jedinka ima tek $33,33\%$ vjerojatnosti odabira $(5/15)$. Iako i dalje ima najveću vjerojatnost odabira, jedinka E više nije toliko dominantna u odnosu na ostale (Tablica 2.2). Suk-



Slika 2.3: Vjerojatnost odabira jedinki kod proporcionalne i rangirajuće selekcije.

ladno prikazanom primjeru, osnovna razlika između proporcionalne i linearno rangirajuće selekcije jest način računanja vjerojatnosti odabira jedinke. On se temelji na osobnoj i kumulativnoj vrijednosti dobrote (Algoritam 2.1), odnosno na osobnom i kumulativnom rangu unutar populacije (Algoritam 2.3, Prilog 9.3).

Tablica 2.2: Vjerojatnost odabira jedinki kod proporcionalne i rangirajuće selekcije.

Jedinka	Dobrota	Vjerojatnost (Prop. sel.)	Rang po dobroti	Vjerojatnost (Rang. sel.)
A	5	3,85%	1	6,67%
B	10	7,69%	3	20,00%
C	15	11,54%	4	26,67%
D	5	3,85%	2	13,33%
E	95	73,08%	5	33,33%
Suma	130		15	

Kada dvije ili više jedinki imaju istu vrijednost dobrote (primjerice, jedinke A i D), one neće imati istu vjerojatnost odabira jer će pripadati različitim rangovima. U takvim situacijama njihov se rang obično određuje prema redoslijedu u populaciji, tj. prema tome koja se jedinka ranije pojavljuje u originalnoj populaciji. Zbog toga u našem primjeru jedinka A ima rang 1, odnosno vjerojatnost odabira $6,67\%$, jer se u populaciji nalazi prije jedinke D, koja ima rang 2 i vjerojatnost odabira $13,33\%$ (Tablica 2.2).

Algoritam 2.3 Linearno rangirajuća selekcija.

```
1: function LINEARNORANGIRAJUCASELEKCIJA(P)
2:     N ← P.velicina                                  ▷ Broj jedinki u populaciji.
3:     Sortiraj P                          ▷ ...prema njihovoj vrijednosti dobrote u uzlaznom poretku.
4:     ukupniRang ← N·(N+1)/2                          ▷ Zbroj rangova svih jedinki.
5:     vjerojatnosti ← [ ]                             ▷ Inicijaliziraj listu vjerojatnosti selekcije.
6:     for i ← 1 to N do
7:         vjerojatnosti[i] ← i/ukupniRang             ▷ Vjerojatnost proporcionalna rangu.
8:     end for
9:     P' ← ∅                                          ▷ Inicijaliziraj novu populaciju.
10:    for k ← 1 to N do                               ▷ Iteriraj za svaki član nove populacije.
11:        r ← Random(0, 1)                            ▷ Generiraj slučajni broj između 0 i 1.
12:        kumulativnaVjerojatnost ← 0
13:        for i ← 1 to N do
14:            kumulativnaVjerojatnost ← kumulativnaVjerojatnost + vjerojatnosti[i]
15:            if r ≤ kumulativnaVjerojatnost then
16:                P' ← P' ∪ {P[i]}                    ▷ Dodaj odabranu jedinku u novu populaciju.
17:                break
18:            end if
19:        end for
20:    end for
21:    return P'                                       ▷ Vrati novu populaciju.
22: end function
```

Kao i kod prethodnih selekcijskih mehanizama, rangirajuće selekcije ne jamče odabir najboljih jedinki u populaciji. Zbog toga je u slučaju njihova korištenja i dalje preporuka primijeniti elitizam na kraju evolucijskog procesa.

2.4 Turnirska selekcija

Umjesto na vrtnji kola ruleta, turnirska selekcija temelji se na simulaciji turnira u kojem sudjeluju dvije ili više slučajno odabranih jedinki. Najbolja među njima odabire se za rezultantnu populaciju, a turniri se ponavljaju sve dok ona ne dosegne veličinu inicijalne (ulazne) populacije (Algoritam 2.4, Prilog 9.4).

Algoritam 2.4 Turnirska selekcija.

```
1: function TURNIRSKASELEKCIJA(P, k)
2:     P' ← ∅                                          ▷ Inicijaliziraj novu populaciju.
3:     for i ← 1 to P.velicina do                      ▷ Iteriraj za svaku jedinku u populaciji.
4:         turnir ← SlucajnoOdaberi(P, k)              ▷ Odaberi k slučajnih jedinki.
5:         najbolji ← PronadiNajboljeg(turnir)         ▷ Odredi jedinku s najboljom dobrotom.
6:         P' ← P' ∪ {najbolji}                        ▷ Dodaj najboljeg iz turnira u novu populaciju.
7:     end for
8:     return P'                                       ▷ Vrati novu populaciju.
9: end function
```

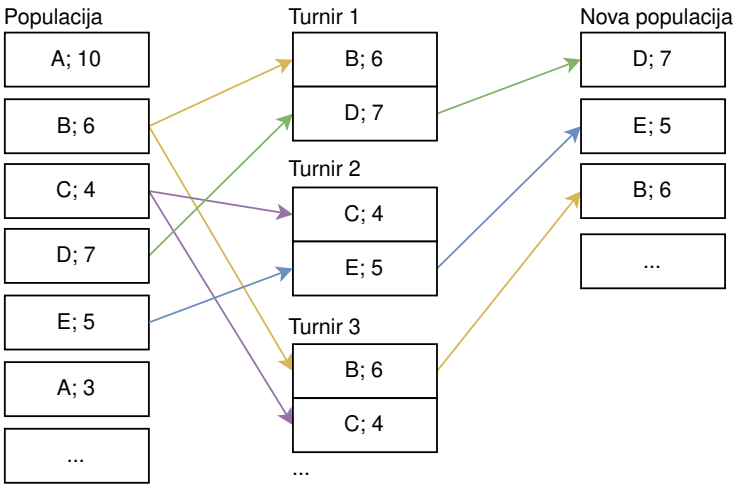
Turnirska selekcija ubraja se među najjednostavnije selekcijske mehanizme, što je vidljivo

i iz njezine implementacije u programskom jeziku C++ (Primjer 2.3).

Primjer 2.3: Implementacija turnirske selekcije u programskom jeziku C++.

```cpp
std::vector<Jedinka> turnirskaSelekcija(const std::vector<Jedinka>& populacija, int k) {
    std::vector<Jedinka> novaPopulacija;
    std::mt19937 generator(std::random_device{}());
    std::uniform_int_distribution<> distribucija(0, populacija.size() - 1);

    for (size_t i = 0; i < populacija.size(); i++) {
        // Odaberi k slucajnih kandidata
        std::vector<int> kandidati;
        for (int j = 0; j < k; j++)
            kandidati.push_back(distribucija(generator));

        // Pronadi najboljeg medu kandidatima
        int pobjednik = kandidati[0];
        for (int j = 1; j < k; j++)
            if (populacija[kandidati[j]].dobrota > populacija[pobjednik].dobrota)
                pobjednik = kandidati[j];

        // Dodaj pobjednika u novu populaciju
        novaPopulacija.push_back(populacija[pobjednik]);
    }
    return novaPopulacija;
}
```

Veličinu turnira određuje broj sudionika, pri čemu jedna jedinka može sudjelovati u više turnira. Štoviše, zbog slučajnog odabira, ista jedinka može se natjecati i protiv same sebe. Veličina turnira najčešće je konstantna u svakom krugu, ali može biti i stohastička (slučajno odabrana) ili dinamička, u svrhu kontrole selekcijskog pritiska (Poglavlje 2.6). Turniri s većim brojem sudionika omogućuju bržu konvergenciju evolucijskog algoritma, ali povećavaju vjerojatnost konvergencije prema lokalnom optimumu.



Slika 2.4: Primjer turnirske selekcije.

Slika 2.4 demonstrira način rada turnirske selekcije nad populacijom s jedinkama i njihovim dobrotama: $A; 10, B; 6, C; 4, D; 7, E; 5$ i $A; 3$. U prvom turniru slučajno su odabrane jedinke $B; 6$ i $D; 7$. Bolja od njih (jedinka D) prelazi u novu (rezultantnu) populaciju. Slično se događa u sljedeća dva turnira gdje bolje od jedinki $C; 4$ i $E; 5$, odnosno $B; 6$ i

$C; 4$ također prelaze u rezultantnu populaciju.

Ovisno o veličini turnira, elitizam ne mora nužno biti primijenjen na kraju evolucijskog procesa. Primjerice, kod velikih turnira postoji mala vjerojatnost da najbolje jedinke već nisu odabrane, što omogućuje uštedu procesorskog vremena izbjegavanjem elitizma. Također, pojedini turniri međusobno su neovisni, što turnirsku selekciju čini pogodnom za paralelizaciju.

2.5 Pohlepna selekcija

Pohlepna selekcija predstavlja jedan od načina za ubrzanje konvergencije algoritma u postupku selekcije. Naime, njome uvijek biramo trenutno najbolju jedinku u populaciji. Ovaj pristup primjenjuje se kada raznolikost populacije nije ključna, odnosno kada želimo maksimalno iskoristiti postojeće kvalitetne jedinke u potrazi za optimalnim rješenjem. Zbog svog načina rada, glavni nedostatak ovog pristupa često je prerana konvergencija algoritma prema lokalnom optimumu.

Algoritam 2.5 Pohlepna selekcija.

function POHLEPNASELEKCIJA(P)
  $P' \leftarrow \emptyset$ ▷ Inicijaliziraj novu populaciju.
  $kopijaP \leftarrow P$ ▷ Kopiraj početnu populaciju.
  for $i \leftarrow 1$ **to** $P.velicina$ **do**
    $najbolja \leftarrow$ NajboljaJedinka($kopijaP$) ▷ Pronađi jedinku s najvećom dobrotom.
    $P' \leftarrow P' \cup \{najbolja\}$ ▷ Dodaj najbolju jedinku u novu populaciju.
    $kopijaP \leftarrow R \setminus \{najbolja\}$ ▷ Ukloni odabranu jedinku iz preostale populacije.
  end for
  return P' ▷ Vrati novu populaciju.
end function

Algoritam 2.5 opisuje način rada pohlepne selekcije. Ulazna populacija P kopira se u pomoćnu populaciju $kopijaP$. Postupak odabira najkvalitetnije jedinke iz $kopijaP$ ponavlja se sve dok nova populacija P' ne dosegne potrebnu veličinu. Svaki put nakon što je najkvalitetnija jedinka odabrana uklanja se iz $kopijaP$ da bi se spriječilo njezino ponovno odabiranje u sljedećoj iteraciji.

Primjer 2.4: Implementacija pohlepne selekcije u programskom jeziku C++.

```
std::vector<Jedinka> pohlepnaSelekcija(std::vector<Jedinka>& Populacija) {
    std::vector<Jedinka> novaPop;
    std::vector<Jedinka> kopijaPop = Populacija; // Kopija pocetne populacije

    // Izbor jedinki...
    for (int i = 0; i < Populacija.size(); ++i) {
        // Pronadi najbolju jedinku u kopijaPop populaciji
        auto najboljaJedinkaIt = std::max_element(kopijaPop.begin(), kopijaPop.end(),
            [](const Jedinka& a, const Jedinka& b) {
                return a.dobrota < b.dobrota;
            });
        novaPop.push_back(*najboljaJedinkaIt); // Dodaj najbolju jedinku u novu populaciju
        kopijaPop.erase(najboljaJedinkaIt); // Ukloni odabranu jedinku
    }
```

```
15      return novaPop;
16  }
```

Ako testiramo C++ implementaciju pohlepne selekcije (Primjer 2.4, Prilog 9.5) na primjeru deset jedinki $(8, 4, 8, 6, 4, 7, 9, 5, 3, 7)$, rezultat će uvijek biti iste jedinke poredane silaznim redoslijedom prema vrijednosti dobrote $(9, 8, 8, 7, 7, 6, 5, 4, 4, 3)$, neovisno o tome je li ulazna populacija prethodno sortirana ili nije.

2.6 Selekcijski pritisak

Selekcijski pritisak određuje intenzitet kojim se u postupku selekcije favoriziraju bolja rješenja unutar populacije. Ovisno o odabranom selekcijskom mehanizmu, selekcijski pritisak može se koristiti za kontrolu brzine konvergencije evolucijskog algoritma prema boljim rješenjima [36]. Iako jači selekcijski pritisak ubrzava konvergenciju, često smanjuje raznolikost populacije, što može dovesti do prerane konvergencije algoritma prema lokalnom umjesto globalnom optimumu.

Rb.	Jedinka	Dobrota
1	G	9
2	C	8
3	A	8
4	J	7
5	F	7
6	D	6
7	H	5
8	E	4
9	B	4
10	I	3

Selekcijski pritisak = 80%

Slika 2.5: Primjer selekcijskog pritiska u iznosu od 80%.

Primjer djelovanja selekcijskog pritiska prikazan je na Slici 2.5. Prikazano je deset jedinki (A–J), silazno sortiranih prema vrijednostima dobrote. U ovom slučaju, selekcijski pritisak iznosi 80%, što znači da u postupku odabira sudjeluje samo prvih 80% najkvalitetnijih jedinki. Time se iz potencijalnog odabira isključuju najlošije jedinke B i I s vrijednostima dobrote 4 i 3, čime se ubrzava konvergencija algoritma.

Uz selekcijski pritisak od 80% rezultantna populacija zadržava relativno visoku raznolikost. Međutim, povećanjem selekcijskog pritiska raznolikost populacije se smanjuje, što može dovesti do prerane konvergencije prema lokalnom optimumu. Primjerice, ako bi selekcijski pritisak iznosio 30%, rezultantna populacija sastojala bi se samo od jedinki G, C i A, što bi negativno utjecalo na evolucijski algoritam koji za pronalazak optimalnog rješenja zahtijeva što veću raznolikost populacije.

Selekcijski pritisak najčešće se povećava kod vremenski zahtjevnih problema (primjerice, kada evaluacija jedinki traje dugo) jer se jačim selekcijskim pritiskom i fokusom na kvalitetnije jedinke nastoji brže pronaći optimalno rješenje. Međutim, zbog izravne povezanosti selekcijskog pritiska i raznolikosti populacije, preporučuje se izbjegavanje njegovog korištenja kod jednostavnijih problema [37].

POGLAVLJE 3

Križanje

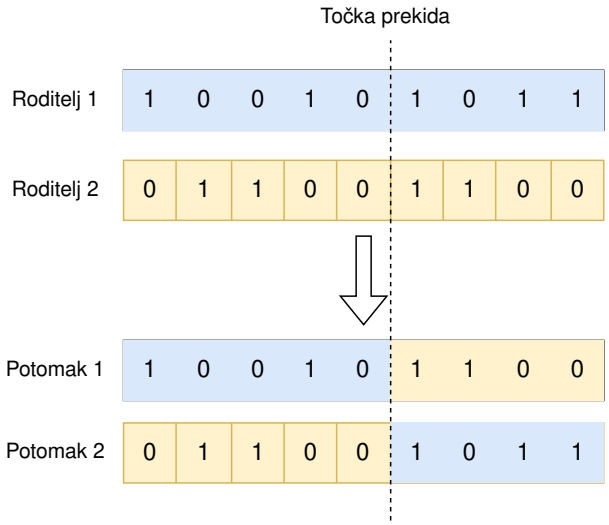
Rezultat selekcije obično su najkvalitetnije jedinke u populaciji koje se u postupku križanja nazivaju roditeljima. Za razliku od selekcije, koja je evolucijski operator usmjeren na odabir već postojećih jedinki, križanje je evolucijski operator kojim se stvaraju nove jedinke rekombiniranjem genetskog materijala roditelja. Križanjem (rekombinacijom) njihovih gena pokušava se dobiti još kvalitetnije jedinke (potomke) koje su bliže optimalnom rješenju. Međutim, za razliku od selekcije, križanje nije obavezan korak u evolucijskom algoritmu te ovisi o vjerojatnosti križanja p_c (engl. $probability\ of\ crossover$). Ovisno o raznolikosti populacije, križanje može rezultirati i potomcima koji su identični svojim roditeljima, što se često događa kada se populacija rješenja približava lokalnom ili globalnom optimumu. Potomci dobiveni križanjem mogu zamijeniti svoje roditelje u populaciji, ali to nije uvijek pravilo. Ovisno o strategiji zamjene, potomci također mogu zamijeniti najlošije jedinke u populaciji ili se natjecati za opstanak zajedno s roditeljima i ostalim članovima populacije (primjerice, kroz selekcijske mehanizme poput turnirske i rangirajuće selekcije).

Slično kao i kod selekcije, više je tipova operatora križanja, poput križanja s jednom ili više točaka prekida [37], uniformnog križanja [38], križanja s tri roditelja [39], aritmetičkog križanja [40], heurističkog križanja [41] i djelomično mapiranog križanja [42] koji će biti opisani u nastavku. Osim navedenih, postoje i mnogi drugi tipovi križanja, poput cikličkog [43], geometrijskog [44] i poligenskog [45].

3.1 Križanje s jednom ili više točaka prekida

Križanje s jednom ili više točaka prekida jedna je od osnovnih tehnika za simulaciju procesa genetske rekombinacije. Ovisno o broju točaka prekida, genetski materijal slučajno odabranih roditelja dijeli se na određenim mjestima, nakon čega se stvaraju potomci koji sadrže kombinacije gena odabranih roditelja. Ovaj pristup omogućuje temeljitije istraživanje prostora mogućih rješenja, s ciljem dobivanja potomaka koji su kvalitetom bolji od svojih roditelja.

Za demonstraciju, pretpostavimo da u svrhu križanja koristimo dva roditelja (jedinke) predstavljena binarnim reprezentacijama $(1, 0, 0, 1, 0, 1, 0, 1, 1)$ i $(0, 1, 1, 0, 0, 1, 1, 0, 0)$. Primjer njihovog križanja u slučaju korištenja samo jedne točke prekida vidljiv je na Slici 3.1. Točka prekida obično se određuje slučajnim odabirom i ovisi o broju gena. Za broj gena N ona može biti u intervalu $[1, N-1]$, a u prikazanom primjeru nalazi se na poziciji 5. Ovisno o odabranoj točki prekida, kao rezultat križanja dobivamo dva potomka koja sadrže kombinacije gena svojih roditelja. Prvi potomak ima gene prvog roditelja do točke prekida $(1, 0, 0, 1, 0)$, a nakon nje gene drugog roditelja $(1, 1, 0, 0)$. Slično je i s drugim potomkom, koji do točke prekida ima gene drugog roditelja $(0, 1, 1, 0, 0)$, a nakon nje gene prvog roditelja $(1, 0, 1, 1)$.



Slika 3.1: Križanje s jednom točkom prekida.

Križanje s jednom točkom prekida možemo demonstrirati i u programskom jeziku C++. Pretpostavimo da imamo sljedeću definiciju jedinke čiji se geni nalaze spremljeni u vektoru binarnih brojeva (puni primjer nalazi se u Prilogu 9.6):

```
class Jedinka {
public:
    std::vector<bool> geni;
    Jedinka(const std::vector<bool>& geni = {}) : geni(geni) {}
};
```

Opisano križanje možemo implementirati sljedećom funkcijom:

Primjer 3.1: Implementacija križanja s jednom točkom prekida u programskom jeziku C++.

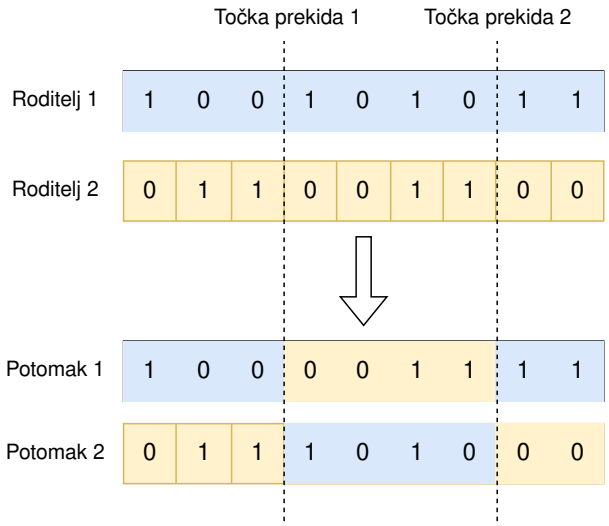
```
void KrizanjeSJednomTockom(const Jedinka& roditelj1, const Jedinka& roditelj2, int*
    tockaPrekida, Jedinka* potomak1, Jedinka* potomak2) {
    // Kreiranje generatora slucajnih brojeva i distribucije
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, roditelj1.geni.size() - 1);

    // Slucajna tocka prekida
    *tockaPrekida = distrib(gen);

```

```
10      // Potomak 1 - geni prvog roditelja do tocke prekida, a zatim geni drugog roditelja
11      potomak1->geni = { roditelj1.geni.begin(), roditelj1.geni.begin() + *tockaPrekida };
12      potomak1->geni.insert(potomak1->geni.end(), roditelj2.geni.begin() + *tockaPrekida,
        roditelj2.geni.end());
13
14      // Potomak 2 - geni drugog roditelja do tocke prekida, a zatim geni prvog roditelja
15      potomak2->geni = { roditelj2.geni.begin(), roditelj2.geni.begin() + *tockaPrekida };
16      potomak2->geni.insert(potomak2->geni.end(), roditelj1.geni.begin() + *tockaPrekida,
        roditelj1.geni.end());
17  }
```

Funkcija `KrizanjeSJednomTockom` (Primjer 3.1) kao ulazne parametre ima dvije jedinke (roditelje), a kao izlazne parametre odabranu točku prekida te dva potomka. Nakon slučajno odabrane točke prekida u intervalu $[1, roditelj1.geni.size() - 1]$, čime se osigurava da točka prekida nikada neće biti na početku ili na kraju vektora gena, funkcija će prethodno opisanim postupkom generirati i vratiti dva potomka. Njen poziv možemo demonstrirati na sljedeći način:

```
1  Jedinka roditelj1({ 1, 0, 0, 1, 0, 1, 0, 1, 1 });
2  Jedinka roditelj2({ 0, 1, 1, 0, 0, 1, 1, 0, 0 });
3  int tockaPrekida;
4  Jedinka potomak1, potomak2;
5
6  KrizanjeSJednomTockom(roditelj1, roditelj2, &tockaPrekida, &potomak1, &potomak2);
7  std::cout << "Tocka prekida: " << tockaPrekida << "\n"; // 3
8
9  for (int gen : potomak1.geni)
10     std::cout << gen << " ";  // 1 0 0 0 0 1 1 0 0
11 std::cout << "\n";
12 for (int gen : potomak2.geni)
13     std::cout << gen << " ";  // 0 1 1 1 0 1 0 1 1
```

Dobiveni izlazi ovise o slučajno odabranoj točki prekida, a u komentarima je prikazan primjer za slučaj kada je odabrana točka broj 3. Međutim, križanje može uključivati i više točaka prekida. Primjer takvog postupka prikazan je na Slici 3.2 koja demonstrira križanje s dvije točke prekida.



Slika 3.2: Križanje s dvije točke prekida.

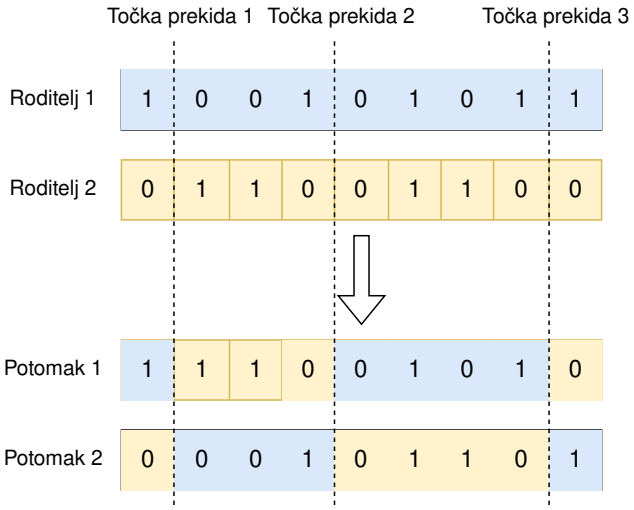
Zbog dvije točke prekida, geni potomaka podijeljeni su na tri segmenta. Prvi potomak nasljeđuje gene iz prvog i trećeg segmenta od prvog roditelja, dok gene iz srednjeg segmenta nasljeđuje od drugog roditelja. Obrnuto vrijedi za drugog potomka. Ipak, to ne znači da pojedini potomak mora imati više gena od roditelja od kojeg nasljeđuje više segmenata. Primjerice, ako su točke prekida rubne (primjerice, 1 i $N-1$), tada će potomak vjerojatno naslijediti veći broj gena od roditelja od kojeg nasljeđuje srednji segment gena.

Primjer 3.2: Implementacija križanja s dvije točke prekida u programskom jeziku C++.

```
void KrizanjeSDvijeTockePrekida(const Jedinka& roditelj1, const Jedinka& roditelj2, int*
    tockaPrekida1, int* tockaPrekida2, Jedinka* potomak1, Jedinka* potomak2) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, roditelj1.geni.size() - 1);

    // Slucajan odabir dvije tocke prekida
    *tockaPrekida1 = distrib(gen);
    *tockaPrekida2 = distrib(gen);

    // Provjeravamo da su tocke razlicite
    while (*tockaPrekida1 == *tockaPrekida2)
        *tockaPrekida2 = distrib(gen);

    // Ako je tocka prekida1 veca od tocke prekida2, zamijenimo ih
    if (*tockaPrekida1 > *tockaPrekida2)
        std::swap(*tockaPrekida1, *tockaPrekida2);

    // Geni prvog potomka...
    potomak1->geni = { roditelj1.geni.begin(), roditelj1.geni.begin() + *tockaPrekida1 };
    potomak1->geni.insert(potomak1->geni.end(), roditelj2.geni.begin() + *tockaPrekida1,
    roditelj2.geni.begin() + *tockaPrekida2);
    potomak1->geni.insert(potomak1->geni.end(), roditelj1.geni.begin() + *tockaPrekida2,
    roditelj1.geni.end());

    // Geni drugog potomka...
    potomak2->geni = { roditelj2.geni.begin(), roditelj2.geni.begin() + *tockaPrekida1 };
    potomak2->geni.insert(potomak2->geni.end(), roditelj1.geni.begin() + *tockaPrekida1,
    roditelj1.geni.begin() + *tockaPrekida2);
    potomak2->geni.insert(potomak2->geni.end(), roditelj2.geni.begin() + *tockaPrekida2,
    roditelj2.geni.end());
}
```

U usporedbi s križanjem koje koristi samo jednu točku prekida, ovaj pristup rezultira većom raznolikošću populacije, no kompliciraniji je za implementaciju (Primjer 3.2, Prilog 9.7). Osim što se geni potomaka sastoje od tri segmenta (što zahtijeva više kopiranja), generirane točke prekida ne smiju biti iste. Štoviše, prva točka prekida mora se obavezno nalaziti prije druge točke.

Križanje u tri ili više točaka funkcionira na vrlo sličan način, gdje svaki potomak nasljeđuje gene roditelja naizmjenično, prema segmentima. U primjeru na Slici 3.3 korištene su tri točke prekida, odnosno četiri segmenta gena. Oba potomka nastaju tako da se geni izmjenjuju u naizmjeničnim segmentima. Prvi segment ostaje nepromijenjen, drugi se mijenja između roditelja, treći ostaje isti, a zadnji (četvrti) segment također se mijenja.



Slika 3.3: Križanje s tri točke prekida.

3.2 Uniformno križanje

Za razliku od križanja temeljenog na točkama prekida, gdje se geni roditelja dijele u jedan ili više segmenata, uniformno križanje razmatra svaki gen roditelja zasebno. Kod uniformnog križanja moguće je primijeniti nekoliko različitih pristupa. Prvi pristup temelji se na slučajnom odabiru i-tog gena za prvi potomak, pri čemu se gen nasljeđuje od prvog ili drugog roditelja. Ako je i-ti gen prvog potomka naslijeđen od prvog roditelja, tada se i-ti gen drugog potomka automatski nasljeđuje od drugog roditelja i obrnuto (izvorni pristup [38]). Drugi pristup omogućuje da se geni oba potomka nasljeđuju potpuno slučajno od jednog ili drugog roditelja. Osim navedenih, moguće je primijeniti ponderirani odabir gena temeljen na vrijednostima dobrote jednog i drugog roditelja ili nasljeđivanje gena korištenjem unaprijed definirane maske.

Algoritam 3.1 Uniformno križanje (slučajni odabir gena jednog potomka).

```
1:  function UniformnoKrizanje(roditelj1, roditelj2, potomak1, potomak2)
2:      for i ← 1 to roditelj1.geni.kolicina do
3:          r ← slučajni broj (0 ili 1)
4:          if r = 0 then
5:              potomak1.geni[i] ← roditelj1.geni[i]
6:              potomak2.geni[i] ← roditelj2.geni[i]
7:          else
8:              potomak1.geni[i] ← roditelj2.geni[i]
9:              potomak2.geni[i] ← roditelj1.geni[i]
10:         end if
11:     end for
12: end function
```

Algoritam 3.1 prikazuje izvorni pristup uniformnom križanju temeljen na slučajnom odabiru gena. Za svaki gen potomaka slučajno se određuje hoće li ga prvi potomak naslijediti od prvog ili drugog roditelja. Ako je slučajno generirana vrijednost jednaka nuli, prvi po-

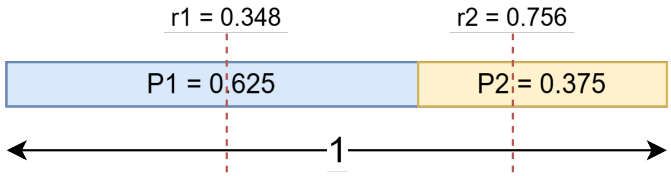
tomak nasljeđuje i-ti gen od prvog roditelja, dok drugi potomak automatski nasljeđuje isti gen od drugog roditelja. Ako je slučajno generirana vrijednost jednaka jedinici, prvi potomak nasljeđuje i-ti gen od drugog roditelja, a drugi potomak automatski od prvog. Ovaj pristup osigurava da se geni oba potomka uvijek nasljeđuju komplementarno, gdje za gen koji jedan potomak naslijedi od jednog roditelja, drugi potomak će uvijek naslijediti od drugog roditelja. To znači da su geni jednog potomka djelomično uvjetovani genima drugog potomka. Demonstracija ove metode nalazi se u Prilogu 9.8.

Algoritam 3.2 Uniformno križanje (slučajni odabir gena oba potomka).

```
1:  function UniformnoKrizanje(roditelj1, roditelj2, potomak1, potomak2)
2:      for i ← 1 to roditelj1.geni.kolicina do
3:          r1 ← slučajni broj (0 ili 1)
4:          r2 ← slučajni broj (0 ili 1)
5:          if r1 = 0 then
6:              potomak1.geni[i] ← roditelj1.geni[i]
7:          else
8:              potomak1.geni[i] ← roditelj2.geni[i]
9:          end if
10:         if r2 = 0 then
11:             potomak2.geni[i] ← roditelj1.geni[i]
12:         else
13:             potomak2.geni[i] ← roditelj2.geni[i]
14:         end if
15:     end for
16: end function
```

Nasuprot tome, Algoritam 3.2 i Prilog 9.9 prikazuju alternativni pristup u kojem se geni oba potomka nasljeđuju potpuno neovisno jedan o drugome. Svaki gen svakog potomka bira se slučajno od jednog od roditelja, bez obzira na prethodni izbor za drugog potomka. Ovakav pristup omogućava veću raznolikost i neovisnost između potomaka, jer svaki gen svakog potomka ima jednaku šansu da bude naslijeđen od bilo kojeg roditelja.

Da bi se ubrzala konvergencija algoritma, moguće je koristiti i ponderirani pristup u uniformnom križanju. U tom slučaju, vjerojatnost nasljeđivanja gena prilagođava se proporcionalno vrijednosti dobrote odabranih roditelja, čime se potiče nasljeđivanje gena od kvalitetnijih roditelja (Algoritam 3.3, Prilog 9.10).



Slika 3.4: Odabir roditelja u ponderiranom uniformnom križanju.

Primjerice, ukoliko imamo dva roditelja s vrijednostima dobrote 5 i 3, vjerojatnost nasljeđivanja svakog od gena od prvog roditelja ($P1$) iznosi 0.625, a od drugog roditelja ($P2$)

Algoritam 3.3 Uniformno križanje (ponderirani slučajni odabir gena oba potomka)

```
1: function UniformnoKrizanje(roditelj1, roditelj2, potomak1, potomak2)
2:     P1 ← roditelj1.dobrota / (roditelj1.dobrota+roditelj2.dobrota)   ▷ Vjerojatnost nasljeđivanja gena od prvog roditelja.
3:     P2 ← 1 − P1                                                      ▷ Vjerojatnost nasljeđivanja gena od drugog roditelja.
4:     for i ← 1 to roditelj1.geni.kolicina do
5:         r1 ← slučajni broj između 0 i 1
6:         r2 ← slučajni broj između 0 i 1
7:         if r1 ≤ P1 then
8:             potomak1.geni[i] ← roditelj1.geni[i]
9:         else
10:            potomak1.geni[i] ← roditelj2.geni[i]
11:        end if
12:        if r2 ≤ P2 then
13:            potomak2.geni[i] ← roditelj1.geni[i]
14:        else
15:            potomak2.geni[i] ← roditelj2.geni[i]
16:        end if
17:    end for
18: end function
```

iznosi 0.375. Ako su za prvi gen oba potomka generirane slučajne vrijednosti $r1 = 0.348$ i $r2 = 0.756$ to znači da prvi potomak nasljeđuje gen od prvog roditelja, dok drugi potomak gen od drugog roditelja (Slika 3.4). Isti postupak ponavlja se i za preostale gene potomaka, gdje se, ovisno o slučajno generiranim vrijednostima $r1$ i $r2$, geni nasljeđuju od prvog ili drugog roditelja. Ovaj postupak osigurava slučajnost u nasljeđivanju gena, pri čemu su vjerojatnosti nasljeđivanja proporcionalne relativnim vrijednostima dobrote roditelja (njihovu omjeru).

Uniformno križanje može se realizirati i korištenjem maske, umjesto slučajnih vjerojatnosti. Maska je niz unaprijed definiranih vrijednosti (nula i jedinica) koje određuju od kojeg roditelja se svaki gen nasljeđuje. Ako je i-ta vrijednost u maski jednaka nuli, prvi potomak nasljeđuje i-ti gen od prvog roditelja, dok drugi potomak nasljeđuje i-ti gen od drugog roditelja. U suprotnom slučaju, kada je i-ta vrijednost u maski jednaka jedinici, prvi potomak nasljeđuje i-ti gen od drugog roditelja, a drugi potomak i-ti gen od prvog roditelja (Algoritam 3.4).

Za demonstraciju pretpostavimo da postoje sljedeći roditelji i maska:

```
Roditelj 1: (1,0,1,1,0)
Roditelj 2: (0,1,0,0,1)
-----------------------
Maska:      (0,1,0,0,1)
```

Slijedeći Algoritam 3.4 dobivamo dva potomka sa sljedećim genima:

```
Potomak 1: (1,1,1,1,1)
Potomak 2: (0,0,0,0,0)
```

Algoritam 3.4 Uniformno križanje pomoću maske.

```
1: function UNIFORMNOKRIZANJEMASKE(roditelj1, roditelj2, maska, potomak1, potomak2)
2:     for i ← 1 to maska.velicina do
3:         if maska[i] = 0 then
4:             potomak1.geni[i] ← roditelj1.geni[i]
5:             potomak2.geni[i] ← roditelj2.geni[i]
6:         else
7:             potomak1.geni[i] ← roditelj2.geni[i]
8:             potomak2.geni[i] ← roditelj1.geni[i]
9:         end if
10:    end for
11: end function
```

Prva vrijednost u maski je nula, pa prvi potomak nasljeđuje svoj prvi gen od prvog roditelja (1), a drugi potomak svoj prvi gen od drugog roditelja (0). Druga vrijednost u maski je jedinica, pa se drugi gen prvog potomka nasljeđuje od drugog roditelja (1), a drugi gen drugog potomka od prvog roditelja (0). Na isti način nasljeđuju se i ostali geni. Demonstracija uniformnog križanja pomoću maske u programskom jeziku C++ nalazi se u Prilogu 9.11.

3.3 Križanje s tri roditelja

U prethodnim metodama uvijek su se koristila dva slučajno odabrana roditelja $P1$ i $P2$, od kojih su potomci nasljeđivali gene. Međutim, križanje se može provesti i korištenjem tri slučajno odabrana roditelja: $P1$, $P2$ i $P3$. U tom slučaju, ako su i-ti geni roditelja $P1$ i $P2$ isti, potomak nasljeđuje i-ti gen od roditelja $P1$. Ako su i-ti geni roditelja $P1$ i $P2$ različiti, potomak nasljeđuje i-ti gen od roditelja $P3$ (Algoritam 3.5).

Algoritam 3.5 Križanje s tri roditelja.

```
1: function KRIZANJETRIRODITELJA(roditelj1, roditelj2, roditelj3, potomak)
2:     for i ← 1 to roditelj1.geni.kolicina do
3:         if roditelj1.geni[i] = roditelj2.geni[i] then
4:             potomak.geni[i] ← roditelj1.geni[i]
5:         else if roditelj1.geni[i] ≠ roditelj2.geni[i] then
6:             potomak.geni[i] ← roditelj3.geni[i]
7:         end if
8:     end for
9: end function
```

Za demonstraciju pretpostavimo da postoje sljedeći roditelji:

```
Roditelj 1: (1,0,1,0,0)
Roditelj 2: (0,1,1,0,1)
Roditelj 3: (1,0,0,1,1)
```

Križanje s tri prethodno prikazana roditelja rezultira potomkom koji ima sljedeće gene: $(1, 0, 1, 0, 1)$. Prvi i drugi gen roditelja 1 i 2 su različiti, pa potomak nasljeđuje prvi i drugi

gen od roditelja 3 (1 i 0). Treći i četvrti gen su isti, pa potomak nasljeđuje gene od prvog roditelja (1 i 0). Zadnji gen nasljeđuje od trećeg roditelja (1) jer su geni roditelja 1 i 2 različiti. Demonstracija ovog načina križanja u programskom jeziku C++ nalazi se u Prilogu 9.12.

3.4 Aritmetičko križanje

Dok prethodne metode križanja mogu uzrokovati nagle skokove u rješenjima potomaka, aritmetičko križanje omogućuje postupniju i glađu tranziciju između vrijednosti gena roditelja. U kontinuiranim optimizacijskim problemima, poput optimizacije realnih brojeva, ovaj pristup olakšava preciznije istraživanje prostora rješenja. Umjesto da se gen naslijedi isključivo od jednog roditelja, u aritmetičkom križanju koristi se linearna kombinacija gena oba roditelja, određena težinskim faktorom α:

$$Potomak.gen[i] = \alpha \cdot Roditelj1.gen[i] + (1 - \alpha) \cdot Roditelj2.gen[i]$$

Ovisno o vrijednosti težinskog faktora, oba roditelja daju svoj doprinos novom rješenju. Njegova vrijednost obično je u intervalu $[0, 1]$. Primjerice, ukoliko je $\alpha = 0$, tada u potpunosti prevladava gen drugog roditelja, dok za $\alpha = 1$ prevladava gen prvog roditelja. Ako je $\alpha = 0.5$, tada su oba roditelja zastupljena u novom rješenju u jednakim omjerima. Iako je težinski faktor moguće inicijalizirati konstantom, vrlo često se zbog poboljšanja raznolikosti inicijalizira slučajnom vrijednošću.

Općenito, razlika između kontinuiranih i diskretnih optimizacijskih problema svodi se na broj mogućih rješenja. U diskretnim problemima broj mogućih rješenja je konačan. Primjerice, ako se kao rješenje traži cijeli broj u intervalu od 1 do 10, to je diskretan problem s 10 mogućih rješenja $(1, 2, 3, ..., 10)$. U kontinuiranim optimizacijskim problemima broj mogućih rješenja gotovo je beskonačan jer rješenje može biti bilo koji realni broj unutar tog raspona (primjerice, 3.14, 7.34364 itd.) [46].

Algoritam 3.6 Aritmetičko križanje.

```
1: function ARITMETICKOKRIZANJE(roditelj1, roditelj2, α, potomak)
2:     for i ← 1 to roditelj1.geni.kolicina do
3:         potomak.geni[i] ← α · roditelj1.geni[i] + (1 − α) · roditelj2.geni[i]
4:     end for
5: end function
```

Za primjer aritmetičkog križanja opisanog u Algoritmu 3.6 pretpostavimo da pokušavamo optimizirati našu računalnu igru. Strategija igre određena je nizom parametara (realnih brojeva) kojima se definira učestalost napadanja drugog igrača, spremnost na rizike i jačina protunapada.

```
Roditelj 1: (6.0, 4.0, 8.0) - agresivnija strategija
Roditelj 2: (2.0, 1.0, 5.0) - defanzivnija strategija
```

Korištenjem linearne kombinacije gena roditelja sa slučajno odabranim težinskim faktorima $\alpha = 0.3$ i $\alpha = 0.7$ kao rezultat dobivamo sljedeće potomke (strategije):

```
Potomak 1:
  Gen 1: 0.3 * 6.0 + (1 - 0.3) * 2.0 = 3.2
  Gen 2: 0.3 * 4.0 + (1 - 0.3) * 1.0 = 1.9
  Gen 3: 0.3 * 8.0 + (1 - 0.3) * 5.0 = 5.9
Potomak 2:
  Gen 1: 0.7 * 6.0 + (1 - 0.7) * 2.0 = 4.8
  Gen 2: 0.7 * 4.0 + (1 - 0.7) * 1.0 = 3.1
  Gen 3: 0.7 * 8.0 + (1 - 0.7) * 5.0 = 7.1
```

Kada analiziramo rezultate, jasno je da vrijednost težinskog faktora α izravno određuje koliko će potomak naslijediti od svakog roditelja. Za prvog potomka $(3.2, 1.9, 5.9)$, kada je $\alpha = 0.3$, njegova strategija bliža je strategiji drugog roditelja $(2.0, 1.0, 5.0)$. To je zato što manja vrijednost α znači da potomak preuzima veći udio genetskog materijala od drugog roditelja. Primjerice, za prvi gen vrijednost 3.2 mnogo je bliža vrijednosti 2.0 nego 6.0, što potvrđuje dominaciju utjecaja drugog roditelja.

S druge strane, kada je $\alpha = 0.7$, strategija drugog potomka $(4.8, 3.1, 7.1)$ bliža je strategiji prvog roditelja $(6.0, 4.0, 8.0)$. U ovom slučaju, viša vrijednost α znači da potomak preuzima više karakteristika od prvog roditelja, što rezultira vrijednostima bližima njegovim genima. Težinski faktor α tako omogućuje prilagodbu ravnoteže između utjecaja oba roditelja, jasno utječući na konačne rezultate križanja. Demonstracija aritmetičkog križanja u programskom jeziku C++ nalazi se u Prilogu 9.13.

3.5 Heurističko križanje

Način rada heurističkog križanja podsjeća na aritmetičko križanje. Dok aritmetičko križanje, ovisno o težinskom faktoru α, omogućuje odrediti u kojoj mjeri potomak nasljeđuje karakteristike jednog ili drugog roditelja, heurističko križanje uvijek pokušava favorizirati boljeg roditelja. U heurističkom križanju potomka se pokušava što više "udaljiti" od lošijeg roditelja, zbog čega potomak može biti izvan intervala vrijednosti roditelja. U heurističkom križanju koristimo sljedeću formulu:

$$Potomak.gen[i] = Roditelj1.gen[i] + r \cdot (Roditelj1.gen[i] - Roditelj2.gen[i])$$

U formuli se pretpostavlja da i-ti gen drugog roditelja ima manju ili jednaku vrijednost od i-tog gen prvog roditelja ($Roditelj2.gen[i] \leq Roditelj1.gen[i]$). U suprotnom, i-ti gen potomka mogao bi se generirati u pogrešnom smjeru (izraz u zagradi bio bi negativan), udaljavajući potomka od optimalnog rješenja. U takvim slučajevima moguće je preskočiti križanje za taj gen ili zamijeniti uloge roditelja (Algoritam 3.7).

Nadalje, faktor r, slično kao i faktor α, može biti u intervalu $[0, 1]$ i može biti unaprijed definiran ili slučajno odabran. Njime preciznije određujemo vrijednost gena potomka, odnosno njegovu udaljenost od gena lošijeg roditelja.

Za demonstraciju heurističkog križanja pretpostavimo da se koriste unaprijed definirani faktori $r = 0.3$ i $r = 0.7$ te da postoje sljedeća dva roditelja:

Algoritam 3.7 Heuristično križanje.

```
1: function HEURISTICKOKRIZANJE(roditelj1, roditelj2, r, potomak)
2:     for i ← 1 to roditelj1.geni.kolicina do
3:         if roditelj1.geni[i] ≥ roditelj2.geni[i] then
4:             potomak.geni[i] ← roditelj1.geni[i] + r · (roditelj1.geni[i] − roditelj2.geni[i])
5:         else
6:             potomak.geni[i] ← roditelj2.geni[i] + r · (roditelj2.geni[i] − roditelj1.geni[i])
7:         end if
8:     end for
9: end function
```

```
Roditelj 1: (6.0, 4.0, 8.0)
Roditelj 2: (2.0, 5.0, 3.0)
```

Upotrebom Algoritma 3.7 dobivamo dva potomka sa sljedećim genima:

```
Potomak 1:
  Gen 1: 6.0 + 0.3 * (6.0 - 2.0) = 6.0 + 1.2 = 7.2
  Gen 2: 5.0 + 0.3 * (5.0 - 4.0) = 5.0 + 0.3 = 5.3
  Gen 3: 8.0 + 0.3 * (8.0 - 3.0) = 8.0 + 1.5 = 9.5
Potomak 2:
  Gen 1: 6.0 + 0.7 * (6.0 - 2.0) = 6.0 + 2.8 = 8.8
  Gen 2: 5.0 + 0.7 * (5.0 - 4.0) = 5.0 + 0.7 = 5.7
  Gen 3: 8.0 + 0.7 * (8.0 - 3.0) = 8.0 + 3.5 = 11.5
```

Potrebno je primijetiti da drugi gen drugog roditelja ima veću vrijednost od istog gena prvog roditelja. U tom slučaju zamijenili smo uloge roditelja jer bi u suprotnom generirani gen potomaka otišao u neželjenom smjeru. Također, dok bi aritmetičko križanje s faktorom α "privuklo" potomka bliže jednom ili drugom roditelju, faktor r je oba potomka "odvukao" dalje od lošijeg roditelja. Štoviše, vrijednosti gena oba potomka su izvan granica vrijednosti gena njihovih roditelja. Zbog toga potreban je oprez pri heurističkom križanju jer novonastali potomak može rezultirati i genima koji su izvan prihvatljivih granica zadanog problema. U tom slučaju mogu se primijeniti neke od tehnika normalizacije rezultata križanja čime se dobivene vrijednosti pokušava očuvati unutar tih granica. Demonstracija heurističkog križanja u programskom jeziku C++ nalazi se u Prilogu 9.14.

Unatoč navedenim nedostacima, heurističko križanje ima određene prednosti u odnosu na aritmetičko križanje. Dok aritmetičko križanje dovodi do "prosječnih" potomaka,heurističko križanje najčešće daje kvalitetnije potomke koji naginju kvalitetnijem roditelju. Zbog toga, heurističko križanje brže konvergira prema optimalnom rješenju, osobito u slučajevima kada je jedan roditelj značajno kvalitetniji od drugog.

3.6 Djelomično mapirano križanje (PMX)

Za razliku od prethodnih metoda križanja koje se uglavnom primjenjuju na binarne i numeričke reprezentacije rješenja, *djelomično mapirano križanje* (engl. *Partially Mapped*

Crossover, PMX) koristi se za probleme s permutacijskom reprezentacijom, primjerice u problemu trgovačkog putnika. PMX se izvodi u nekoliko koraka opisanih Algoritmom 3.8:

Algoritam 3.8 Djelomično mapirano križanje (PMX).

function PMX($roditelj1$, $roditelj2$, $potomak1$, $potomak2$)
  $n \leftarrow roditelj1.geni.kolicina$
  $potomak1.geni[1..n] \leftarrow$ prazno
  $potomak2.geni[1..n] \leftarrow$ prazno
  mapa1 $\leftarrow$ prazan skup parova
  mapa2 $\leftarrow$ prazan skup parova
  ▷ Odredi dvije nasumične točke križanja.
  $t_1 \leftarrow$ SLUCAJANBROJ$(1, n)$
  $t_2 \leftarrow$ SLUCAJANBROJ$(1, n)$
  if $t_1 > t_2$ **then**
    ZAMIJENI(t_1, t_2)
  end if
  ▷ Razmijeni srednji segment i uspostavi mapiranje između vrijednosti.
  for $i \leftarrow t_1$ **to** t_2 **do**
    $potomak1.geni[i] \leftarrow roditelj2.geni[i]$
    $potomak2.geni[i] \leftarrow roditelj1.geni[i]$
    mapa1.dodaj(roditelj2.geni$[i]$, roditelj1.geni$[i]$)
    mapa2.dodaj(roditelj1.geni$[i]$, roditelj2.geni$[i]$)
  end for
  ▷ Popuni pozicije izvan segmenta, eliminirajući duplikate pomoću mapiranja.
  for $i \leftarrow 1$ **to** n **do**
    if $i < t_1$ **or** $i > t_2$ **then** ▷ Za potomka 1: pokušaj izravno kopirati gen iz roditelja 1.
      $gen1 \leftarrow roditelj1.geni[i]$
      while $gen1 \in potomak1.geni[t_1..t_2]$ **do** ▷ Duplikat? Pronađi zamjenu preko mape
        $gen1 \leftarrow$ mapa1[gen1]
      end while
      $potomak1.geni[i] \leftarrow gen1$
      ▷ Za potomka 2: isto, ali u suprotnom smjeru.
      $gen2 \leftarrow roditelj2.geni[i]$
      while $gen2 \in potomak2.geni[t_1..t_2]$ **do** ▷ Duplikat? Pronađi zamjenu preko mape.
        $gen2 \leftarrow$ mapa2[gen2]
      end while
      $potomak2.geni[i] \leftarrow gen2$
    end if
  end for
end function

Prvo se nasumično odaberu dvije točke križanja koje definiraju srednji segment kromosoma. Taj se segment zatim razmijeni između roditelja. Time mogu nastati duplikati u kromosomu potomka, što nije dozvoljeno kod permutacijskih problema poput problema trgovačkog putnika gdje svaka vrijednost mora biti jedinstvena. Da bi se to izbjeglo, PMX uspostavlja mapiranje između vrijednosti u razmijenjenim segmentima. Svaka vrijednost iz segmenta jednog roditelja povezuje se s odgovarajućom vrijednošću iz segmenta drugog. Prilikom popunjavanja ostatka kromosoma potomka, ako je određeni gen već prisutan u srednjem segmentu koristi se ovo mapiranje da bi se pronašla odgovarajuća zamjena,

odnosno gen koji nije već korišten. Zamjena može uključivati više koraka sve dok se ne pronađe vrijednost koja zadovoljava uvjet unikatnosti.

Za potrebe demonstracije pretpostavimo da imamo dva roditelja u problemu trgovačkog putnika, pri čemu su gradovi u ruti koju treba obići označeni cjelobrojnim vrijednostima:

- Roditelj 1: `1 2 3 4 5 6 7 8 9`
- Roditelj 2: `5 4 6 9 2 1 7 8 3`

Nasumično odabrane točke križanja neka su na pozicijama 3 do 6:

- Roditelj 1: `1 2 |3 4 5 6| 7 8 9`
- Roditelj 2: `5 4 |6 9 2 1| 7 8 3`

Korak 1: Razmjena srednjih segmenata

- Potomak 1: `_ _ 6 9 2 1 _ _ _`
- Potomak 2: `_ _ 3 4 5 6 _ _ _`

Korak 2: Uspostava mapiranja između razmijenjenih vrijednosti.

$$3 \leftrightarrow 6$$
$$4 \leftrightarrow 9$$
$$5 \leftrightarrow 2$$
$$6 \leftrightarrow 1$$

Korak 3: Popunjavanje preostalih pozicija
Za potomka 1 popunjavaju se pozicije izvan srednjeg segmenta (1, 2, 7, 8, 9) genima iz roditelja 1: `1, 2, 7, 8, 9`. Budući da neki od tih gena već postoje u srednjem segmentu potomka (`6, 9, 2, 1`), koriste se mapiranja:

- `1` je već prisutan $\Rightarrow$ `1` $\rightarrow$ `6`, ali `6` također postoji $\Rightarrow$ `6` $\rightarrow$ `3` $\Rightarrow$ upisuje se `3`
- `2` je već prisutan $\Rightarrow$ `2` $\rightarrow$ `5` $\Rightarrow$ upisuje se `5`
- `7`, `8` nisu u srednjem segmentu $\Rightarrow$ upisuju se izravno
- `9` je prisutan $\Rightarrow$ `9` $\rightarrow$ `4` $\Rightarrow$ upisuje se `4`

Potomak 1 nakon popunjavanja: `3 5 6 9 2 1 7 8 4`

Za potomka 2 koriste se geni iz roditelja 2: `5, 4, 7, 8, 3`, a srednji segment je `3, 4, 5, 6`.

- `5` $\Rightarrow$ `5` $\rightarrow$ `2` $\Rightarrow$ upisuje se `2`
- `4` $\Rightarrow$ `4` $\rightarrow$ `9` $\Rightarrow$ upisuje se `9`

- 7, 8 $\Rightarrow$ upisuju se izravno
- 3 $\Rightarrow$ 3 $\rightarrow$ 6, koji je već prisutan $\Rightarrow$ 6 $\rightarrow$ 1 $\Rightarrow$ upisuje se 1

Potomak 2 nakon popunjavanja: 2 9 3 4 5 6 7 8 1. Konačno za roditelje

- Roditelj 1: 1 2 3 4 5 6 7 8 9
- Roditelj 2: 5 4 6 9 2 1 7 8 3

PMX križanjem dobiju se potomci

- Potomak 1: 3 5 6 9 2 1 7 8 4
- Potomak 2: 2 9 3 4 5 6 7 8 1

Cjelovita demonstracija djelomično mapiranog križanja u programskom jeziku C++ nalazi se u Prilogu 9.15.

Osim djelomično mapiranog križanja (PMX), u literaturi su predložene i brojne druge metode križanja posebno prilagođene permutacijskim reprezentacijama, kao što su *Order Crossover (OX)* [47], *Cycle Crossover (CX)* [37] i *Edge Recombination Crossover (ERX)* [48]. Različitim pristupima kombiniranju roditelja one u nekim slučajevima mogu bolje očuvati korisne dijelove rješenja (poput redoslijeda elemenata, njihovih relativnih položaja ili susjednih veza), što u nekim problemima može uzrokovati bržu konvergenciju i kvalitetnije potomke.

POGLAVLJE 4

Mutacija

U procesu križanja odabranih jedinki (*roditelja*) nastaju *djeca* koja, ovisno o korištenoj metodi križanja, nasljeđuju određene karakteristike svojih *roditelja*. U sljedećem koraku evolucijskog procesa provodi se mutacija (Slika 1.2), odnosno evolucijski operator kojim se, ovisno o vjerojatnosti mutacije p_m (engl. *probability of mutation*), slučajnim odabirom mijenjaju vrijednosti gena pojedinih rješenja. Općenito, mutacija sprječava preranu konvergenciju algoritma u lokalnom optimumu, povećava raznolikost populacije i omogućuje istraživanje novih dijelova prostora rješenja. Vjerojatnost mutacije obično je vrlo mala (najčešće $\leq$ 1%) da bi se očuvala dobra rješenja u populaciji, dok veće vrijednosti vjerojatnosti mutacije najčešće dovode do duže pretrage optimalnog rješenja. Štoviše, s rastom vjerojatnosti mutacije evolucijski algoritam sve više poprima karakteristike slučajne umjesto ciljane pretrage.

Ovisno o tipu rješenja, postoje različite vrste mutacije, poput uniformne mutacije [49], Gaussove mutacije [21], mutacije zamjenom [37], inverzijske [50] i adaptivne mutacije [51]. Uniformna i Gaussova mutacija najčešće se koriste za rješenja reprezentirana realnim brojevima i binarnim nizovima. Mutacija zamjenom i inverzijska mutacija često se primjenjuju u permutacijskim problemima poput problema trgovačkog putnika. Adaptivna mutacija često se koristi u problemima koji imaju više lokalnih optimuma da bi se izbjegla prerana konvergencija algoritma prema neoptimalnom rješenju.

4.1 Uniformna mutacija

U problemima gdje su geni rješenja predstavljeni binarnim, realnim ili diskretnim vrijednostima (predefinirani skup) često se koristi uniformna mutacija. Ona predstavlja jednostavan oblik mutacije koji koristi parametar p_m da bi za svaki gen odlučila hoće li biti podložan promjeni. Za svaki gen generira se slučajna vrijednost, a ako je ona manja ili jednaka p_m, gen mutira.

Rezultat mutacije može ovisiti o tipu vrijednosti gena, njegovoj trenutnoj vrijednosti te skupu svih mogućih vrijednosti koje taj gen može poprimiti (Algoritam 4.1). Primjerice,

Algoritam 4.1 Uniformna mutacija (za binarne, realne i diskretne vrijednosti).

function UNIFORMNAMUTACIJA(jedinka, $p_m, tip\_gena, V, X_{min}, X_{max}$)
  for $i \leftarrow 1$ **to** jedinka.geni.kolicina **do**
    if random$(0,1) \leq p_m$ **then**
      if $tip\_gena =$ binarni **then** ▷ Binarni .
        $jedinka.geni[i] \leftarrow 1 - jedinka.geni[i]$
      else if $tip\_gena =$ realni **then** ▷ Realni geni.
        $jedinka.geni[i] \leftarrow$ random(X_{min}, X_{max})
      else ▷ Diskretni geni.
        $jedinka.geni[i] \leftarrow$ random_choice$(V \setminus \{jedinka.geni[i]\})$
      end if
    end if
  end for
end function

ako su geni realni brojevi definirani u intervalu $[X_{\min}, X_{\max}]$, rezultat mutacije može biti bilo koji realni broj unutar tog raspona. Ako geni imaju diskretne vrijednosti, uniformna mutacija rezultira slučajno odabranom vrijednošću iz skupa V, pri čemu se osigurava da nova vrijednost bude različita od originalne. Binarni geni mutiraju na isti način, s time da je nova vrijednost uvijek inverzna od prethodne ($0 \to 1$, $1 \to 0$). Budući da diskretni i binarni geni obično imaju mali skup mogućih vrijednosti, potrebno je dodatno provjeriti i osigurati da mutacija zaista uvodi novi genetski materijal usporedbom originalne i mutirane vrijednosti gena.

Primjerice, demonstrirajmo mutaciju gena koji imaju diskretne vrijednosti:

```
class JedinkaDiskretna {
public:
    std::vector<int> geni;
    std::vector<int> V; // moguce vrijednosti gena

    JedinkaDiskretna(const std::vector<int>& geni, const std::vector<int>& V)
        : geni(geni), V(V) {
    }
};
```

Prikazana klasa $JedinkaDiskretna$ sadrži vektor cjelobrojnih gena $geni$ te vektor V koji sadrži skup svih mogućih vrijednosti koje pojedini gen može poprimiti. Sukladno prethodno definiranoj klasi, implementirana je sljedeća funkcija za mutaciju njenih gena (Primjer 4.1).

Primjer 4.1: Implementacija uniformne mutacije jedinke s diskretnim genima u programskom jeziku C++.

```
void uniformnaMutacijaDiskretna(JedinkaDiskretna* jedinka, double pm) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_real_distribution<double> distr(0.0, 1.0);

    for (size_t i = 0; i < jedinka->geni.size(); i++) {
        if (distr(gen) <= pm) { // da li ce doci do mutacije?
            std::uniform_int_distribution<size_t> indexDistr(0, jedinka->V.size() - 1);
```

```
            int novaVrijednost;
            do {
                novaVrijednost = jedinka->V[indexDistr(gen)];
            } while (novaVrijednost == jedinka->geni[i]); // osigurava promjenu vrijednosti
            jedinka->geni[i] = novaVrijednost;
        }
    }
}
```

Za svaki gen jedinke prvo provjeravamo hoće li doći do mutacije, što ovisi o slučajno generiranoj vrijednosti i parametru p_m. Ako se mutacija treba dogoditi, potrebno je odabrati novu vrijednost za ciljani gen. Budući da geni poprimaju diskretne vrijednosti iz skupa definiranog vektorom V, nova se vrijednost bira slučajnim odabirom indeksa u tom vektoru. Taj se postupak ponavlja sve dok nova vrijednost ciljanog gena ne bude različita od njegove originalne vrijednosti.

Za testiranje mutacije možemo upotrijebiti sljedeći programski odsječak:

```
std::vector<int> V = { 0, 2, 4, 6, 8 }; // skup mogucih vrijednosti gena
JedinkaDiskretna diskretnaJedinka({ 0, 2, 0, 6, 8 }, V);
double pm = 0.2; // vjerojatnost mutacije (20%)

std::cout << "Prije mutacije: ";
for (int gen : diskretnaJedinka.geni) std::cout << gen << " ";
std::cout << "\n";

uniformnaMutacijaDiskretna(&diskretnaJedinka, pm);
std::cout << "Nakon mutacije: ";
for (int gen : diskretnaJedinka.geni) std::cout << gen << " ";
```

Kreirana je jedinka $diskretnaJedinka$ s vrijednostima gena $(0, 2, 0, 6, 8)$. Budući da se radi o jedinki s diskretnim genima, prethodno je definiran skup V koji sadrži sve moguće vrijednosti koje pojedini gen može poprimiti, odnosno $(0, 2, 4, 6, 8)$. Nadalje, uniformna mutacija ovisi o parametru p_m, čija vrijednost iznosi 0.2 (20%). U svrhu demonstracije, vjerojatnost mutacije u ovom slučaju namjerno je postavljena na puno višu vrijednost nego što bi to obično bio slučaj (≤ 0.01, odnosno $\leq 1\%$).

Jedan od mogućih rezultata izvršavanja prethodnog programskog odsječka je sljedeći:

```
Prije mutacije: 0 2 0 6 8
Nakon mutacije: 0 8 0 6 8
```

Potrebno je primijetiti da je mutirao samo drugi gen, odnosno njegova vrijednost se promijenila s 2 na 8. Time je na jednom mjestu u jedinki dodan novi genetski materijal, pri čemu je nova vrijednost gena dobivena slučajnim odabirom iz prethodno definiranog skupa V. Na sličan način možemo primijeniti uniformnu mutaciju i na gene s binarnim i realnim vrijednostima, kao što je prikazano u Prilogu 9.16.

Implementacija mutacije često je vrlo slična generiranju slučajne populacije u inicijalizacijskoj fazi evolucijskog algoritma (Primjer 4.2).

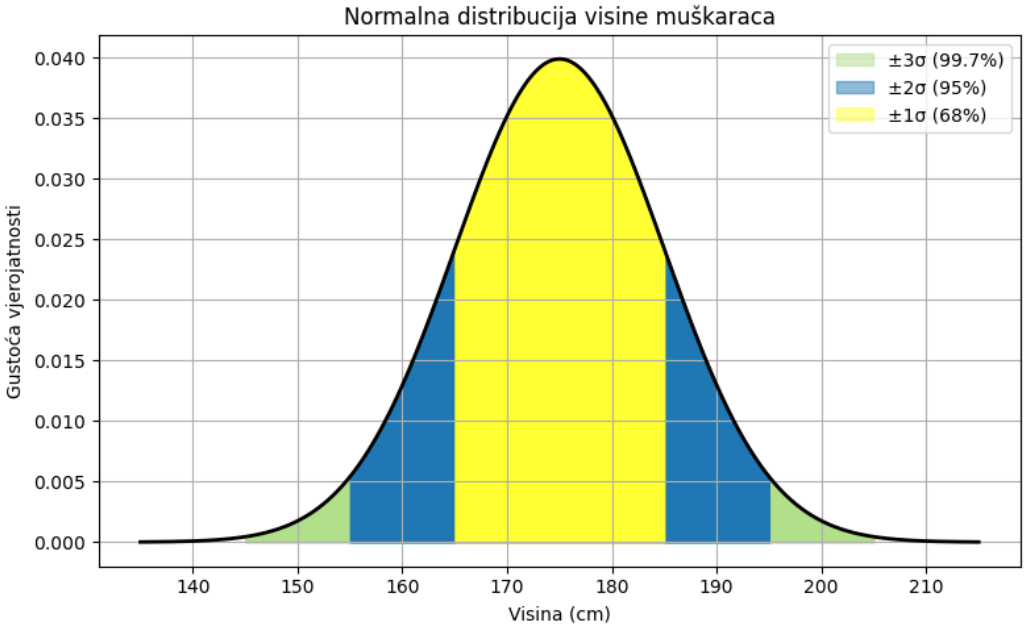
Primjer 4.2: Generiranje slučajne populacije diskretnih jedinki u programskom jeziku C++.

```
std::vector<JedinkaDiskretna> slucajnaPopulacija(int velicina, int brojGena, const std::
    vector<int>& V) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution<size_t> indexDistr(0, V.size() - 1); // raspon indeksa
    std::vector<JedinkaDiskretna> populacija; // rezultantna populacija

    for (int i = 0; i < velicina; ++i) {
        std::vector<int> geni;
        for (int j = 0; j < brojGena; ++j)
            geni.push_back(V[indexDistr(gen)]); // slucajna vrijednost gena iz skupa V
        populacija.emplace_back(geni, V);
    }
    return populacija;
}
```

Dok mutacija gena najčešće ovisi o vrlo maloj vrijednosti parametra p_m, pri generiranju slučajne populacije svaki gen pojedine jedinke uvijek se inicijalizira slučajnom vrijednošću.

4.2 Gaussova mutacija

Gaussova mutacija, temeljena na Gaussovoj (normalnoj) distribuciji [52], mijenja ciljane gene tako što se generirane slučajne vrijednosti dodjeljuju postojećim vrijednostima gena. Pri generiranju slučajnih vrijednosti koristi se Gaussova distribucija, a konačne vrijednosti gena moraju biti ograničene na interval $[X_{\min}, X_{\max}]$. U Gaussovoj distribuciji najčešće se koristi srednja vrijednost $\mu = 0$, dok standardna devijacija σ ovisi o problemu i može se prilagoditi prema veličini prostora za pretraživanje. Ovaj oblik mutacije najprikladniji je za probleme u kojima su poželjne manje promjene u vrijednostima gena ili za primjenu u tehnikama lokalne pretrage.



Slika 4.1: Primjer Gaussove distribucije visine muškaraca ($\mu = 175$ cm, $\sigma = 10$ cm).

Razlog zbog kojeg su promjene u Gaussovoj mutaciji najčešće male može se izravno iščitati iz primjera prikazanog na Slici 4.1, koja ilustrira normalnu distribuciju visina muškaraca. Sa srednjom vrijednošću od 175 cm i standardnom devijacijom od ± 10 cm, prema pravilu $68 - 95 - 99.7$ [53], možemo očekivati da će otprilike 68% populacije imati visinu između 165 cm i 185 cm (područje označeno žutom bojom na slici). To znači da je generiranje visina unutar ovog raspona, koji predstavlja relativno malo odstupanje od prosjeka, znatno vjerojatnije u usporedbi s udaljenijim vrijednostima. Isto vrijedi i za Gaussovu mutaciju: budući da se promjene generiraju prema normalnoj distribuciji s navedenim parametrima, male promjene mnogo su češće. Unatoč tome, Gaussova mutacija povremeno može generirati i veće promjene (područja izvan žutog), što je korisno za izbjegavanje lokalnih ekstrema i istraživanje šireg prostora rješenja.

Algoritam 4.2 Gaussova mutacija.

function GAUSSOVAMUTACIJA(jedinka, $p_m, \mu, \sigma, X_{\min}, X_{\max}$)
    for $i \leftarrow 1$ **to** jedinka.geni.količina **do**
        if random$(0,1) \leq p_m$ **then** ▷ Provjera vjerojatnosti mutacije.
            $promjena \leftarrow$ randomNormalDistr(μ, σ) ▷ Generiraj Gaussov slučajni broj s μ i σ.
            $novaVrijednost \leftarrow jedinka.geni[i] + promjena$
            $jedinka.geni[i] \leftarrow \max(X_{\min}, \min(novaVrijednost, X_{\max}))$ ▷ Interval $[X_{\min}, X_{\max}]$
        end if
    end for
end function

Algoritam 4.2 opisuje jedan od mogućih načina primjene Gaussove mutacije. Prvo, za svaki gen jedinke provjerava se hoće li biti mutiran na temelju vjerojatnosti mutacije p_m. Ako se gen treba mutirati, generira se slučajna vrijednost ($promjena$) korištenjem Gaussove (normalne) distribucije s parametrima μ i σ, nakon čega se ta vrijednost dodaje trenutnoj vrijednosti gena. Nova vrijednost gena zatim se ograničava na zadani interval $[X_{\min}, X_{\max}]$ čime se osigurava da ostane unutar dopuštenog raspona.

U programskom jeziku C++ vrlo je jednostavno implementirati Gaussovu mutaciju, budući da sama standardna biblioteka pruža podršku za Gaussovu (normalnu) distribuciju korištenjem klase `std::normal_distribution`.

Pretpostavimo da imamo sljedeću definiciju jedinke $VisinaMuskaraca$:

```
class VisinaMuskaraca {
public:
    std::vector<int> geni;
    int Xmin, Xmax; // interval vrijednosti
    VisinaMuskaraca(const std::vector<int>& geni, int Xmin, int Xmax)
        : geni(geni), Xmin(Xmin), Xmax(Xmax) {
    }
};
```

Jedinka sadrži niz (vektor) visina muškaraca te njihove granične vrijednosti $Xmin$ i $Xmax$. Implementacija Gaussove mutacije ove jedinke u programskom jeziku C++ prikazana je Primjerom 4.3.

Primjer 4.3: Implementacija Gaussove mutacije u programskom jeziku C++.

```
void gaussMutacija(VisinaMuskaraca* jedinka, double pm, double srednjaVr, double stdDev) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<> normalDist(srednjaVr, stdDev); // Gaussova distribucija!
    std::uniform_real_distribution<> uniformDist(0.0, 1.0);

    for (size_t i = 0; i < jedinka->geni.size(); ++i) {
        // Za svaki gen odlucujemo hoce li se mutirati na temelju pm
        if (uniformDist(gen) <= pm) {
            int promjena = static_cast<int>(normalDist(gen));  // Slucajna promjena
            int novaVr = jedinka->geni[i] + promjena;

            // Osiguravamo da nova vrijednost ostane unutar dozvoljenog opsega
            novaVr = std::max(jedinka->Xmin, std::min(novaVr, jedinka->Xmax));
            jedinka->geni[i] = novaVr;
        }
    }
}
```

Za generiranje slučajnih promjena koristi se Gaussova (normalna) distribucija. Dobivena promjena dodaje se trenutnoj vrijednosti mutiranog gena. Ukoliko nova vrijednost, nastala zbrajanjem promjene i trenutne vrijednosti gena, prelazi granice intervala $[X_{\min}, X_{\max}]$, tada gen poprima vrijednost bliže granice intervala (tj. $X_{\min}$ ili $X_{\max}$). Navedenu mutaciju možemo testirati na sljedećem primjeru:

```
std::vector<int> geni = { 170, 180, 190, 165, 185 };
VisinaMuskaraca vM(geni, 120, 220); // Xmin = 120, Xmax = 220

std::cout << "Prije mutacije: ";
for (int gene : vM.geni) std::cout << gene << " ";
std::cout << std::endl;

// Parametri: pm = 0.2, srednja vrijednost = 0, std. dev. = 10
gaussMutacija(&vM, 0.2, 0, 10);

std::cout << "Nakon mutacije: ";
for (int gene : vM.geni) std::cout << gene << " ";
std::cout << std::endl;
```

Pri testiranju standardno je korištena srednja vrijednost $\mu = 0$, što znači da je jednako vjerojatno da će mutacija povećati ili smanjiti vrijednost gena. Sukladno primjeru, standardna devijacija σ iznosi 10. Kao rezultat izvođenja koda dobiveni su sljedeći rezultati:

```
Prije mutacije: 170 180 190 165 185
Nakon mutacije: 165 180 190 182 185
```

Primjećujemo da su samo dva od pet gena mutirala, što je očekivano s obzirom na vjerojatnost mutacije p_m koja u ovom slučaju iznosi 0.2. Prvi gen mutirao je na manju vrijednost (sa 170 na 165), dok je četvrti gen mutirao na veću vrijednost (sa 165 na 182). Obje su promjene unutar približno dvije standardne devijacije ($\sigma = 10$) od početnih vrijednosti, što je u skladu s karakteristikama Gaussove distribucije. Također, sve vrijednosti ostaju unutar očekivanog raspona za visinu odraslih muškaraca, što ukazuje na učinkovitost ograničavanja na interval $[X_{\min} = 120, X_{\max} = 220]$. Potpuni primjer demonstracije ove mutacije u programskom jeziku C++ nalazi se u Prilogu 9.17.

4.3 Mutacija zamjenom

U prethodno opisanim metodama mutacija djeluje na način da, ovisno o vjerojatnosti mutacije p_m, dodjeljuje slučajne vrijednosti genima. Međutim, mutacija zamjenom, ovisno o vjerojatnosti mutacije p_m, zamjenjuje vrijednosti dvaju gena unutar iste jedinke (Algoritam 4.3). Ovaj oblik mutacije pogodniji je za probleme permutacijskog tipa, poput problema trgovačkog putnika, problema optimizacije rasporeda poslova i sličnih.

Algoritam 4.3 Mutacija zamjenom.

function MUTACIJAZAMJENOM(jedinka, p_m)
  for $i \leftarrow 1$ **to** jedinka.geni.kolicina **do**
    if random$(0,1) \leq p_m$ **then** ▷ Provjera vjerojatnosti mutacije.
      repeat
        $j \leftarrow$ random$(1,$jedinka.geni.kolicina$)$
      until $j \neq i$ ▷ Osiguravanje različitog gena.
      swap$(jedinka.geni[i], jedinka.geni[j])$ ▷ Zamjena gena.
    end if
  end for
end function

Algoritam prolazi kroz sve gene jedinke. Za svaki gen, s vjerojatnošću p_m, odlučuje se hoće li se izvršiti mutacija. Ako se mutacija treba dogoditi, bira se slučajni gen iz iste jedinke, osiguravajući da je različit od trenutnog. Zatim se vrijednosti ova dva gena zamjenjuju. Iako problemi permutacijskog tipa imaju diskretan skup vrijednosti, na ovaj način se i u njima može pridonijeti raznolikosti populacije.

Za primjer imamo sljedeću definiciju jedinke koja predstavlja rješenje problema trgovačkog putnika. Cilj je u što kraćem vremenu posjetiti sva zadana mjesta, gdje je svako mjesto označeno jedinstvenim cijelim brojem.

```
class TrgovackiPutnik {
public:
    std::vector<int> ruta; // Geni predstavljaju redoslijed gradova
    TrgovackiPutnik(const std::vector<int>& ruta) : ruta(ruta) {}
};
```

Sukladno Algoritmu 4.3, implementirana je sljedeća C++ funkcija za mutaciju zamjenom (Primjer 4.4):

Primjer 4.4: Implementacija mutacije zamjenom u programskom jeziku C++.

```
void mutacijaZamjenom(TrgovackiPutnik* jedinka, double pm) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> uniformDist(0.0, 1.0);
    std::uniform_int_distribution<> intDist(0, jedinka->ruta.size() - 1);

    for (size_t i = 0; i < jedinka->ruta.size(); ++i) {
        // Za svaki grad odlucujemo hoce li se mutirati na temelju pm
        if (uniformDist(gen) <= pm) {
            int j;
```

```
            do {
                j = intDist(gen);
            } while (j == i);  // Osiguravamo da je j razlicit od i

            // Zamjena gradova
            std::swap(jedinka->ruta[i], jedinka->ruta[j]);
        }
    }
}
```

Zanimljivo je primijetiti da pojedini gen može biti slučajno mutiran čak i u slučaju da prethodno nije izabran za mutaciju. Primjerice, iako je za prvi gen odlučeno da neće biti mutiran, on svejedno slučajnim odabirom može biti izabran za zamjenu pri mutiranju nekog drugog gena. Štoviše, to se može dogoditi i više puta.

Mutaciju zamjenom sada možemo testirati na sljedećem primjeru:

```
std::vector<int> ruta = { 0, 1, 2, 3, 4, 5 }; // Gradovi oznaceni brojevima 0-5
TrgovackiPutnik tp(ruta);

std::cout << "Prije mutacije: ";
for (int grad : tp.ruta) std::cout << grad << " ";
std::cout << std::endl;

mutacijaZamjenom(&tp, 0.2); // Parametar: pm = 0.2
std::cout << "Nakon mutacije: ";
for (int grad : tp.ruta) std::cout << grad << " ";
```

Jedan od mogućih rezultata izvršavanja je sljedeći:

```
Prije mutacije: 0 1 2 3 4 5
Nakon mutacije: 5 1 2 3 0 4
```

Iz rezultata je vidljivo da se mutacija zamjenom dogodila nekoliko puta. Vrijednost nula sada se nalazi na petom mjestu, vrijednost četiri na zadnjem (šestom), a vrijednost pet kao vrijednost prvog gena u nizu. Potpuni primjer mutacije zamjenom u programskom jeziku C++ nalazi se u Prilogu 9.18.

4.4 Inverzijska mutacija

Inverzijska mutacija oblik je mutacije pogodan za probleme permutacijskog tipa. Za razliku od mutacije zamjenom, koja za svaki gen provjerava hoće li mutirati, inverzijska mutacija tu provjeru radi samo jednom za cijelu jedinku. Ukoliko se treba dogoditi inverzijska mutacija, biraju se dvije slučajne pozicije gena u jedinki te se invertira redoslijed svih gena između njih, uključujući i odabrane pozicije (Algoritam 4.4).

Kao i u slučaju mutacije zamjenom, inverzijsku mutaciju također možemo demonstrirati na problemu trgovačkog putnika. Umjesto mutacije zamjenom, potrebno je implementirati funkciju za inverzijsku mutaciju (Primjer 4.5, Prilog 9.19).

Algoritam 4.4 Inverzijska mutacija.

```
1:  function INVERZIJSKAMUTACIJA(jedinka, p_m)
2:      if random(0, 1) ≤ p_m then                    ▷ Provjera vjerojatnosti mutacije.
3:          repeat
4:              i ← random(1, jedinka.geni.kolicina)
5:              j ← random(1, jedinka.geni.kolicina)
6:          until i ≠ j                               ▷ Osiguravanje različitih indeksa.
7:          if i > j then
8:              SWAP(i, j)                            ▷ Osiguravanje ispravnog poretka.
9:          end if
10:         REVERSE(jedinka.geni[i : j])              ▷ Inverzija odabranog segmenta.
11:     end if
12: end function
```

Primjer 4.5: Implementacija inverzijske mutacije u programskom jeziku C++.

```
void inverzijskaMutacija(TrgovackiPutnik* jedinka, double pm) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> uniformDist(0.0, 1.0);

    // Odlucujemo hoce li se mutacija dogoditi na temelju pm
    if (uniformDist(gen) <= pm) {
        std::uniform_int_distribution<> intDist(0, jedinka->ruta.size() - 1);

        int start, end;
        // Osiguravamo da su start i end razliciti
        do {
            start = intDist(gen);
            end = intDist(gen);
        } while (start == end);

        // Osiguravamo da je start manji od end
        if (start > end) std::swap(start, end);

        // Inverzija
        std::reverse(jedinka->ruta.begin() + start, jedinka->ruta.begin() + end + 1);
    }
}
```

Ukoliko inverzijsku mutaciju testiramo na istom primjeru kao i mutaciju zamjenom, jedan od mogućih rezultata je sljedeći:

```
Prije mutacije: 0 1 2 3 4 5
Nakon mutacije: 0 3 2 1 4 5
```

Inverzijska mutacija se dogodila, pri čemu su za inverziju odabrani geni od drugog do četvrtog (vrijednosti od 1 do 3). Stoga se vrijednosti tih gena u obrnutom redoslijedu pojavljuju nakon mutacije. Važno je primijetiti da su geni izvan odabranog raspona (0, 4 i 5) ostali nepromijenjeni na svojim originalnim pozicijama.

4.5 Adaptivna mutacija

Neovisno o odabranom tipu mutacije, vjerojatnost mutacije p_m može biti adaptivna. Intenzitet mutacije može se prilagođavati kvaliteti jedinke te stagnaciji vrijednosti dobrote

da bi se izašlo iz područja lokalnog optimuma. Adaptivna mutacija često je u upotrebi kod problema koji imaju višestruke lokalne ekstreme. Najčešće se u početku koriste veće vjerojatnosti mutacije radi opsežnijeg istraživanja prostora svih mogućih rješenja, dok se u kasnijim fazama evolucijskog algoritma vjerojatnost mutacije smanjuje da bi se omogućilo fino podešavanje dobrih rješenja i konvergencija prema globalnom optimumu.

Algoritam 4.5 Pojednostavljeni opći oblik adaptivne mutacije.

```
1: function AdaptivnaMutacija(jedinka, globalniNapredak)
2:     jedinka.stopaMutacije ← PrilagodiStopu(jedinka.dobrota, globalniNapredak)
3:     for i ← 1 to jedinka.geni.kolicina do                ▷ Iteracija kroz sve gene.
4:         if random(0, 1) ≤ jedinka.stopaMutacije then     ▷ Ako gen treba mutirati.
5:             jedinka.geni[i] ← NovaVrijednost(jedinka.geni[i])   ▷ Dodijeli novu vrijednost.
6:         end if
7:     end for
8: end function
```

Opći oblik adaptivne mutacije prikazan je u Algoritmu 4.5. U njemu je vidljivo da vjerojatnost mutacije ne mora biti ista za sve jedinke u populaciji, već se može pratiti za svaku jedinku posebno. Štoviše, ona se može dinamički odrediti ovisno o dobroti jedinke i globalnom napretku populacije (razlici između najbolje dobrote u trenutnoj i prethodnoj generaciji). Ukoliko se dobrota populacije s generacijama ne poboljšava, vjerojatnost mutacije se postepeno povećava da bi se izbjegli lokalni optimumi. Zatim se za svaki gen (ili cijelu jedinku, ovisno o odabranoj metodi mutacije) provjerava hoće li mutirati te se spremaju generirane promjene.

U svrhu demonstracije adaptivne mutacije neka postoji sljedeća definicija jedinke:

```cpp
class JedinkaRealna {
public:
    std::vector<double> geni;
    double minVal, maxVal;
    double dobrota;
    double pm; // vjerojatnost mutacije

    JedinkaRealna(const std::vector<double>& geni, double minVal, double maxVal)
        : geni(geni), minVal(minVal), maxVal(maxVal), dobrota(0.0), pm(0.01) {}
};
```

Osim vrijednosti dobrote, svaka realna jedinka sada ima i svoju vlastitu vjerojatnost mutacije. Pri inicijalizaciji, podrazumijevana vrijednost dobrote jedinke iznosi 0, dok vjerojatnost mutacije iznosi 0.01. Jedan od pristupa adaptivnoj uniformnoj mutaciji ovakve jedinke prikazan je u Primjeru 4.6.

Primjer 4.6: Implementacija adaptivne uniformne mutacije u programskom jeziku C++.

```cpp
void adaptivnaMutacijaRealna(JedinkaRealna& jedinka, double globalniNapredak) {
    static std::random_device rd;
    static std::mt19937 rng(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);
    std::uniform_real_distribution<> genDis(jedinka.minVal, jedinka.maxVal);

    // Prilagodba vjerojatnosti mutacije
```

```
8       if (globalniNapredak < 0.001)
9           jedinka.pm = std::min(0.1, jedinka.dobrota * 1.5);
10      else
11          jedinka.pm = std::max(0.001, jedinka.dobrota * 0.9);
12
13      // Mutacija gena
14      for (auto& gen : jedinka.geni)
15          if (dis(rng) <= jedinka.pm)
16              gen = genDis(rng);
17  }
```

Ukoliko je globalni napredak populacije zanemariv (manji od 0.001) povećava se vjerojatnost mutacije da bi se potaknulo istraživanje novih područja. Pri tome se ograničava maksimalna vjerojatnost mutacije na 0.1 (10%) da evolucijski algoritam ne bi rezultirao previše nasumičnom pretragom. Nasuprot tome, ako postoji značajniji globalni napredak populacije, onda se vjerojatnost mutacije smanjuje, pri čemu najmanja moguća vjerojatnost mutacije iznosi 0.001 (0.1%). Ovisno o dodijeljenoj vjerojatnosti mutacije, svaki pojedini gen jedinke može mutirati i pritom dobiti novu vrijednost u intervalu $[minVal, maxVal]$.

Adaptivna mutacija dobro balansira između istraživanja novih i eksploatacije već postojećih (dobrih) područja rješenja. Primjenjiva je u raznim reprezentacijama rješenja (binarna, cjelobrojna, realna, permutacijska itd.) te se ovisno o njima mogu razlikovati strategije adaptacije mutacije. U konačnici, adaptivna mutacija omogućuje evolucijskom algoritmu da se sam prilagođava problemu te, sukladno tome, može rezultirati bržom ili kasnijom konvergencijom.

Iako prethodno nisu bili spomenuti, osim adaptivne mutacije postoje i adaptivno križanje i adaptivna selekcija u evolucijskim algoritmima. Adaptivno križanje prilagođava način kombiniranja genetskog materijala roditelja [54], dok adaptivna selekcija dinamički mijenja kriterije odabira jedinki za reprodukciju [55]. Zajedno s adaptivnom mutacijom, ove tehnike čine evolucijske algoritme iznimno fleksibilnima i sposobnima za samostalno prilagođavanje različitim problemima optimizacije.

POGLAVLJE 5

Strategije izvedbe evolucijskih algoritama

Evolucijski algoritmi, premda efikasni i fleksibilni, u svojoj osnovnoj formi mogu biti neučinkoviti kada se primjenjuju na velike ili računalno zahtjevne probleme. Njihova izvedba uvelike ovisi o načinu na koji se implementiraju evolucijski operatori, kao i o dodatnim strategijama koje se koriste za ubrzanje izvođenja, očuvanje genetske raznolikosti te smanjenje nepotrebnih evaluacija. U ovom poglavlju razmatraju se upravo takve strategije, odnosno one koje poboljšavaju učinkovitost evolucijskih algoritama bez kompromitiranja kvalitete rješenja.

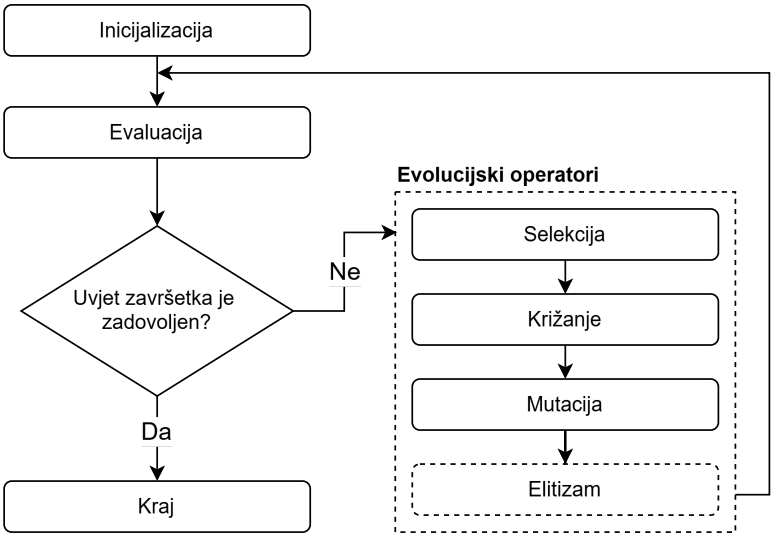
Prikazane metode obuhvaćaju i klasične pristupe poput elitizma koji osigurava prijenos najkvalitetnijih jedinki između generacija, kao i moderne pristupe temeljene na paralelizmu i memoizaciji. Dok elitizam omogućuje očuvanje već pronađenih dobrih rješenja, paralelizam znatno ubrzava proces evaluacije, što je ključno kod složenih problema ili velikih populacija. Memoizacija, s druge strane, smanjuje broj nepotrebnih evaluacija pamćenjem već obrađenih rješenja i njihovih svojstava.

Osim navedenih tehnika, u ovom poglavlju dodatno se razmatra i strategija temeljena na Kartezijevom produktu gena, koja omogućuje širenje učinka jedne evaluacije na veći broj sličnih rješenja. U situacijama kada više genetskih varijacija dovodi do iste vrijednosti dobrote, moguće je unaprijed detektirati skup takvih jedinki i za sve njih pohraniti jedinstvenu evaluaciju. Ovaj pristup posebno dolazi do izražaja u domenama poput genetskog programiranja ili problema raspoređivanja, gdje se uočava visok stupanj redundantnosti među jedinkama.

5.1 Elitizam

Nakon selekcije, križanja i mutacije, može se primijeniti strategija elitizma. Njome se osigurava preživljavanje najkvalitetnijih jedinki u populaciji kroz generacije prema načelu *survival of the fittest* [56, 57]. Određeni broj elitnih jedinki iz prethodne generacije rješenja prebacuje se u trenutnu, najčešće zamjenjujući najlošije jedinke. Korak elitizma obično se smješta na kraj evolucijskog procesa (primjerice, nakon mutacije, vidi

Sliku 5.1), da bi se izbjeglo slučajno mijenjanje ili gubitak najboljih jedinki tijekom evolucijskog procesa.



Slika 5.1: Elitizam u evolucijskom algoritmu.

Korištenje elitizma nije uvijek nužno potrebno, a najčešće ovisi o odabranom selekcijskom mehanizmu. Primjerice, ukoliko se koriste velike turnirske selekcije, vjerojatnost da najbolja jedinka u populaciji neće biti izabrana jako je mala, pa se korak elitizma u ovakvim slučajevima može i preskočiti. Time se može uštedjeti procesorsko vrijeme potrebno za sortiranje prethodne generacije rješenja i kopiranje elitnih jedinki u trenutnu populaciju.

Algoritam 5.1 Elitizam u evolucijskom algoritmu.

```
1: function ELITIZAM(prethodnaPop, trenutnaPop, kolElitni)
2:     SORTIRAJ(prethodnaPop, poDobroti)        ▷ Sortiraj prethodnu populaciju po dobroti.
3:     SORTIRAJ(trenutnaPop, poDobroti)         ▷ Sortiraj trenutnu populaciju po dobroti.
4:     for i ← 1 to kolElitni do
5:         trenutnaPop[kraj - i] ← prethodnaPop[i]        ▷ Zamjena najlošijih jedinki.
6:     end for
7: end function
```

Implementacija elitizma odvija se u nekoliko koraka opisanih u Algoritmu 5.1. Prvo se prethodna i nova populacija sortiraju prema vrijednostima dobrote svojih jedinki. Ovi su koraci nužni da bi se iz prethodne generacije rješenja mogle dohvatiti one najbolje (elitne) jedinke, kao i da bi se identificirale najlošije jedinke u trenutnoj populaciji koje treba zamijeniti elitnim jedinkama. C++ implementacija prikazana je u Primjeru 5.1.

Primjer 5.1: Implementacija elitizma u programskom jeziku C++.

```
1 void dodajElitneJedinke(Populacija& izvor, Populacija& odrediste, int brojElitnih) {
2     // Silazno sortiraj jedinke u izvornoj populaciji prema vrijednosti dobrote
3     std::sort(izvor.jedinka.begin(), izvor.jedinka.end(),
4               [](const Jedinka& a, const Jedinka& b) {
5                   return a.dobrota > b.dobrota;  // Od vece vrijednosti prema manjoj
```

```
            });

    // Silazno sortiraj jedinke u odredisnoj populaciji prema vrijednosti dobrote
    std::sort(odrediste.jedinka.begin(), odrediste.jedinka.end(),
              [](const Jedinka& a, const Jedinka& b) {
                  return a.dobrota > b.dobrota;  // Od vece vrijednosti prema manjoj
              });

    // Zamijeni najgore jedinke u odredisnoj populaciji s elitnim jedinkama
    for (int i = 0; i < brojElitnih; ++i)
        odrediste.jedinka[odrediste.jedinka.size() - i - 1] = izvor.jedinka[i];
}
```

Broj elitnih jedinki definiran je parametrom. Stvarni broj elitnih jedinki ovisi o specifičnom problemu te se vrlo često određuje eksperimentalno, testiranjem različitih vrijednosti. Obično se odabire mali postotak, primjerice, 1% - 10% od ukupnog broja jedinki u populaciji. Pri tome treba imati na umu da unatoč prednostima elitizma (očuvanje najboljih rješenja kroz generacije) on može negativno utjecati na raznolikost populacije. Preveliki broj elitnih jedinki može smanjiti tu raznolikost, osobito u kasnijim generacijama kada elitne jedinke već dominiraju populacijom. Zbog toga broj elitnih jedinki može biti varijabilan, tj. ovisan o stagnaciji populacije. Također, prilikom odabira elitnih jedinki može se primijeniti strategija izbjegavanja duplikata, čime se dodatno pogoduje raznolikosti populacije.

5.2 Paralelizam

Paralelizam u evolucijskim algoritmima dobro je istražena tema koja se koristi u svrhu poboljšanja performansi evaluacije populacije, kao i jedna od strategija za očuvanje raznolikosti rješenja. Kada je cilj ubrzati evaluaciju kod računski zahtjevnih problema primjenjuje se fino granulirani paralelizam, pri čemu se evaluacija jedinki izvodi paralelno korištenjem više dretvi ili procesora, ili čak više računala [24, 58]. Ovaj pristup omogućuje značajno poboljšanje performansi u velikim optimizacijskim problemima gdje je potrebno obraditi velike količine podataka u kratkom vremenskom razdoblju.

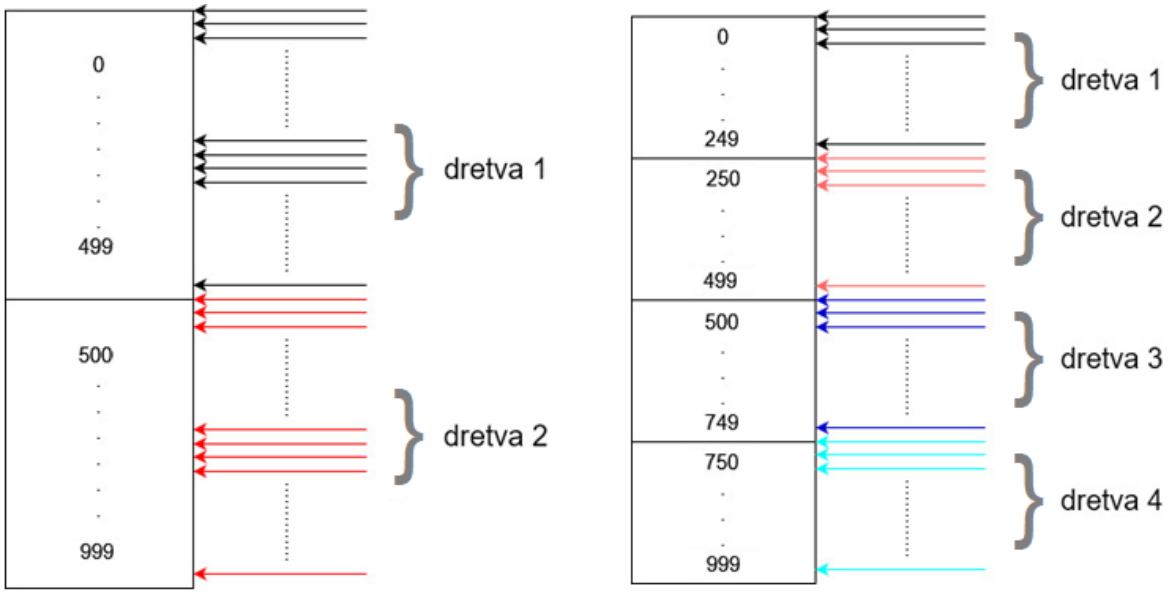
S druge strane, postoji i grubo granulirani paralelizam koji se fokusira na neovisnu evoluciju više subpopulacija uz primjenu evolucijskih operatora (selekcija, križanje i mutacija) [59, 60, 61]. Najčešće se implementira na distribuiranim sustavima korištenjem više računala, ali može se provesti i na višeprocesorskim arhitekturama. Ovaj pristup dovodi do veće raznolikosti konačne populacije te smanjuje rizik od prerane konvergencije algoritma, pod uvjetom da je migracija jedinki među subpopulacijama rijetka i dobro postavljena. Naime, ako je migracija prečesta ili preopsežna, populacije se mogu prebrzo izjednačiti, što smanjuje genetsku raznolikost i može dovesti do prerane konvergencije.

U daljnjem tekstu fokus će biti na poboljšanju performansi evaluacije rješenja u evolucijskim algoritmima, odnosno na primjeni fino granularnog paralelizma u višedretvenim i višeračunalnim okruženjima.

5.2.1 Paralelna evaluacija populacije

Kada je evaluacija jedinki glavno usko grlo u evolucijskom procesu, fino granulirani paralelizam nudi učinkovit pristup za poboljšanje performansi. Ovaj oblik paralelizma, koji se najčešće implementira na jednom računalu koristeći višedretveno programiranje, omogućuje istovremenu evaluaciju većeg broja jedinki populacije.

Višedretvena evaluacija posebno je učinkovita kada je evaluacija dobrote jedinke računski (vremenski) zahtjevna, a populacija dovoljno velika da opravda troškove kreiranja, inicijalizacije i rada dretvi. Implementacija ovog pristupa obično uključuje podjelu populacije na manje segmente gdje svaka dretva evaluira svoj dio populacije (Slika 5.2). Time se postiže bolje iskorištavanje resursa modernih višejezgrenih procesora što rezultira značajnim ubrzanjem evolucijskog procesa.



Slika 5.2: Višedretveni pristup evaluaciji populacije s 1000 jedinki.

Za primjer, u problemu semantičkog zaključivanja [1] korištena je višedretvena evaluacija populacije. Zbog složenosti pojedinih problema izvođenje evolucijskog procesa korištenjem samo jedne dretve trajalo bi čak do 42 dana. Međutim, implementacijom višedretvenog pristupa i korištenjem računala s 40 procesorskih jezgri trajanje istog procesa smanjeno je na približno 38 sati. U ovom slučaju, populacije su imale po 20.000 jedinki.

Važno je napomenuti da, iako višedretvena evaluacija ne zahtijeva složene algoritme za distribuciju zadataka kao što je to slučaj s grubo granuliranim paralelizmom, ipak zahtijeva pažljivo upravljanje sinkronizacijom i balansiranjem opterećenja među dretvama. Osim toga, ako evaluacija dobrote jedinki nije dovoljno računski zahtjevna, upravljanje dretvama može postati usko grlo, odnosno može dovesti do čak i slabijih performansi višedretvenog pristupa u odnosu na evaluaciju populacije sa samo jednom dretvom.

Za demonstraciju pretpostavimo da rješavamo problem pronalaska binarne reprezentacije nekog prirodnog broja, pri čemu je kvalitetnije ono rješenje koje je bliže traženoj vri-

jednosti (problem minimizacije). Primjer 5.2 prikazuje C++ implementaciju metode za evaluaciju kompletne populacije, pri čemu se ovisno o parametru, evaluacija može izvoditi sekvencijalno, koristeći samo jednu dretvu, ili višedretveno, koristeći sve dostupne procesorske jezgre. Kompletan programski kod nalazi se u Prilogu 9.20.

Primjer 5.2: Implementacija sekvencijalne i višedretvene evaluacije populacije u programskom jeziku C++ korištenjem standardne biblioteke.

```
void Populacija::evaluiraj(bool visedretveno, int optimalnaVrijednost) {
    if (!visedretveno) {
        // Sekvencijalna evaluacija svih jedinki (bez paralelizacije)
        for (unsigned int i = 0; i < jedinka.size(); i++)
            jedinka[i].evaluiraj(optimalnaVrijednost);
        return;
    }

    // Dohvati broj dostupnih procesorskih jezgri ili koristi barem jednu dretvu
    unsigned brojDretvi = std::max(1u, std::thread::hardware_concurrency());
    if (jedinka.empty()) return; // Ako nema jedinki, prekini izvrsavanje

    std::vector<std::thread> dretve;
    int pocetak = 0, iskoristeniZadaci = 0, zadaciZaOvuDretvu = 0;

    // Kreiranje i pokretanje dretvi
    for (unsigned int i = 0; i < std::min((unsigned)jedinka.size(), brojDretvi); i++) {
        // Odredivanje broja zadataka za ovu dretvu (ravnomjerno rasporedivanje)
        zadaciZaOvuDretvu = jedinka.size()/brojDretvi + (i < jedinka.size() % brojDretvi);
        int kraj = std::min(pocetak + zadaciZaOvuDretvu - 1, (int)(jedinka.size()) - 1);

        // Pokretanje dretve za evaluaciju dijela populacije
        std::thread dretva([=](unsigned _pocetak, unsigned _kraj) {
            for (unsigned int j = _pocetak; j <= _kraj; j++)
                jedinka[j].evaluiraj(optimalnaVrijednost); // !
            }, pocetak, kraj);
        dretve.push_back(std::move(dretva));

        // Azuriranje iskoristenih zadataka nakon pokretanja dretve
        iskoristeniZadaci += zadaciZaOvuDretvu;
        pocetak = iskoristeniZadaci;
    }

    // Cekanje da sve dretve zavrse izvrsavanje
    for (auto& dretva : dretve)
        dretva.join();
}
```

Iako standardna C++ biblioteka nudi podršku za implementaciju višedretvenosti, na programeru je odgovornost da samostalno odredi i ravnomjerno rasporedi zadatke među dretvama. Pravilna i ravnomjerna raspodjela poslova ključna je za postizanje optimalnih performansi višedretvenog pristupa, prvenstveno zbog značajnih troškova povezanih s kreiranjem i uništavanjem dretvi. Nepravilna raspodjela može rezultirati neravnomjernim opterećenjem, pri čemu neke dretve završavaju znatno ranije od drugih, što dovodi do neiskorištenosti procesorskih jezgri i ukupno slabijih performansi algoritma.

Naime, kada bi se za evaluaciju svake pojedine jedinke u populaciji zasebno stvarala i uništavala dretva, ukupni trošak operacijskog sustava (vrijeme i resursi) za upravljanje dretvama premašio bi dobit od paralelizacije, što bi rezultiralo lošijim performansama. Umjesto toga, učinkovitiji pristup je korištenje unaprijed alociranog skupa dretvi (sukladno broju dostupnih procesorskih jezgri), pri čemu svaka dretva obrađuje više zadataka um-

jesto da se ponovno stvara za svaki zadatak. Takav pristup smanjuje spomenute troškove i poboljšava iskorištenost resursa. Upravo to demonstrira Primjer 5.2, gdje se populacija dijeli na više segmenata, a svaka dretva evaluira dodijeljeni dio populacije.

Tablica 5.1: Primjer raspodjele poslova za 2, 4 i 8 dretvi, za populaciju s 1.000 jedinki.

Broj dretvi	Dretva	Interval	Broj jedinki
2	1	[0, 499]	500
2	2	[500, 999]	500
4	1	[0, 249]	250
4	2	[250, 499]	250
4	3	[500, 749]	250
4	4	[750, 999]	250
8	1	[0, 124]	125
8	2	[125, 249]	125
8	3	[250, 374]	125
8	4	[375, 499]	125
8	5	[500, 624]	125
8	6	[625, 749]	125
8	7	[750, 874]	125
8	8	[875, 999]	125

Primjer takve raspodjele poslova vidljiv je u Tablici 5.1. Za populaciju od 1.000 jedinki te 2, 4 i 8 dretvi raspodjela posla jednaka je po svakoj dretvi. Međutim, da je riječ o populaciji od, primjerice, 1.003 jedinke uz korištenje 4 dretve, raspodjela bi bila sljedeća:

- Dretva 1: [0, 250] (251 jedinka)
- Dretva 2: [251, 501] (251 jedinka)
- Dretva 3: [502, 752] (251 jedinka)
- Dretva 4: [753, 1002] (250 jedinki)

Sada broj jedinki ne može biti jednak za svaku dretvu, ali je i dalje ravnomjerno raspoređen da bi sve dretve imale podjednako opterećenje.

Općenito, ovisno o korištenom programskom jeziku i programskom okviru, preporuka je koristiti bazen dretvi (engl. *thread pool*) da bi upravljanje dretvama bilo još jednostavnije i učinkovitije. Bazen dretvi pruža bolju kontrolu nad brojem aktivnih dretvi, sprječavajući preopterećenje sustava [24]. Osim toga, ovaj pristup često pojednostavljuje implementaciju jer prebacuje odgovornost za optimalnu raspodjelu poslova s programera na sam sustav, istovremeno često pružajući bolje performanse u praksi.

Primjer 5.3: Implementacija sekvencijalne i višedretvene evaluacije populacije korištenjem bazena dretvi u C++ Builder razvojnom okruženju i korištenjem VCL biblioteke.

```
1 void Populacija::evaluiraj(bool visedretveno, int optimalnaVrijednost) {
```

```
    if (!visedretveno) {
        // Sekvencijalna evaluacija svih jedinki (bez paralelizacije)
        for (unsigned int i = 0; i < jedinka.size(); i++)
            jedinka[i].evaluiraj(optimalnaVrijednost);
        return;
    }
    if (jedinka.empty()) return; // Ako nema jedinki, prekini izvrsavanje

    // Visedretveno izvrsavanje koristenjem bazena dretvi (C++ Builder)
    int size = static_cast<int>(jedinka.size());
    TParallel::For(0, size - 1, _di_TProc__1<int>([=](int i) {
        if (i >= 0 && i < size) {
            jedinka[i].evaluiraj(optimalnaVrijednost);
        }
    }));
}
```

Za razliku od nekih programskih jezika poput Pythona, Jave i C#, C++ standardna biblioteka ne pruža izravnu podršku za korištenje bazena dretvi. Ovo znači da programeri koji žele implementirati bazen dretvi u C++ moraju posegnuti za alternativnim rješenjima. Jedna od takvih alternativa je korištenje VCL (Visual Component Library) biblioteke u C++ Builder razvojnom okruženju (Primjer 5.3) [62]. VCL pruža mogućnost korištenja bazena dretvi kroz klasu $TParallel$ koja omogućuje jednostavnije i učinkovitije upravljanje višedretvenim izvršavanjem zadataka. Ova biblioteka olakšava implementaciju paralelizma u C++ Builder aplikacijama, nudeći funkcionalnosti slične onima koje su dostupne u drugim jezicima s ugrađenom podrškom za bazene dretvi.

Usporedbom Primjera 5.2 i 5.3 jasno je vidljivo zašto je korištenje bazena dretvi bolji pristup. Znatno je jednostavniji za implementaciju, a u konačnici, zbog svog načina rada, najčešće daje i bolje performanse.

Na performanse višedretvene evaluacije populacije može utjecati i potreba za sinkronizacijom dretvi. Primjerice, ukoliko bi bilo potrebno brojati odrađene evaluacije u višedretvenom načinu rada, dretve bi bilo potrebno zaključavati pri pristupu zajedničkim resursima (zajedničkom brojaču) korištenjem kritične sekcije, muteksa ili sličnih mehanizama. U određenom trenutku sve dretve morat će čekati svoj red i pauzirati izvršavanje da bi uvećale stanje brojača, da bi tek nakon toga mogle dalje nastaviti s radom.

5.2.2 Paralelizam u distribuiranom okruženju

Paralelna evaluacija može se odvijati i u distribuiranom okruženju, korištenjem više računala povezanih u mrežu. Distribuirani paralelizam u evolucijskim algoritmima može se ostvariti na nekoliko načina, među kojima su model gospodar-sluga , otočni, hijerarhijski model itd. [63]. Ovi se modeli mogu primijeniti i na jednom računalu u višedretvenom okruženju, no njihova je primarna svrha iskorištavanje računalnih resursa distribuiranog sustava.

Jedan od najjednostavnijih za implementaciju upravo je model gospodar-sluga (engl. *master-slave*), u kojem jedno računalo (gospodar) obavlja pripremu populacije za evaluaciju (selekciju, križanje i mutaciju), dok ostala računala u mreži (sluge) evaluiraju pojedine dijelove te populacije i vraćaju rezultate gospodaru (Algoritam 5.2).

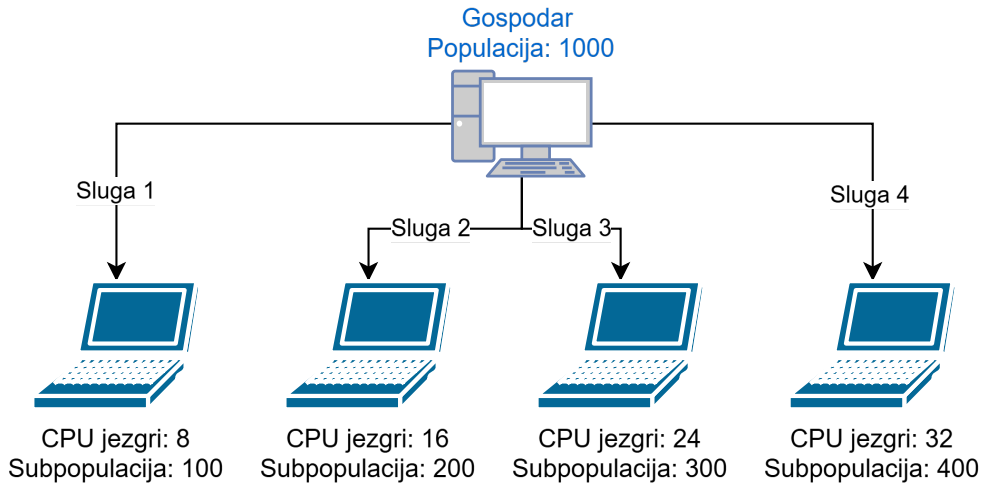
Algoritam 5.2 Evaluacija populacije korištenjem modela gospodar-sluga u distribuiranom okruženju.

```
1: function GOSPODARSLUGA(populacija, brojSluga)
2:     while uvjet završetka nije zadovoljen do
3:         novaPopulacija ← SELEKCIJA(populacija)
4:         KRIZANJE(novaPopulacija)
5:         MUTACIJA(novaPopulacija)
                                          ▷ Podjela populacije na segmente za sluge.
6:         podpopulacije ← PODIJELI(novaPopulacija, brojSluga)
                                          ▷ Slanje podpopulacija slugama na evaluaciju.
7:         for i ← 1 to brojSluga do
8:             POSALJISLUGI(i, podpopulacije[i])
9:         end for
                                          ▷ Čekanje rezultata evaluacije od sluga.
10:        for i ← 1 to brojSluga do
11:            evaluiraniDioPopulacije ← PRIMIODSLUGE(i)
12:            ZAMIJENI(novaPopulacija.dioPopulacije[i], evaluiraniDioPopulacije)
13:        end for
14:        populacija ← novaPopulacija
15:    end while
16: end function
```

Navedeni algoritam opisuje općeniti pristup paralelnoj evaluaciji populacije u distribuiranom okruženju korištenjem modela gospodar-sluga, pri čemu se detalji implementacije mogu razlikovati ovisno o specifičnostima pojedinog slučaja. Primjerice, podjela populacije na dijelove ne mora nužno biti ravnomjerna, već može ovisiti o karakteristikama svakog pojedinog sluge. Svaki sluga, odnosno računalo, može imati različite resurse poput broja procesorskih jezgri, količine memorije i drugih čimbenika. U takvoj situaciji raspodjela poslova među slugama može biti optimizirana prema mogućnostima pojedinog računala. Primjerice, slugi s dvostruko više procesorskih jezgri može se dodijeliti veći (ravnomjerno izračunati) dio populacije da bi sve sluge završile s radom otprilike u isto vrijeme, čime se minimizira čekanje na obradu sljedeće populacije (Slika 5.3).

U primjeru na Slici 5.3 pretpostavljamo da gospodar ne sudjeluje u evaluacijskom procesu, već samo priprema populaciju, razdjeljuje zadatke i čeka rezultate. Također, pretpostavljamo da sve sluge imaju procesore iste brzine, jer bi u suprotnom i to bio čimbenik u izračunavanju ravnomjerne raspodjele poslova među slugama.

Način komunikacije između gospodara i sluga u distribuiranom evolucijskom algoritmu koji koristi model gospodar-sluga također može biti specifičan za svaki slučaj. Mogući je sinkroni oblik komunikacije, gdje gospodar šalje zadatke slugama i čeka njihove rezultate prije nego što nastavi dalje. Ovaj pristup može biti učinkovit u slučajevima gdje je potrebna stroga kontrola nad procesom, ali može uzrokovati kašnjenja ako neki od sluga rade sporije [64]. S druge strane, u asinkronom obliku komunikacije gospodar ne mora čekati na sve rezultate, već može procesirati one koji su mu već dostupni, čime se smanjuje vrijeme čekanja i omogućava bolja skalabilnost, osobito u heterogenim sustavima gdje



Slika 5.3: Model gospodar-sluga u distribuiranom okruženju.

računala imaju različite kapacitete ili opterećenja [65]. Međutim, da bi u asinkronom obliku komunikacije gospodar znao kada su sve evaluacije završene, potrebno je implementirati mehanizam koji će signalizirati kada su svi zadaci dovršeni (primjerice, svaki sluga može poslati signal gospodaru kada završi s izvršavanjem svih dodijeljenih zadataka).

5.3 Memoizacija

U evolucijskim algoritmima, tehnika korištenja tablice raspršivanja (engl. *hash table*) ili slične strukture podataka za otkrivanje i izbjegavanje ponovne evaluacije već evaluiranih jedinki u populaciji često se naziva memoizacijskom strategijom. Ovaj pristup posebno je koristan u scenarijima gdje je evaluacija jedinki računski zahtjevna, jer njegova primjena smanjuje ukupno procesorsko vrijeme potrebno za evaluaciju populacije [66].

Memoizacija postaje još važnija u kasnijim fazama evolucijskog procesa, kada populacija nakon većeg broja generacija počne sadržavati sve više sličnih rješenja. U tim fazama evolucijski operatori, poput križanja i mutacije, mogu proizvesti jedinke koje su već bile evaluirane u ranijim generacijama. Bez memoizacije te bi se jedinke ponovno evaluirale, što bi rezultiralo nepotrebnim trošenjem računalnih resursa. Memoizacija omogućuje korištenje prethodno pohranjenih vrijednosti dobrote umjesto ponovne evaluacije, što ne samo da smanjuje trošak evaluacije već i ubrzava cijeli proces.

Memoizacija u kontekstu evolucijskih algoritama uključuje pohranjivanje vrijednosti dobrote evaluiranih jedinki u tablici raspršivanja (ili drugoj učinkovitoj strukturi za pretraživanje), zajedno s njihovim genetskim reprezentacijama. Kada se jedinka evaluira, algoritam prvo provjerava tablicu raspršivanja da bi utvrdio je li ta jedinka (ili genetski identična njoj) već bila evaluirana. Ako jest, dohvaća se pohranjena vrijednost dobrote, čime se izbjegava računski trošak ponovne evaluacije. Ako nije, jedinka se evaluira, a njezina se dobrota zajedno s genetskom reprezentacijom pohranjuje u tablicu raspršivanja za buduće korištenje.

Uspješnost memoizacije u evolucijskim algoritmima ovisi o dizajnu funkcije raspršivanja i korištenoj strukturi podataka. Funkcija raspršivanja mora učinkovito preslikavati genetsku reprezentaciju jedinki u jedinstvene ključeve, a struktura podataka treba omogućiti brze operacije umetanja i pretraživanja.

Primjerice, pretpostavimo da u programskom jeziku C++ želimo koristiti strategiju memoizacije pri evaluaciji jedinki čiji binarni geni predstavljaju cijele brojeve. Tada možemo koristiti sljedeću strukturu podataka ključ-vrijednost:

```
std::unordered_map<unsigned int, unsigned int> memoizacija;
```

Struktura `std::unordered_map` ima prosječnu vremensku složenost $O(1)$ za operacije pretraživanja i umetanja podataka, što je čini idealnom za primjenu u memoizaciji. Ključ u ovoj strukturi predstavlja genetsku strukturu rješenja (u našem slučaju cijeli broj čiju evaluaciju želimo provjeriti ili pohraniti), dok vrijednost predstavlja dobrotu jedinke. Navedena struktura najčešće se nalazi u globalnom prostoru da bi se mogla višestruko koristiti pri evaluaciji svake nove generacije rješenja.

Primjer 5.4: Evaluacija jedinke korištenjem strategije memoizacije.

```
1  void Jedinka::evaluiraj(int optimalnaVrijednost) {
2      unsigned int broj = 0;
3      for (unsigned int i = 0; i < gen.size(); i++)
4          broj += gen[i] * (1u << (gen.size() - i - 1));
5
6      // Provjera postoji li vec izracunata vrijednost dobrote za ovu binarnu vrijednost.
7      auto it = memoizacija.find(broj);
8      if (it != memoizacija.end())
9          dobrota = it->second; // Ako je vrijednost vec izracunata, dohvacamo je iz tablice
10     else { // Ako vrijednost nije vec izracunata
11         dobrota = (broj < optimalnaVrijednost) ? (optimalnaVrijednost - broj) : (broj -
       optimalnaVrijednost);
12         memoizacija[broj] = dobrota; // vrijednost dodajemo u tablicu rasprsivanja
13     }
14 }
```

Primjer 5.4 demonstrira primjenu strategije memoizacije prilikom evaluacije jedinke u evolucijskom algoritmu. Na temelju binarne strukture jedinke izračunava se cijeli broj koji ona predstavlja. Taj broj istovremeno služi kao genetska reprezentacija jedinke te se njegova prisutnost provjerava u memoizacijskoj strukturi. Ako se tamo nalazi, vrijednost dobrote dohvaća se iz memoizacijske strukture i dodjeljuje jedinki. U suprotnom, jedinka se evaluira, a njezina genetska struktura i izračunata vrijednost dobrote spremaju se u memoizacijsku strukturu. Puni primjer demonstracije strategije memoizacije nalazi se u Prilogu 9.21.

5.4 Kartezijev produkt gena

Ovisno o problemu, strategija memoizacije može se dodatno poboljšati tako da se na temelju evaluacije jednog rješenja izračuna i spremi vrijednost dobrote za cijeli skup sličnih rješenja. Primjerice, ako jedan ili više gena imaju slične vrijednosti (varijante) koje

ne utječu na dobrotu jedinke, tada postoji više jedinki koje dijele istu vrijednost dobrote. Kartezijevim produktom vrijednosti gena možemo automatski odrediti sve takve jedinke te, prilikom evaluacije jedne od njih, istu vrijednost dobrote zabilježiti svim drugim sličnim jedinkama u memoizacijskoj strukturi. U ovakvom slučaju, strategija proširenja memoizacije na cijeli skup sličnih rješenja može značajno smanjiti broj evaluacija te ubrzati evolucijski proces.

Kartezijev produkt (ili Kartezijev umnožak) operacija je iz teorije skupova koja generira novi skup sastavljen od svih mogućih uređenih parova elemenata iz dva ili više skupova. Ovaj koncept koristi se u različitim granama matematike i računarstva, uključujući geometriju, relacije i baze podataka [67]. Za primjer, neka imamo skupove

$$A = \{1, 2, 3\}$$
$$B = \{a, b, c\}$$

Kartezijev produkt $A \times B$ je:

$$A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$$

Ako skupovi A i B predstavljaju gene, a vrijednosti unutar njih varijacije koje ne utječu na dobrotu jedinke, tada Kartezijevim produktom dobivamo skup jedinki koje dijele istu vrijednost dobrote. Kao primjer u evolucijskim algoritmima (genetskom programiranju), promotrimo rješenje (jedinku) koje predstavlja atributnu gramatiku za domensko specifični jezik `ExprLA` (Primjer 5.5) [24].

Primjer 5.5: Atributna gramatika za domensko specifični jezik ExprLA (netočno rješenje).

```
language ExprLA {
  lexicon {
    Comment /\*[^\*]+\*/
    Operator \+ | \*
    Separator \( | \)
    Int [0-9]+
    ignore [\ \0x0D\0x0A\0x09]+ | #Comment
  }
  attributes int *.val, *.inVal;
  rule Expr {
    E ::= T EE compute {
      E.val = EE.val*EE.inVal;  // gen 1
      EE.inVal = T.val*EE.val;  // gen 2
    };
    EE ::= + T EE compute {
      EE.val = EE[1].inVal+EE.inVal;  // gen 3
      EE[1].inVal = EE.val*T.val;     // gen 4
    } | epsilon compute {
      EE.val = EE.inVal*EE.inVal;   // gen 5
    };
    T ::= F TT compute {
      T.val = TT.inVal*TT.val;  // gen 6
      TT.inVal = TT.val*T.val;  // gen 7
    };
    TT ::= * F TT compute {
      TT.val = F.val*TT[1].val;   // gen 8
      TT[1].inVal = TT.inVal+TT.val;  // gen 9
    } | epsilon compute {
      TT.val = TT.inVal+TT.inVal;  // gen 10
    };
```

```
31    }
32    rule Term {
33      F ::= #Int compute {
34        F.val = Integer.valueOf(#Int.value()).intValue();  // gen 11
35      };
36      F ::= ( E ) compute {
37        F.val = E.val;  // gen 12
38      };
39    }
40  }
```

Navedeni primjer prikazuje netočno rješenje jedinke koja ima 12 gena. Većina tih gena (semantičkih jednadžbi) može postojati u više varijacija, prikazanih u Tablici 5.2.

Tablica 5.2: Semantičke jednadžbe i njihove varijacije za gene iz Primjera 5.5.

Rb.	Semantic equation and variations
1	E.val = EE.val*EE.inVal; E.val = EE.inVal*EE.val;
2	EE.inVal = T.val*EE.val; EE.inVal = EE.val*T.val;
3	EE.val = EE[1].inVal+EE.inVal; EE.val = EE.inVal+EE[1].inVal;
4	EE[1].inVal = EE.val*T.val; EE[1].inVal = T.val*EE.val;
5	EE.val = EE.inVal*EE.inVal;
6	T.val = TT.inVal*TT.val; T.val = TT.val*TT.inVal;
7	TT.inVal = TT.val*T.val; TT.inVal = T.val*TT.val;
8	TT.val = F.val*TT[1].val; TT.val = TT[1].val*F.val;
9	TT[1].inVal = TT.inVal+TT.val; TT[1].inVal = TT.val+TT.inVal;
10	TT.val = TT.inVal+TT.inVal;
11	F.val = Integer.valueOf(#Int.value()).intValue();
12	F.val = E.val;

Geni (semantičke jednadžbe) pod rednim brojevima $1, 2, 3, 4, 6, 7, 8$ i 9 mogu postojati u dvije varijante, pri čemu nijedna od njih ne utječe na vrijednost dobrote jedinke iz Primjera 5.5. Kartezijev produkt svih 12 skupova gena (njihovih varijacija) rezultira s 256 mogućih varijanti jedinki ($2*2*2*2*1*2*2*2*2*1*1*1$), pri čemu sve imaju istu vrijednost dobrote.

Nakon evaluacije samo jedne od mogućih kombinacija, njezina vrijednost dobrote sprema se u tablicu raspršivanja, a ista vrijednost dodjeljuje se i preostalim varijantama te jedinke u tablici raspršivanja (njih 255). Ovim pristupom strategija memoizacije postaje znatno učinkovitija jer se u ovom slučaju evaluacijom samo jedne jedinke zapravo određuje vrijednost dobrote za 256 jedinki.

Jedan dodatni primjer primjene Kartezijevog produkta gena u evolucijskim algoritmima može se pronaći u problemima raspoređivanja, posebno u problemu rasporeda predavanja na fakultetu. Ako jedinka predstavlja rješenje rasporeda predavanja, onda geni u toj jedinki mogu predstavljati **dvoranu** i **vrijeme** u kojoj se odvija nastava iz određenog kolegija. U nekim situacijama postoji više alternativa koje ne mijenjaju kvalitetu rješenja (dobrotu jedinke), primjerice:

- **Različite dvorane iste kategorije**: Ako dvije dvorane imaju isti kapacitet i tehničku opremu, zamjena jedne dvorane drugom ne utječe na kvalitetu rasporeda.
- **Moguće zamjene termina predavanja**: Ako kolegij može biti održan u dva vremenska termina bez narušavanja ograničenja (primjerice, ne preklapa se s drugim kolegijima istog nastavnika ili iste grupe studenata), tada se ti termini mogu smatrati ekvivalentnim.

Ako uzmemo skup mogućih dvorana $A = \{D_1, D_2\}$ i skup mogućih termina $B = \{T_1, T_2\}$, Kartezijev produkt $A \times B$ daje četiri kombinacije:

$$A \times B = \{(D_1, T_1), (D_1, T_2), (D_2, T_1), (D_2, T_2)\}$$

Sve ove kombinacije generiraju jedinke koje imaju istu vrijednost dobrote jer su dvorane i termini ekvivalentni u kontekstu postojećeg problema. Koristeći strategiju memoizacije, možemo evaluirati samo jednu od ovih kombinacija i istu vrijednost dobrote dodijeliti svim ostalim kombinacijama, čime značajno smanjujemo broj evaluacija potrebnih za optimizaciju rasporeda.

POGLAVLJE 6

Metode lokalnog pretraživanja

Evolucijski algoritmi predstavljaju skup heurističkih metoda koje, inspirirane prirodnim evolucijskim procesima, nastoje pronaći optimalna rješenja za složene optimizacijske probleme. Temeljni mehanizmi ovih algoritama uključuju selekciju, križanje, mutaciju te evaluaciju pojedinačnih rješenja. Osim primjene ovih mehanizama, u praksi se često koriste i tehnike lokalnog pretraživanja. Ove strategije mogu djelovati samostalno ili kao dio šireg optimizacijskog pristupa, ovisno o specifičnostima problema i ciljevima optimizacije.

Lokalno pretraživanje usmjereno je na poboljšanje trenutnih rješenja kroz postupne, često inkrementalne, promjene unutar ograničenog prostora za pretraživanje. Prednosti ovog pristupa su brza konvergencija i efikasnost u pronalaženju poboljšanja u neposrednoj blizini (okolici) početnog rješenja. Među najčešće korištene metode u okviru lokalnog pretraživanja su:

- **Metoda penjanja uz brijeg** – metoda koja iterativno prelazi na susjedna rješenja s boljom vrijednosti dobrote [68].
- **Simulirano kaljenje** – stohastička tehnika koja, kroz kontrolirano "hlađenje" sustava, dopušta prelazak i na lošija rješenja u početnoj fazi pretraživanja, čime se smanjuje rizik zaglavljivanja u lokalnim optimumima [69].
- **Tabu pretraživanje** – metoda koja uvodi memorijske strukture (tabu liste) da bi se izbjeglo vraćanje na prethodno obrađena rješenja te da bi se istražila nova područja prostora za pretraživanje [70].
- **Memetski algoritam** – hibridni pristup koji kombinira evolucijske algoritme s lokalnim pretraživanjem [71].

Lokalne metode pretraživanja omogućuju brzo poboljšavanje postojećih rješenja te su posebno korisne kada je poznata dobra početna točka ili kada se želi dodatno poboljšati rješenje dobiveno nekom drugom metodom. Međutim, ove metode mogu biti osjetljive na lokalne optimume, zbog čega je u složenijim problemima poželjno kombinirati ih s drugim

tehnikama koje omogućuju šire istraživanje prostora rješenja.

Lokalne metode pretraživanja nalaze široku primjenu u mnogim stvarnim problemima optimizacije, posebice onima gdje je izračun globalnog optimuma računalno preskup ili nedostižan zbog veličine i složenosti prostora rješenja [72]. U takvim situacijama lokalno pretraživanje predstavlja praktičan pristup koji omogućuje brzo pronalaženje zadovoljavajućih rješenja unutar ograničenog broja iteracija.

Nadalje, lokalne metode često se integriraju u šire heurističke ili metaheurističke okvire, poput memetskih algoritama ili hibridnih evolucijskih strategija, gdje preuzimaju ulogu eksploatacijskog mehanizma usmjerenog na fino podešavanje postojećih rješenja [73, 74]. Upravo zbog svoje jednostavnosti, fleksibilnosti i mogućnosti prilagodbe, lokalne metode pretraživanja i dalje ostaju temeljna komponenta mnogih naprednih algoritama u području računalne inteligencije i operacijskih istraživanja.

6.1 Generiranje okolnih rješenja

Ključna karakteristika metoda lokalnog pretraživanja jest generiranje okolnih rješenja s ciljem pronalaska boljeg rješenja unutar prostora pretraživanja. Okolna rješenja čine skup mogućih rješenja do kojih se može doći iz trenutnog rješenja primjenom malih promjena prema unaprijed definiranim pravilima. Taj skup rješenja naziva se prostor okolice (engl. *neighborhood*), a njegovo pravilno definiranje od ključnog je značaja za učinkovitost odabrane metode lokalnog pretraživanja jer izravno utječe na brzinu konvergencije i kvalitetu konačnih rješenja.

Broj okolnih rješenja može biti konstantan tijekom cijelog postupka lokalne pretrage, odnosno isti za svako rješenje nad kojim se pretraga provodi. Međutim, taj broj može biti i adaptivan, što znači da se mijenja tijekom izvođenja algoritma ovisno o određenim kriterijima, poput trenutne faze pretrage, kvalitete pronađenih rješenja ili udaljenosti od optimuma. Primjerice, u početnim fazama algoritma broj okolnih rješenja može biti jednak za svaki gen, dok se kasnije, ovisno o kvaliteti okolnih rješenja, taj broj može smanjiti za gene čija okolina daje lošija rješenja, a proporcionalno povećati za ostale gene, tako da ukupni broj generiranih okolnih rješenja ostane približno isti.

Važno je napomenuti da izbor strategije generiranja okolnih rješenja uključuje kompromis između kvalitete lokalne pretrage i računalne složenosti. Veće okoline omogućuju bolju pokrivenost prostora pretraživanja i povećavaju šansu za pronalazak boljeg rješenja, ali istovremeno povećavaju i broj evaluacija, što može biti značajan čimbenik kod složenijih problema. S druge strane, ograničavanje okolice na manji broj jednostavnih izmjena rezultira bržim izvođenjem, ali povećava rizik zaglavljivanja u lokalnim optimumima.

Za demonstraciju, pretpostavimo da imamo binarno rješenje duljine 8:

$$s = [1,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0]$$

U ovom slučaju, prostor okolnih rješenja možemo definirati kao skup svih rješenja koja

se mogu dobiti invertiranjem (promjenom) točno jednog bita u početnom rješenju. Stoga okolna rješenja od s izgledaju ovako:

$$s_1 = [\underline{\mathbf{0}}, 0, 0, 1, 1, 0, 1, 0] \quad \text{(invertiran 1. bit)}$$
$$s_2 = [1, \underline{\mathbf{1}}, 0, 1, 1, 0, 1, 0] \quad \text{(invertiran 2. bit)}$$
$$s_3 = [1, 0, \underline{\mathbf{1}}, 1, 1, 0, 1, 0] \quad \text{(invertiran 3. bit)}$$
$$s_4 = [1, 0, 0, \underline{\mathbf{0}}, 1, 0, 1, 0] \quad \text{(invertiran 4. bit)}$$
$$s_5 = [1, 0, 0, 1, \underline{\mathbf{0}}, 0, 1, 0] \quad \text{(invertiran 5. bit)}$$
$$s_6 = [1, 0, 0, 1, 1, \underline{\mathbf{1}}, 1, 0] \quad \text{(invertiran 6. bit)}$$
$$s_7 = [1, 0, 0, 1, 1, 0, \underline{\mathbf{0}}, 0] \quad \text{(invertiran 7. bit)}$$
$$s_8 = [1, 0, 0, 1, 1, 0, 1, \underline{\mathbf{1}}] \quad \text{(invertiran 8. bit)}$$

Veličina ovako definirane okolice iznosi 8 budući da imamo 8 mogućih promjena bita. Svako od okolnih rješenja razlikuje se od početnog rješenja u točno jednom bitu, a tek nakon evaluacije svih ovih rješenja moguće je utvrditi postoji li među njima rješenje koje je bolje od početnog.

Invertiranje jednog bita u binarnoj reprezentaciji tek je jedna od mogućnosti generiranja okolnih rješenja. Ovisno o specifičnostima problema, moguće su i druge strategije za generiranje okolice, primjerice:

- **Invertiranje više bitova odjednom**: u jednom koraku invertira se unaprijed definirani broj bitova (primjerice, dva ili tri).
- **Rotacija bitova**: kružno pomicanje cijelog ili dijela binarnog niza ulijevo ili udesno.
- **Zamjena pozicija bitova**: zamjena mjesta dvaju ili više odabranih bitova.
- **Dodavanje ili uklanjanje bitova**: ako je dopuštena promjenjiva duljina rješenja, okolica se može generirati dodavanjem ili uklanjanjem bitova.
- **Permutacija podniza bitova**: preuređivanje redoslijeda unaprijed definiranog podniza bitova unutar rješenja.

U sljedećem primjeru pretpostavimo da rješavamo problem trgovačkog putnika te da imamo sljedeće početno rješenje u permutacijskoj reprezentaciji (svaki grad smije se pojaviti točno jednom):

$$A \to B \to C \to D \to E$$

Za generiranje okolnih rješenja u ovom problemu možemo koristiti neke od sljedećih strategija:

- **Zamjena pozicija dvaju gradova**:

$$\underline{\mathbf{B}} \to \underline{\mathbf{A}} \to C \to D \to E \quad \text{(zamjena gradova A i B)}$$
$$A \to \underline{\mathbf{C}} \to \underline{\mathbf{B}} \to D \to E \quad \text{(zamjena gradova B i C)}$$
$$A \to B \to \underline{\mathbf{D}} \to \underline{\mathbf{C}} \to E \quad \text{(zamjena gradova C i D)}$$
$$A \to B \to C \to \underline{\mathbf{E}} \to \underline{\mathbf{D}} \quad \text{(zamjena gradova D i E)}$$

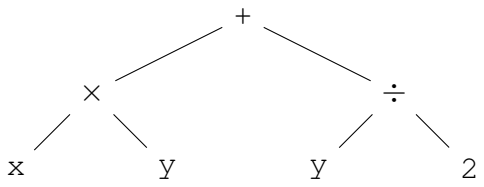
- **Inverzija segmenta puta**:

$$A \rightarrow \underline{\mathbf{D}} \rightarrow \underline{\mathbf{C}} \rightarrow \underline{\mathbf{B}} \rightarrow E \quad \text{(invertiran segment od grada B do grada D)}$$

- **Pomicanje jednog grada**:

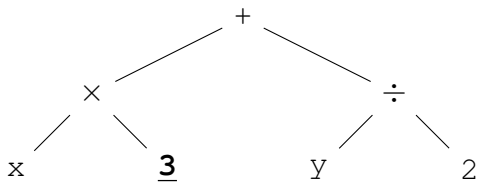
$$A \rightarrow \underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}} \rightarrow \underline{\mathbf{B}} \rightarrow E \quad \text{(grad B premješten iza grada D)}$$

Lokalne pretrage moguće su i u genetskom programiranju u kojem su rješenja predstavljena kao stabla koja simbolički opisuju računalne programe, matematičke izraze ili logičke konstrukcije. Stabla su sastavljena od operatora (čvorova u stablu) i terminala (listova stabla), primjerice, varijabli (x, y) ili konstanti.



Slika 6.1: Primjer početnog rješenja u obliku stabla u GP-u (izraz $(x \times y) + \frac{y}{2}$).

Slika 6.1 prikazuje početno rješenje (izraz $(x \times y) + \frac{y}{2}$) čiju okolicu rješenja želimo generirati. U nastavku je primjer tri moguća okolna rješenja (Slika 6.2, Slika 6.3 i Slika 6.4).

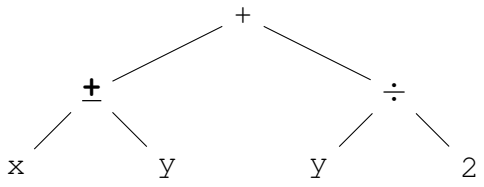


Slika 6.2: Primjer okolnog rješenja stabla nastalog zamjenom podstabla $(\times\, x\, y)$ podstablom $(\times\, x\, 3)$.

U prvom okolnom rješenju podstablo $(x \times y)$ zamijenjeno je novim podstablom $(x \times 3)$. Izraz nakon ove promjene postaje:

$$(x \times 3) + \frac{y}{2}$$

Promjenom podstabla pokušava se ispitati može li konstanta 3 umjesto varijable y dovesti do boljeg rješenja.

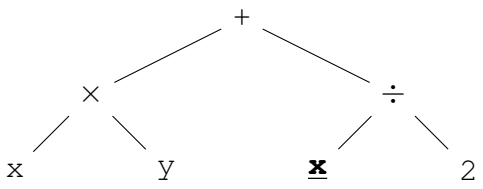


Slika 6.3: Primjer okolnog rješenja nastalog mutacijom operatora ($\times$ zamijenjen operatorom $+$).

U drugom okolnom rješenju operator $\times$ u lijevom podstablu zamijenjen je operatorom $+$. Izraz nakon ove promjene glasi:

$$(x + y) + \frac{y}{2}$$

Ova promjena ispituje utjecaj drugačije matematičke operacije na kvalitetu rješenja, odnosno provjerava hoće li zbrajanje varijabli x i y dovesti do boljeg rješenja od njihovog množenja.



Slika 6.4: Primjer okolnog rješenja nastalog mutacijom lista (y zamijenjen x-om).

U trećem okolnom rješenju varijabla y u desnom podstablu zamijenjena je varijablom x. Izraz nakon ove promjene postaje:

$$(x \times y) + \frac{x}{2}$$

Cilj je ove promjene istražiti hoće li upotreba varijable x umjesto y u drugom dijelu izraza dati bolje rješenje.

Neovisno o tipu problema i reprezentaciji rješenja, odabir strategije generiranja okolice ključan je za učinkovitost algoritma lokalnog pretraživanja jer izravno utječe na istraživanje prostora pretraživanja. Ipak, važno je napomenuti da se u postupcima lokalne pretrage obično izbjegava istovremeno mijenjati veći broj gena jer se takvim promjenama značajnije povećava udaljenost od početnog rješenja, zbog čega bi se generirana rješenja mogla naći izvan definirane okolice. Cilj je lokalne pretrage upravo istraživanje malih promjena oko trenutnog rješenja, čime se osigurava temeljita eksploatacija prostora neposredno uz početno rješenje. Veće promjene tipično pripadaju globalnim strategijama pretraživanja, čiji je cilj istraživanje udaljenijih područja prostora pretraživanja.

6.2 Metoda penjanja uz brijeg

Prva metoda lokalnog pretraživanja metoda je penjanja uz brijeg (engl. *hill climbing*). Često se naziva i pohlepnom metodom lokalnog pretraživanja (engl. *greedy local search*) [75] jer algoritam u svakom koraku odabire najbolje trenutno dostupno rješenje, bez razmatranja dugoročnih posljedica, što je karakteristika pohlepnih pristupa. Zbog takve prirode ovog algoritma često dolazi do zaglavljivanja u lokalnim optimumima iz kojih se teško može izaći bez dodatnih mehanizama poput upotrebe metode penjanja uz brijeg s nasumičnim ponovnim pokretanjem (engl. *random-restart hill climbing*) [76] ili metode simuliranog kaljenja (engl. *simulated annealing*) [77].

Ako proces traženja optimalnog rješenja zamislimo kao penjanje na vrh brijega, tada se ova metoda može opisati kao iterativno poboljšavanje trenutnog rješenja (jedinke) tako da

se u svakom koraku pokušava prijeći u susjedno rješenje s višom (boljom) vrijednošću dobrote. Ako bolje rješenje ne postoji (primjerice, u slučaju lokalnog maksimuma ili platoa) algoritam se zaustavlja.

Slika 6.5: Lokalno pretraživanje – metoda penjanja uz brijeg (engl. *hill climbing*).

Na Slici 6.5, za početna rješenja $x = 1.68$ i $x = 2.18$ lokalna pretraga metodom penjanja uz brijeg završit će u krajnje lijevom lokalnom maksimumu, budući da se okolica tih rješenja nalazi upravo na njegovom brijegu. Zatim, za početno rješenje $x = 6.39$ penjanje uz brijeg dovest će do globalnog maksimuma. Slično vrijedi i za druga početna rješenja $x = 8.28$ i $x = 9.72$ čija će lokalna pretraga ovom metodom rezultirati zadnjim (krajnje desnim) lokalnim maksimumom.

Iako bi algoritam penjanja uz brijeg u slučaju $x = 5.5$ uobičajeno prekinuo pretragu (vrijednost se nalazi na platou), u nekim varijantama dopušten je i prelazak na susjedno rješenje s jednakom vrijednošću dobrote. Tako se omogućuje nastavak pretraživanja jer neka od tih rješenja na platou mogu imati bolje susjede. Primjerice, početno rješenje $x = 5.5$ ima istu vrijednost dobrote kao i njegova okolina $(5.2, 5.3, ..., 5.8)$, ali rješenje 5.8 u svom susjedstvu ima bolje rješenje koje vodi do globalnog maksimuma.

Opći oblik metode penjanja uz brijeg prikazan je Algoritmom 6.1. Lokalna pretraga odvija se sve dok nije zadovoljen kriterij zaustavljanja, primjerice, kada je dosegnut maksimalni broj iteracija, postignuta zadovoljavajuća vrijednost dobrote ili neki drugi unaprijed definirani uvjet. Nakon generiranja i evaluacije okoline početnog rješenja `r`, najbolje susjedno rješenje pohranjuje se u `r'`. Ako je vrijednost dobrote tog rješenja manja ili jednaka vrijednosti dobrote početnog rješenja, tada napretka nema i pretraga se prekida (dosegnut je plato, lokalni ili globalni optimum). U suprotnom, trenutno najbolje rješenje `r'` postaje novo početno rješenje `r`, te se lokalna pretraga ponavlja. Na kraju se vraća rješenje `r` koje predstavlja lokalni, a u najboljem slučaju i globalni optimum.

Algoritam 6.1 Penjanje uz brijeg (engl. *hill climbing*).

```
1: function PENJANJEUZBRIJEG(r)                          ▷ Početno rješenje r.
2:     while nije kraj do          ▷ Petlja se izvodi dok nije zadovoljen kriterij zaustavljanja.
3:         r' ← najboljiSusjed(r)                        ▷ Odabir najboljeg susjednog rješenja.
4:         if dobrota(r') ≤ dobrota(r) then              ▷ Ako nema poboljšanja, prekidamo.
5:             break
6:         end if
7:         r ← r'                                        ▷ Prijelaz na bolje rješenje.
8:     end while
9:     return r                                          ▷ Vraća se lokalni (ili globalni) optimum.
10: end function
```

Važno je napomenuti da ova inačica algoritma pretpostavlja da se u svakoj iteraciji pregledava cijela okolina trenutnog rješenja. Međutim, zbog veličine te okoline, to često nije izvedivo u praksi. U takvim slučajevima može se primijeniti stohastička inačica algoritma [78] koja ne pretpostavlja da će bolje susjedno rješenje uvijek biti pronađeno. Umjesto toga, radi s ograničenim brojem slučajno generiranih susjeda u svakoj iteraciji i omogućuje nastavak pretrage čak i kada trenutno ne dolazi do poboljšanja.

Također je moguće koristiti i lokalno snopovsko pretraživanje (engl. *local beam search*) [79]. Algoritam započinje generiranjem k slučajnih rješenja (primjerice, njih 5). U svakoj iteraciji, za svako trenutno rješenje generira se skup svih susjednih rješenja. Od svih generiranih susjeda (bez uključivanja roditelja) odabire se novih k najboljih rješenja koja se koriste u sljedećoj iteraciji.

Spomenuti pristupi posebno su korisni u problemima s velikim ili diskretnim prostorom pretraživanja, poput optimizacije rasporeda nastave, slaganja termina ispita ili odabira konfiguracije sustava gdje nije moguće učinkovito pretražiti sve susjede zbog njihove brojnosti ili složenosti.

Primjer 6.1: Implementacija metode penjanja uz brijeg u programskom jeziku C++.

```
Jedinka PenjanjeUzBrijeg(Jedinka Rjesenje, unsigned int optimum) {
    Rjesenje.evaluiraj(optimum);
    bool zapeo = false;
    int brojIteracija = 0;
    const int maksIteracija = 1000;

    // ponavljaj lokalnu pretragu sve dok algoritam nije zapneo u lokalnom optimumu,
    // maksimalni broj iteracija nije preden, i nije pronadeno optimalno rjesenje
    while (!zapeo && brojIteracija <= maksIteracija && Rjesenje.uBroj() != optimum) {
        zapeo = true;
        Jedinka najbolji = Rjesenje;
        for (int i = 0; i < Rjesenje.gen.size(); ++i) {
            Jedinka susjed = Rjesenje;
            susjed.flipBit(i);
            susjed.evaluiraj(optimum);

            if (susjed.dobrota < najbolji.dobrota) {
                najbolji = susjed;
                zapeo = false;
            }
        }
        Rjesenje = najbolji;
```

```
23          ++brojIteracija;
24      }
25      return Rjesenje; // vrati najbolje pronadeno rjesenje
26  }
```

Za demonstraciju, pretpostavimo da pomoću metode penjanja uz brijeg želimo dohvatiti binarnu reprezentaciju broja 175, pri čemu krećemo od početnog rješenja – broja 53. Vrijednost dobrote pojedinog rješenja određena je njegovom udaljenošću od tražene vrijednosti, što ovaj problem čini problemom minimizacije (najbolja vrijednost dobrote iznosi 0). U tu svrhu korištena je metoda `PenjanjeUzBrijeg` (Primjer 6.1, Prilog 9.22), koja za svaku jedinku (rješenje) generira okolna rješenja inverzijom bitova. Svako rješenje predstavljeno je binarnom reprezentacijom duljine 8 bitova, pa će stoga za svako početno rješenje postojati ukupno 8 okolnih rješenja (Tablica 6.1).

Tablica 6.1: Koraci algoritma penjanja uz brijeg za problem dohvata binarne reprezentacije cijelog broja 175 s početnim rješenjem - brojem 53.

	Broj (bin)	**Broj (dek)**	**Dobrota**
Iteracija 1	00110101	**53**	**122**
Susjed 1	10110101	**181**	**6**
Susjed 2	01110101	117	58
Susjed 3	00010101	21	154
Susjed 4	00100101	37	138
Susjed 5	00111101	61	114
Susjed 6	00110001	49	126
Susjed 7	00110111	55	120
Susjed 8	00110100	52	123
Iteracija 2	10110101	**181**	**6**
Susjed 1	00110101	53	122
Susjed 2	11110101	245	70
Susjed 3	10010101	149	26
Susjed 4	10100101	165	10
Susjed 5	10111101	189	14
Susjed 6	10110001	**177**	**2**
Susjed 7	10110111	183	8
Susjed 8	10110100	180	5
Iteracija 3	10110001	**177**	**2**
Susjed 1	00110001	49	126
Susjed 2	11110001	241	66
Susjed 3	10010001	145	30
Susjed 4	10100001	161	14
Susjed 5	10111001	185	10
Susjed 6	10110101	181	6
Susjed 7	10110011	179	4

	Broj (bin)	Broj (dek)	Dobrota
Susjed 8	10110000	**176**	**1**
Iteracija 4	10110000	**176**	**1**
Susjed 1	00110000	48	127
Susjed 2	11110000	240	65
Susjed 3	10010000	144	31
Susjed 4	10100000	160	15
Susjed 5	10111000	184	9
Susjed 6	10110100	180	5
Susjed 7	10110010	178	3
Susjed 8	10110001	177	2

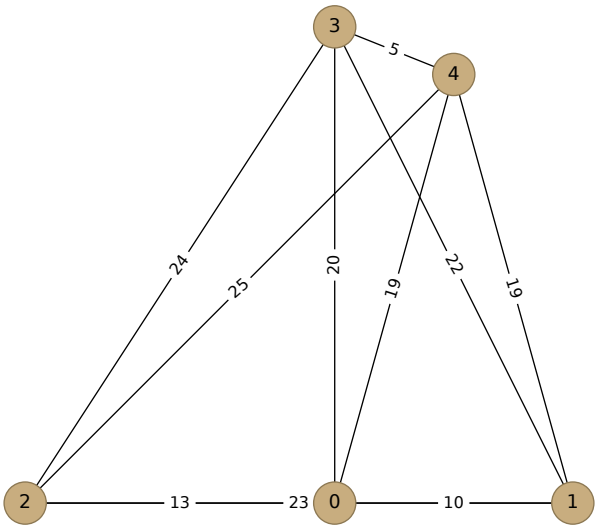
U prvoj iteraciji algoritam započinje s početnim rješenjem, brojem 53, čija je binarna reprezentacija 00110101. Dobrota ovog rješenja iznosi 122, što ukazuje na to da je prilično udaljeno od ciljne vrijednosti 175. Da bi pronašao bolje rješenje, algoritam generira sve moguće susjede početnog rješenja tako da preokreće po jedan bit u svakom koraku. Među tim susjedima najpovoljniji se pokazuje broj 181 (10110101), čija je dobrota znatno kvalitetnija — svega 6. Time se već u prvom koraku postiže veliko poboljšanje, što ilustrira učinkovitost metode penjanja uz brijeg pri brzom napredovanju iz loših početnih stanja.

U drugoj iteraciji algoritam kreće od rješenja 181. Premda se već radi o kvalitetnom rješenju, pretraga susjedstva pokazuje da postoji još povoljniji kandidat — broj 177 s binarnim zapisom 10110001 i dobrotom 2. Ovdje se izmjena događa u manje značajnom bitu, što pokazuje da algoritam sada ulazi u fazu preciznijeg prilagođavanja vrijednosti.

Treća iteracija počinje s rješenjem 177, a u njegovom susjedstvu se pojavljuje broj 176 (10110000) s još manjom (kvalitetnijom) dobrotom, svega 1. U četvrtoj iteraciji analiza susjedstva tog rješenja pokazuje da nijedno susjedno rješenje ne nudi daljnje poboljšanje (svi imaju veću ili jednaku dobrotu). Primjerice, najbolji susjed u četvrtoj iteraciji s vrijednošću 177 već je obrađen u prethodnoj iteraciji i ima veću (lošiju) dobrotu od trenutnog rješenja. Budući da ne postoji mogućnost napretka, algoritam se u ovom trenutku zaustavlja, što je tipično ponašanje metode penjanja uz brijeg pri dosezanju lokalnog (ili globalnog) optimuma.

Istu metodu možemo upotrijebiti i pri rješavanju problema trgovačkog putnika gdje pokušavamo pronaći najkraći put među zadanim gradovima. Slika 6.6 demonstrira jedan primjer takvog problema.

Ako kao početno rješenje problema sa Slike 6.6 uzmemo putanju $0 \to 1 \to 2 \to 4 \to 3 \to 0$ s vrijednošću dobrote (ukupnom duljinom puta) 83, konačno rješenje bit će putanja $2 \to 0 \to 1 \to 4 \to 3 \to 2$ s dobrotom 71. Do tog rješenja dolazi se u svega tri iteracije metode penjanja uz brijeg (Tablica 6.2), dok se cjelovita implementacija rješenja ovog problema u programskom jeziku C++ nalazi u Prilogu 9.23.



Slika 6.6: Primjer problema trgovačkog putnika (pet gradova).

Tablica 6.2: Koraci algoritma penjanja uz brijeg za problem trgovačkog putnika s pet gradova.

	Put	**Dobrota**
Iteracija 1	**0 → 1 → 2 → 4 → 3 → 0**	**83**
Susjed 1	**1** → **0** → 2 → 4 → 3 → 1	**75**
Susjed 2	**2** → 1 → **0** → 4 → 3 → 2	81
Susjed 3	**4** → 1 → 2 → **0** → 3 → 4	80
Susjed 4	**3** → 1 → 2 → 4 → **0** → 3	109
Susjed 5	0 → **2** → **1** → 4 → 3 → 0	80
Susjed 6	0 → **4** → 2 → **1** → 3 → 0	109
Susjed 7	0 → **3** → 2 → 4 → **1** → 0	98
Susjed 8	0 → 1 → **4** → **2** → 3 → 0	98
Susjed 9	0 → 1 → **3** → 4 → **2** → 0	75
Susjed 10	0 → 1 → 2 → **3** → **4** → 0	81
Iteracija 2	**1 → 0 → 2 → 4 → 3 → 1**	**75**
Susjed 1	**0** → **1** → 2 → 4 → 3 → 0	83
Susjed 2	**2** → 0 → **1** → 4 → 3 → 2	**71**
Susjed 3	**4** → 0 → 2 → **1** → 3 → 4	82
Susjed 4	**3** → 0 → 2 → 4 → **1** → 3	99
Susjed 5	1 → **2** → **0** → 4 → 3 → 1	82
Susjed 6	1 → **4** → 2 → **0** → 3 → 1	99
Susjed 7	1 → **3** → 2 → 4 → **0** → 1	100
Susjed 8	1 → 0 → **4** → **2** → 3 → 1	100
Susjed 9	1 → 0 → **3** → 4 → **2** → 1	83
Susjed 10	1 → 0 → 2 → **3** → **4** → 1	71
Iteracija 3	**2 → 0 → 1 → 4 → 3 → 2**	**71**

	Put	Dobrota
Susjed 1	$\underline{\mathbf{0}} \to \underline{\mathbf{2}} \to 1 \to 4 \to 3 \to 0$	80
Susjed 2	$\underline{\mathbf{1}} \to 0 \to \underline{\mathbf{2}} \to 4 \to 3 \to 1$	75
Susjed 3	$\underline{\mathbf{4}} \to 0 \to 1 \to \underline{\mathbf{2}} \to 3 \to 4$	81
Susjed 4	$\underline{\mathbf{3}} \to 0 \to 1 \to 4 \to \underline{\mathbf{2}} \to 3$	98
Susjed 5	$2 \to \underline{\mathbf{1}} \to \underline{\mathbf{0}} \to 4 \to 3 \to 2$	81
Susjed 6	$2 \to \underline{\mathbf{4}} \to 1 \to \underline{\mathbf{0}} \to 3 \to 2$	98
Susjed 7	$2 \to \underline{\mathbf{3}} \to 1 \to 4 \to \underline{\mathbf{0}} \to 2$	97
Susjed 8	$2 \to 0 \to \underline{\mathbf{4}} \to \underline{\mathbf{1}} \to 3 \to 2$	97
Susjed 9	$2 \to 0 \to \underline{\mathbf{3}} \to 4 \to \underline{\mathbf{1}} \to 2$	80
Susjed 10	$2 \to 0 \to 1 \to \underline{\mathbf{3}} \to \underline{\mathbf{4}} \to 2$	75

Algoritam započinje s početnom putanjom $0 \to 1 \to 2 \to 4 \to 3 \to 0$, čija je ukupna duljina (dobrota) 83. U prvoj iteraciji algoritam generira sve moguće susjedne putanje, pri čemu se svaka susjedna putanja dobiva zamjenom para gradova u trenutnom rješenju. Među svim generiranim susjedima, najbolju dobrotu ima **Susjed 1**, koji zamjenjuje gradove na pozicijama 0 i 1, pri čemu dobivamo novu putanju $1 \to 0 \to 2 \to 4 \to 3 \to 1$ s dobrotom 75. Budući da je nova putanja bolja, algoritam prihvaća to rješenje i prelazi u drugu iteraciju.

U drugoj iteraciji ponovno se generira deset susjeda zamjenom svih parova gradova. Najbolje rješenje postiže **Susjed 2**, koji zamjenjuje gradove na pozicijama 0 i 2. Dobivena putanja $2 \to 0 \to 1 \to 4 \to 3 \to 2$ ima dobrotu 71, što predstavlja poboljšanje, pa algoritam prihvaća to rješenje kao početno rješenje u trećoj iteraciji algoritma. Međutim, iako se i u trećoj iteraciji generira deset susjeda, niti jedan od njih ne nudi poboljšanje u odnosu na najbolje rješenje iz druge iteracije. Najbolji susjedi imaju dobrotu 75, što je lošije od trenutnog (početnog) rješenja te iteracije - 71. Budući da nema susjeda s boljom dobrotom, algoritam se zaustavlja.

U svega tri iteracije, algoritam penjanja uz brijeg pronašao je lokalno optimalnu putanju duljine 71. Ovaj primjer zorno ilustrira prednosti algoritma u brzom pronalaženju dobrih rješenja, ali i njegovu sklonost zaustavljanju u lokalnim optimumima zbog nedostatka mehanizama za istraživanje šireg prostora rješenja.

6.3 Penjanje uz brijeg s nasumičnim pokretanjem

Uspjeh metode penjanja uz brijeg uvelike ovisi o odabiru početnog rješenja, zbog čega algoritam često zaglavi u lokalnom optimumu. Da bi se taj problem ublažio, moguće je više puta pokrenuti algoritam, svaki put s drukčijim (nasumično odabranim) početnim rješenjem. Takav pristup, poznat kao *penjanje uz brijeg s nasumičnim ponovnim pokretanjem* (engl. *random-restart hill climbing*) [76], povećava vjerojatnost pronalaska globalnog optimuma budući da se pretražuju različiti dijelovi prostora rješenja.

Primjerice, u prethodnom problemu pronalaska binarne reprezentacije cijelog broja 175,

algoritam penjanja uz brijeg s početnim rješenjem - brojem 53, zaglavio je u lokalnom optimumu u četvrtoj iteraciji (Tablica 6.1). Da bismo pronašli točno rješenje, možemo primijeniti metodu penjanja uz brijeg s nasumičnim ponovnim pokretanjem.

Primjer 6.2: Metoda penjanja uz brijeg s nasumičnim ponovnim pokretanjem implementirana u programskom jeziku C++.

```
Jedinka PenjanjeUzBrijegRestart(unsigned int optimum, int brojRestarta) {
    Jedinka najboljeRjesenje;
    najboljeRjesenje.dobrota = UINT8_MAX;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 1);

    for (int i = 0; i < brojRestarta && najboljeRjesenje.uBroj() != optimum; ++i) {
        // Generiraj slucajno pocetno rjesenje
        Jedinka pocetno;
        generate(pocetno.gen.begin(), pocetno.gen.end(), [&]() { return distrib(gen) % 2;
    });
        std::cout << "\n>>> Nasumicni restart " << (i + 1)
                  << " | pocetak = " << pocetno.uBroj() << std::endl;

        // Lokalna pretraga iz trenutnog pocetnog rjesenja
        Jedinka lokalno = PenjanjeUzBrijeg(pocetno, optimum);

        if (lokalno.dobrota < najboljeRjesenje.dobrota)
            najboljeRjesenje = lokalno;

        std::cout << ">>> Kraj restarta " << (i + 1) << ": " << lokalno.uBroj()
            << " (dobrota = " << lokalno.dobrota << ")\n";
    }
    return najboljeRjesenje;
}
```

Koristeći već implementiranu funkciju `PenjanjeUzBrijeg` (Primjer 6.1, Prilog 9.22), nova funkcija `PenjanjeUzBrijegRestart` (Primjer 6.2) generira nasumična početna rješenja te ponavlja pozive funkcije `PenjanjeUzBrijeg` sve dok se ne pronađe globalni optimum ili dok se ne dosegne unaprijed zadani broj restarta. Primjerice,

```
unsigned int optimum = 175;
int pocetnoRjesenjeBroj = 53;
Jedinka pocetnoRjesenje(brojUBitove(pocetnoRjesenjeBroj));
std::cout << "pocetak = " << pocetnoRjesenje.uBroj() << std::endl;

// ako penjanje uz brijeg nije rezultiralo globalnim optimumom
if (PenjanjeUzBrijeg(pocetnoRjesenje, optimum).uBroj() != optimum) {
    // penjanje uz brijeg s nasumicnim ponovnim pokretanjem (maks. 5 poziva)
    Jedinka rezultat = PenjanjeUzBrijegRestart(optimum, 5);
    std::cout << "\n>>> Najbolje globalno rjesenje = " << rezultat.uBroj()
              << ", Dobrota: " << rezultat.dobrota << std::endl;
}
```

Sada ponovno krećemo od početnog rješenja (broja 53). Budući da u ovom slučaju optimalno rješenje (broj 175) neće biti pronađeno standardnom metodom penjanja uz brijeg, poziva se funkcija `PenjanjeUzBrijegRestart` koja će najviše 5 puta pokušati pronaći optimalno rješenje nasumičnim odabirom početnog rješenja. Jedan od mogućih rezultata izvršavanja prethodnog koda je sljedeći:

```
pocetak = 53
```

```
Iteracija: 1 = 181
Iteracija: 2 = 177
Iteracija: 3 = 176

>>> Nasumicni restart 1 | pocetak = 96
Iteracija: 1 = 224
Iteracija: 2 = 160
Iteracija: 3 = 176
>>> Kraj restarta 1: 176 (dobrota = 1)

>>> Nasumicni restart 2 | pocetak = 32
Iteracija: 1 = 160
Iteracija: 2 = 176
>>> Kraj restarta 2: 176 (dobrota = 1)

>>> Nasumicni restart 3 | pocetak = 124
Iteracija: 1 = 126
Iteracija: 2 = 127
>>> Kraj restarta 3: 127 (dobrota = 48)

>>> Nasumicni restart 4 | pocetak = 39
Iteracija: 1 = 167
Iteracija: 2 = 175
>>> Kraj restarta 4: 175 (dobrota = 0)

>>> Najbolje globalno rjesenje = 175, Dobrota: 0
```

Na početku, algoritam kreće s fiksnim početnim rješenjem — brojem 53. Već nakon tri iteracije penjanjem dolazi do lokalnog optimuma s vrijednošću 176, koja je vrlo blizu cilja, ali nije jednaka optimumu. Budući da se globalni optimum nije postigao, aktivira se mehanizam nasumičnog ponovnog pokretanja gdje se algoritam pokreće s novim, nasumično odabranim početnim rješenjima.

Tijekom prvog restarta, algoritam nasumično kreće s rješenjem 96 koje se u tri koraka poboljšava do 176. Iako je to kvalitetno rješenje, ono nije idealno. Drugi restart započinje s brojem 32 i brzo konvergira na istu vrijednost 176. Ova pojava potvrđuje da je 176 vrlo čest lokalni optimum u prostoru rješenja te se lako dostiže iz različitih početnih točaka.

Treći restart započinje s brojem 124 i nakon dvije iteracije završava na 127, što je znatno udaljenije od optimuma (dobrota $= 48$). Ovo pokazuje da nisu sva početna rješenja jednako korisna jer neka vode i prema lošim lokalnim optimumima. Konačno, u četvrtom restartu, algoritam kreće od 39 i u dvije iteracije uspješno dolazi do globalnog optimuma 175 (dobrota $= 0$). Na kraju se kao najbolje globalno rješenje izdvaja upravo to rješenje. Cjelovita implementacija i demonstracija ove metode u programskom jeziku C++ nalazi se u Prilogu 9.24.

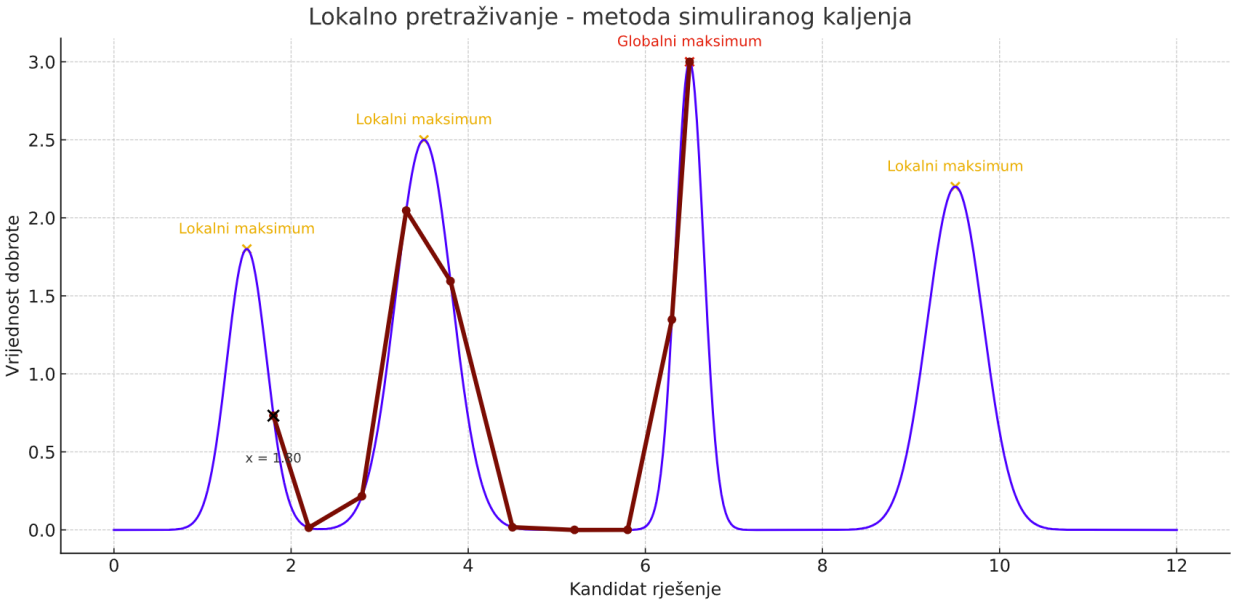
Ovaj primjer jasno pokazuje zašto je korisno koristiti nasumična ponovna pokretanja: iako jedno izvođenje algoritma može zaglaviti u lokalnom optimumu, višestruki pokušaji iz ra-

zličitih početnih točaka povećavaju vjerojatnost pronalaska globalnog optimuma. Također, izlaz ilustrira kako različita početna rješenja mogu dovesti do vrlo različitih rezultata. U nekim slučajevima (primjerice, kod restarta s početnim brojem 96 ili 32) algoritam brzo konvergira prema lokalnom optimumu 176 koji je blizu cilja, ali nije točan. U trećem pokušaju, početno rješenje 124 vodi prema znatno lošijem rješenju 127 koje je na velikoj udaljenosti od cilja. S druge strane, početna vrijednost 39 u četvrtom pokušaju dovodi do točnog rješenja 175 već nakon dvije iteracije.

Ipak, metoda nasumičnog ponovnog pokretanja može imati i određene nedostatke. Kod složenih problema s velikim prostorom rješenja ili puno lokalnih optimuma, vjerojatnost da će slučajna početna rješenja dovesti do globalnog optimuma može biti vrlo mala. To će onda kao posljedicu imati veliki broj nepotrebnih ponovnih pokretanja. Također, ova metoda ne koristi podatke iz prethodnih pokušaja, tj. ne "uči" iz neuspjelih pokušaja. Štoviše, može se dogoditi čak više pokretanja s identičnim početnim rješenjima.

6.4 Simulirano kaljenje

Učestalo zapinjanje u lokalnim optimumima, karakteristično za pohlepni pristup metode penjanja uz brijeg, može se ublažiti primjenom metode simuliranog kaljenja (engl. *simulated annealing* [69]). Umjesto prihvaćanja samo najboljih rješenja (pohlepni pristup), ova metoda povremeno prihvaća i lošija rješenja. Time se omogućuje izlazak iz lokalnih optimuma i prelazak u druga područja prostora pretraživanja koja potencijalno vode prema globalnom optimumu. Jedan takav primjer prikazan je na Slici 6.7 gdje bi početno rješenje, korištenjem metode penjanja uz brijeg, ostalo zarobljeno u krajnje lijevom lokalnom maksimumu. Ipak, primjenom metode simuliranog kaljenja, algoritam uspijeva izbjeći čak dva lokalna maksimuma i dosegnuti globalni maksimum.



Slika 6.7: Lokalno pretraživanje – metoda simuliranog kaljenja (engl. *simulated annealing*).

Algoritam simuliranog kaljenja inspiriran je fizikalnim procesom termalnog kaljenja metala u kojem se materijal najprije zagrijava do visoke temperature, a zatim polako hladi, omogućujući atomima da preurede svoju strukturu u energetski najstabilniji raspored, odnosno *globalni minimum* energije. Ideja je prenesena u domenu optimizacije gdje se traži globalno optimalno rješenje unutar kompleksnog prostora rješenja s mnogo lokalnih optimuma.

Algoritam 6.2 Simulirano kaljenje (engl. *simulated annealing*).

function SIMULIRANOKALJENJE(r, T_0, α, $T_{\min}$) ▷ Početno rješenje r, temperatura T_0, faktor hlađenja α, minimalna temperatura $T_{\min}$.
  $T \leftarrow T_0$ ▷ Inicijalizacija temperature.
  $r_{\text{najbolji}} \leftarrow r$ ▷ Pamti se najbolje rješenje.
  while $T > T_{\min}$ **and nije kraj do** ▷ Petlja dok traje kaljenje.
    $r' \leftarrow$ generirajSusjeda(r) ▷ Nasumično rješenje iz okoline.
    $\Delta \leftarrow$ dobrota(r) $-$ dobrota(r') ▷ Gubitak u kvaliteti.
    if $\Delta < 0$ **then** ▷ Ako je novo rješenje bolje (maksimizacija).
      $r \leftarrow r'$ ▷ Prihvaća se novo rješenje.
      if dobrota(r) $>$ dobrota(r_{najbolji}) **then**
        $r_{\text{najbolji}} \leftarrow r$ ▷ Ažurira se najbolje rješenje.
      end if
    else
      $v \leftarrow \text{rand}(0, 1)$ ▷ Slučajan broj iz $[0, 1]$.
      $p \leftarrow e^{-\Delta/T}$ ▷ Metropolisova vjerojatnost.
      if $v < p$ **then**
        $r \leftarrow r'$ ▷ Prihvaća se lošije rješenje s vjerojatnošću p.
      end if
    end if
    $T \leftarrow T \cdot \alpha$ ▷ Hlađenje temperature.
  end while
  return r_{najbolji} ▷ Vraća se najbolje pronađeno rješenje.
end function

Algoritam započinje s inicijalnim (početnim) rješenjem i dodjeljuje mu neku visoku početnu temperaturu. U svakoj iteraciji algoritam nasumično generira jedno susjedno rješenje (primjerice, promjenom jednog gena u kromosomu). Ako novo rješenje ima bolju vrijednost dobrote, tada se ono uvijek prihvaća. Međutim, ako je novo rješenje lošije, ono se ipak može prihvatiti s određenom vjerojatnošću koja ovisi o trenutnoj temperaturi (Algoritam 6.2).

Vjerojatnost prihvaćanja lošijeg rješenja temelji se na Metropolisovoj formuli [80]:

$$P = e^{-\frac{\Delta E}{T}}$$

gdje ΔE označava razliku u vrijednosti dobrote između trenutnog i novog rješenja. Ako je novo rješenje bolje, prihvaća se bezuvjetno. U suprotnom, lošije rješenje može se prihvatiti s određenom vjerojatnošću koja opada s padom temperature.

Nakon svake iteracije temperatura se smanjuje prema nekoj funkciji hlađenja. Najčešće se koristi geometrijsko hlađenje prema formuli $T = T \cdot \alpha$, gdje je $\alpha \in (0, 1)$ faktor hlađenja. Alternativno se koristi logaritamsko ili linearno hlađenje. Algoritam završava kada temperatura padne ispod unaprijed definiranog minimuma, kad je postignut zadani broj iteracija ili kad je postignuto idealno rješenje.

Primjer 6.3: Metoda simuliranog kaljenja implementirana u programskom jeziku C++.

```
Jedinka SimuliranoKaljenje(
    Jedinka r,                  // pocetno rjesenje
    unsigned int optimum,       // ciljana vrijednost
    double T0,                  // pocetna temperatura
    double Tmin,                // minimalna temperatura
    double alpha,               // faktor hladenja
    int maksIteracija           // maksimalni broj iteracija
) {
    std::random_device rd;
    std::mt19937 rng(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    double T = T0;
    r.evaluiraj(optimum);
    Jedinka rNajbolji = r;
    int iter = 0;
    while (T > Tmin && iter < maksIteracija && rNajbolji.uBroj() != optimum) {
        Jedinka rSusjed = generirajSusjeda(r, rng);
        rSusjed.evaluiraj(optimum);
        int delta = rSusjed.dobrota - r.dobrota;

        if (delta < 0) { // ako je susjedno rjesenje bolje (blize optimumu)
            r = rSusjed;
            if (r.dobrota < rNajbolji.dobrota)
                rNajbolji = r;
        }
        else { // susjedno rjesenje je losije
            double vjerojatnost = std::exp(-delta / T);
            if (dist(rng) < vjerojatnost)
                r = rSusjed;
        }
        std::cout << "Iteracija: " << iter + 1
            << ", Broj: " << r.uBroj()
            << ", Dobrota: " << r.dobrota
            << ", Temp: " << T << std::endl;
        T *= alpha; // smanjivanje temperature (hladenje)
        ++iter;
    }
    return rNajbolji;
}
```

Primjer 6.3 (Prilog 9.25) prikazuje C++ implementaciju metode simuliranog kaljenja za problem pronalaska binarne reprezentacije cijelog broja. Budući da je u ovom slučaju riječ o problemu minimizacije, ΔE označava razliku u vrijednosti dobrote između susjednog i trenutnog rješenja. Ako je $\Delta E < 0$, tada je susjedno rješenje bolje (bliže je traženoj vrijednosti) i automatski se prihvaća. S druge strane, lošije rješenje također može biti prihvaćeno s određenom vjerojatnošću koja ovisi o Metropolisovoj formuli, odnosno o trenutnoj temperaturi. Što je temperatura viša (u početnim iteracijama), to je veća i vjerojatnost prihvaćanja lošijih rješenja. Kako se broj iteracija povećava, temperatura se postupno smanjuje, a time opada i vjerojatnost prihvaćanja lošijih rješenja.

Poziv ove metode možemo testirati na istom primjeru kao i pri demonstraciji metode pen-

janja uz brijeg, gdje se tražila binarna reprezentacija cijelog broja 175 uz početno rješenje - broj 53:

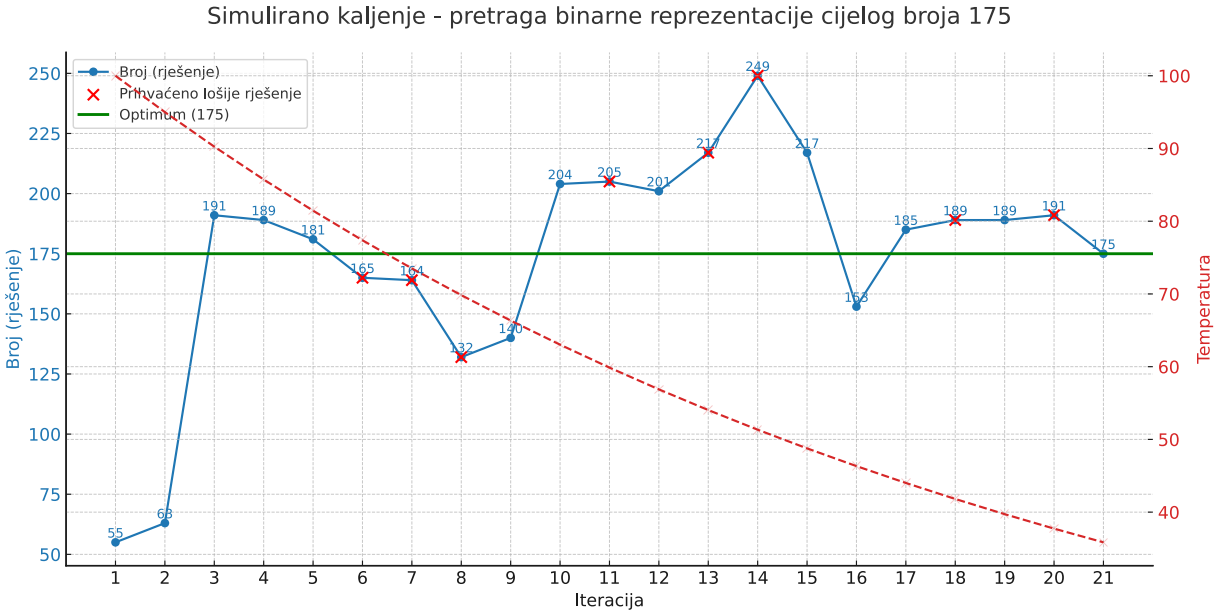
```
unsigned int optimum = 175;
Jedinka pocetnoRjesenje(brojUBitove(53));

// Parametri SA algoritma
double T0 = 100.0;
double Tmin = 0.01;
double alpha = 0.95;
int maksIteracija = 1000;

Jedinka rezultat = SimuliranoKaljenje(
        pocetnoRjesenje, optimum, T0, Tmin, alpha, maksIteracija
    );
std::cout << "Najbolji broj: " << rezultat.uBroj()
          << ", Dobrota: " << rezultat.dobrota << std::endl;
```

Budući da je metoda stohastička, svaki put može dati različit rezultat. Jedan od mogućih rezultata prikazan je u nastavku:

```
Iteracija: 1, Broj: 55, Dobrota: 120, Temp: 100
Iteracija: 2, Broj: 63, Dobrota: 112, Temp: 95
Iteracija: 3, Broj: 191, Dobrota: 16, Temp: 90.25
Iteracija: 4, Broj: 189, Dobrota: 14, Temp: 85.7375
Iteracija: 5, Broj: 181, Dobrota: 6, Temp: 81.4506
Iteracija: 6, Broj: 165, Dobrota: 10, Temp: 77.3781
Iteracija: 7, Broj: 164, Dobrota: 11, Temp: 73.5092
Iteracija: 8, Broj: 132, Dobrota: 43, Temp: 69.8337
Iteracija: 9, Broj: 140, Dobrota: 35, Temp: 66.342
Iteracija: 10, Broj: 204, Dobrota: 29, Temp: 63.0249
Iteracija: 11, Broj: 205, Dobrota: 30, Temp: 59.8737
Iteracija: 12, Broj: 201, Dobrota: 26, Temp: 56.88
Iteracija: 13, Broj: 217, Dobrota: 42, Temp: 54.036
Iteracija: 14, Broj: 249, Dobrota: 74, Temp: 51.3342
Iteracija: 15, Broj: 217, Dobrota: 42, Temp: 48.7675
Iteracija: 16, Broj: 153, Dobrota: 22, Temp: 46.3291
Iteracija: 17, Broj: 185, Dobrota: 10, Temp: 44.0127
Iteracija: 18, Broj: 189, Dobrota: 14, Temp: 41.812
Iteracija: 19, Broj: 189, Dobrota: 14, Temp: 39.7214
Iteracija: 20, Broj: 191, Dobrota: 16, Temp: 37.7354
Iteracija: 21, Broj: 175, Dobrota: 0, Temp: 35.8486
Najbolji broj: 175, Dobrota: 0
```

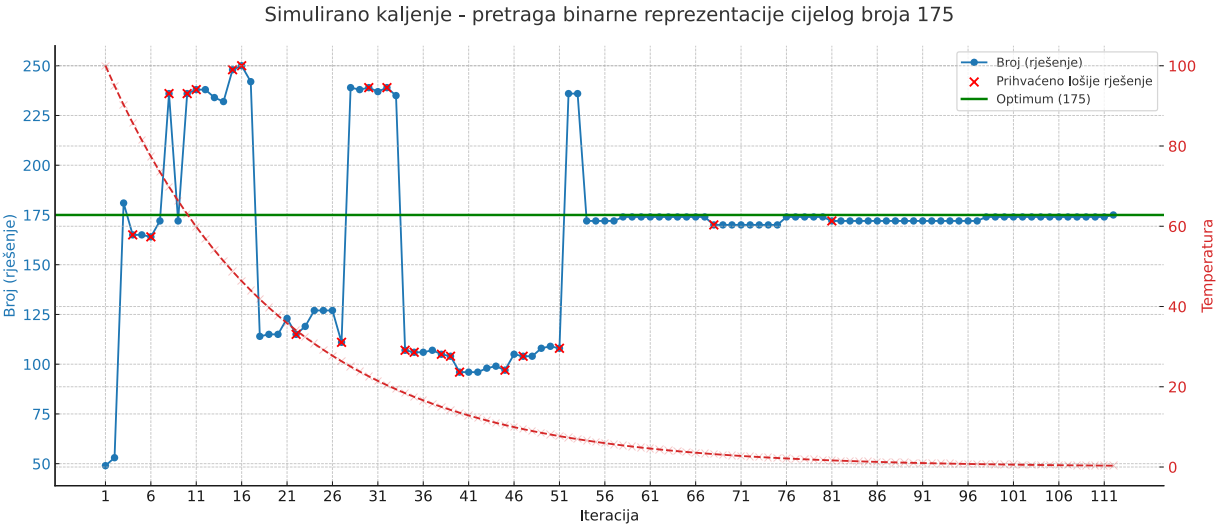
U prikazanom primjeru vidimo kako algoritam simuliranog kaljenja tijekom 21 iteracije pronalazi optimalnu binarnu reprezentaciju za broj 175. Na početku algoritam brzo prelazi iz početnog broja 53 na znatno bolje vrijednosti, poput 191 i 181, koje su relativno blizu optimuma. Tijekom srednjih iteracija (primjerice, između 7. i 16.) algoritam oscilira između boljih i lošijih rješenja, što je posljedica još uvijek veće prisutne vjerojatnosti prihvaćanja lošijih rješenja. Na kraju, kako temperatura opada, ta vjerojatnost postaje niža, a algoritam postupno konvergira prema optimumu kojeg dostiže točno u 21. iteraciji.

Ovaj rezultat ilustrira prednost stohastičkog pristupa koji u ranim fazama istražuje prostor rješenja šire, dok u kasnijima postupno usmjerava pretragu prema optimumu.



Slika 6.8: Simulirano kaljenje – pretraga binarne reprezentacije cijelog broja 175.

Trenutna rješenja i pripadne vrijednosti temperature po svakoj iteraciji prikazani su na Slici 6.8. Graf dodatno potvrđuje tipično ponašanje algoritma simuliranog kaljenja - u početnim fazama, pri višim temperaturama, pretraga uključuje i lošija rješenja, što omogućuje izlazak iz lokalnih optimuma. S postupnim opadanjem temperature rješenja se stabiliziraju i sve više usmjeravaju prema globalnom optimumu. To je pogotovo izraženo kod lokalne pretrage simuliranim kaljenjem s većim brojem iteracija (Slika 6.9).



Slika 6.9: Simulirano kaljenje – druga pretraga binarne reprezentacije cijelog broja 175.

Kao što je vidljivo iz prethodnih rezultata i grafova, simulirano kaljenje karakteriziraju

dvije faze ponašanja. U početku, kada je temperatura visoka, algoritam često prihvaća lošija rješenja. To mu omogućuje da „iskače" iz lokalnih optimuma i istražuje različita područja prostora rješenja. Kasnije, kako temperatura opada, ponašanje algoritma postaje sličnije metodi penjanja uz brijeg, odnosno prihvaćaju se samo bolja (ili jednako dobra) rješenja, a algoritam se usredotočuje na fino ugađanje rješenja.

Također, teoretski je dokazano u [81] da, ako se temperatura hladi dovoljno sporo, algoritam konvergira prema globalnom optimumu s vjerojatnošću 1. To potvrđuje da simulirano kaljenje nije heuristički pristup bez teorijske podloge, već ozbiljan optimizacijski algoritam temeljen na fizikalnim zakonima.

Prednosti algoritma uključuju otpornost na lokalne optimume, jednostavnost implementacije i robusnost u slučaju kompleksnih, nelinearnih prostora rješenja. Najveći nedostatak nešto je sporija izvedba u odnosu na deterministične metode kao što je penjanje uz brijeg, te osjetljivost na odabir parametara – početne temperature, brzine hlađenja i definicije susjednih rješenja.

6.5 Tabu pretraživanje

Jedan od pristupa kojim se mogu ublažiti učestali problemi lokalnih optimuma u heurističkim algoritmima je tabu pretraživanje (engl. *tabu search*) koje je uveo Fred Glover 1986. godine [82]. Sama riječ *tabu* označava nešto zabranjeno, a u kontekstu lokalne pretrage odnosi se na privremenu zabranu prihvaćanja već isprobanih rješenja ili poteza. Cilj je usmjeriti pretragu prema novim, neistraženim područjima prostora rješenja, čime se povećava vjerojatnost pronalaska boljih (često i globalnih) rješenja.

Temeljna ideja tabu pretraživanja jest vođenje tzv. tabu liste, u koju se privremeno pohranjuju nedavno posjećena rješenja ili izvršeni potezi (primjerice, zamjene vrijednosti, permutacije i sl.) da bi se spriječilo vraćanje na prethodno istražene konfiguracije. Duljina liste je ograničena, a kada dosegne svoj maksimalni kapacitet najstariji zapisi se uklanjaju prema načelu *First-In-First-Out* (FIFO) [83]. Ipak, važno je napomenuti da naprednije varijante tabu pretraživanja mogu uključivati i dinamičku duljinu tabu liste koja se prilagođava tijekom izvođenja pretrage. Primjerice, duljina se povećava kada algoritam stagnira, a smanjuje kada napreduje prema boljim rješenjima.

U praksi se najčešće koriste dvije vrste pamćenja – *pamćenje rješenja* (engl. *solution-based memory*) i *pamćenje poteza* (engl. *move-based memory*). Pri pamćenju rješenja u tabu listi bilježe se cijela rješenja, dok se pri pamćenju poteza pamti samo transformacija kojom se došlo do novog rješenja (primjerice, koje su dvije vrijednosti zamijenjene u permutaciji ili koji je bit bio invertiran). Takav pristup omogućuje učinkovitije vođenje evidencije o nedavnim koracima pretrage uz manju potrošnju memorije te sprječava ponavljanje istih transformacija, čak i ako one vode do različitih rješenja.

Važno je istaknuti da se u metodi tabu pretraživanja lokalna pretraga u svakoj iteraciji ne provodi iz globalno najbolje do tada pronađenog rješenja (kao što je to slučaj u metodi

penjanja uz brijeg), već iz trenutno prihvaćenog rješenja koje je rezultat prethodne iteracije. Takav pristup omogućuje algoritmu da napusti trenutno dobro rješenje i istraži nova područja pretraživačkog prostora, čak i ako to u početku vodi prema rješenju lošije kvalitete. Zahvaljujući mehanizmu tabu liste, algoritam se pritom štiti od povratka u već istražena rješenja tijekom određenog broja iteracija (ovisno o veličini tabu liste). Istovremeno, najbolje globalno rješenje pamti se zasebno i ažurira se isključivo kada se pronađe rješenje bolje kvalitete. Na taj način algoritam postiže ravnotežu između eksploatacije (lokalnog poboljšanja) i istraživanja novih dijelova prostora pretraživanja, što u praksi često rezultira pronalaskom kvalitetnijih rješenja u odnosu na klasične metode lokalne pretrage, poput penjanja uz brijeg.

Zahvaljujući svojoj sposobnosti balansiranja između eksploatacije i istraživanja prostora rješenja, tabu pretraživanje uspješno se primjenjuje u mnogim kombinatoričkim optimizacijskim problemima, uključujući raspoređivanje, rutiranje, problem trgovačkog putnika i mnoge druge.

Algoritam 6.3 Tabu pretraživanje (engl. *tabu search*).

function TABUPRETRAZIVANJE(r, L, $k_{\max}$) ▷ Početno rješenje r, tabu duljina L, maksimalan broj iteracija $k_{\max}$.
  $r_{\text{najbolji}} \leftarrow r$ ▷ Početno najbolje rješenje.
  tabuLista $\leftarrow$ prazna lista ▷ Tabu lista započinje prazna.
  $k \leftarrow 0$ ▷ Brojač iteracija.
  while $k < k_{\max}$ **and nije kraj do** ▷ Glavna petlja pretrage.
    $R' \leftarrow$ generirajSveSusjede(r) ▷ Skup svih susjeda trenutnog rješenja.
    $r' \leftarrow$ najbolji iz R' koji nije u tabuListi ▷ Tabu filtriranje.
    if r' **je tabu and** dobrota(r') $>$ dobrota(r_{najbolji}) **then** ▷ Aspiration kriterij.
      dopusti r' ▷ Prekida se tabu ako je rješenje dovoljno dobro.
    end if
    $r \leftarrow r'$ ▷ Pomakni se na novo rješenje.
    if dobrota(r) $>$ dobrota(r_{najbolji}) **then**
      $r_{\text{najbolji}} \leftarrow r$ ▷ Ažuriraj najbolje rješenje.
    end if
    dodajPotezUListu(tabuLista, r) ▷ Dodaj novo rješenje ili potez u tabu listu.
    if duljina(tabuLista) $> L$ **then**
      ukloniNajstariji(tabuLista) ▷ FIFO izbacivanje najstarijeg poteza.
    end if
    $k \leftarrow k + 1$
  end while
  return r_{najbolji} ▷ Vraća se najbolje pronađeno rješenje.
end function

Algoritam tabu pretraživanja (Algoritam 6.3) inicijalizira se početnim rješenjem r, koje se ujedno postavlja kao trenutno najbolje poznato rješenje r_{najbolji}. Također se definira prazna tabu lista koja privremeno pohranjuje poteze ili rješenja s ciljem izbjegavanja njihovog ponovnog korištenja tijekom ograničenog broja iteracija. U svakoj iteraciji algoritma generira se skup susjeda trenutnog rješenja, označen s R'. Unutar tog skupa traži se najbolje rješenje koje nije tabu, odnosno ono koje ne pripada tabu listi. Time se postiže kontrolirani mehanizam lokalne pretrage s pamćenjem koji usmjerava pretragu prema

novim regijama prostora rješenja.

Ako je trenutno najbolje rješenje u skupu susjeda ipak tabu, ali je bolje od dosad najboljeg poznatog rješenja r_{najbolji}, tada se primjenjuje tzv. *aspiracijski kriterij* koji omogućuje prekršaj tabu pravila ako bi to dovelo do globalnog napretka pretrage [70]. Međutim, primjena aspiracijskog kriterija ima smisla samo u određenim kontekstima. U slučaju kada se koristi pamćenje rješenja, kriterij je opravdan uglavnom u dinamičkim problemima gdje se funkcija dobrote može mijenjati tijekom vremena. U statičnim problemima s determinističkom evaluacijom, ponovno prihvaćanje već poznatog rješenja koje je ranije bilo odbijeno obično nema smisla jer se njegova evaluacija nije promijenila.

S druge strane, kada se koristi pamćenje poteza, aspiracijski kriterij ima znatno veću primjenu. Budući da isti potez može u različitim kontekstima voditi do različitih rješenja, moguće je da se potez koji je formalno tabu u nekoj iteraciji ipak isplati izvršiti ako dovodi do globalno boljeg rješenja. U tom slučaju, aspiracijski kriterij omogućuje da se tabu ograničenje zaobiđe, čime se izbjegava nepotrebno blokiranje napretka algoritma.

Nakon izbora najboljeg prihvatljivog susjeda r', algoritam ažurira trenutno rješenje r i, ako je potrebno, ažurira i globalno najbolje rješenje r_{najbolji}. Potom se novi potez ili rješenje dodaje u tabu listu, a ako je duljina liste premašila zadani prag L, najstariji element se uklanja po načelu FIFO (engl. *First-In-First-Out*) [83]. Ciklus pretrage ponavlja se sve dok ne bude zadovoljen kriterij zaustavljanja, nakon čega se vraća globalno najbolje pronađeno rješenje r_{najbolji}.

Primjer 6.4: Tabu pretraživanje s pamćenjem rješenja za problem trgovačkog putnika u programskom jeziku C++.

```
Jedinka TabuPretraga(Jedinka rjesenje, const std::vector<std::vector<int>>& matrica, int tabuDuljina = 5, int maksIteracija = 100) {
    rjesenje.evaluiraj(matrica);          // Evaluiraj pocetno rjesenje
    Jedinka najboljeRjesenje = rjesenje; // Inicijalno postavi kao najbolje
    std::list<Jedinka> tabuLista;  // Lista zabranjenih (nedavno posjecenih) rjesenja
    int iteracija = 0;

    while (iteracija < maksIteracija) {
        Jedinka najboljiKandidat;  // Trenutno najbolji susjed
        najboljiKandidat.dobrota = INT_MAX;
        bool nadjen = false;  // Pracenje je li pronaden kandidat izvan tabu liste

        // Generiranje i evaluacija svih susjeda
        for (size_t i = 0; i < rjesenje.put.size(); ++i) {
            for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
                Jedinka susjed = rjesenje;
                susjed.zamijeniGradove(i, j);  // Zamjena dvaju gradova = novi susjed
                susjed.evaluiraj(matrica);     // Izracunaj duljinu rute

                // Provjera nalazi li se susjed u tabu listi (usporeduje se put)
                auto it = std::find_if(tabuLista.begin(), tabuLista.end(),
                    [&](const Jedinka& e) { return e.put == susjed.put; });

                // Ako nije tabu i bolji je od dosadasnjeg kandidata - prihvati
                if (it == tabuLista.end() && susjed.dobrota < najboljiKandidat.dobrota) {
                    najboljiKandidat = susjed;
                    nadjen = true;
                }
            }
        }
```

```
30          if (!nadjen) {
31              // Ako su svi susjedi tabu - ignoriraj tabu listu i uzmi najbolji moguci susjed
32              for (size_t i = 0; i < rjesenje.put.size(); ++i) {
33                  for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
34                      Jedinka susjed = rjesenje;
35                      susjed.zamijeniGradove(i, j);
36                      susjed.evaluiraj(matrica);
37
38                      if (susjed.dobrota < najboljiKandidat.dobrota)
39                          najboljiKandidat = susjed;
40                  }
41              }
42          }
43          rjesenje = najboljiKandidat; // Pomak na novo rjesenje
44
45          // Azuriraj najbolje globalno rjesenje ako je novo bolje
46          if (rjesenje.dobrota < najboljeRjesenje.dobrota)
47              najboljeRjesenje = rjesenje;
48
49          // Dodaj novo rjesenje u tabu listu
50          tabuLista.push_back(rjesenje);
51
52          // Odrzavanje duljine tabu liste (FIFO)
53          if (tabuLista.size() > static_cast<size_t>(tabuDuljina))
54              tabuLista.pop_front(); // Izbaci najstarije
55          ++iteracija;
56      }
57      return najboljeRjesenje; // Vrati najbolje pronadeno rjesenje
58  }
```

Implementacija tabu pretrage s pamćenjem rješenja za problem trgovačkog putnika vidljiva je u Primjeru 6.4, odnosno Prilogu 9.26. U usporedbi s metodom penjanja uz brijeg (Primjer 6.1), metoda tabu pretraživanja koristi tabu listu koja sprečava algoritam da se vraća na nedavno posjećena rješenja. Na taj se način izbjegava kružno pretraživanje i omogućuje kretanje izvan lokalnih minimuma. Ako je trenutno najbolje susjedno rješenje označeno kao tabu, algoritam ga preskače i nastoji pronaći alternativno, još neproučeno rješenje. U slučaju da su svi susjedi tabu, tabu ograničenje privremeno se zanemaruje da bi se osiguralo napredak. Ovaj mehanizam omogućuje metodi da istražuje širi prostor rješenja, čak i ako to uključuje privremeno prihvaćanje lošijih rješenja.

U slučaju da se prethodno prikazano tabu pretraživanje za rješavanje problema trgovačkog putnika (Primjer 6.4) implementira tako da umjesto cijelih rješenja pamti poteze, tada tabu lista treba sadržavati informacije o zamjenama indeksa (tj. gradova) koje su izvršene u prethodnim iteracijama (Primjer 6.5, Prilog 9.27).

Primjer 6.5: Tabu pretraživanje s pamćenjem poteza za problem trgovačkog putnika u programskom jeziku C++.

```
1  Jedinka TabuPretraga(Jedinka rjesenje, const std::vector<std::vector<int>>& matrica, int
       tabuDuljina = 5, int maksIteracija = 100) {
2      rjesenje.evaluiraj(matrica);
3      Jedinka najboljeRjesenje = rjesenje;
4
5      // Umjesto pamcenja cijelog rjesenja, pamte se samo POTEZI (zamijenjeni indeksi)
6      std::list<std::pair<int, int>> tabuLista;
7
8      int iteracija = 0;
9      while (iteracija < maksIteracija) {
10         Jedinka najboljiKandidat;
11         najboljiKandidat.dobrota = INT_MAX;
12         int najboljiI = -1, najboljiJ = -1; // Sprema se najbolji potez
```

```
13
14          // Lokalna pretraga: pokusaj svih mogucih zamjena gradova
15          for (size_t i = 0; i < rjesenje.put.size(); ++i) {
16              for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
17                  Jedinka susjed = rjesenje;
18                  susjed.zamijeniGradove(i, j);
19                  susjed.evaluiraj(matrica);
20
21                  // Provjera je li POTEZ (i,j) u tabu listi
22                  bool jeTabu = std::find(tabuLista.begin(), tabuLista.end(), std::make_pair(
    i, j)) != tabuLista.end() ||
23                                std::find(tabuLista.begin(), tabuLista.end(), std::make_pair(
    j, i)) != tabuLista.end();
24
25                  // ASPIRACIJSKI KRITERIJ: dozvoljava tabu potez
26                  // ako vodi do boljeg globalnog rjesenja
27                  if (!jeTabu || susjed.dobrota < najboljeRjesenje.dobrota) {
28                      if (susjed.dobrota < najboljiKandidat.dobrota) {
29                          najboljiKandidat = susjed;
30                          najboljiI = i;
31                          najboljiJ = j;
32                      }
33                  }
34              }
35          }
36          // Azuriraj trenutno rjesenje
37          rjesenje = najboljiKandidat;
38
39          // Azuriraj globalno najbolje rjesenje
40          if (rjesenje.dobrota < najboljeRjesenje.dobrota)
41              najboljeRjesenje = rjesenje;
42
43          // Dodaj odabrani potez u tabu listu
44          if (najboljiI != -1 && najboljiJ != -1)
45              tabuLista.push_back({ najboljiI, najboljiJ });
46
47          // Odrzavaj duljinu tabu liste (FIFO)
48          if (tabuLista.size() > static_cast<size_t>(tabuDuljina))
49              tabuLista.pop_front();
50          ++iteracija;
51      }
52      return najboljeRjesenje;
53  }
```

U ovakvom pristupu gdje se pamte pojedini potezi neophodno je uvesti i aspiracijski kriterij koji omogućava prihvaćanje tabu poteza ako on vodi do boljeg rješenja od dosad globalno najboljeg. Naime, iako neka specifična zamjena gradova u problemu trgovačkog putnika u ranim iteracijama možda ne doprinosi poboljšanju rješenja (ili čak pogoršava rezultat), ista ta zamjena u drugačijem kontekstu (nakon što se ostali elementi puta promijene) može postati ključna za pronalazak globalno optimalnog rješenja.

Pristup pamćenja poteza najčešće omogućuje veću fleksibilnost u pretrazi prostora rješenja, a ujedno smanjuje i memorijsku složenost. Usporedba poteza zahtijeva manje resursa od usporedbe čitavih permutacija (kompletnih rješenja) jer se u tabu listu tada pohranjuju samo parovi indeksa (primjerice, (i, j)) koji označavaju izvršenu zamjenu gradova. Provjera je li neka nova zamjena tabu svodi se na jednostavnu provjeru prisutnosti tog para (ili simetričnog para (j, i)) u tabu listi. Nasuprot tome, kod pristupa koji pamti cijela rješenja potrebno je uspoređivati čitave vektore permutacija što je računalno skuplje, kako u vremenskom pogledu (veći broj elemenata za usporedbu), tako i u pogledu potrošnje memorije (potrebno je pohraniti cijelu permutaciju za svako rješenje u tabu listi).

6.6 Memetski algoritam

Memetski algoritmi (engl. *memetic algorithms*) predstavljaju klasu metaheurističkih optimizacijskih metoda koje kombiniraju globalno pretraživanje evolucijskih algoritama s tehnikama lokalne pretrage. Ovaj pristup omogućuje iskorištavanje prednosti oba svijeta: istraživanja šireg prostora rješenja putem operatora poput križanja i mutacije te iskorištavanja lokalnih informacija radi dodatnog poboljšanja rješenja [71].

Naziv *memetski* inspiriran je pojmom *mem*, jedinicom kulturne informacije koju je, analogno genu u biologiji, uveo Richard Dawkins [84]. U kontekstu ovih algoritama, *mem* simbolizira ideju da se rješenja, osim nasljeđivanjem, mogu i lokalno usavršavati, čime se oponaša proces kulturne adaptacije i učenja unutar populacije. Na taj način svaka jedinka ima mogućnost samostalnog poboljšanja pregledom svog neposrednog okruženja (okolice) prije nego što sudjeluje u daljnjem razmnožavanju unutar populacije.

Cilj memetskog algoritma nije zamijeniti genetski pristup, već ga nadograditi učinkovitom lokalnom pretragom koja omogućuje bržu konvergenciju prema kvalitetnim rješenjima. Ipak, lokalno pretraživanje koje je sastavni dio memetskog algoritma primjenjuje se na ograničen i selektivan način da bi se izbjegla prebrza konvergencija i gubitak raznolikosti u populaciji. Obično se provodi nad manjim brojem jedinki i uz ograničen broj iteracija, čime se održava ravnoteža između pretraživanja i eksploatacije.

Zbog svoje fleksibilnosti i učinkovitosti, memetski algoritmi uspješno su primijenjeni u rješavanju niza složenih optimizacijskih problema, uključujući problem trgovačkog putnika, raspoređivanje resursa, bio-informatički zadaci, automatizirani dizajn i projektiranje itd. [85, 86].

Algoritam 6.4 Memetski algoritam s lokalnom pretragom nad najboljim jedinkama.

1: $t \leftarrow 0$
2: $P(t) \leftarrow$ INICIJALIZACIJA
3: EVALUACIJA($P(t)$)
4: **while** uvjet završetka nije zadovoljen **do**
5: $\quad P'(t) \leftarrow$ SELEKCIJA($P(t)$)
6: $\quad P''(t) \leftarrow$ KRIŽANJE($P'(t)$)
7: $\quad P'''(t) \leftarrow$ MUTACIJA($P''(t)$)
8: $\quad$ EVALUACIJA($P'''(t)$)
9: $\quad P(t+1) \leftarrow$ LOKALNAPRETRAGANADNAJBOLJIMA($P'''(t), N$)
10: $\quad t \leftarrow t+1$
11: **end while**

Ovisno o prirodi problema, u memetskom algoritmu moguće je primijeniti različite strategije lokalne pretrage. Primjerice, Algoritam 6.4 prikazuje slučaj u kojem se lokalna pretraga provodi nad N najboljih jedinki u populaciji. U takvom pristupu, evaluacija je nužna odmah nakon mutacije, odnosno prije primjene lokalne pretrage da bi se moglo odabrati koje će jedinke biti predmet poboljšanja. Štoviše, između koraka mutacije i lokalne pretrage često se primjenjuje i mehanizam elitizma kojim se najbolje jedinke iz prethodne

generacije prenose u novu generaciju.

Osim poboljšavanja najboljih jedinki moguće je primijeniti i druge strategije. Jedna od njih lokalna je pretraga nad slučajno odabranim jedinkama s unaprijed definiranom vjerojatnošću. Druga je mogućnost *adaptivna lokalna pretraga*, kod koje se broj jedinki ili intenzitet pretrage dinamički prilagođavaju tijekom vremena, ovisno o brzini konvergencije ili promjenama u prosječnoj kvaliteti populacije. Također, lokalna pretraga može se ograničiti samo na one jedinke koje zadovoljavaju određeni kriterij raznolikosti (tzv. *selektivna lokalna pretraga*) čime se smanjuje rizik od preuranjene konvergencije. Nadalje, moguće je definirati i *frekvencijsku kontrolu* kojom se lokalna pretraga ne provodi u svakoj generaciji, već periodički (npr. svakih k generacija) ili stohastički.

Odabir odgovarajuće strategije primjene lokalne pretrage ima velik utjecaj na ravnotežu između eksploatacije i istraživanja, a time i na ukupnu učinkovitost memetskog algoritma. Također, učinak lokalne pretrage u memetskom algoritmu uvelike ovisi i o načinu na koji se rezultati te pretrage integriraju u populaciju.

Postoji više mogućih strategija zamjene, ovisno o tome nad kojom se jedinkom pretraga vrši te kako i gdje se poboljšana rješenja umeću. Jedan od najčešćih pristupa je strategija poznata kao *pohlepna zamjena* (engl. *greedy replacement*), prema kojoj najbolje susjedno rješenje zamjenjuje izvornu jedinku samo ako je kvalitetnije. Druga je mogućnost da najbolje susjedno rješenje ne zamjenjuje izvorno, već se umetne u populaciju, primjerice, zamjenom najlošijeg člana. Druga je mogućnost uključiti više rješenja dobivenih lokalnom pretragom u populaciju, primjerice, kroz zamjenu više (najčešće najlošijih) članova populacije. Tablica 6.3 prikazuje pregled nekih od najčešće korištenih strategija integracije rezultata lokalne pretrage u memetskom algoritmu.

Tablica 6.3: Česte strategije integracije rezultata lokalne pretrage u memetskom algoritmu [85].

Naziv strategije	**Opis**	**Zamjenjuje**	**Uvjet za zamjenu**
Pohlepna zamjena	Najbolji susjed zamjenjuje izvornu jedinku samo ako je kvalitetniji.	Izvorna jedinka	Ako je $f(x') > f(x)$
Bezuvjetna zamjena	Najbolji susjed uvijek zamjenjuje izvornu jedinku, neovisno o njegovoj kvaliteti.	Izvorna jedinka	Uvijek
Zamjena najlošijeg	Najbolji susjed se uključuje u populaciju zamjenom najlošeg člana.	Najlošija jedinka	Uvijek ili uvjetno (primjerice, samo ako je $f(x') > f(x)$)
Umetanje više rješenja	Više rješenja dobivenih lokalnom pretragom različitih jedinki uključuje se u populaciju.	Višestruki članovi populacije	Ovisno o kriteriju umetanja (npr. zamjenom najlošijih članova)

Kao što je prikazano, učinkovitost memetskog algoritma uvelike ovisi o dvjema ključnim dimenzijama – strategiji odabira jedinki nad kojima se lokalna pretraga primjenjuje te načinu integracije dobivenih rješenja u populaciju. Odabir jedinki može biti temeljen na elitizmu (najbolje jedinke), slučajnosti (stohastički odabir s određenom vjerojatnošću), kriterijima raznolikosti (selektivna pretraga), adaptivnim pravilima ili frekvencijskoj kontroli. Svaka od ovih strategija ima specifične prednosti – elitizam ubrzava konvergenciju,

dok selektivna i adaptivna primjena održavaju raznolikost i prilagodbu algoritma tijekom vremena.

S druge strane, strategije integracije rezultata lokalne pretrage određuju kako se novo rješenje koristi, odnosno zamjenjuje li izvorno rješenje (pohlepno ili bezuvjetno), zamjenjuje li nekog drugog člana populacije (primjerice, najlošijeg), ili se dodatno umetne u populaciju. I ovdje je odabir uvjetovan prirodom problema – pohlepna zamjena pogodna je za stabilne funkcije dobrote, dok umetanje više rješenja može povećati selekcijski pritisak.

U praksi, optimalna kombinacija ovih dvaju aspekata često se ne može unaprijed odrediti, već se donosi empirijski na temelju eksperimentiranja i analize ponašanja algoritma na konkretnom problemu. Stoga se u razvoju memetskih algoritama preporučuje dizajn koji omogućuje jednostavnu zamjenu ili kombinaciju strategija, čime se algoritam može prilagoditi specifičnostima različitih problema.

6.6.1 Metoda pohlepne zamjene

U nastavku prikažimo implementaciju i usporedbu genetskog i memetskog algoritma na jednostavnom, ali ilustrativnom problemu pretrage binarnog zapisa zadane cjelobrojne vrijednosti. Cilj je pronaći binarni niz maksimalne duljine 32 bita koji, kada se interpretira kao cijeli broj, bude što bliži unaprijed definiranoj ciljanoj vrijednosti (primjerice, broju `1234567890`). Svaka jedinka u populaciji predstavlja potencijalno rješenje, a vrijednost dobrote računa se kao apsolutna razlika između dekodirane vrijednosti i ciljne vrijednosti (problem minimizacije). Rješenje genetskim algoritmom implementirano je i prikazano u Prilogu 9.28 te koristi sljedeće evolucijske operatore:

- **Turnirska selekcija (veličine 2)**: za svaku novu jedinku nasumično se biraju dva kandidata, a dalje se prenosi onaj s boljom dobrotom.
- **Uniformno križanje**: svaki gen djeteta nasumično se nasljeđuje od jednog od roditelja.
- **Mutacija**: svaki bit (gen) ima fiksnu vjerojatnost promjene.
- **Elitizam**: postotak najboljih jedinki iz prethodne generacije prenosi se u novu.

Algoritam se izvršava sve dok se ne pronađe optimalno rješenje (dobrota $= 0$) ili dok se ne dosegne maksimalan broj generacija (2500). Navedenu implementaciju usporedit ćemo s implementacijom memetskog algoritma koji proširuje taj genetski algoritam dodavanjem lokalne pretrage nad određenim brojem najboljih jedinki trenutne populacije (Primjer 6.6). Ta pretraga omogućit će bolju eksploataciju lokalnog prostora rješenja, čime se povećava preciznost i ubrzava konvergencija prema optimalnom rješenju.

Primjer 6.6: Lokalna pretraga nad N najboljih jedinki u memetskom algoritmu (metoda pohlepne zamjene).

```
1 // Lokalna pretraga se primjenjuje nad predefiniranim brojem najboljih jedinki
2 void MemetskiAlgoritam::lokalnaPretraga(Populacija* P) {
3     // Sortiranje populacije po dobroti - najbolje jedinke dolaze na pocetak
```

```
    std::sort(P->jedinke.begin(), P->jedinke.end(), [](const Jedinka& a, const Jedinka& b){
        return a.dobrota < b.dobrota;
    });

    // Iteracija kroz najbolje jedinke koje ce biti lokalno poboljsane
    for (int i = 0; i < brojZaLokalnuPretragu; ++i) {
        Jedinka najbolji = P->jedinke[i]; // Pocetna jedinka koju pokusavamo poboljsati

        // Generiranje svih susjeda promjenom svakog pojedinog gena
        for (size_t g = 0; g < najbolji.geni.size(); ++g) {
            Jedinka susjed = najbolji;
            susjed.geni[g] = !susjed.geni[g]; // Invertiraj g-ti gen
            susjed.evaluiraj(ciljanaVrijednost); // Evaluiraj dobrotu susjeda

            // Ako je susjed kvalitetniji (ima manju dobrotu), postaje najbolji
            if (susjed.dobrota < najbolji.dobrota)
                najbolji = susjed;
        }
        P->jedinke[i] = najbolji;
    }
}
```

Nakon što se u populaciji provedu križanje i mutacija, nad unaprijed definiranim brojem najboljih jedinki provodi se lokalna pretraga. U sklopu te pretrage svaka se jedinka poboljšava tako da se generiraju svi njeni mogući susjedi (s razlikom u jednom bitu). Lokalna pretraga u ovom slučaju koristi pristup pohlepne zamjene gdje se najbolji susjed mijenja s izvornim rješenjem, ukoliko je kvaliteta najboljeg susjeda bolja od izvornog rješenja. Lokalna pretraga provodi se prije elitizma da eventualna degradacija rješenja izazvana lokalnom pretragom ne bi utjecala na očuvanje najboljih jedinki iz prethodne generacije (Prilog 9.29). Na taj se način osigurava da globalno najbolja rješenja nisu prebrisana ako bi lokalna pretraga slučajno odvela jedinku prema lošijem rješenju.

Tablica 6.4: Usporedba broja evaluacija za različit broj elitnih jedinki u lokalnoj pretrazi (metoda pohlepne zamjene).

Sjeme	El=0 (GA)	El=5 (MA)	El=10 (MA)	El=20 (MA)
1687280607	11.000	43.710	13.930	1.140
834276	8.000	1.890	31.030	3.810
298347162	NaN	3.120	11.650	1.140
40321789	10.750	14.190	4.810	9.150
3817265984	12.000	3.940	6.250	8.620
Prosjek	10.438	13.370	13.534	4.772

U svrhu usporedbe genetskog i memetskog algoritma, odabrano je pet nasumičnih početnih vrijednosti za generator slučajnih brojeva. Za svako od tih sjemena pokrenuto je izvršavanje genetskog algoritma (GA), kao i triju varijanti memetskog algoritma (MA) koje se razlikuju prema broju elitnih jedinki (5, 10 i 20) nad kojima se primjenjuje lokalna pretraga. Da bi usporedba bila što preciznija, analizirani su rezultati prema ukupnom broju evaluacija, a ne prema broju generacija. Naime, u slučaju memetskog algoritma broj generacija ne odražava točno računalnu složenost jer se dodatno, unutar svake generacije, obavlja lokalna pretraga s velikim brojem evaluacija susjeda (do 32 po jedinki) što značajno može utjecati na ukupno vrijeme izvođenja.

Rezultati prikazani u Tablici 6.4 pokazuju da genetski algoritam u jednom od pokretanja (sjeme 298347162) nije uspio pronaći optimalno rješenje u zadanom broju generacija. U ostalim slučajevima, prosječan broj evaluacija iznosio je 10.438. Nasuprot tome, memetski je algoritam u svim pokretanjima pronašao optimalno rješenje. Međutim, broj potrebnih evaluacija uvelike ovisi o broju elitnih jedinki obuhvaćenih lokalnom pretragom. Kod najmanjeg broja (5 jedinki), broj evaluacija u nekim je slučajevima bio veći nego kod GA, dok je kod najvećeg broja (20 jedinki) postignut najniži prosječni broj evaluacija, što ukazuje na bržu konvergenciju. Štoviše, u zadnjem je slučaju svako pokretanje memetskog algoritma dovelo do bržeg pronalaska optimalnog rješenja u usporedbi s genetskim algoritmom.

Ovi rezultati ukazuju na to da, ovisno o odabranim parametrima, lokalna pretraga može omogućiti bržu konvergenciju algoritma, ali i dovesti do većeg ukupnog broja evaluacija, osobito ako lokalna pretraga nije dovoljno učinkovita ili se primjenjuje na prevelikom broju jedinki. Odabir optimalnih vrijednosti parametara (kao što su veličina populacije, postotak elitizma, intenzitet mutacije, ali i broj jedinki za lokalnu pretragu) nije univerzalan i u pravilu ovisi o specifičnostima problema koji se rješava. Stoga se ti parametri najčešće određuju eksperimentalno, usporedbom performansi algoritma na većem broju slučajeva i analizom metrika poput broja generacija, ukupnih evaluacija, vremena izvođenja i pouzdanosti pronalaska optimalnog rješenja.

6.6.2 Metoda višestruke zamjene

Osim optimalnih vrijednosti parametara kojima se postiže najbrža konvergencija algoritma, predmet eksperimentiranja može biti i sama strategija lokalne pretrage. Tako smo, primjerice, prethodno koristili lokalnu pretragu s pohlepnom zamjenom, gdje najbolji susjed zamjenjuje izvorno rješenje ako je bolje kvalitete (Primjer 6.6). Radi demonstracije, sada ispitajmo alternativnu strategiju – lokalnu pretragu s višestrukom zamjenom (Primjer 6.7, Prilog 9.30) u kojoj ćemo pokušati generirati manju količinu okolnih rješenja.

Primjer 6.7: Lokalna pretraga nad N najboljih jedinki u memetskom algoritmu (metoda višestruke zamjene).

```
// Lokalna pretraga se primjenjuje nad predefiniranim brojem najboljih jedinki
void MemetskiAlgoritam::lokalnaPretraga(Populacija* P, std::mt19937& generator) {
    // Sortiranje populacije po dobroti - najbolje jedinke dolaze na pocetak
    std::sort(P->jedinke.begin(), P->jedinke.end(), [](const Jedinka& a, const Jedinka& b){
        return a.dobrota < b.dobrota;
    });
    std::vector<Jedinka> susjedi;
    std::uniform_int_distribution<> distribGen(0, 31);
    std::uniform_real_distribution<> distribVjerojatnost(0.0, 1.0);

    // Generiraj susjede za najbolji dio populacije
    for (int i = 0; i < brojZaLokalnuPretragu; ++i) {
        Jedinka original = P->jedinke[i];
        Jedinka kandidat = original;
        for (size_t g = 0; g < kandidat.geni.size(); ++g) {
            if (distribVjerojatnost(generator) <= 0.5)
                kandidat.geni[g] = !kandidat.geni[g];
        }
        kandidat.evaluiraj(ciljanaVrijednost);
        susjedi.push_back(kandidat);
    }
```

```
22
23      // Sortiraj generirane susjede po dobroti
24      std::sort(susjedi.begin(), susjedi.end(), [](const Jedinka& a, const Jedinka& b) {
25          return a.dobrota < b.dobrota;
26      });
27
28      // Zamijeni najlosije jedinke u populaciji generiranim susjedima
29      for (int i = 0; i < susjedi.size(); ++i)
30          P->jedinke[P->jedinke.size() - 1 - i] = susjedi[i];
31  }
```

U ovom slučaju, za svako od najboljih izvornih rješenja ne generiraju se svi mogući susjedi, već se susjedi stvaraju nad slučajno odabranim genima (bitovima) s unaprijed zadanom vjerojatnošću od 50 %. Nadalje, umjesto pohlepne zamjene izvornih jedinki, ova strategija zamjenjuje najlošije jedinke u populaciji generiranim susjedima, pri čemu izvorne jedinke ostaju netaknute.

Tablica 6.5: Usporedba broja evaluacija za različit broj elitnih jedinki u lokalnoj pretrazi (metoda umetanja više rješenja).

Sjeme	El=0 (GA)	El=5 (MA)	El=10 (MA)	El=20 (MA)
1687280607	11.000	13.000	17.410	20.230
834276	8000	373.570	8.570	675.250
298347162	NaN	20.395	7.790	14.560
40321789	10.750	11.725	NaN	8.890
3817265984	12.000	14.530	10.650	15.640
Prosjek	10.438	86.664	11.105	146.914

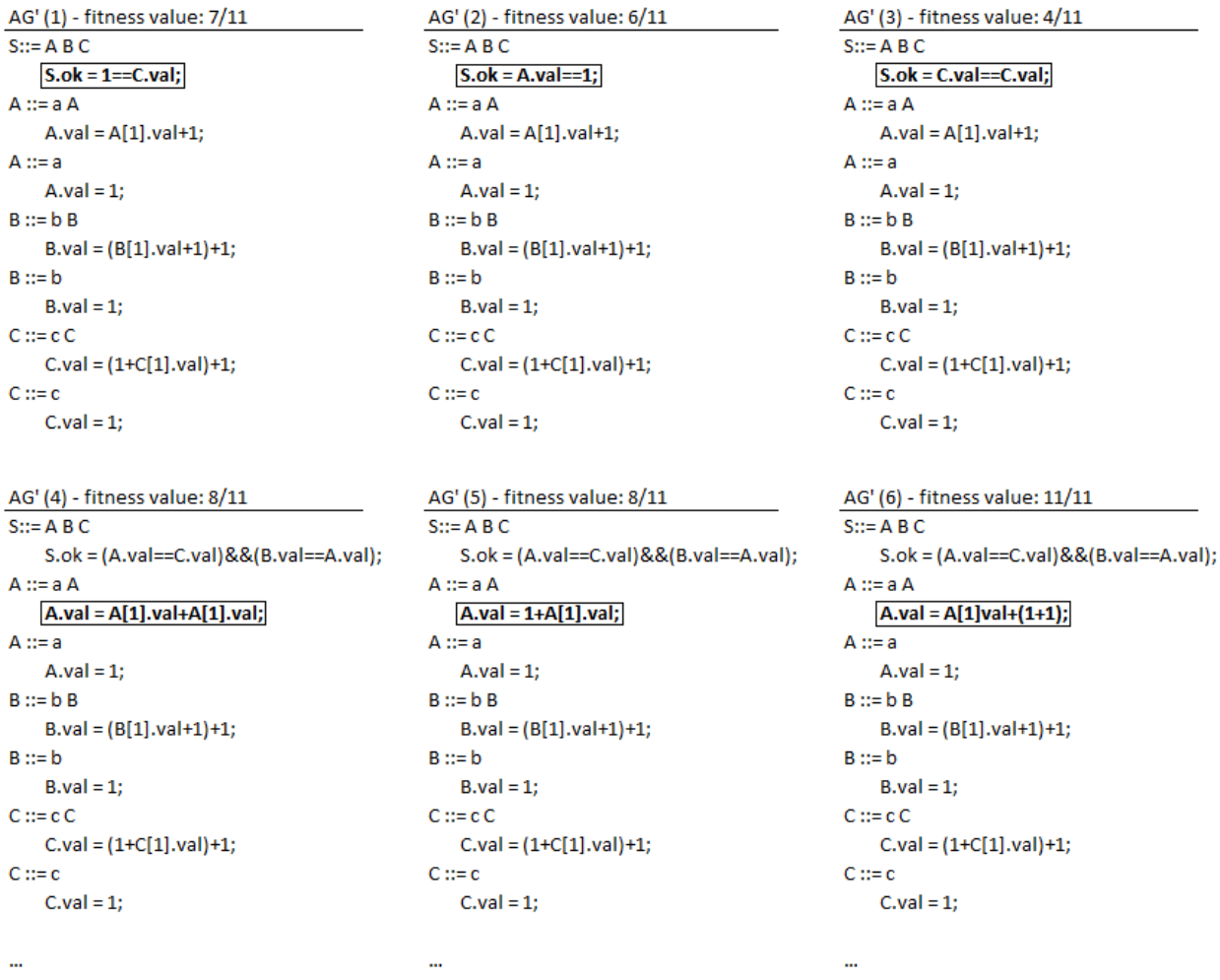
Eksperimentiranjem nove strategije lokalne pretrage (metoda umetanja više rješenja) dobiveni su rezultati prikazani u Tablici 6.5. Usporedbom s rezultatima prethodne strategije (metoda pohlepne zamjene) prikazanima u Tablici 6.4 može se primijetiti da nova metoda, unatoč konceptualnoj jednostavnosti i potencijalno manjem broju generiranih susjeda po jedinki, u većini slučajeva uzrokuje veći broj evaluacija do konvergencije. Posebno se ističe varijanta s 20 elitnih jedinki gdje višestruka zamjena pokazuje značajno lošije performanse (prosječno 146.914 evaluacija naspram samo 4.772 u slučaju pohlepne zamjene). Slično vrijedi i za varijantu s 5 elitnih jedinki gdje prosječan broj evaluacija znatno raste (86.664 u usporedbi s 13.370 kod pohlepne strategije).

U novoj strategiji, okolina se istražuje neselektivno i stohastički, bez jamstva da će se pronaći bolje lokalno rješenje. Stoga rezultati i nisu toliko neočekivani: višestruka zamjena, iako potencijalno jeftinija po pojedinačnoj iteraciji, može dovesti do slabijeg lokalnog poboljšanja, što zahtijeva veći broj generacija do konvergencije. Ipak, za potpuniju evaluaciju strategije bilo bi korisno dodatno istražiti utjecaj vjerojatnosti promjene gena, broja elitnih jedinki te kriterija za prihvaćanje rješenja da bi se bolje razumjela dinamika strategije i mogućnosti njezine optimizacije.

6.6.3 Lokalna pretraga s parametriziranom dubinom

U prikazanom problemu svaki gen može imati samo dvije vrijednosti (binarni genotip), pa je lokalna pretraga podrazumijevala promjenu jednog bita (invertiranje), čime se generirao samo jedan susjed po mogućem genu. Međutim, u mnogim realnim problemima svaki gen može imati više alternativnih vrijednosti, primjerice, cijele brojeve iz diskretnog skupa $\{0, 1, \ldots, k-1\}$. U takvim slučajevima, broj susjeda po genu može značajno rasti, što omogućuje uvođenje dodatnog parametra – *dubine pretrage po genu*. Umjesto da se svaki gen samo jednom promijeni na alternativnu vrijednost, možemo definirati koliko ćemo alternativnih vrijednosti po genu ispitivati primjerice, sve osim trenutne, samo jednu ili N nasumičnih).

Ovakva generalizacija omogućuje uvođenje dodatne fleksibilnosti i kontrolu nad intenzitetom lokalne pretrage. S druge strane, povećanjem broja mogućih vrijednosti po genu rastu i računalni zahtjevi lokalne pretrage pa je potrebno pažljivo balansirati između kvalitete lokalnog poboljšanja i troška evaluacije. U takvim slučajevima preporučuje se korištenje najčešće stohastičkih metoda za odabir alternativnih vrijednosti po genu.



Slika 6.10: Primjer susjedstva atributne gramatike jezika $a^n b^n c^n$ [87].

Slika 6.10 prikazuje primjer takve lokalne pretrage jednog netočnog rješenja atributne gramatike za jezik $a^n b^n c^n$ [87], i to za prve dvije semantičke jednadžbe. U prikazanom primjeru svaki gen (u ovom slučaju jednadžba) može imati velik broj varijanti, a svaka se od njih u postupku lokalne pretrage bira stohastički (nasumičnim odabirom). *Dubina pretrage po genu* iznosi 3, odnosno za svaki gen nasumično se odabiru i ispituju tri varijante jednadžbi. Zanimljivo je da jedna od tako odabranih jednadžbi, usprkos ograničenom i neselektivnom pretraživanju, vodi čak do globalnog optimuma, što pokazuje potencijal stohastičke lokalne pretrage i u složenijim primjerima problema.

Kao i kod većine metaheurističkih pristupa, odabir strategije lokalne pretrage također treba temeljiti na eksperimentalnoj evaluaciji, uzimajući u obzir specifičnosti problema te željeni balans između eksploatacije (iskorištavanja postojećeg znanja) i istraživanja novih rješenja. Učinkovitost pojedine strategije uvelike ovisi o prirodi problema, omjeru između lokalnog i globalnog pretraživanja koji se želi postići, kao i o dostupnim računalnim resursima. Zbog toga vrijedi naglasiti da bi kod drugačijih problema prethodno prikazane metode lokalne pretrage mogle dati suprotne rezultate od prikazanih. Stoga se ne može unaprijed generalizirati koja je strategija bolja, te je potrebna empirijska validacija u kontekstu svakog konkretnog problema.

POGLAVLJE 7

Dodatni primjeri optimizacijskih problema

U prethodnim poglavljima opisane su temeljne tehnike korištene u evolucijskim algoritmima, uključujući različite oblike selekcije, križanja, mutacije i lokalne pretrage. Njihove demonstracije uglavnom su se temeljile na problemima pretraživanja binarne reprezentacije zadanog cijelog broja (primjerice, prilozi 9.28, 9.29 i 9.30) te pronalaska najkraće rute trgovačkog putnika (primjerice, prilozi 9.23, 9.26 i 9.27). Stoga ovo poglavlje donosi dodatne primjere optimizacijskih problema iz različitih područja, ne samo radi ilustracije učinkovitosti evolucijskih algoritama, već i kao poticaj čitateljima za njihovu primjenu u vlastitim istraživanjima, projektima i razvoju rješenja.

Također je važno istaknuti da se svaki od sljedećih problema može riješiti na više različitih načina, ovisno o odabiru strategije (algoritma), reprezentaciji rješenja, definiciji funkcije dobrote i postavkama parametara. Budući da učinkovitost pojedinog pristupa uvelike ovisi o specifičnostima problema i kontekstu njegove primjene, ne postoji jednoznačno najbolje rješenje. Upravo zato preporučuje se čitateljima da navedene probleme pokušaju riješiti i nekim od alternativnih strategija opisanih u ovoj knjizi, kako bi uočili razlike u rezultatima i razumjeli prednosti i nedostatke pojedinih pristupa.

7.1 Minimizacija funkcije $|x - 5| + \sin(3x)$

U ovom primjeru želimo minimizirati funkciju:

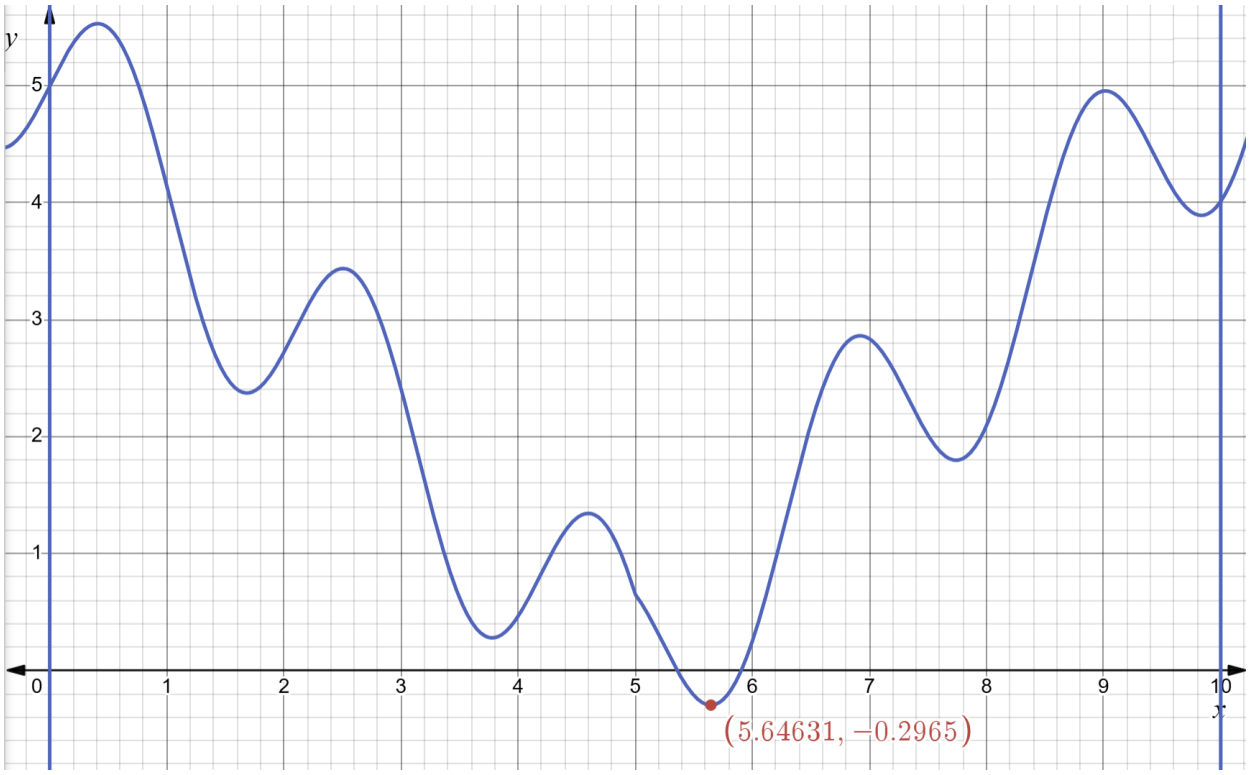
$$f(x) = |x - 5| + \sin(3x)$$

pri čemu je funkcija definirana na kontinuiranom intervalu $x \in [0, 10]$ (Slika 7.1). U teoriji, njezin minimum moguće je pronaći korištenjem pristupa *grube sile* (engl. *brute force*), ispitivanjem velikog broja točaka unutar tog intervala. No, takav pristup brzo postaje računalno nepraktičan kada se traži visoka preciznost. Primjerice, ako želimo pronaći minimum s preciznošću od 10^{-12}, morali bismo evaluirati barem 10^{12} točaka. Čak i ako evaluacija funkcije traje samo mikrosekundu, ukupno vrijeme izvođenja u ovom slučaju

prelazilo bi sate.

Osim toga, klasične metode temeljene na derivacijama također nisu prikladne za ovu funkciju. Zbog apsolutne vrijednosti, funkcija nema jasno definiran nagib (derivaciju) u točki $x = 5$, a oscilacije koje uzrokuje sinusni dio mogu uzrokovati da se pojedini algoritmi lokalne pretrage zaustave u nekom lokalnom minimumu, umjesto da pronađu globalni minimum (Slika 7.1).

Genetski algoritam pokazuje se boljim u takvim slučajevima. Umjesto da iscrpno pregledava sve moguće vrijednosti, on iterativno stvara i evaluira nova rješenja pomoću mehanizama selekcije, križanja i mutacije. Cjelovito rješenje genetskim algoritmom u programskom jeziku C++ prikazano je u Prilogu 9.31, a u nastavku će biti objašnjeni njegovi najvažniji dijelovi.



Slika 7.1: Graf funkcije $f(x) = |x-5| + \sin(3x)$, $x \in [0, 10]$.

Za rješavanje ovog problema koristi se realna reprezentacija rješenja. Svaka jedinka u populaciji predstavlja jednu realnu vrijednost $x \in [0, 10]$, dok se kvaliteta jedinke (dobrota) računa kao vrijednost funkcije $f(x)$, pri čemu je cilj minimizacija.

```
class Jedinka {
public:
    double x; // stvarna vrijednost, u intervalu [0, 10]
    double dobrota; // f(x)
...
```

U nastavku algoritam koristi standardne operatore genetskih algoritama, prilagođene realnim vrijednostima:

- **Inicijalizacija:** Generira se početna populacija jedinki s vrijednostima x nasumično odabranima unutar zadanog intervala $[0, 10]$.
- **Evaluacija:** Svaka se jedinka evaluira funkcijom $f(x) = |x - 5| + \sin(3x)$, pri čemu težimo minimizaciji (manjoj vrijednosti za $f(x)$).
- **Selekcija:** Primijenjena je *turnirska selekcija* (Poglavlje 2.4) veličine 3. Iz populacije se nasumično odaberu tri jedinke, a najbolja među njima ulazi u novu populaciju. Ovaj postupak ponavlja se dok se ne generira nova populacija jedinki iste veličine (100).
- **Križanje:** Korišteno je *aritmetičko križanje* (Poglavlje 3.4). Za dva roditelja x_1 i x_2, potomci se dobivaju kao:
$$x_1' = \alpha x_1 + (1 - \alpha)x_2$$
$$x_2' = (1 - \alpha)x_1 + \alpha x_2$$
gdje je $\alpha \in [0, 1]$ nasumično odabrana vrijednost. Ovakav pristup omogućuje stvaranje potomaka unutar raspona vrijednosti roditelja.
- **Mutacija:** Korištena je *Gaussova mutacija* (Poglavlje 4.2) s vjerojatnošću od 20 %. Ako se dogodi, vrijednosti x se dodaje nasumična vrijednost iz normalne raspodjele $N(0, 0.1)$. Većina generiranih vrijednosti bit će unutar intervala $[-0{,}1, 0{,}1]$, no moguća su i veća odstupanja. Ako nova vrijednost x izađe izvan dopuštenog intervala $[0, 10]$, vraća se unutar granica.
- **Elitizam:** Nije korišten.
- **Zaustavljanje:** Algoritam se izvodi unaprijed definirani broj generacija (200), a tijekom evolucije bilježi se najbolja pronađena jedinka. U ovom slučaju nije moguće postaviti uvjet za prekid temeljen na vrijednosti funkcije, budući da optimalna vrijednost nije unaprijed poznata.

Tijekom izvođenja, algoritam brzo konvergira prema globalnom minimumu. Primjerice:

```
Generacija 1: f(x)=-0.212626, x=5.50076
Generacija 2: f(x)=-0.2963576314154188, x=5.6521244909580926
Generacija 4: f(x)=-0.2964594764131629, x=5.6494522995462608
Generacija 5: f(x)=-0.2965014166735382, x=5.6464295570220404
Generacija 9: f(x)=-0.2965014790170860, x=5.6463213083535173
Generacija 10: f(x)=-0.2965014798157656, x=5.6463067137701657
Generacija 13: f(x)=-0.2965014798187986, x=5.6463074963591149
Generacija 16: f(x)=-0.2965014798188167, x=5.6463075619372143
Generacija 19: f(x)=-0.2965014798188170, x=5.6463075635141866
```

Vidljivo je da se već nakon desetak generacija algoritam približio optimalnom rješenju s visokom preciznošću. Najbolja pronađena vrijednost funkcije iznosi $-0{,}2965014798188170$,

a pripadna vrijednost $x = 5{,}6463075635141866$. Vrijednosti su stabilne već nakon 13. generacije, a preciznost rezultata odgovara razini preciznosti tipa `double` u programskom jeziku C++, odnosno oko 15 znamenki.

Ovaj primjer demonstrira kako jednostavan genetski algoritam može pronaći globalni minimum nelinearne i oscilatorne funkcije s više lokalnih minimuma. Primjena realne reprezentacije rješenja i aritmetičkog križanja čini algoritam jednostavnim za implementaciju i učinkovitim u praksi. Sličan pristup moguće je primijeniti na širok raspon problema iz numeričke optimizacije.

7.2 Problem osam kraljica

S problemom osam kraljica već smo se ukratko upoznali pri opisu permutacijske reprezentacije rješenja u Poglavlju 1.3.4, a u ovom ćemo poglavlju detaljno prikazati njegovo rješavanje primjenom genetskog algoritma (vidi Prilog 9.32).

Problem osam kraljica jedan je od najpoznatijih problemskih primjera iz područja umjetne inteligencije i kombinatorne optimizacije [88, 89]. Cilj je postaviti osam kraljica na standardnu šahovsku ploču veličine 8×8 tako da nijedna kraljica ne napada neku drugu. Budući da se kraljice mogu kretati vodoravno, okomito i dijagonalno, rješenje zahtijeva da u svakom retku, stupcu i dijagonali postoji najviše jedna kraljica.

Naizgled jednostavan, ovaj problem skriva iznimno veliku kombinatornu složenost. Ako se na ploču postavlja osam međusobno različitih kraljica na različita polja, ukupan broj mogućih pozicija iznosi:

$$\binom{64}{8} = 4\,426\,165\,368$$

To uključuje sve načine postavljanja osam neoznačenih kraljica na različita polja ploče, neovisno o tome napadaju li se međusobno. S obzirom na to da je samo 92 od tih konfiguracija valjano rješenje problema (gdje se kraljice međusobno ne napadaju), jasno je da bi pristup temeljen na pretraživanju *grubom silom* (engl. *brute force*) bio krajnje neučinkovit i računalno nepraktičan. Zbog navedenih karakteristika, problem osam kraljica često se koristi kao referentni primjer za ispitivanje učinkovitosti heurističkih i metaheurističkih algoritama.

Za rješavanje problema osam kraljica koristi se permutacijska reprezentacija rješenja. Svaka jedinka predstavlja jednu permutaciju brojeva od 0 do 7, pri čemu svaki broj označava stupac na koji se postavlja kraljica u odgovarajućem retku. Na taj se način automatski izbjegavaju međusobni napadi kraljica po retcima i stupcima, a preostaje samo provjera dijagonalnih konflikata. Evaluacija rješenja provodi se brojanjem parova kraljica koje se nalaze na istoj dijagonali. Cilj je algoritma minimizirati broj takvih konflikata.

```
1 class Jedinka {
2 public:
3     std::vector<int> pozicije; // pozicije[i] = stupac kraljice u retku i
```

```
4     int dobrota; // broj konflikata po dijagonalama (sto manje, to bolje)
5 ...
```

U nastavku algoritam koristi klasične operatore genetskih algoritama, prilagođene permutacijama:

- **Inicijalizacija:** Svaka se jedinka inicijalizira nasumičnom permutacijom brojeva 0 do 7, čime se osigurava jedinstvenost pozicija kraljica u svakom retku i stupcu.
- **Evaluacija:** Svaka se jedinka evaluira funkcijom koja broji broj parova kraljica na istoj dijagonali. Konflikt postoji ako je $|i - j| = |x_i - x_j|$, gdje su i i j indeksi redaka, a x_i, x_j stupci kraljica. Primjerice, za jedinku definiranu vektorom:

  $$\texttt{pozicije} = \begin{bmatrix} 0 & 2 & 4 & 6 & 1 & 3 & 5 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

  vrijednosti u gornjem retku predstavljaju stupce u koje su smještene kraljice, dok donji redak označava pripadajuće retke (indekse).

  Kraljice u retku 0 i 7 nalaze se na istoj dijagonali jer vrijedi:

  $$|i - j| = |0 - 7| = 7 \quad \text{i} \quad |x_i - x_j| = |0 - 7| = 7$$

- **Selekcija:** Primijenjena je *turnirska selekcija* (Poglavlje 2.4) veličine 3. Iz populacije se nasumično odaberu tri jedinke, a najbolja (s najmanje konflikata) ulazi u novu populaciju. Ovaj postupak ponavlja se dok se ne generira nova populacija.
- **Križanje:** Korišteno je *PMX* (engl. *Partially Mapped Crossover*) križanje (Poglavlje 3.6), koje je posebno pogodno za permutacijsku reprezentaciju jer osigurava da potomci ostanu valjane permutacije, odnosno da se kraljice ne ponove u istom stupcu. Primjerice, pretpostavimo da roditelji predstavljaju sljedeće permutacije, gdje svaki broj označava stupac kraljice u retku i:

  $$\begin{aligned} &\text{Roditelj 1:} \quad [3, \mathbf{1}, \mathbf{4}, \mathbf{6}, 0, 7, 5, 2] \\ &\text{Roditelj 2:} \quad [4, \mathbf{6}, \mathbf{2}, \mathbf{0}, 3, 7, 1, 5] \end{aligned}$$

  Segmenti između pozicija 1 i 3 razmijenjeni su između roditelja (označeno podebljano), pri čemu nastaju djelomični potomci:

  $$\begin{aligned} &\text{Potomak 1 (nepopravljeno):} \quad [3, \mathbf{6}, \mathbf{2}, \mathbf{0}, 0, 7, 5, 2] \\ &\text{Potomak 2 (nepopravljeno):} \quad [4, \mathbf{1}, \mathbf{4}, \mathbf{6}, 3, 7, 1, 5] \end{aligned}$$

  Oba potomka sadrže duplikate (primjerice, broj 0 i 2 u potomku 1), što bi značilo da su dvije kraljice postavljene u isti stupac, što nije dopušteno u kontekstu problema osam kraljica. Zato se primjenjuje popravak na temelju preslikavanja razmijenjenih vrijednosti kako bi se uklonili duplikati i zadržala valjana permutacija. Time se dobivaju nove jedinke koje predstavljaju ispravno rješenje bez ponavljanja stupaca, a samim time i bez napada po stupcima.

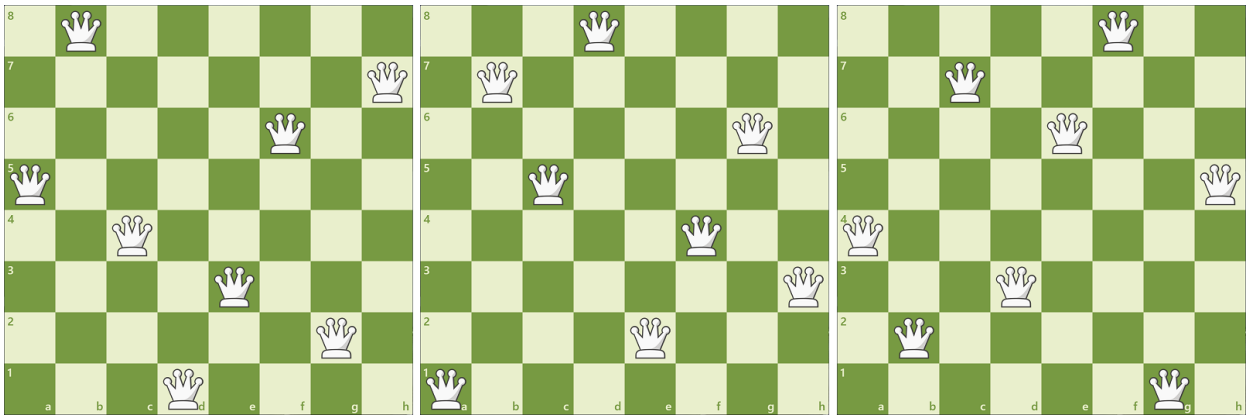
Potomak 1 (popravljeno): $[3, 6, 2, 0, \underline{1}, 7, 5, \underline{4}]$
Potomak 2 (popravljeno): $[\underline{2}, 1, 4, 6, 3, 7, \underline{0}, 5]$

- **Mutacija:** Korištena je jednostavna zamjena dvaju elemenata u permutaciji (mutacija zamjenom), s vjerojatnoćom od 20 %. Ovakva mutacija ne narušava permutacijsku strukturu rješenja.
- **Elitizam:** Nije korišten.
- **Zaustavljanje:** Algoritam se izvodi najviše 500 generacija ili dok se ne pronađe jedinka s nula konflikata, što predstavlja valjano rješenje problema.

Tijekom izvođenja, algoritam u većini pokretanja brzo konvergira prema optimalnom rješenju. Primjeri dobivenih rješenja u šahovskoj notaciji:

```
A5,B8,C4,D1,E3,F6,G2,H7
A1,B7,C5,D8,E2,F4,G6,H3
A4,B2,C7,D3,E6,F8,G1,H5
```

Ista rješenja vizualno su prikazana Slikom 7.2.



Slika 7.2: Neka od rješenja problema osam kraljica.

Primjena genetskog algoritma na problem osam kraljica pokazala se učinkovitom i elegantnom metodom za rješavanje ovog klasičnog problema kombinatorne optimizacije. Korištenjem permutacijske reprezentacije rješenja pojednostavljuje se prostor pretrage eliminacijom konflikata po redcima i stupcima, dok se dijagonalni konflikti uspješno razrješavaju evaluacijskom funkcijom.

Iako je problem osam kraljica relativno jednostavan, isti se pristup može primijeniti i na općeniti problem n-kraljica (cilj je postaviti n kraljica na šahovsku ploču dimenzija $n \times n$ tako da nijedna kraljica ne napada neku drugu), kao i na srodne probleme raspoređivanja, alokacije resursa i slično.

7.3 Problem ruksaka

Problem ruksaka već je ranije predstavljen u Poglavlju 1.3.1 kao primjer problema u kojem se može upotrijebiti binarna reprezentacija rješenja. Problem ruksaka (eng. *knapsack problem*) jedan je od najpoznatijih i najistraživanijih problema u području kombinatorne optimizacije [90, 91]. U svojoj osnovnoj varijanti, poznatoj kao "problem 0/1 ruksaka", zadan je skup n predmeta, pri čemu je svaki predmet i opisan svojom vrijednošću v_i i težinom w_i, uz ukupni kapacitet ruksaka W. Cilj je pronaći podskup predmeta koji maksimizira ukupnu vrijednost u ruksaku, uz ograničenje da ukupna težina ne prelazi dopušteni kapacitet ruksaka:

$$\begin{aligned} \text{maksimizirati} \quad & \sum_{i=1}^{n} v_i x_i, \quad x_i \in \{0,1\} \\ \text{uz ograničenje} \quad & \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0,1\} \end{aligned}$$

Rješenje se tipično predstavlja binarnim nizom duljine n, gdje binarna vrijednost $x_i = 1$ označava da je i-ti predmet uključen u ruksak, a vrijednost $x_i = 0$ da nije (primjer: Tablica 1.1). Ova vrsta reprezentacije čini problem posebno pogodnim za rješavanje različitim evolucijskim tehnikama, uključujući genetske algoritme, zbog svoje jednostavne strukture i lako definirane funkcije dobrote. U ovom slučaju, dobrota obično predstavlja ukupnu vrijednost predmeta u ruksaku. Međutim, u slučaju prekoračenja kapaciteta ruksaka, takvo rješenje ima najnižu moguću vrijednost dobrote (0 - prazan ruksak).

Cjelovito rješenje problema ruksaka pomoću genetskog algoritma u programskom jeziku C++ nalazi se u Prilogu 9.33, dok su u nastavku opisani njegovi najvažniji dijelovi:

- **Inicijalizacija:** Svaka se jedinka inicijalizira nizom slučajnih binarnih vrijednosti (0 ili 1), čime se generiraju različite kombinacije nasumično odabranih predmeta u ruksaku.
- **Evaluacija:** Svaka se jedinka evaluira tako da se zbrajaju vrijednosti i težine predmeta označenih s 1. Ako ukupna težina ne premašuje kapacitet ruksaka, ukupna vrijednost predstavlja dobrotu rješenja. U suprotnom, dobrota se postavlja na 0 jer takvo rješenje nije prihvatljivo.
- **Selekcija:** Primijenjena je *stohastička univerzalna selekcija* (Poglavlje 9.2). Izgrađuje se kumulativna distribucija normalizirane dobrote, a jedinke se biraju pomoću ravnomjerno razmaknutih točaka na intervalu $[0, 1]$, čime se osigurava raznolika, ali pravedna selekcija.
- **Križanje:** Koristi se *križanje u jednoj točki prekida* (Poglavlje 3.1). Za svaki par roditelja nasumično se određuje jedna točka prekida, nakon čega se zamjenjuju svi geni nakon te točke između roditelja. Time nastaju dva nova potomka koji kombiniraju gene oba roditelja.

- **Mutacija:** Primijenjena je *uniformna mutacija* (Poglavlje 4.1), pri čemu se svaki gen u jedinki može neovisno promijeniti s određenom vjerojatnosti (1%). Na taj se način održava raznolikost populacije i omogućuje izbjegavanje lokalnih optimuma.
- **Elitizam:** Jedna (najbolja) jedinka iz prethodne generacije prenosi se u novu generaciju kako bi se sačuvalo trenutno najbolje pronađeno rješenje (Poglavlje 5.1). U praksi je moguće prenijeti i veći broj jedinki iz prethodne generacije, no pretjerano elitistično ponašanje može smanjiti raznolikost nove populacije te dovesti do prebrze konvergencije prema lokalnom optimumu.
- **Zaustavljanje:** S obzirom da optimalna vrijednost nije unaprijed poznata, algoritam se izvodi unaprijed definiran broj generacija (100).

Pri testiranju implementiranog rješenja iz Priloga 9.33, za ulazne podatke

```
// popis mogucih predmeta u ruksaku
std::vector<Predmet> predmeti = {
    {"Knjiga",      12.99, 1.2}, // naziv, cijena (EUR), tezina (kg)
    {"Boca vode",    1.50, 1.5},
    {"Laptop",     599.00, 2.5},
    {"Hrana",       20.00, 1.8},
    {"Kabanica",    49.99, 1.0},
    {"Fotoaparat", 320.00, 2.0},
    {"Upaljac",      2.00, 0.2},
    {"Jakna",       89.99, 1.4},
    {"Tablete",     15.50, 0.3},
    {"Baterije",     6.99, 0.5}
};
double kapacitet_ruksaka = 8.5; // u kg
```

jedan od dobivenih izlaza je sljedeći:

```
Naziv          Vrijednost (EUR) Tezina (kg)
-----------------------------------------------
Knjiga         12.99            1.20
Laptop         599.00           2.50
Kabanica       49.99            1.00
Fotoaparat     320.00           2.00
Jakna          89.99            1.40
Tablete        15.50            0.30
-----------------------------------------------
Ukupna vrijednost: 1087.47 EUR
Ukupna tezina:     8.40 kg
Kapacitet ruksaka: 8.50 kg
```

Na temelju zadanih ulaznih podataka, gdje je kapacitet ruksaka ograničen na $8.5\,\text{kg}$, a skup dostupnih predmeta definiran realnim vrijednostima u eurima i pripadajućim težinama u kilogramima, genetski algoritam pronalazi optimalnu (ili bar blizu optimalnu)

kombinaciju predmeta čija ukupna vrijednost je maksimalna, a težina ne prelazi dopušteno ograničenje. Naime, uočavamo da algoritam uspješno odabire kombinaciju predmeta koja gotovo u potpunosti iskorištava raspoloživi kapacitet ruksaka, pri čemu je ukupna težina $8.40,\text{kg}$, odnosno samo 0.1 kg ispod ograničenja. Istovremeno, ukupna vrijednost predmeta iznosi čak 1087.47 eura, što ukazuje na visoku učinkovitost algoritma. Štoviše, učinkovitost algoritma očituje se i u malom broju generacija potrebnih za pronalazak ovakvog rješenja (najčešće već unutar prvih 10 generacija).

Zanimljivo je primijetiti da se u rješenju nalaze predmeti visoke vrijednosti poput *laptopa*, *fotoaparata* i *jakne*, dok su neki lakši, ali manje vrijedni predmeti (primjerice,*boca vode* ili *baterije*) izostavljeni jer njihov omjer vrijednosti i težine nije konkurentan. Takvo ponašanje algoritma odražava sposobnost dobre procjene kompromisa između težine i vrijednosti, što je srž problema ruksaka.

Primjena genetskog algoritma na problem ruksaka pokazuje kako se heurističke metode mogu učinkovito primijeniti i na probleme optimizacije s ograničenjima. Za razliku od problema osam kraljica, gdje je cilj minimizirati broj konflikata, ovdje se traži maksimizacija ukupne vrijednosti uz poštivanje strogo definiranog kapaciteta. Evaluacija rješenja tako uključuje i penalizaciju onih jedinki koje premašuju dopuštenu težinu. Time se algoritam usmjerava prema realno izvedivim, ali i ekonomski isplativijim kombinacijama predmeta.

7.4 Optimizacija školskog rasporeda

Jedan od čestih izazova u obrazovnim institucijama jest izrada tjednog rasporeda nastave koji zadovoljava niz praktičnih ograničenja. Svakom predmetu potrebno je dodijeliti termin i učionicu, pritom vodeći računa da nijedan profesor ne drži nastavu u više termina istovremeno, da su učionice dostupne i da nastava bude unutar zadanog vremenskog okvira. Osim toga, često postoji ograničen broj dana i sati u kojima se nastava može održavati, što dodatno otežava izradu ispravnog i učinkovitog rasporeda.

U ovom primjeru bit će prikazano kako se problem rasporeda nastave može formulirati i rješavati korištenjem genetskog algoritma. Fokus je stavljen na evaluaciju sukoba (primjerice, preklapanja termina za istog profesora ili zauzetost učionica), a cilj je pronaći raspored s minimalnim brojem takvih konflikata. Cjeloviti primjer rješenja ovog problema u programskom jeziku C++ nalazi se u Prilogu 9.34.

U ovom slučaju, ulaz u problem optimizacije rasporeda čine tri glavne komponente: popis predmeta, broj dostupnih učionica i broj nastavnih dana:

```
// popis predmeta
std::vector<Predmet> predmeti = {
    {"Predmet 1",  "Prof. 1"}, {"Predmet 2",  "Prof. 1"}, {"Predmet 3",  "Prof. 1"},
    {"Predmet 4",  "Prof. 2"}, {"Predmet 5",  "Prof. 2"}, {"Predmet 6",  "Prof. 2"},
    {"Predmet 7",  "Prof. 3"}, {"Predmet 8",  "Prof. 3"}, {"Predmet 9",  "Prof. 3"},
    {"Predmet 10", "Prof. 3"}, {"Predmet 11", "Prof. 4"}, {"Predmet 12", "Prof. 4"},
    {"Predmet 13", "Prof. 4"}, {"Predmet 14", "Prof. 4"}, {"Predmet 15", "Prof. 4"},
    {"Predmet 16", "Prof. 5"}, {"Predmet 17", "Prof. 5"}, {"Predmet 18", "Prof. 5"},
    {"Predmet 19", "Prof. 5"}, {"Predmet 20", "Prof. 5"}
};
```

```
int brojUcionica = 2;   // dvije ucionice na raspolaganju
int brojDana = 3;       // nastava traje 3 dana (pon-sri)
```

Svaki predmet ima pridruženog profesora, pri čemu isti profesor može držati nastavu iz više predmeta. Time se u rasporedu uvodi važno ograničenje: profesor ne smije biti raspoređen na više termina nastave koji se odvijaju u isto vrijeme.

Drugi važan ulazni parametar je broj učionica koji ograničava broj paralelnih termina nastave koja se mogu održati u isto vrijeme. Ako se dva termina nastave pokušaju smjestiti u istu učionicu u isto vrijeme, to se tretira kao konflikt koji negativno utječe na kvalitetu rasporeda.

I konačno, treći ulazni parametar je broj dana unutar kojih se nastava može rasporediti (primjerice, od ponedjeljka do srijede). Svaki dan sadrži isti broj dostupnih sati za nastavu u tim učionicama (primjerice, od 8 do 12 sati), a nastava je fiksne duljine (primjerice, jedan sat).

Zajedno, svi ovi ulazni podaci određuju prostor svih mogućih rješenja unutar kojeg genetski algoritam traži optimalni raspored. Rješenje se oblikuje tako da svakom predmetu dodijeli konkretan dan, sat i učionica, a procjena kvalitete takvog rješenja temelji se na broju sukoba u rasporedu (problem minimizacije).

Najvažnije komponente genetskog algoritma implementiranog u Prilogu 9.34 opisane su u nastavku:

- **Inicijalizacija:** Svaka jedinka predstavlja jedan mogući raspored nastave. Za svaki predmet nasumično se određuje dan, sat i učionica, uz poštivanje zadanih granica (broj dana, radno vrijeme i broj učionica). Generiranjem većeg broja takvih jedinki formira se početna populacija rješenja s kojom algoritam započinje svoj rad.
- **Evaluacija:** Dobrota jedinke definirana je kao broj penalizacija, pri čemu se penaliziraju sljedeće situacije:

  1. termin nastave izlazi izvan dozvoljenog vremenskog raspona
  2. dvije različite nastave zakazane su u istoj učionici u istom terminu
  3. profesor ima više paralelnih termina nastave.

  Što je broj penalizacija manji, to je jedinka kvalitetnija. Idealna jedinka ima vrijednost dobrote jednaku nuli, što označava potpuno valjan raspored bez sukoba.
- **Selekcija:** Korištena je *turnirska selekcija* (vidi Poglavlje 2.4). Za svako novo rješenje u populaciji nasumično se biraju dvije jedinke, a prenosi se ona s manjim brojem penalizacija. Time se preferiraju kvalitetniji rasporedi, ali se i dalje, zbog minimalne veličine turnira, omogućuje prisutnost slabijih rješenja kako bi se očuvala raznolikost populacije.
- **Križanje:** Korišteno je *uniformno ponderirano križanje* (vidi Poglavlje 3.2), pri čemu se za svaki gen (odnosno termin nastave) neovisno određuje hoće li biti naslijeđen

od prvog ili drugog roditelja, s jednakom vjerojatnošću od 50%. Ovaj oblik križanja omogućuje ravnomjernu kombinaciju genetskog materijala oba roditelja, bez obzira na redoslijed gena, te se može primijeniti i na reprezentacije koje uključuju više komponenti, kao što su dan, sat i učionica.

- **Mutacija:** Primijenjena je *uniformna mutacija* (vidi Poglavlje 4.1), u kojoj svaki gen (termin nastave) ima određenu vjerojatnost mutacije. Ako se dogodi mutacija, vrijednosti dana, sata i učionice za taj gen nasumično se ponovno generiraju. Ovakav pristup zadržava jednostavnost implementacije, a ujedno omogućuje dovoljnu raznolikost populacije, što je ključno za izbjegavanje lokalnih minimuma.
- **Elitizam:** Jedna ili više najboljih jedinki (s najmanjim brojem penalizacija) iz prethodne generacije prenosi se u novu generaciju. Time se osigurava očuvanje trenutno najboljeg rješenja, čime se izbjegava regresija kvalitete rješenja kroz generacije (vidi Poglavlje 5.1). U ovom slučaju prenosi se samo jedna elitna jedinka.
- **Zaustavljanje:** Algoritam se zaustavlja nakon unaprijed definiranog broja generacija (200) ili čim pronađe rješenje bez ikakvih sukoba (dobrota jednaka 0).

Prilikom pokretanja navedenog programa, jedan od dobivenih izlaza (rješenja) bio je sljedeći:

```
Generacija 31: najbolja dobrota = 0
Predmet 1 (Prof. 1) -> Pon 8-9h, ucionica 1
Predmet 2 (Prof. 1) -> Pon 10-11h, ucionica 1
Predmet 3 (Prof. 1) -> Sri 9-10h, ucionica 2
Predmet 4 (Prof. 2) -> Sri 9-10h, ucionica 1
Predmet 5 (Prof. 2) -> Pon 9-10h, ucionica 2
Predmet 6 (Prof. 2) -> Sri 10-11h, ucionica 1
Predmet 7 (Prof. 3) -> Pon 9-10h, ucionica 1
Predmet 8 (Prof. 3) -> Sri 8-9h, ucionica 1
Predmet 9 (Prof. 3) -> Pon 11-12h, ucionica 1
Predmet 10 (Prof. 3) -> Uto 8-9h, ucionica 2
Predmet 11 (Prof. 4) -> Sri 11-12h, ucionica 2
Predmet 12 (Prof. 4) -> Uto 8-9h, ucionica 1
Predmet 13 (Prof. 4) -> Uto 9-10h, ucionica 1
Predmet 14 (Prof. 4) -> Pon 11-12h, ucionica 2
Predmet 15 (Prof. 4) -> Uto 11-12h, ucionica 1
Predmet 16 (Prof. 5) -> Uto 9-10h, ucionica 2
Predmet 17 (Prof. 5) -> Pon 10-11h, ucionica 2
Predmet 18 (Prof. 5) -> Sri 10-11h, ucionica 2
Predmet 19 (Prof. 5) -> Sri 8-9h, ucionica 2
Predmet 20 (Prof. 5) -> Uto 10-11h, ucionica 1
```

Za ukupno 20 predmeta raspoređenih između pet profesora i dvije učionice genetski algoritam pronašao je optimalno rješenje bez sukoba u samo 31 generaciji. Radi lakše provjere ispravnosti dobivenog rješenja Tablica 7.1 prikazuje navedeni raspored nastave organiziran po danima i satima.

Tablica 7.1: Tjedni raspored nastave prema danima i satima.

Sat	Pon	Uto	Sri
8:00-9:00	Pred. 1 (Prof. 1) [uč. 1]	Pred. 10 (Prof. 3) [uč. 2] | Pred. 12 (Prof. 4) [uč. 1]	Pred. 8 (Prof. 3) [uč. 1] | Pred. 19 (Prof. 5) [uč. 2]
9:00-10:00	Pred. 5 (Prof. 2) [uč. 2] | Pred. 7 (Prof. 3) [uč. 1]	Pred. 13 (Prof. 4) [uč. 1] | Pred. 16 (Prof. 5) [uč. 2]	Pred. 3 (Prof. 1) [uč. 2] | Pred. 4 (Prof. 2) [uč. 1]
10:00-11:00	Pred. 2 (Prof. 1) [uč. 1] | Pred. 17 (Prof. 5) [uč. 2]	Pred. 20 (Prof. 5) [uč. 1]	Pred. 6 (Prof. 2) [uč. 1] | Pred. 18 (Prof. 5) [uč. 2]
11:00-12:00	Pred. 9 (Prof. 3) [uč. 1] | Pred. 14 (Prof. 4) [uč. 2]	Pred. 15 (Prof. 4) [uč. 1]	Pred. 11 (Prof. 4) [uč. 2]

U rasporedu nema preklapanja termina za iste profesore, niti dvostrukog zauzeća učionica, a sva nastava smještena je unutar dopuštenog vremenskog raspona. Stoga možemo potvrditi da dobiveno rješenje zaista predstavlja valjani (optimalni) tjedni raspored nastave koji zadovoljava sva zadana ograničenja.

7.5 Simbolička regresija

Regresijska analiza statistička je metoda koja se koristi za modeliranje odnosa između jedne *zavisne* (ili izlazne) varijable i jedne ili više *nezavisnih* (ili ulaznih) varijabli [92]. Cilj je regresije pronaći matematičku funkciju koja opisuje kako promjene u ulaznim varijablama utječu na izlaz. Primjerice, ako želimo modelirati kako temperatura zraka utječe na potrošnju električne energije, regresijskom analizom pokušavamo pronaći funkciju tipa:

$$\text{potrošnja} = f(\text{temperatura})$$

gdje je f funkcija koju trebamo pronaći. Ako pretpostavimo da je f linearna, govorimo o *linearnoj regresiji*:

$$\text{potrošnja} = a \cdot \text{temperatura} + b$$

gdje su a i b parametri koje procjenjujemo na temelju podataka.

Regresijska analiza koristi se u gotovo svim područjima znanosti i inženjerstva – od ekonomije, biologije i sociologije, do strojnog učenja i umjetne inteligencije.

S druge strane, simbolička regresija metoda je regresijske analize čiji je cilj pronaći matematički izraz koji najbolje opisuje vezu između ulaznih i izlaznih podataka, pri čemu se ne pretpostavlja unaprijed poznati oblik funkcije [19]. Za razliku od klasičnih metoda (poput linearne ili polinomne regresije), simbolička regresija istovremeno otkriva strukturu i parametre modela. Drugim riječima, algoritam simboličke regresije ne traži optimalne koeficijente unutar unaprijed zadane funkcije, već samostalno gradi funkciju korištenjem osnovnih matematičkih operatora (primjerice, $+$, $-$, $\times$, $\div$, $\sin$, $\exp$, itd.).

Najčešći je način implementacije simboličke regresije korištenje genetskog programiranja koje razvija populaciju kandidata u obliku stabala izraza, pri čemu svako stablo predstavlja jedan mogući matematički izraz [19]. Operatori i varijable kombiniraju se na različite načine da bi se formirali izrazi koji minimiziraju pogrešku na skupu podataka. Genetsko programiranje pokazalo se osobito korisnim kada je cilj dobiti ne samo točne, nego i interpretabilne modele, odnosno izraze koje čovjek može lako razumjeti, analizirati i koristiti u daljnjoj obradi [93].

Jedna od zanimljivih primjena simboličke regresije u kontekstu evolucijskih algoritama jest modeliranje ili aproksimacija prostora rješenja (engl. *fitness landscape*). U situacijama kada je izračun funkcije dobrote skup, spor ili numerički zahtjevan, primjerice u inženjerskim simulacijama, simbolička regresija može poslužiti kao zamjenski model (engl. *surrogate model*) koji aproksimira ponašanje funkcije dobrote. Time se evolucijskom algoritmu omogućuje da brže i učinkovitije istražuje prostor rješenja, bez potrebe za skupim evaluacijama [94].

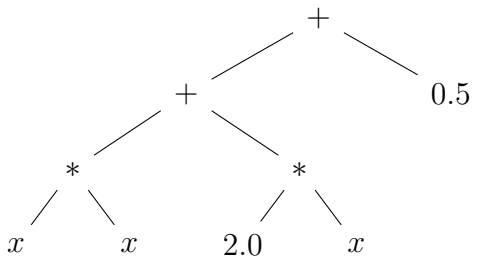
Zahvaljujući sposobnosti da iz podataka otkrije jednostavne, razumljive i matematički točne izraze, simbolička regresija nalazi primjenu u znanstvenim istraživanjima, inženjerskim sustavima i biomedicini, odnosno u svim područjima gdje je interpretabilnost modela jednako važna kao i njegova prediktivna snaga [95].

Za demonstraciju, uzmimo da imamo sljedeće ulazne točke (uzorke):

$$\{(-3,4),\ (-2,1),\ (-1,0),\ (0,1),\ (1,4),\ (2,9),\ (3,16)\}.$$

Primjenom genetskog programiranja, u Prilogu 9.35 prikazano je kako pronaći najbolju aproksimaciju funkcije koja prolazi kroz zadane točke. U nastavku su objašnjeni najvažniji dijelovi algoritma.

Na početku algoritma (inicijalizacija) nasumično se generira početna populacija matematičkih izraza sastavljenih od varijabli, konstanti i zadanih operatora $(+, -, *)$. Primjer jednog takvog izraza prikazan je na Slici 7.3. Svaki izraz interpretira se kao funkcija te se evaluira nad ulaznim točkama, pri čemu se izračunava pogreška (dobrota) u odnosu na stvarne izlazne vrijednosti. Cilj je inicijalizacije proizvesti početnu raznolikost te identificirati izraze koji najbolje aproksimiraju zadane točke.



Slika 7.3: Stablo izraza za funkciju $f(x) = x^2 + 2x + 0.5$.

Evaluacija izraza temelji se na usporedbi izračunate vrijednosti $f(x)$ s očekivanom vrijednošću y za svaki ulazni uzorak (x, y). Dobrota jedinke određuje se kao suma kvadrata pogrešaka:

$$\text{dobrota} = \sum_{i=1}^{n} (f(x_i) - y_i)^2$$

Ova mjera standardna je u regresiji jer veće pogreške strože kažnjava, a izraze s manjim ukupnim odstupanjem nagrađuje. Primjerice, ako jedinka daje $f(-3) = 2$, a stvarna vrijednost je $y = 4$, tada doprinos pogrešci iznosi:

$$(2-4)^2 = 4$$

Što je razlika veća, kvadrat pogreške raste brže, pa algoritam preferira stabilne izraze koji daju približno točne rezultate za sve točke.

Primjerice, pretpostavimo da želimo evaluirati rješenje (funkciju) $f(x) = x^2 + 2x + 0.5$ sa Slike 7.3 nad setom ulaznih točaka. Izračun dobrote suma je kvadrata pogrešaka:

$(3.50-4)^2 + (0.50-1)^2 + (-0.50-0)^2 + (0.50-1)^2 + (3.50-4)^2 + (8.50-9)^2 + (15.50-16)^2 =$
$0.5^2 + 0.5^2 + 0.5^2 + 0.5^2 + 0.5^2 + 0.5^2 + 0.5^2 =$
$0.25 + 0.25 + 0.25 + 0.25 + 0.25 + 0.25 + 0.25 =$ **1.75**

U nastavku je korištena *turnirska selekcija* (vidi Poglavlje 2.4). Za svako novo rješenje u populaciji nasumično se biraju dvije jedinke (stabla matematičkih izraza), a prenosi se ona s manjom, odnosno boljom vrijednosti funkcije dobrote. Na taj način preferiraju se izrazi koji bolje aproksimiraju zadane ulazne točke. Turnirska selekcija često se koristi jer je jednostavna za implementaciju, ne zahtijeva sortiranje populacije i lako se parametrizira (primjerice, veličinom turnira). Osim toga, omogućuje dobar balans između iskorištavanja najboljih rješenja i očuvanja raznolikosti u populaciji.

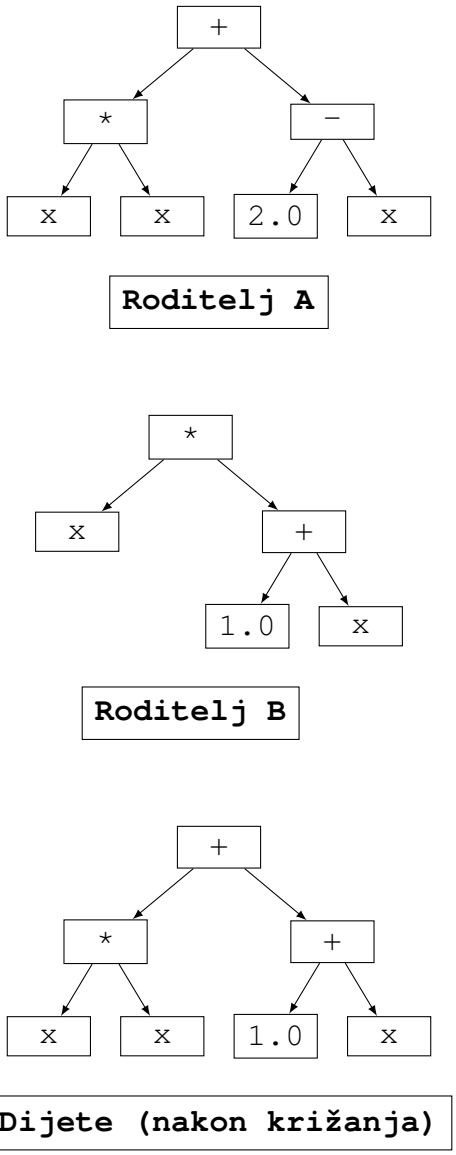
Križanje se provodi nad dvjema jedinkama (matematičkim izrazima). Postupak je sljedeći:

1. Stvara se kopija stabla prvog roditelja (donora).
2. Nasumično se odabire podstablo iz drugog roditelja (donora zamjene).
3. U kopiji prvog roditelja nasumično se odabire čvor koji će biti zamijenjen.
4. Odabrano podstablo drugog roditelja zamjenjuje čvor u kopiranom stablu.

Na taj način dijete nasljeđuje osnovnu strukturu od jednog roditelja, ali s dijelom izraza drugog roditelja. Križanje omogućuje rekombinaciju korisnih podizraza i uvođenje raznolikosti u populaciju.

Na Slici 7.4 prikazan je konkretan primjer križanja dviju jedinki u genetskom programiranju.

- **Roditelj A** sadrži izraz `((x * x) + (2.0 - x))`, pri čemu je desno podstablo čvor s operatorom `-`, čiji su potomci `2.0` i `x`.
- **Roditelj B** predstavlja izraz `(x * (1.0 + x))`. U ovom primjeru, iz roditelja B nasumično je odabrano podstablo koje odgovara izrazu `(1.0 + x)`.
- Tijekom križanja, to podstablo iz roditelja B zamjenjuje nasumično odabrano podstablo iz roditelja A, u ovom slučaju `(2.0 - x)`.
- Time nastaje nova jedinka (dijete) s izrazom `((x * x) + (1.0 + x))`.

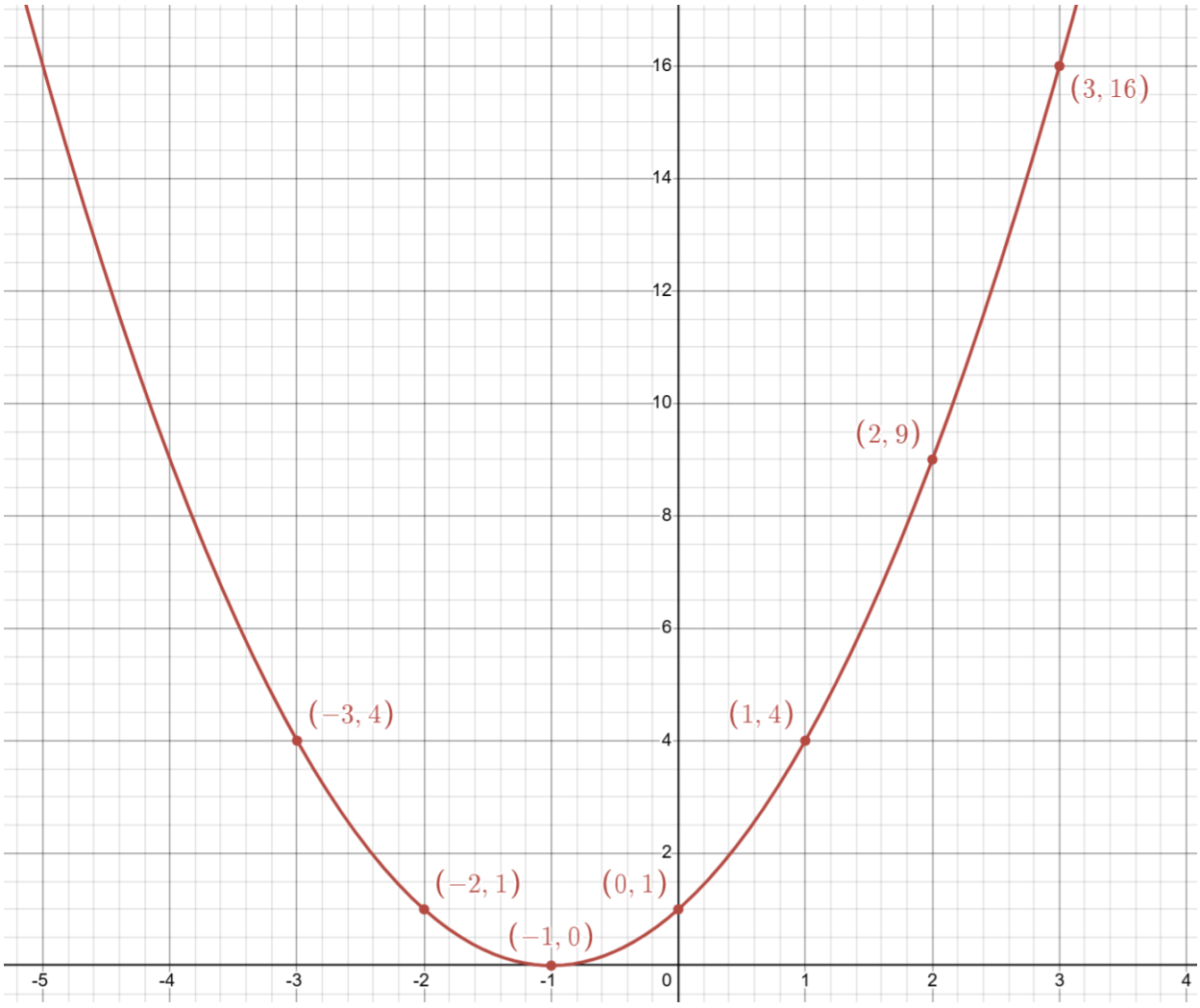


Slika 7.4: Primjer križanja: podstablo `(+ 1.0 x)` iz Roditelja B zamjenjuje podstablo `(- 2.0 x)` iz Roditelja A.

Ovim postupkom dijete nasljeđuje strukturu roditelja `A`, ali sadrži i genetski materijal roditelja `B` u obliku zamijenjenog podizraza. To omogućuje razmjenu funkcionalnih dijelova između različitih jedinki, čime se povećava vjerojatnost dolaska do kvalitetnijeg rješenja u sljedećim generacijama.

Nadalje, mutacija se provodi tako da se za svaku novonastalu jedinku generira slučajni broj iz intervala $[0, 1]$. Ako je taj broj manji ili jednak od unaprijed zadane vjerojatnosti mutacije (primjerice, $0{,}2$), unutar stabla izraza nasumično se odabire jedan čvor ili podstablo, koje se potom zamjenjuje novim, nasumično generiranim stablom (Prilog 9.35). Na taj način struktura izraza djelomično se mijenja, čime se omogućuje istraživanje novih dijelova prostora rješenja koji ne bi bili dostupni samo primjenom križanja.

Algoritam koristi i strategiju *elitizma*, pri čemu se jedna (najbolja) jedinka iz prethodne generacije zamjenjuje s najlošijom jedinkom u trenutnoj generaciji, čime se osigurava očuvanje najboljeg pronađenog rješenja. Algoritam završava tek nakon što se dosegne unaprijed zadani broj generacija (100), budući da ciljna funkcija, odnosno konačni matematički izraz, nije unaprijed poznata.



Slika 7.5: Graf funkcije $f(x) = x^2 + 2x + 1$ s prikazom zadanih ulaznih točaka.

Stvarna funkcija koju tražimo je $f(x) = x^2 + 2x + 1$. Slika 7.5 prikazuje njezin graf te potvrđuje da se sve ulazne točke (uzorci) nalaze upravo na toj funkciji. Međutim, pogledajmo sada neka od rješenja pronađena genetskim programiranjem:

Izraz 1:

$$((3.335274 + (x - 2.373807)) + ((x \cdot x) + x))$$

Skraćeni oblik:

$$f(x) = x^2 + 2x + 0.961467$$

Dobrota:

$$= 0.010393$$

Dobiveni izraz relativno je jednostavan i blizu tražene funkcije $f(x) = x^2+2x+1$. Dobrota iznosi 0.010393, što znači da je pogreška mala, ali ipak uočljiva. Ovaj izraz koristi osnovne operatore i ima umjerenu dubinu stabla pa je dobar primjer jedinke koja je razumljiva i točna, ali još uvijek nije savršena aproksimacija.

Izraz 2:

$$((0.497490 + (0.497490 + x)) + (x + (x \cdot x)))$$

Skraćeni oblik:

$$f(x) = x^2 + 2x + 0.994980$$

Dobrota:

$$= 0.000176$$

Ovaj izraz je kraći, simetričan i čitljiv, te je numerički još precizniji (dobrota je samo 0.000176). Time je funkcija gotovo identična traženoj, uz minimalnu razliku. Izraz je i dalje lagan za razumijevanje i ima ograničen broj operacija.

Izraz 3:

$$\begin{aligned}&(((((((x \cdot x) - (0.020664 \cdot (x - 0.020664))) - ((0.020664 \cdot (x - x)) \cdot \\ &((x - ((3.370040 - (x \cdot 1.640801)) + x)) - x))) - ((0.184826 \cdot 0.579098) - \\ &(0.020664 \cdot ((0.020664 - ((x - x) - 0.184826)) - ((x + (3.178887 - x)) - x))))) \\ &+ x) - 0.579098) + (1.747589 + x))\end{aligned}$$

Skraćeni oblik:

$$f(x) = x^2 + 2x + 1.00044335834$$

Dobrota:

$$= 0.0000014$$

Zadnji izraz znatno je duži, duboko ugniježđen i teže čitljiv, ali se evaluacijom pokazuje da daje najbližu aproksimaciju ciljnoj funkciji $f(x) = x^2+2x+1$ s izuzetno niskom pogreškom od 0.0000014. Iako je to numerički najbolje rješenje, njegov je oblik daleko najsloženiji, što ga čini lošijim izborom u smislu interpretabilnosti.

Genetsko programiranje u ovom je slučaju pokazalo iznimnu sposobnost u pronalasku vrlo dobrih aproksimacija tražene funkcije $f(x) = x^2 + 2x + 1$, i to bez eksplicitnog znanja o njezinu obliku. Tijekom evolucije, algoritam je generirao niz raznolikih i često vrlo složenih izraza. Ipak, među njima su se spontano pojavili izrazi koji su gotovo identični stvarnoj funkciji, s vrlo malim odstupanjima u konstantnim članovima. Primjerice, evoluirani izrazi:

$$f_1(x) = x^2 + 2x + 0.994980 \quad \text{i} \quad f_2(x) = x^2 + 2x + 1.000443$$

imaju pogreške (dobrote) reda veličine 10^{-4} odnosno 10^{-6}.

Važno je istaknuti da genetsko programiranje:

- nije imalo unaprijed zadanu formu funkcije
- nije znalo da se radi o kvadratnom polinomu
- nije imalo informaciju da je optimalna konstanta točno 1.

Unatoč tome, zahvaljujući mehanizmima selekcije, križanja i mutacije, uspjelo je pronaći vrlo precizne izraze koji odlično aproksimiraju sve ulazne točke. Ovakav rezultat jasno demonstrira snagu i fleksibilnost simboličke regresije temeljene na genetskom programiranju, osobito u slučajevima kada oblik funkcije nije unaprijed poznat.

7.6 Razvoj strategije za odobravanja kredita

U ovom primjeru analiziramo primjenu genetskog programiranja na zadatak klasifikacije zahtjeva za kredit. Cilj je evolucijom logičkog stabla pronaći strategiju koja će, na temelju kombinacije ulaznih atributa, odlučiti treba li klijentov zahtjev biti odobren ili odbijen.

Modelirana su četiri binarna ulazna atributa:

- **A**: korisnik ima stalno zaposlenje (1) ili nema (0)
- **B**: korisnik ima pozitivnu kreditnu povijest (1) ili nema (0)
- **C**: korisnik ima dugovanja (1) ili nema (0)
- **D**: korisnik je mlađi od 30 godina (1) ili nije (0).

Svaki primjer klasificira se u jednu od dviju klasa:

- **1** – odobriti zahtjev
- **0** – odbiti zahtjev.

Primjerice, korisnik koji ima posao (`A=1`), urednu kreditnu povijest (`B=1`), nema dugova (`C=0`) i nije mlad (`D=0`) predstavlja idealan profil, pa bi stablo trebalo klasificirati taj slučaj kao `1` – odobriti zahtjev. S druge strane, korisnik koji je mlad (`D=1`), nema posao ni kreditnu povijest (`A=0`, `B=0`) i ima dugove (`C=1`) predstavlja visokorizičan slučaj i trebao bi biti klasificiran kao `0` – odbiti zahtjev.

Iako se klasifikacijski problemi u praksi često rješavaju primjenom neuronskih mreža zbog njihove visoke učinkovitosti [96, 97], takvi modeli nerijetko rezultiraju strukturama koje je teško interpretirati. Stoga je u ovom primjeru naglasak stavljen na razvoj strategije odlučivanja u obliku uvjetnih izraza, pri čemu se koristi genetsko programiranje koje omogućuje jasno razumijevanje i analizu dobivenih modela. Cilj genetskog programiranja u ovom kontekstu jest generirati i usavršavati logička stabla sastavljena od izraza poput `(B == 1)`, `(C && D)` ili uvjetnih izraza `IF(... THEN ... ELSE ...)` koja što vjernije odražavaju skrivenu strategiju odlučivanja na temelju danih primjera.

Ulazni skup podataka prikazan je u Tablici 7.2. Svaki redak predstavlja jednog korisnika i pripadajuću klasu (odobriti ili odbiti zahtjev). Osim osnovnih primjera koji jasno razlikuju

dobre i loše kandidate, dodani su i posebni slučajevi koji sprječavaju da algoritam pronađe prejednostavna rješenja, poput pravila "ako korisnik nije mlad, automatski odobri zahtjev". Time se potiče genetsko programiranje da uči složenije kombinacije uvjeta među više varijabli.

Tablica 7.2: Ulazni primjeri za klasifikaciju zahtjeva za kredit.

Klijent	A	B	C	D	Klasa	Objašnjenje
1	1	1	0	0	1	Sve idealno
2	1	1	1	0	1	Dugovi, ali inače dobar profil
3	1	0	0	0	1	Ima posao i nema dugova
4	0	1	0	0	1	Dobra povijest i bez dugova
5	0	0	1	1	0	Mlad, nema posao ni povijest, ima dugove
6	0	1	1	1	0	Samo povijest, mlad i s dugovima
7	1	0	1	1	0	Samo posao, ali sve ostalo loše
8	0	0	0	1	0	Mlad i bez ikakvih pozitivnih osobina
9	0	0	1	0	0	Stariji, ali sve ostalo negativno
10	1	1	0	1	1	Mlad, ali savršen profil
11	1	0	0	1	1	Mlad, ali ima posao i bez dugova
12	0	1	0	1	1	Mlad, ali dobra povijest i bez dugova
13	0	0	0	0	0	Stariji, ali bez pozitivnih karakteristika

Legenda: A – stalno zaposlenje, B – pozitivna kreditna povijest, C – ima dugovanja, D – mlađi od 30 godina, Klasa – odluka.

Cjelovito rješenje primjenom genetskog programiranja u programskom jeziku C++ vidljivo je u Prilogu 9.36, dok su u nastavku opisani njegovi najvažniji dijelovi.

Na početku algoritma (inicijalizacija) nasumično se generira početna populacija logičkih stabala sastavljenih od varijabli (A, B, C, D), logičkih operatora ($==$, $!=$, $\&\&$, $||$), konstanti (0 ili 1) te uvjetnih izraza oblika `IF(uvjet) THEN (grana) ELSE (grana)`. Svako stablo predstavlja strategiju odlučivanja, odnosno pravilo koje za konkretan ulazni primjer (korisnika) vraća odluku o tome treba li kredit biti odobren (1) ili odbijen (0).

Evaluacija svakog stabla provodi se usporedbom njegovih predikcija s točno zadanim klasama za sve ulazne primjere. Za svaki ispravno klasificirani primjer dodjeljuje se jedan bod. Dobrota jedinke računa se kao broj ispravno klasificiranih primjera. Ako stablo za neki primjer donese ispravnu odluku (predikcija odgovara stvarnoj klasi), dodaje se jedan bod. Primjerice, ako jedinka ispravno klasificira 10 od ukupno 13 primjera, njezina dobrota je 10.

Cilj je evolucije maksimizirati broj točno klasificiranih primjera. Kroz generacije populacija stabala razvija se koristeći klasične evolucijske operatore:

- **Selekcija**: koristi se turnirska selekcija. Za svaku novu jedinku iz populacije nasumično se odabire nekoliko kandidata (veličine turnira, primjerice, 2), a najbolji među njima (s najvećom dobrotom) odabire se kao roditelj. Na taj se način osigurava prednost kvalitetnijih rješenja, ali uz očuvanje raznolikosti.

- **Križanje (rekombinacija)**: za dvoje roditelja odabrana podstabla nasumično se zamjenjuju – u prvu kopiju roditelja A umeće se kopija podstabla iz roditelja B. Ovim postupkom stvaraju se potomci koji mogu naslijediti korisne dijelove strukture oba roditelja, potencijalno spajajući učinkovite pododluke.
- **Mutacija**: uz određenu vjerojatnost (primjerice, 20%), potomku se zamijeni neko od njegovih podstabala novim nasumično generiranim stablom. Mutacija unosi dodatnu varijaciju u populaciju, što pomaže izbjegavanju lokalnih optimuma.
- **Elitizam**: najbolja jedinka iz trenutne generacije uvijek se prenosi u sljedeću, čime se osigurava očuvanje najkvalitetnijeg rješenja.

Nakon određenog broja generacija ili postizanja maksimalne moguće točnosti, prikazuje se najbolje pronađeno stablo odlučivanja. Primjerice, za ulazne podatke prikazane u Tablici 7.2 jedno od dobivenih rješenja nakon samo 13 generacija bilo je sljedeće:

```
IF ((A OR IF (1)
        THEN
            B
        ELSE
            (1 OR B)))
THEN
    ((C AND D) != (1 OR (((D != D) == (0 OR 1)) AND (IF (D)
            THEN
                1
            ELSE
                1 AND (D AND C)))))
ELSE
    ((1 OR ((IF (A)
            THEN
                1
            ELSE
                A AND (0 OR B)) != (B != ((C AND D) != B)))) AND
     IF (1)
        THEN A
        ELSE A)
```

Iako kompleksan, ovaj izraz predstavlja logičku funkciju koja kombinira sve četiri varijable (A, B, C, D) kako bi odlučila o ishodu klasifikacije. Uvjeti poput `A || IF(1) THEN B ELSE ...` ukazuju na pokušaj generalizacije ponašanja korisnika koji ima posao ili pozitivnu povijest, dok izrazi poput `C && D` provjeravaju postojanje dugovanja kod mlađih korisnika.

Zbog višestrukih trivijalnih izraza (primjerice, `IF(1) THEN ...`), izraz se može značajno pojednostaviti u interpretaciji. Njegova sažeta verzija može glasiti:

```
IF (A OR B)
    THEN (C AND D) != 1
ELSE A
```

odnosno, u programskom jeziku C++:

```
bool strategija(bool A, bool B, bool C, bool D) {
    if (A || B) {
        return (C && D) != true;
    } else
        return A;
}
```

Drugim riječima, ako klijent ima posao (A) ili dobru kreditnu povijest (B), rezultat ovisi o tome ima li dugove (C) i je li mlad (D); inače se odluka svodi na status zaposlenja (varijablu A). Ova pojednostavljena logika ilustrira kako genetsko programiranje može generirati odluke koje, iako formalno složene, sadrže korisnu logiku koja se može dodatno pojednostaviti i objasniti.

Tablica 7.3: Evaluacija strategije `IF (A OR B) THEN (C AND D) != 1 ELSE A` nad skupom ulaznih podataka.

Klijent	A	B	C	D	Klasa	Evaluacija strategije	Točno
1	1	1	0	0	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
2	1	1	1	0	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
3	1	0	0	0	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
4	0	1	0	0	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
5	0	0	1	1	0	A OR B = 0 $\Rightarrow$ ELSE $\Rightarrow$ A $\Rightarrow$ 0	✓
6	0	1	1	1	0	A OR B = 1 $\Rightarrow$ C AND D = 1 $\Rightarrow$ $1 \neq 1 \Rightarrow 0$	✓
7	1	0	1	1	0	A OR B = 1 $\Rightarrow$ C AND D = 1 $\Rightarrow$ $1 \neq 1 \Rightarrow 0$	✓
8	0	0	0	1	0	A OR B = 0 $\Rightarrow$ ELSE $\Rightarrow$ A $\Rightarrow$ 0	✓
9	0	0	1	0	0	A OR B = 0 $\Rightarrow$ ELSE $\Rightarrow$ A $\Rightarrow$ 0	✓
10	1	1	0	1	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
11	1	0	0	1	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
12	0	1	0	1	1	A OR B = 1 $\Rightarrow$ C AND D = 0 $\Rightarrow$ $0 \neq 1 \Rightarrow 1$	✓
13	0	0	0	0	0	A OR B = 0 $\Rightarrow$ ELSE $\Rightarrow$ A $\Rightarrow$ 0	✓

Evaluacija strategije odlučivanja `IF (A OR B) THEN (C AND D) != 1 ELSE A` za svakog klijenta zasebno prikazana je u Tablici 7.3. Da bismo bolje razumjeli njeno ponašanje, detaljnije analizirajmo par sljedećih ulaznih primjera:

- **Klijent 1:** `A = 1, B = 1, C = 0, D = 0`, **Klasa = 1**
  - `A OR B = 1` $\Rightarrow$ prelazi se na THEN granu.
  - `C AND D = 0`, zatim `0 != 1` $\Rightarrow$ rezultat je `1`.
  - Strategija vraća `1`, što odgovara ispravnoj klasi.
- **Klijent 5:** `A = 0, B = 0, C = 1, D = 1`, **Klasa = 0**
  - `A OR B = 0` $\Rightarrow$ prelazi se na ELSE granu.
  - ELSE grana vraća vrijednost `A`, koja je `0`.
  - Strategija vraća `0`, što odgovara ispravnoj klasi.

- **Klijent 6:** `A = 0, B = 1, C = 1, D = 1`, **Klasa = 0**
  - `A OR B = 1` $\Rightarrow$ prelazi se na THEN granu.
  - `C AND D = 1`, zatim `1 != 1` $\Rightarrow$ rezultat je `0`.
  - Strategija vraća `0`, što odgovara ispravnoj klasi.

Ovaj primjer pokazuje kako genetsko programiranje, iako stohastičko i potencijalno neefikasno u nekim slučajevima, može proizvesti strategije odlučivanja koje su ne samo učinkovite u klasifikaciji nego i lako razumljive čovjeku. Time se GP nameće kao prikladna alternativa i u zadacima gdje je interpretabilnost strategije jednako važna kao i njezina točnost.

POGLAVLJE 8

Komparativna analiza i smjernice za odabir algoritma

U ovom poglavlju daje se sažeta komparativna analiza glavnih tipova evolucijskih algoritama obrađenih u knjizi te smjernice za odabir odgovarajućeg algoritma ovisno o karakteristikama problema. Cilj poglavlja je studentima i istraživačima pružiti orijentaciju pri odabiru metode koja najbolje odgovara prirodi optimizacijskog zadatka koji rješavaju.

8.1 Usporedna analiza evolucijskih algoritama

Evolucijski algoritmi mogu se međusobno razlikovati prema načinu reprezentacije rješenja, tipu problema koji rješavaju, korištenim operatorima i svojstvima konvergencije. U tom pogledu, Tablica 8.1 daje pregled ključnih evolucijskih algoritama.

Tablica 8.1: Komparativna analiza ključnih evolucijskih algoritama.

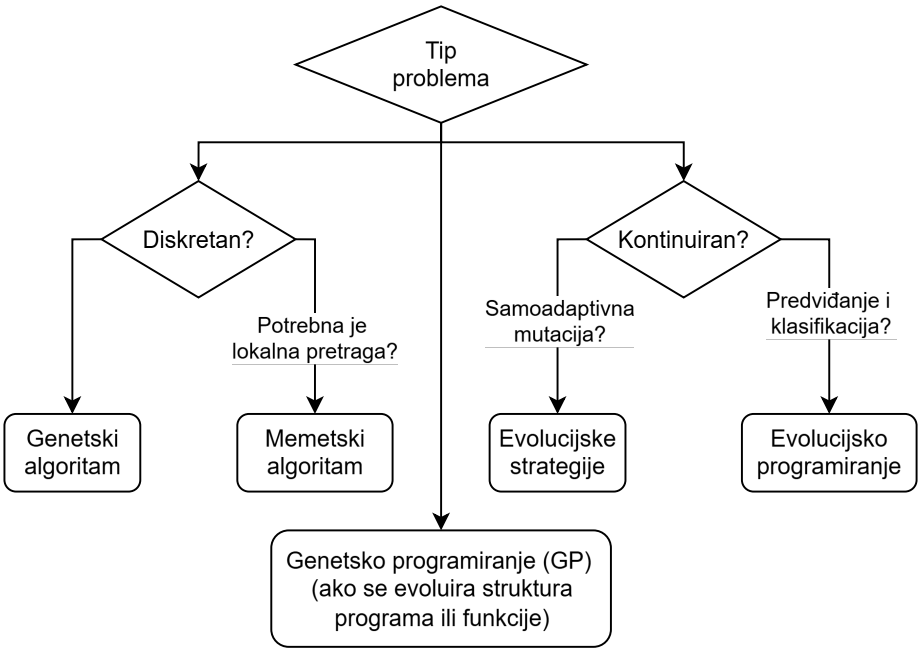
Algoritam	**Reprezentacija**	**Tip problema**	**Prednosti**	**Ograničenja / Primjene**
Genetski algoritam	Binarna, realna, permutacijska, vektorska	Kombinatorni, diskretni, kontinuirani	Jednostavna implementacija, robusnost, široka primjena	Prerana konvergencija; ruksak, rasporedi, optimizacija parametara
Genetsko programiranje	Stablo	Simbolički, modeliranje	Generira funkcije i programe	Velika složenost; regresija, klasifikacija
Evolucijske strategije	Realna	Kontinuirani, numerički	Samoadaptivna mutacija, preciznost	Neprimjeren za diskretne; optimizacija parametara
Evolucijsko programiranje	Realna ili graf	Predviđanje, klasifikacija	Jednostavnost, stabilnost	Sporija konvergencija; vremenske serije, odluke
Memetski algoritam	Ovisi o osnovnom algoritmu	Kombinatorni, složeni	Visoka kvaliteta rješenja, iskorištavanje lokalne pretrage	Računski zahtjevan; planiranje, logistika, rasporedi

Genetski algoritam najčešće se koristi zbog njegove uspješne primjene u raznim problemima optimizacije, gdje jednostavna binarna ili permutacijska reprezentacija omogućuje učinkovito pretraživanje velikog prostora rješenja. Genetsko programiranje ide korak dalje jer evoluira same programske strukture te se koristi u zadacima poput simboličke regresije, generiranja izraza i evolucije programa gdje cilj nije samo pronaći rješenje, već i razumjeti njegovu formu.

Nadalje, evolucijske strategije istaknule su se u optimizaciji realnih parametara, primjerice kod prilagodbe modela u kontinuiranim prostorima, zahvaljujući samoadaptivnim mehanizmima mutacije koji omogućuju precizno usmjeravanje pretrage. Evolucijsko programiranje primijenjeno je u problemima predviđanja i klasifikacije, poput modeliranja vremenskih serija, gdje su stabilnost i robusnost važniji od brzine konvergencije. Konačno, memetski algoritam kombinira globalnu evolucijsku pretragu s lokalnim poboljšavanjem i pokazao se posebno učinkovitim u složenim zadacima, gdje lokalna optimizacija dodatno unapređuje kvalitetu rješenja i ubrzava konvergenciju.

8.2 Smjernice za odabir algoritma

Prilikom odabira odgovarajućeg evolucijskog algoritma potrebno je uzeti u obzir tip problema, prirodu funkcije dobrote, veličinu prostora pretraživanja te potrebu za lokalnim pretraživanjem ili interpretabilnošću rezultata. Na Slici 8.1 prikazan je dijagram toka koji može poslužiti kao jednostavan vodič.



Slika 8.1: Osnovne smjernice za odabir odgovarajućeg evolucijskog algoritma.

Uz osnovne smjernice, dijagram na Slici 8.1 može se dodatno proširiti i precizirati. Primjerice, genetsko programiranje može se primijeniti i u područjima predviđanja, klasifikacije i simboličke regresije, kao što je to već prikazano u poglavljima 7.5 i 7.6. Također, ako problem uključuje višekriterijsku optimizaciju, prikladno je primijeniti višekriterijske

evolucijske algoritme poput NSGA-II [98]. U slučajevima kombinatorne optimizacije, genetski i memetski algoritmi obično daju najbolje rezultate, dok se za optimizaciju realnih parametara preporučuju evolucijske strategije ili diferencijalna evolucija [99].

Prikazana Tablica 8.1 i dijagram toka (Slika 8.1) omogućuju brzu orijentaciju pri odabiru prikladnog algoritma. U praksi, optimalan izbor često uključuje eksperimentalno podešavanje parametara i kombiniranje pristupa radi postizanja ravnoteže između brzine konvergencije, kvalitete rješenja i računske složenosti.

POGLAVLJE 9

Prilozi

U ovom poglavlju kao prilozi nalaze se gotovi primjeri i demonstracije raznih algoritama koji su prethodno detaljno objašnjeni u knjizi. Primjeri su implementirani korištenjem programskog jezika C++ s ciljem da čitatelju omoguće praktično razumijevanje i lakšu primjenu obrađenih teorijskih koncepata. Svaki primjer uključuje strukturiran i dokumentiran izvorni kod koji služi kao dodatna podrška za implementaciju algoritama u stvarnim projektima ili za daljnje eksperimentiranje.

9.1 Proporcionalna selekcija

Referenca: Poglavlje 2.1

```cpp
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
public:
    char oznaka; // identifikator jedinke
    double dobrota;  // Vrijednost dobrote
    Jedinka(char _oznaka, double _dobrota) : oznaka(_oznaka), dobrota(_dobrota) {};
};

std::vector<Jedinka> proporcionalnaSelekcija(const std::vector<Jedinka>& P) {
    // Ukupna suma vrijednosti dobrote svih jedinki
    double S = 0;
    for (const auto& p : P) {
        S += p.dobrota;
    }

    // Kumulativne vjerojatnosti
    std::vector<double> C;
    double C_i = P[0].dobrota / S;
    C.push_back(C_i);

    for (size_t i = 1; i < P.size(); ++i) {
        C_i += P[i].dobrota / S;
        C.push_back(C_i);
    }
```

```
    // Nova populacija koja ce biti vracena
    std::vector<Jedinka> P_;

    // Generator nasumicnih brojeva
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    // Petlja za odabir jedinki
    for (size_t i = 0; i < P.size(); ++i) {
        double r = dis(gen);  // Nasumican broj izmedu 0 i 1
        for (size_t j = 0; j < P.size(); ++j) {
            if (C[j] >= r) {
                P_.push_back(P[j]);  // Dodaj odabranu jedinku u novu populaciju
                break;
            }
        }
    }
    return P_;  // Vraca novu populaciju
}

int main() {
    std::vector<Jedinka> P = { {'A', 20}, {'B', 25}, {'C', 30} , {'D', 20}, {'E', 15} };
    std::vector<Jedinka> novaPopulacija = proporcionalnaSelekcija(P);

    std::cout << "Nova populacija (jedinka, dobrota):\n";
    for (const auto& jedinka : novaPopulacija)
        std::cout << jedinka.oznaka << ", " << jedinka.dobrota << std::endl;

    return 0;
}
```

9.2 Stohastička univerzalna selekcija

Referenca: Poglavlje 2.2

```
#include <iostream>
#include <vector>
#include <random>

// Klasa za reprezentaciju jedinke
class Jedinka {
public:
    double dobrota; // Vrijednost funkcije dobrote
    Jedinka(double f) : dobrota(f) {}
};

// Funkcija za stohasticku univerzalnu selekciju
std::vector<Jedinka> SUS(const std::vector<Jedinka>& populacija) {
    // Korak 1: Izracunaj ukupnu vrijednost dobrote
    double ukupnaDobrota = 0.0;
    for (const auto& jedinka : populacija) {
        ukupnaDobrota += jedinka.dobrota;
    }
    // Korak 2: Odredi razmak izmedu pokazivaca
    double D = ukupnaDobrota / populacija.size();

    // Korak 3: Generiraj pocetni nasumicni broj u intervalu [0, D)
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(0, D);
    double r = dist(gen);

    // Korak 4: Kreiraj novu populaciju
    std::vector<Jedinka> novaPopulacija;
    double kumulativnaDobrota = 0.0;
```

```
    size_t i = 0;

    for (size_t k = 0; k < populacija.size(); ++k) {
        while (r > kumulativnaDobrota) {
            kumulativnaDobrota += populacija[i].dobrota;
            ++i;
        }
        // Dodaj trenutnu jedinku u novu populaciju
        novaPopulacija.push_back(populacija[i - 1]);
        r += D;
    }
    return novaPopulacija;
}

// Testiranje u funkciji main
int main() {
    // Kreiranje inicijalne populacije
    std::vector<Jedinka> populacija = { 50, 60, 30, 40, 25, 10, 20 };

    // Pozivanje SUS funkcije
    std::vector<Jedinka> novaPopulacija = SUS(populacija);

    // Ispis nove populacije
    std::cout << "Nova populacija (dobrote):" << std::endl;
    for (const auto& jedinka : novaPopulacija) {
        std::cout << jedinka.dobrota << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

9.3 Linearno rangirajuća selekcija

Referenca: Poglavlje 2.3

```
#include <iostream>
#include <vector>
#include <random>

struct Jedinka {
    char oznaka;
    double dobrota;
    Jedinka(char _oznaka, double _dobrota) : oznaka(_oznaka), dobrota(_dobrota) {};
};

// Linearno rangirajuca selekcija
std::vector<Jedinka> linearnaRangSelekcija(const std::vector<Jedinka>& populacija) {
    int brojJedinki = populacija.size();
    std::vector<Jedinka> sortiranaPopulacija = populacija;
    std::vector<double> vjerojatnosti(brojJedinki);
    std::vector<Jedinka> novaPopulacija;

    // Sortiraj populaciju prema dobroti
    std::sort(sortiranaPopulacija.begin(), sortiranaPopulacija.end(), [](const Jedinka& a,
    const Jedinka& b) {
        return a.dobrota < b.dobrota; // Sortiraj po uzlaznom poretku
        });

    // Izracunaj ukupni rang
    int ukupniRang = brojJedinki * (brojJedinki + 1) / 2;

    // Izracunaj vjerojatnosti na temelju ranga
    for (int i = 0; i < brojJedinki; ++i)
        vjerojatnosti[i] = (double)(i + 1) / ukupniRang; // Rang krece od 1
```

```
    // Inicijalizacija generatora slucajnih brojeva
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_real_distribution<> distribucija(0.0, 1.0);

    // Kreiraj novu populaciju
    for (int k = 0; k < populacija.size(); ++k) {
        double r = distribucija(generator);
        double kumulativnaVjerojatnost = 0.0;

        for (int i = 0; i < brojJedinki; ++i) {
            kumulativnaVjerojatnost += vjerojatnosti[i];
            if (r <= kumulativnaVjerojatnost) {
                novaPopulacija.push_back(sortiranaPopulacija[i]);
                break;
            }
        }
    }
    return novaPopulacija;
}

int main() {
    std::vector<Jedinka> populacija = {
        {'A', 10.0}, {'B', 20.0}, {'C', 15.0}, {'D', 5.0}, {'E', 90.0}
    };

    // linearno rangirajuca selekcija
    std::vector<Jedinka> novaPopulacija = linearnaRangSelekcija(populacija);

    // Ispis nove populacije
    std::cout << "Nova populacija nakon rangirajuce selekcije:\n";
    for (const auto& jedinka : novaPopulacija)
        std::cout << jedinka.oznaka << ": " << jedinka.dobrota << "\n";

    return 0;
}
```

9.4 Turnirska selekcija

Referenca: Poglavlje 2.4

```
#include <iostream>
#include <vector>
#include <random>

struct Jedinka {
    char oznaka;
    int dobrota;
    Jedinka(char _oznaka, double _dobrota) : oznaka(_oznaka), dobrota(_dobrota) {};
};

std::vector<Jedinka> turnirskaSelekcija(const std::vector<Jedinka>& populacija, int k) {
    std::vector<Jedinka> novaPopulacija;
    std::mt19937 generator(std::random_device{}());
    std::uniform_int_distribution<> distribucija(0, populacija.size() - 1);

    for (size_t i = 0; i < populacija.size(); i++) {
        // Odaberi k nasumicnih kandidata
        std::vector<int> kandidati;
        for (int j = 0; j < k; j++) {
            int indeks = distribucija(generator);
            kandidati.push_back(indeks);
            std::cout << "Kandidat " << populacija[indeks].oznaka
                << ": " << populacija[indeks].dobrota << "\n";
        }
```

```
        // Pronadi najboljeg medu kandidatima
        int pobjednik = kandidati[0];
        for (int j = 1; j < k; j++)
            if (populacija[kandidati[j]].dobrota > populacija[pobjednik].dobrota)
                pobjednik = kandidati[j];

        // Dodaj pobjednika u novu populaciju
        novaPopulacija.push_back(populacija[pobjednik]);
        std::cout << "Pobjednik: " << populacija[pobjednik].oznaka
            << ": " << populacija[pobjednik].dobrota << "\n\n";
    }
    return novaPopulacija;
}

int main() {
    std::vector<Jedinka> populacija = { {'A', 10}, {'B', 6}, {'C', 4}, {'D', 7}, {'E', 5},
    {'A', 3} };
    std::vector<Jedinka> novaPopulacija = turnirskaSelekcija(populacija, 2);

    std::cout << "Nova populacija:\n";
    for (const auto& jedinka : novaPopulacija)
        std::cout << jedinka.oznaka << ": " << jedinka.dobrota << "\n";
    return 0;
}
```

9.5 Pohlepna selekcija

Referenca: Poglavlje 2.5

```
#include <iostream>
#include <vector>

struct Jedinka {
    char oznaka;
    int dobrota;
    Jedinka(char _oznaka, double _dobrota) : oznaka(_oznaka), dobrota(_dobrota) {};
};

std::vector<Jedinka> pohlepnaSelekcija(std::vector<Jedinka>& Populacija) {
    std::vector<Jedinka> novaPop;
    std::vector<Jedinka> kopijaPop = Populacija;     // Kopija pocetne populacije

    // Izbor jedinki...
    for (int i = 0; i < Populacija.size(); ++i) {
        // Pronadi najbolju jedinku u preostalim jedinkama
        auto najboljaJedinkaIt = std::max_element(kopijaPop.begin(), kopijaPop.end(),
            [](const Jedinka& a, const Jedinka& b) {
                return a.dobrota < b.dobrota;
            });

        novaPop.push_back(*najboljaJedinkaIt); // Dodaj najbolju jedinku u novu populaciju
        kopijaPop.erase(najboljaJedinkaIt); // Ukloni odabranu jedinku iz preostale
    populacije
    }
    return novaPop;
}

int main() {
    std::vector<Jedinka> populacija = { {'A', 8}, {'D', 6}, {'F', 7}, {'I', 3}, {'B', 4}, {
    'G', 9}, {'E', 4}, {'H', 5}, {'J', 7}, {'C', 8} };
    std::vector<Jedinka> novaPopulacija = pohlepnaSelekcija(populacija);

    std::cout << "Nova populacija:\n";
    for (const auto& jedinka : novaPopulacija)
        std::cout << jedinka.oznaka << ": " << jedinka.dobrota << "\n";
    return 0;
```

```
}
```

9.6 Križanje s jednom točkom prekida

Referenca: Poglavlje 3.1

```
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
public:
    std::vector<bool> geni;
    Jedinka(const std::vector<bool>& geni = {}) : geni(geni) {}
};

void KrizanjeSJednomTockom(const Jedinka& roditelj1, const Jedinka& roditelj2, int*
    tockaPrekida, Jedinka* potomak1, Jedinka* potomak2) {
    // Kreiranje generatora slucajnih brojeva i distribucije
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, roditelj1.geni.size() - 1); // [1, N-1]

    // Nasumicna tocka prekida
    *tockaPrekida = distrib(gen);

    // Prvi potomak - uzimamo gene od prvog roditelja do tocke prekida,
    potomak1->geni = { roditelj1.geni.begin(), roditelj1.geni.begin() + *tockaPrekida };
    // zatim od drugog roditelja nakon prekida
    potomak1->geni.insert(potomak1->geni.end(), roditelj2.geni.begin() + *tockaPrekida,
    roditelj2.geni.end());

    // Drugi potomak - uzimamo gene od drugog roditelja do tocke prekida,
    potomak2->geni = { roditelj2.geni.begin(), roditelj2.geni.begin() + *tockaPrekida };
    // zatim od prvog roditelja nakon prekida
    potomak2->geni.insert(potomak2->geni.end(), roditelj1.geni.begin() + *tockaPrekida,
    roditelj1.geni.end());
}

int main() {
    Jedinka roditelj1({ 1, 0, 0, 1, 0, 1, 0, 1, 1 });
    Jedinka roditelj2({ 0, 1, 1, 0, 0, 1, 1, 0, 0 });

    int tockaPrekida;
    Jedinka potomak1, potomak2;
    KrizanjeSJednomTockom(roditelj1, roditelj2, &tockaPrekida, &potomak1, &potomak2);

    std::cout << "Tocka prekida: " << tockaPrekida << "\n"; // 6
    for (int gen : potomak1.geni) std::cout << gen << " ";  // 1 0 0 1 0 1 1 0 0
    std::cout << "\n";
    for (int gen : potomak2.geni) std::cout << gen << " ";  // 0 1 1 0 0 1 0 1 1
    return 0;
}
```

9.7 Križanje s dvije točke prekida

Referenca: Poglavlje 3.1

```
#include <iostream>
#include <vector>
#include <random>
```

```
class Jedinka {
public:
    std::vector<bool> geni;
    Jedinka(const std::vector<bool>& geni = {}) : geni(geni) {}
};

void KrizanjeSDvijeTockePrekida(const Jedinka& roditelj1, const Jedinka& roditelj2, int*
    tockaPrekida1, int* tockaPrekida2, Jedinka* potomak1, Jedinka* potomak2) {
    // Kreiranje generatora slucajnih brojeva i distribucije
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, roditelj1.geni.size() - 1);

    // Nasumicno odabir dvije tocke prekida
    *tockaPrekida1 = distrib(gen);
    *tockaPrekida2 = distrib(gen);

    // Provjeravamo da su tocke razlicite
    while (*tockaPrekida1 == *tockaPrekida2)
        *tockaPrekida2 = distrib(gen);

    // Ako je tocka prekida1 veca od tocke prekida2, zamijenimo ih
    if (*tockaPrekida1 > *tockaPrekida2)
        std::swap(*tockaPrekida1, *tockaPrekida2);

    // Prvi potomak - uzimamo gene od prvog roditelja do prve tocke prekida,
    potomak1->geni = { roditelj1.geni.begin(), roditelj1.geni.begin() + *tockaPrekida1 };
    // zatim od drugog roditelja izmedu tocaka prekida,
    potomak1->geni.insert(potomak1->geni.end(), roditelj2.geni.begin() + *tockaPrekida1,
    roditelj2.geni.begin() + *tockaPrekida2);
    // i gene od prvog roditelja nakon druge tocke
    potomak1->geni.insert(potomak1->geni.end(), roditelj1.geni.begin() + *tockaPrekida2,
    roditelj1.geni.end());

    // Drugi potomak - uzimamo gene od drugog roditelja do prve tocke prekida,
    potomak2->geni = { roditelj2.geni.begin(), roditelj2.geni.begin() + *tockaPrekida1 };
    // zatim od prvog roditelja izmedu tocaka prekida,
    potomak2->geni.insert(potomak2->geni.end(), roditelj1.geni.begin() + *tockaPrekida1,
    roditelj1.geni.begin() + *tockaPrekida2);
    // i gene od drugog roditelja nakon druge tocke prekida
    potomak2->geni.insert(potomak2->geni.end(), roditelj2.geni.begin() + *tockaPrekida2,
    roditelj2.geni.end());
}

int main() {
    Jedinka roditelj1({ 1, 0, 0, 1, 0, 1, 0, 1, 1 });
    Jedinka roditelj2({ 0, 1, 1, 0, 0, 1, 1, 0, 0 });

    int tockaPrekida1, tockaPrekida2;
    Jedinka potomak1, potomak2;
    KrizanjeSDvijeTockePrekida(roditelj1, roditelj2, &tockaPrekida1, &tockaPrekida2, &
    potomak1, &potomak2);

    std::cout << "Tocke prekida: " << tockaPrekida1 << " i " << tockaPrekida2 << "\n";
    for (int gen : potomak1.geni) std::cout << gen << " ";  // Ispis gena prvog potomka
    std::cout << "\n";
    for (int gen : potomak2.geni) std::cout << gen << " ";  // Ispis gena drugog potomka
    return 0;
}
```

9.8 Uniformno križanje (nasumični odabir gena jednog potomka)

Referenca: Poglavlje 3.2

```
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
public:
    std::vector<bool> geni;
    Jedinka(const std::vector<bool>& geni = {}) : geni(geni) {}
};

void uniformnoKrizanje(const Jedinka& roditelj1, const Jedinka& roditelj2, Jedinka*
    potomak1, Jedinka* potomak2) {
    // Kreiranje generatora slucajnih brojeva i distribucije
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 1);

    for (int i = 0; i < roditelj1.geni.size(); i++) {
        // Nasumicno biramo roditelja za svaki gen
        if (distrib(gen) == 0) {
            // Prvi potomak nasljeduje gen od prvog roditelja, drugi od drugog
            potomak1->geni.push_back(roditelj1.geni[i]);
            potomak2->geni.push_back(roditelj2.geni[i]);
        }
        else {
            // Prvi potomak nasljeduje gen od drugog roditelja, drugi od prvog
            potomak1->geni.push_back(roditelj2.geni[i]);
            potomak2->geni.push_back(roditelj1.geni[i]);
        }
    }
}

int main() {
    Jedinka roditelj1({ 1, 0, 0, 1, 0, 1, 0, 1, 1 });
    Jedinka roditelj2({ 0, 1, 1, 0, 0, 1, 1, 0, 0 });

    Jedinka potomak1, potomak2;
    uniformnoKrizanje(roditelj1, roditelj2, &potomak1, &potomak2);

    for (int gen : potomak1.geni) std::cout << gen << " ";  // Ispis gena prvog potomka
    std::cout << "\n";
    for (int gen : potomak2.geni) std::cout << gen << " ";  // Ispis gena drugog potomka
    return 0;
}
```

9.9 Uniformno križanje (nasumični odabir gena oba potomka)

Referenca: Poglavlje 3.2

```
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
```

```
public:
    std::vector<int> geni;
    Jedinka(const std::vector<int>& geni = {}) : geni(geni) {}
};

// Uniformno krizanje (nasumicni odabir gena oba potomka)
void uniformnoKrizanje(const Jedinka& roditelj1, const Jedinka& roditelj2, Jedinka*
    potomak1, Jedinka* potomak2) {
    // Kreiranje generatora slucajnih brojeva
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 1);

    for (int i = 0; i < roditelj1.geni.size(); i++) {
        // Generiranje nasumicnih brojeva za odabir gena svakog potomka
        int r1 = distrib(gen);
        int r2 = distrib(gen);

        // Odabir gena za potomak1
        if (r1 == 0)
            potomak1->geni.push_back(roditelj1.geni[i]);
        else
            potomak1->geni.push_back(roditelj2.geni[i]);

        // Odabir gena za potomak2
        if (r2 == 0)
            potomak2->geni.push_back(roditelj1.geni[i]);
        else
            potomak2->geni.push_back(roditelj2.geni[i]);
    }
}

int main() {
    // Primjeri roditelja
    Jedinka roditelj1({ 1, 0, 0, 1, 0, 1, 0, 1, 1 });
    Jedinka roditelj2({ 0, 1, 1, 0, 0, 1, 1, 0, 0 });

    // Potomci
    Jedinka potomak1, potomak2;

    // Poziv funkcije za uniformno krizanje
    uniformnoKrizanje(roditelj1, roditelj2, &potomak1, &potomak2);

    // Ispis gena potomaka
    std::cout << "Geni potomka 1: ";
    for (int gen : potomak1.geni) std::cout << gen << " ";
    std::cout << "\n";

    std::cout << "Geni potomka 2: ";
    for (int gen : potomak2.geni) std::cout << gen << " ";
    std::cout << "\n";

    return 0;
}
```

9.10 Ponderirano uniformno križanje

Referenca: Poglavlje 3.2

```
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
public:
    std::vector<bool> geni;
```

```
    double dobrota;  // Dobrota jedinke

    Jedinka(const std::vector<bool>& geni = {}, double dobrota = 1.0) : geni(geni), dobrota
    (dobrota) {}
};

// Uniformno krizanje s ponderiranim nasumicnim odabirom
void ponderiranoUniformnoKrizanje(const Jedinka& roditelj1, const Jedinka& roditelj2,
    Jedinka* potomak1, Jedinka* potomak2) {
    // Kreiranje generatora slucajnih brojeva i distribucija
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> distrib(0.0, 1.0);

    // Racunanje vjerojatnosti nasljedivanja za oba roditelja
    double P1 = roditelj1.dobrota / (roditelj1.dobrota + roditelj2.dobrota);
    double P2 = 1 - P1;

    std::cout << "Vjerojatnost za roditelja 1: " << P1 << std::endl;
    std::cout << "Vjerojatnost za roditelja 2: " << P2 << std::endl;

    for (int i = 0; i < roditelj1.geni.size(); i++) {
        // Generiranje nasumicnih brojeva za svaki gen
        double r1 = distrib(gen);  // Nasumicni broj za prvi potomak
        double r2 = distrib(gen);  // Nasumicni broj za drugi potomak

        // Ispis generiranih brojeva za svaki gen
        std::cout << "Gen " << i + 1 << ":\n";
        std::cout << "  Nasumicni broj za prvi potomak: " << r1 << std::endl;
        std::cout << "  Nasumicni broj za drugi potomak: " << r2 << std::endl;

        // Nasljedivanje gena za prvi potomak na temelju ponderirane vjerojatnosti
        if (r1 <= P1) {
            potomak1->geni.push_back(roditelj1.geni[i]);
            std::cout << "  Potomak 1 nasljeduje gen od roditelja 1: " << roditelj1.geni[i]
     << std::endl;
        }
        else {
            potomak1->geni.push_back(roditelj2.geni[i]);
            std::cout << "  Potomak 1 nasljeduje gen od roditelja 2: " << roditelj2.geni[i]
     << std::endl;
        }

        // Nasljedivanje gena za drugi potomak na temelju ponderirane vjerojatnosti
        if (r2 <= P2) {
            potomak2->geni.push_back(roditelj1.geni[i]);
            std::cout << "  Potomak 2 nasljeduje gen od roditelja 1: " << roditelj1.geni[i]
     << std::endl;
        }
        else {
            potomak2->geni.push_back(roditelj2.geni[i]);
            std::cout << "  Potomak 2 nasljeduje gen od roditelja 2: " << roditelj2.geni[i]
     << std::endl;
        }
    }
}

int main() {
    // Inicijalizacija roditelja s genima i dobrota
    Jedinka roditelj1({ 1, 0, 0 }, 5.0);  // Dobrota roditelja 1
    Jedinka roditelj2({ 0, 1, 1 }, 3.0);  // Dobrota roditelja 2

    Jedinka potomak1, potomak2;

    // Krizanje
    ponderiranoUniformnoKrizanje(roditelj1, roditelj2, &potomak1, &potomak2);

    // Ispis rezultata
    std::cout << "\nPotomak 1: ";
    for (int gen : potomak1.geni) std::cout << gen << " ";  // Ispis gena prvog potomka
```

```
    std::cout << "\n";

    std::cout << "Potomak 2: ";
    for (int gen : potomak2.geni) std::cout << gen << " ";  // Ispis gena drugog potomka
    std::cout << "\n";

    return 0;
}
```

9.11 Uniformno križanje pomoću maske

Referenca: Poglavlje 3.2

```
#include <iostream>
#include <vector>

class Jedinka {
public:
    std::vector<bool> geni;
    Jedinka(const std::vector<bool>& geni = {}) : geni(geni) {}
};

// Uniformno krizanje pomocu maske
void uniformnoKrizanjeMaske(const Jedinka& roditelj1, const Jedinka& roditelj2, const std::
    vector<bool>& maska, Jedinka& potomak1, Jedinka& potomak2) {
    for (size_t i = 0; i < maska.size(); i++) {
        if (maska[i] == 0) {
            potomak1.geni.push_back(roditelj1.geni[i]);
            potomak2.geni.push_back(roditelj2.geni[i]);
        }
        else {
            potomak1.geni.push_back(roditelj2.geni[i]);
            potomak2.geni.push_back(roditelj1.geni[i]);
        }
    }
}

int main() {
    Jedinka roditelj1({ 1, 0, 1, 1, 0 });
    Jedinka roditelj2({ 0, 1, 0, 0, 1 });
    std::vector<bool> maska = { 0, 1, 0, 0, 1 }; // Maska za krizanje

    Jedinka potomak1, potomak2;
    uniformnoKrizanjeMaske(roditelj1, roditelj2, maska, potomak1, potomak2);

    // Ispis rezultata
    std::cout << "Potomak 1: ";
    for (int gen : potomak1.geni) std::cout << gen << " ";
    std::cout << "\n";

    std::cout << "Potomak 2: ";
    for (int gen : potomak2.geni) std::cout << gen << " ";
    std::cout << "\n";

    return 0;
}
```

9.12 Križanje s tri roditelja

Referenca: Poglavlje 3.3

```
#include <iostream>
#include <vector>

class Jedinka {
public:
    std::vector<int> geni;
    Jedinka(const std::vector<int>& geni = {}) : geni(geni) {}
};

// Krizanje s tri roditelja
void krizanjeTriRoditelja(const Jedinka& roditelj1, const Jedinka& roditelj2, const Jedinka
    & roditelj3, Jedinka& potomak) {
    for (size_t i = 0; i < roditelj1.geni.size(); i++) {
        if (roditelj1.geni[i] == roditelj2.geni[i])
            potomak.geni.push_back(roditelj1.geni[i]);
        else
            potomak.geni.push_back(roditelj3.geni[i]);
    }
}

int main() {
    Jedinka roditelj1({ 1, 0, 1, 0, 0 });
    Jedinka roditelj2({ 0, 1, 1, 0, 1 });
    Jedinka roditelj3({ 1, 0, 0, 1, 1 });

    Jedinka potomak;
    krizanjeTriRoditelja(roditelj1, roditelj2, roditelj3, potomak);

    // Ispis rezultata
    std::cout << "Potomak: ";
    for (int gen : potomak.geni) std::cout << gen << " ";
    std::cout << "\n";

    return 0;
}
```

9.13 Aritmetičko križanje

Referenca: Poglavlje 3.4

```
#include <iostream>
#include <vector>

class Jedinka {
public:
    std::vector<double> geni;
    Jedinka(const std::vector<double>& geni = {}) : geni(geni) {}
};

// Aritmeticko krizanje s tezinskim faktorom alpha
void aritmetickoKrizanje(const Jedinka& roditelj1, const Jedinka& roditelj2, double alpha,
    Jedinka& potomak) {
    potomak.geni.clear();
    for (size_t i = 0; i < roditelj1.geni.size(); i++) {
        double gen = alpha * roditelj1.geni[i] + (1 - alpha) * roditelj2.geni[i];
        potomak.geni.push_back(gen);
    }
}

int main() {
    Jedinka roditelj1({ 6.0, 4.0, 8.0 });
    Jedinka roditelj2({ 2.0, 1.0, 5.0 });
    Jedinka potomak;

    aritmetickoKrizanje(roditelj1, roditelj2, 0.3, potomak);
    std::cout << "Potomak: ";
```

```
    for (double gen : potomak.geni) std::cout << gen << " ";
    std::cout << "\n";

    aritmetickoKrizanje(roditelj1, roditelj2, 0.7, potomak);
    std::cout << "Potomak: ";
    for (double gen : potomak.geni) std::cout << gen << " ";
    std::cout << "\n";

    return 0;
}
```

9.14 Heurističko križanje

Referenca: Poglavlje 3.5

```
#include <iostream>
#include <vector>

class Jedinka {
public:
    std::vector<double> geni;
    Jedinka(const std::vector<double>& geni = {}) : geni(geni) {}
};

// Heuristicko krizanje s faktorom r
void heuristickoKrizanje(const Jedinka& roditelj1, const Jedinka& roditelj2, double r,
    Jedinka& potomak) {
    potomak.geni.clear();
    for (size_t i = 0; i < roditelj1.geni.size(); i++) {
        if (roditelj1.geni[i] >= roditelj2.geni[i]) {
            double gen = roditelj1.geni[i] + r * (roditelj1.geni[i] - roditelj2.geni[i]);
            potomak.geni.push_back(gen);
        }
        else {
            double gen = roditelj2.geni[i] + r * (roditelj2.geni[i] - roditelj1.geni[i]);
            potomak.geni.push_back(gen);
        }
    }
}

int main() {
    Jedinka roditelj1({ 6.0, 4.0, 8.0 });
    Jedinka roditelj2({ 2.0, 5.0, 3.0 });
    Jedinka potomak;

    heuristickoKrizanje(roditelj1, roditelj2, 0.3, potomak);
    std::cout << "Potomak (r = 0.3): ";
    for (double gen : potomak.geni) std::cout << gen << " ";
    std::cout << "\n";

    heuristickoKrizanje(roditelj1, roditelj2, 0.7, potomak);
    std::cout << "Potomak (r = 0.7): ";
    for (double gen : potomak.geni) std::cout << gen << " ";
    std::cout << "\n";

    return 0;
}
```

9.15 Djelomično mapirano križanje (PMX)

Referenca: Poglavlje 3.6

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <random>

class Jedinka {
public:
    std::vector<int> geni;
    Jedinka(const std::vector<int>& geni = {}) : geni(geni) {}
};

int mapirajGen(int gen, const std::unordered_map<int, int>& mapa, const std::vector<int>&
    segment) {
    // sve dok se gen vec nalazi u segmentu...
    while (std::find(segment.begin(), segment.end(), gen) != segment.end()) {
        auto it = mapa.find(gen);
        if (it != mapa.end()) gen = it->second; // pokusaj koristiti zamjenski gen
        else break;
    }
    return gen;
}

// PMX krizanje s parametriziranim tockama krizanja
void PMX(const Jedinka& roditelj1, const Jedinka& roditelj2, int t1, int t2, Jedinka&
    potomak1, Jedinka& potomak2) {
    int n = roditelj1.geni.size();
    potomak1.geni.resize(n, -1);
    potomak2.geni.resize(n, -1);

    std::unordered_map<int, int> mapa1, mapa2;

    // Razmjena srednjih segmenata i mapiranje
    for (int i = t1; i <= t2; ++i) {
        potomak1.geni[i] = roditelj2.geni[i];
        potomak2.geni[i] = roditelj1.geni[i];
        mapa1[roditelj2.geni[i]] = roditelj1.geni[i];
        mapa2[roditelj1.geni[i]] = roditelj2.geni[i];
    }

    // Popunjavanje ostalih pozicija
    for (int i = 0; i < n; ++i) {
        if (i >= t1 && i <= t2) continue;

        int gen1 = roditelj1.geni[i];
        gen1 = mapirajGen(gen1, mapa1, { potomak1.geni.begin() + t1, potomak1.geni.begin()
    + t2 + 1 });
        potomak1.geni[i] = gen1;

        int gen2 = roditelj2.geni[i];
        gen2 = mapirajGen(gen2, mapa2, { potomak2.geni.begin() + t1, potomak2.geni.begin()
    + t2 + 1 });
        potomak2.geni[i] = gen2;
    }
}

void ispisiJedinku(const Jedinka& jed, const std::string& oznaka) {
    std::cout << oznaka << ": ";
    for (int gen : jed.geni) std::cout << gen << " ";
    std::cout << "\n";
}

int main() {
    Jedinka roditelj1({ 1, 2, 3, 4, 5, 6, 7, 8, 9 });
    Jedinka roditelj2({ 5, 4, 6, 9, 2, 1, 7, 8, 3 });
    Jedinka potomak1, potomak2;

    // Tocke krizanja
    int t1 = 2; // ili slucajni odabir
    int t2 = 5; // ...
```

```
    std::cout << "Tocke krizanja: " << t1 << " - " << t2 << "\n\n";

    ispisiJedinku(roditelj1, "Roditelj 1");
    ispisiJedinku(roditelj2, "Roditelj 2");

    PMX(roditelj1, roditelj2, t1, t2, potomak1, potomak2);

    ispisiJedinku(potomak1, "Potomak 1");
    ispisiJedinku(potomak2, "Potomak 2");
    return 0;
}
```

9.16 Uniformna mutacija

Referenca: Poglavlje 4.1

```
#include <iostream>
#include <vector>
#include <random>

// Binarna jedinka
class JedinkaBinarna {
public:
    std::vector<bool> geni;
    JedinkaBinarna(const std::vector<bool>& geni = {}) : geni(geni) {}
};

// Realna jedinka
class JedinkaRealna {
public:
    std::vector<double> geni;
    double minVal, maxVal;

    JedinkaRealna(const std::vector<double>& geni, double minVal, double maxVal)
        : geni(geni), minVal(minVal), maxVal(maxVal) {
    }
};

// Diskretna jedinka
class JedinkaDiskretna {
public:
    std::vector<int> geni;
    std::vector<int> V;

    JedinkaDiskretna(const std::vector<int>& geni, const std::vector<int>& V)
        : geni(geni), V(V) {
    }
};

// Uniformna mutacija binarnih gena (inverzija)
void uniformnaMutacijaBinarna(JedinkaBinarna* jedinka, double pm) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_real_distribution<double> distr(0.0, 1.0);

    for (size_t i = 0; i < jedinka->geni.size(); i++) {
        if (distr(gen) <= pm)
            jedinka->geni[i] = 1 - jedinka->geni[i];
    }
}

// Uniformna mutacija realnih gena
void uniformnaMutacijaRealna(JedinkaRealna* jedinka, double pm) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
```

```
    std::uniform_real_distribution<double> distr(0.0, 1.0);
    std::uniform_real_distribution<double> realDistr(jedinka->minVal, jedinka->maxVal);

    for (size_t i = 0; i < jedinka->geni.size(); i++) {
        if (distr(gen) <= pm)
            jedinka->geni[i] = realDistr(gen);
    }
}

// Uniformna mutacija diskretnih gena
void uniformnaMutacijaDiskretna(JedinkaDiskretna* jedinka, double pm) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_real_distribution<double> distr(0.0, 1.0);

    for (size_t i = 0; i < jedinka->geni.size(); i++) {
        if (distr(gen) <= pm) {
            std::uniform_int_distribution<size_t> indexDistr(0, jedinka->V.size() - 1);
            int novaVrijednost;
            do {
                novaVrijednost = jedinka->V[indexDistr(gen)];
            } while (novaVrijednost == jedinka->geni[i]); // Osigurava promjenu vrijednosti
            jedinka->geni[i] = novaVrijednost;
        }
    }
}

int main() {
    double pm = 0.2; // Vjerojatnost mutacije (20%)

    // Binarna mutacija
    JedinkaBinarna binarnaJedinka({ 1, 0, 0, 1, 0, 1, 0, 1, 1 });
    std::cout << "Binarna prije mutacije: ";
    for (int gen : binarnaJedinka.geni) std::cout << gen << " ";
    std::cout << "\n";

    uniformnaMutacijaBinarna(&binarnaJedinka, pm);
    std::cout << "Binarna nakon mutacije: ";
    for (int gen : binarnaJedinka.geni) std::cout << gen << " ";
    std::cout << "\n\n";

    // Realna mutacija
    JedinkaRealna realnaJedinka({ 1.2, 3.4, 5.6, 7.8, 9.6 }, 0.0, 10.0);
    std::cout << "Realna prije mutacije: ";
    for (double gen : realnaJedinka.geni) std::cout << gen << " ";
    std::cout << "\n";

    uniformnaMutacijaRealna(&realnaJedinka, pm);
    std::cout << "Realna nakon mutacije: ";
    for (double gen : realnaJedinka.geni) std::cout << gen << " ";
    std::cout << "\n\n";

    // Diskretna mutacija
    std::vector<int> V = { 0, 2, 4, 6, 8 }; // Skup mogucih vrijednosti
    JedinkaDiskretna diskretnaJedinka({ 0, 2, 0, 6, 8 }, V);
    std::cout << "Diskretna prije mutacije: ";
    for (int gen : diskretnaJedinka.geni) std::cout << gen << " ";
    std::cout << "\n";

    uniformnaMutacijaDiskretna(&diskretnaJedinka, pm);
    std::cout << "Diskretna nakon mutacije: ";
    for (int gen : diskretnaJedinka.geni) std::cout << gen << " ";
    std::cout << "\n";

    return 0;
}
```

9.17 Gaussova mutacija

Referenca: Poglavlje 4.2

```
#include <iostream>
#include <vector>
#include <random>

class VisinaMuskaraca {
public:
    std::vector<int> geni;
    int Xmin, Xmax; // interval vrijednosti
    VisinaMuskaraca(const std::vector<int>& geni, int Xmin, int Xmax)
        : geni(geni), Xmin(Xmin), Xmax(Xmax) {
    }
};

// Funkcija za Gaussovu mutaciju s parametrom pm (vjerojatnost mutacije)
void gaussMutacija(VisinaMuskaraca* jedinka, double pm, double srednjaVr, double stdDev) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<> normalDist(srednjaVr, stdDev);
    std::uniform_real_distribution<> uniformDist(0.0, 1.0);

    for (size_t i = 0; i < jedinka->geni.size(); ++i) {
        // Za svaki gen odlucujemo hoce li se mutirati na temelju pm
        if (uniformDist(gen) <= pm) {
            int promjena = static_cast<int>(normalDist(gen));  // Slucajna promjena
            //std::cout << promjena << "\n";
            int novaVr = jedinka->geni[i] + promjena;

            // Osiguravamo da nova vrijednost ostane unutar dozvoljenog opsega
            novaVr = std::max(jedinka->Xmin, std::min(novaVr, jedinka->Xmax));
            jedinka->geni[i] = novaVr;
        }
    }
}

int main() {
    std::vector<int> geni = { 170, 180, 190, 165, 185 };
    VisinaMuskaraca vM(geni, 120, 220); // Xmin = 120, Xmax = 220

    std::cout << "Prije mutacije: ";
    for (int gene : vM.geni) std::cout << gene << " ";
    std::cout << std::endl;

    // Parametri: pm = 0.2, srednja vrijednost = 0, std. dev. = 10
    gaussMutacija(&vM, 0.2, 0, 10);

    std::cout << "Nakon mutacije: ";
    for (int gene : vM.geni) std::cout << gene << " ";
    std::cout << std::endl;

    return 0;
}
```

9.18 Mutacija zamjenom

Referenca: Poglavlje 4.3

```
#include <iostream>
#include <vector>
#include <random>
```

```
#include <algorithm>

class TrgovackiPutnik {
public:
    std::vector<int> ruta; // Geni predstavljaju redoslijed gradova
    TrgovackiPutnik(const std::vector<int>& ruta) : ruta(ruta) {}
};

// Funkcija za mutaciju zamjenom s parametrom pm (vjerojatnost mutacije)
void mutacijaZamjenom(TrgovackiPutnik* jedinka, double pm) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> uniformDist(0.0, 1.0);
    std::uniform_int_distribution<> intDist(0, jedinka->ruta.size() - 1);

    for (size_t i = 0; i < jedinka->ruta.size(); ++i) {
        // Za svaki grad odlucujemo hoce li se mutirati na temelju pm
        if (uniformDist(gen) <= pm) {
            int j;
            do {
                j = intDist(gen);
            } while (j == i);  // Osiguravamo da je j razlicit od i

            // Zamjena gradova
            std::swap(jedinka->ruta[i], jedinka->ruta[j]);
        }
    }
}
int main() {
    std::vector<int> ruta = { 0, 1, 2, 3, 4, 5 }; // Gradovi oznaceni brojevima 0-5
    TrgovackiPutnik tp(ruta);

    std::cout << "Prije mutacije: ";
    for (int grad : tp.ruta) std::cout << grad << " ";
    std::cout << std::endl;

    mutacijaZamjenom(&tp, 0.2); // Parametar: pm = 0.2
    std::cout << "Nakon mutacije: ";
    for (int grad : tp.ruta) std::cout << grad << " ";
    std::cout << std::endl;

    return 0;
}
```

9.19 Inverzijska mutacija

Referenca: Poglavlje 4.4

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>

class TrgovackiPutnik {
public:
    std::vector<int> ruta; // Geni predstavljaju redoslijed gradova
    TrgovackiPutnik(const std::vector<int>& ruta) : ruta(ruta) {}
};

// Funkcija za inverzijsku mutaciju s parametrom pm (vjerojatnost mutacije)
void inverzijskaMutacija(TrgovackiPutnik* jedinka, double pm) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> uniformDist(0.0, 1.0);

    // Odlucujemo hoce li se mutacija dogoditi na temelju pm
```

```
    if (uniformDist(gen) <= pm) {
        std::uniform_int_distribution<> intDist(0, jedinka->ruta.size() - 1);

        int start, end;
        // Osiguravamo da su start i end razliciti
        do {
            start = intDist(gen);
            end = intDist(gen);
        } while (start == end);

        // Osiguravamo da je start manji od end
        if (start > end) std::swap(start, end);

        // Inverzija
        std::reverse(jedinka->ruta.begin() + start, jedinka->ruta.begin() + end + 1);
    }
}

int main() {
    std::vector<int> ruta = { 0, 1, 2, 3, 4, 5 }; // Gradovi oznaceni brojevima 0-5
    TrgovackiPutnik tp(ruta);

    std::cout << "Prije mutacije: ";
    for (int grad : tp.ruta) std::cout << grad << " ";
    std::cout << std::endl;

    // Parametar: pm = 0.8 (visoka vjerojatnost za demonstraciju)
    inverzijskaMutacija(&tp, 0.8);

    std::cout << "Nakon mutacije: ";
    for (int grad : tp.ruta) std::cout << grad << " ";
    std::cout << std::endl;

    return 0;
}
```

9.20 Sekvencijalna i višedretvena evaluacija populacije

Referenca: Poglavlje 5.2.1

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>

class Jedinka {
public:
    std::vector<bool> gen = std::vector<bool>(32); // broj bitova
    unsigned int dobrota{ std::numeric_limits<unsigned int>::max() };
    void evaluiraj(int optimalnaVrijednost);
};
void Jedinka::evaluiraj(int optimalnaVrijednost) {
    // binarno u cijeli broj
    unsigned int broj = 0;
    for (unsigned int i = 0; i < gen.size(); i++)
        broj += gen[i] * (int)pow(2, gen.size() - i - 1);

    // vrijednost dobrote (udaljenost od optimalne vrijednosti)
    if (broj < optimalnaVrijednost)
        dobrota = optimalnaVrijednost - broj;
    else
        dobrota = broj - optimalnaVrijednost;
    // za testiranje s visedretvenih pristupom
    std::this_thread::sleep_for(std::chrono::nanoseconds(500));
}
```

```
class Populacija {
public:
    std::vector<Jedinka> jedinka;
    void generirajSlucajnu(int popVel);
    void evaluiraj(bool visedretveno, int optimalnaVrijednost);
};
void Populacija::generirajSlucajnu(int popVel) {
    jedinka.clear();
    for (int i = 0; i < popVel; i++) {
        Jedinka pom;
        for (unsigned int j = 0; j < pom.gen.size(); j++)
            pom.gen[j] = rand() % 2;
        jedinka.push_back(pom);
    }
}
void Populacija::evaluiraj(bool visedretveno, int optimalnaVrijednost) {
    if (!visedretveno) {
        // Sekvencijalna evaluacija svih jedinki (bez paralelizacije)
        for (unsigned int i = 0; i < jedinka.size(); i++)
            jedinka[i].evaluiraj(optimalnaVrijednost);
        return;
    }

    // Dohvati broj dostupnih procesorskih jezgri ili koristi barem jednu dretvu
    unsigned brojDretvi = std::max(1u, std::thread::hardware_concurrency());
    if (jedinka.empty()) return; // Ako nema jedinki, prekini izvrsavanje

    std::vector<std::thread> dretve;
    int pocetak = 0, iskoristeniZadaci = 0, zadaciZaOvuDretvu = 0;

    // Kreiranje i pokretanje dretvi
    for (unsigned int i = 0; i < std::min((unsigned)jedinka.size(), brojDretvi); i++) {
        // Odredivanje broja zadataka za ovu dretvu (ravnomjerno rasporedivanje)
        zadaciZaOvuDretvu = jedinka.size() / brojDretvi + (i < jedinka.size() % brojDretvi)
    ;
        int kraj = std::min(pocetak + zadaciZaOvuDretvu - 1, (int)(jedinka.size()) - 1);

        // Pokretanje dretve za evaluaciju dijela populacije
        std::thread dretva([=](unsigned _pocetak, unsigned _kraj) {
            for (unsigned int j = _pocetak; j <= _kraj; j++)
                jedinka[j].evaluiraj(optimalnaVrijednost);
            }, pocetak, kraj);
        dretve.push_back(std::move(dretva));

        // Azuriranje iskoristenih zadataka nakon pokretanja dretve
        iskoristeniZadaci += zadaciZaOvuDretvu;
        pocetak = iskoristeniZadaci;
    }

    // Cekanje da sve dretve zavrse izvrsavanje
    for (auto& dretva : dretve)
        dretva.join();
}

int main() {
    int seed = 1687280607; //(unsigned)time(NULL);
    srand(seed);

    int popVel = 1000;
    int optimalnaVrijednost = 1234567890;
    bool visedretveno = 0;

    auto start_t = std::chrono::high_resolution_clock::now();
    std::vector<Populacija> generacija;

    Populacija slucajnaPop;
    slucajnaPop.generirajSlucajnu(popVel);
    slucajnaPop.evaluiraj(visedretveno, optimalnaVrijednost);
    generacija.push_back(slucajnaPop);
```

```
    auto kraj_t = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(kraj_t - start_t)
    .count();
    std::cout << "Trajanje: " << duration << " ms.\n";
    return 0;
}
```

9.21 Evaluacija populacije korištenjem strategije memoizacije

Referenca: Poglavlje 5.3

```
#include <iostream>
#include <vector>
#include <random>
#include <unordered_map>

std::unordered_map<unsigned int, unsigned int> memoizacija; // memoizacijska struktura

class Jedinka {
public:
    std::vector<bool> gen = std::vector<bool>(32);
    unsigned int dobrota{ std::numeric_limits<unsigned int>::max() };
    void evaluiraj(int optimalnaVrijednost);
    inline static int memBrojac = 0;
};

void Jedinka::evaluiraj(int optimalnaVrijednost) {
    unsigned int broj = 0;
    for (unsigned int i = 0; i < gen.size(); i++)
        broj += gen[i] * (1u << (gen.size() - i - 1));

    // Provjera postoji li vec izracunata vrijednost dobrote za ovu binarnu vrijednost.
    auto it = memoizacija.find(broj);
    if (it != memoizacija.end()) {
        dobrota = it->second; // Ako je vrijednost vec izracunata, dohvacamo je iz tablice
        memBrojac++;
        std::cout << broj << "\n"; // vec pronadena vrijednost u tablici rasprsivanja
    }
    else { // Ako vrijednost nije vec izracunata,
        dobrota = (broj < optimalnaVrijednost) ? (optimalnaVrijednost - broj) : (broj -
    optimalnaVrijednost);
        memoizacija[broj] = dobrota; // vrijednost dodajemo u tablicu rasprsivanja
    }
}

class Populacija {
public:
    std::vector<Jedinka> jedinka;
    void generirajSlucajnu(int popVel);
    void evaluiraj(int optimalnaVrijednost);
};

void Populacija::generirajSlucajnu(int popVel) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 1);

    jedinka.clear();
    for (int i = 0; i < popVel; i++) {
        Jedinka pom;
        for (unsigned int j = 0; j < pom.gen.size(); j++)
            pom.gen[j] = dis(gen);
        jedinka.push_back(pom);
```

```
    }
}

void Populacija::evaluiraj(int optimalnaVrijednost) {
    for (unsigned int i = 0; i < jedinka.size(); i++)
        jedinka[i].evaluiraj(optimalnaVrijednost);
}

int main() {
    int popVel = 250000;
    int optimalnaVrijednost = 1234567890;
    std::vector<Populacija> generacija;

    Populacija slucajnaPop;
    slucajnaPop.generirajSlucajnu(popVel);
    slucajnaPop.evaluiraj(optimalnaVrijednost);
    generacija.push_back(slucajnaPop);

    std::cout << "Memoizacija: " << Jedinka::memBrojac << " preskocenih evaluacija!\n";
    return 0;
}
```

9.22 Penjanje uz brijeg (binarna reprezentacija broja)

Referenca: Poglavlje 6.2

```
#include <iostream>
#include <vector>

class Jedinka {
public:
    std::vector<bool> gen = std::vector<bool>(8);
    unsigned int dobrota = UINT8_MAX;

    Jedinka(const std::vector<bool>& _gen = {});
    void evaluiraj(unsigned int optimum);
    unsigned int uBroj();
    void flipBit(int i);
};

Jedinka::Jedinka(const std::vector<bool>& _gen) {
    if (_gen.empty())
        gen = std::vector<bool>(8);
    else
        gen = _gen;
}

void Jedinka::flipBit(int i) {
    if (i >= 0 && i < gen.size()) {
        gen[i] = !gen[i];
    }
}

void Jedinka::evaluiraj(unsigned int optimum) {
    unsigned int broj = uBroj();
    dobrota = (broj < optimum) ? (optimum - broj) : (broj - optimum);
}

unsigned int Jedinka::uBroj() {
    unsigned int sum = 0;
    for (unsigned int i = 0; i < gen.size(); i++)
        sum += gen[i] * (1U << (gen.size() - i - 1));
    return sum;
}

std::vector<bool> brojUBitove(unsigned int broj, size_t duljina = 8) {
```

```
    std::vector<bool> bitovi(duljina, 0);
    for (int i = 0; i < duljina; ++i)
        bitovi[duljina - i - 1] = (broj >> i) & 1;
    return bitovi;
}

Jedinka PenjanjeUzBrijeg(Jedinka Rjesenje, unsigned int optimum) {
    Rjesenje.evaluiraj(optimum);
    bool zapeo = false;
    int brojIteracija = 0;
    const int maksIteracija = 1000;
    int prethodnaVrijednost = Rjesenje.uBroj();

    // ponavljaj dok nije lokalni optimum, maksimum iteracija, ili dok nije pronadeno
    rjesenje
    while (!zapeo && brojIteracija <= maksIteracija && Rjesenje.uBroj() != optimum) {
        zapeo = true;
        Jedinka najbolji = Rjesenje;
        for (int i = 0; i < Rjesenje.gen.size(); ++i) {
            Jedinka susjed = Rjesenje;
            susjed.flipBit(i);
            susjed.evaluiraj(optimum);

            if (susjed.dobrota < najbolji.dobrota) {
                najbolji = susjed;
                zapeo = false;
            }
        }
        if (najbolji.uBroj() != prethodnaVrijednost) {
            std::cout << "Iteracija: " << brojIteracija + 1
                << " = " << najbolji.uBroj() << std::endl;
            prethodnaVrijednost = najbolji.uBroj();
        }
        Rjesenje = najbolji;
        ++brojIteracija;
    }
    return Rjesenje;
}

int main() {
    unsigned int optimum = 175;
    Jedinka pocetnoRjesenje(brojUBitove(53));
    Jedinka rezultat = PenjanjeUzBrijeg(pocetnoRjesenje, optimum);
    std::cout << "Broj: " << rezultat.uBroj() << ", Dobrota: " << rezultat.dobrota << std::
    endl;
    return 0;
}
```

9.23 Penjanje uz brijeg (problem trgovačkog putnika)

Referenca: Poglavlje 6.2

```
#include <iostream>
#include <vector>

class Jedinka {
public:
    std::vector<int> put;
    int dobrota = INT_MAX;

    Jedinka(const std::vector<int>& _put = {});
    void evaluiraj(const std::vector<std::vector<int>>& matricaUdaljenosti);
    void zamijeniGradove(int i, int j);
};

Jedinka::Jedinka(const std::vector<int>& _put) {
```

```
    if (_put.empty())
        put = { 0, 1, 2, 3, 4 }; // inicijalni redoslijed gradova
    else
        put = _put;
}

void Jedinka::zamijeniGradove(int i, int j) {
    if (i >= 0 && j >= 0 && i < put.size() && j < put.size())
        std::swap(put[i], put[j]);
}

void Jedinka::evaluiraj(const std::vector<std::vector<int>>& matricaUdaljenosti) {
    int ukupnaUdaljenost = 0;
    for (size_t i = 0; i < put.size() - 1; ++i)
        ukupnaUdaljenost += matricaUdaljenosti[put[i]][put[i + 1]];

    // Povratak u pocetni grad
    ukupnaUdaljenost += matricaUdaljenosti[put.back()][put.front()];
    dobrota = ukupnaUdaljenost;
}

void ispisiPut(const Jedinka& jedinka) {
    for (int grad : jedinka.put)
        std::cout << grad << " -> ";
    std::cout << jedinka.put.front()
        << "\tDuljina puta (dobrota): "
        << jedinka.dobrota << std::endl; // povratak
}

Jedinka PenjanjeUzBrijeg(Jedinka rjesenje, const std::vector<std::vector<int>>& matrica) {
    rjesenje.evaluiraj(matrica);
    //std::cout << "Pocetno rjesenje: ";
    //ispisiPut(rjesenje);

    bool zapeo = false;
    int brojIteracija = 0;
    const int maksIteracija = 1000;

    while (!zapeo && brojIteracija < maksIteracija) {
        zapeo = true;
        Jedinka najbolji = rjesenje;
        int brojSusjeda = 0;

        //std::cout << "\n--- Iteracija " << brojIteracija + 1 << " ---\n";
        for (size_t i = 0; i < rjesenje.put.size(); ++i) {
            for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
                ++brojSusjeda;
                Jedinka susjed = rjesenje;
                susjed.zamijeniGradove(i, j);
                susjed.evaluiraj(matrica);

                //std::cout << "Susjed " << brojSusjeda << " (zamjena gradova "
                //    << i << " i " << j << "): ";
                //ispisiPut(susjed);

                if (susjed.dobrota < najbolji.dobrota) {
                    najbolji = susjed;
                    zapeo = false;
                  //  std::cout << ">>> Poboljsanje! Novo najbolje rjesenje (dobrota "
                  //      << najbolji.dobrota << ")\n";
                }
            }
        }
        rjesenje = najbolji;
        ++brojIteracija;
    }
    return rjesenje;
}

int main() {
```

```
    // matrica udaljenosti izmedu 5 gradova
    std::vector<std::vector<int>> udaljenosti = {
        { 0, 10, 13, 20, 19 },
        { 10,  0, 23, 22, 19 },
        { 13, 23,  0, 24, 25 },
        { 20, 22, 24,  0,  5 },
        { 19, 19, 25,  5,  0 }
    };

    std::vector<int> pocetniPut = { 0, 1, 2, 4, 3 };
    Jedinka pocetnoRjesenje(pocetniPut);
    Jedinka rezultat = PenjanjeUzBrijeg(pocetnoRjesenje, udaljenosti);

    std::cout << "Najbolji pronadeni put: ";
    ispisiPut(rezultat);
    return 0;
}
```

9.24 Penjanje uz brijeg s nasumičnim ponovnim pokretanjem

Referenca: Poglavlje 6.3

```
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
public:
    std::vector<bool> gen = std::vector<bool>(8);
    unsigned int dobrota = UINT8_MAX;

    Jedinka(const std::vector<bool>& _gen = {});
    void evaluiraj(unsigned int optimum);
    unsigned int uBroj();
    void flipBit(int i);
};

Jedinka::Jedinka(const std::vector<bool>& _gen) {
    if (_gen.empty())
        gen = std::vector<bool>(8);
    else
        gen = _gen;
}

void Jedinka::flipBit(int i) {
    if (i >= 0 && i < gen.size())
        gen[i] = !gen[i];
}

void Jedinka::evaluiraj(unsigned int optimum) {
    unsigned int broj = uBroj();
    dobrota = (broj < optimum) ? (optimum - broj) : (broj - optimum);
}

unsigned int Jedinka::uBroj() {
    unsigned int sum = 0;
    for (unsigned int i = 0; i < gen.size(); i++)
        sum += gen[i] * (1U << (gen.size() - i - 1));
    return sum;
}

std::vector<bool> brojUBitove(unsigned int broj, size_t duljina = 8) {
    std::vector<bool> bitovi(duljina, 0);
```

```
    for (int i = 0; i < duljina; ++i)
        bitovi[duljina - i - 1] = (broj >> i) & 1;
    return bitovi;
}

Jedinka PenjanjeUzBrijeg(Jedinka Rjesenje, unsigned int optimum) {
    Rjesenje.evaluiraj(optimum);
    bool zapeo = false;
    int brojIteracija = 0;
    const int maksIteracija = 1000;
    int prethodnaVrijednost = Rjesenje.uBroj();

    // ponavljaj dok nije lokalni optimum, maksimum iteracija, ili dok nije pronadeno
    rjesenje
    while (!zapeo && brojIteracija <= maksIteracija && Rjesenje.uBroj() != optimum) {
        zapeo = true;
        Jedinka najbolji = Rjesenje;
        for (int i = 0; i < Rjesenje.gen.size(); ++i) {
            Jedinka susjed = Rjesenje;
            susjed.flipBit(i);
            susjed.evaluiraj(optimum);

            if (susjed.dobrota < najbolji.dobrota) {
                najbolji = susjed;
                zapeo = false;
            }
        }
        if (najbolji.uBroj() != prethodnaVrijednost) {
            std::cout << "Iteracija: " << brojIteracija + 1
                << " = " << najbolji.uBroj() << std::endl;
            prethodnaVrijednost = najbolji.uBroj();
        }
        Rjesenje = najbolji;
        ++brojIteracija;
    }
    return Rjesenje;
}

Jedinka PenjanjeUzBrijegRestart(unsigned int optimum, int brojRestarta) {
    Jedinka najboljeRjesenje;
    najboljeRjesenje.dobrota = UINT8_MAX;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 1);

    for (int i = 0; i < brojRestarta && najboljeRjesenje.uBroj() != optimum; ++i) {
        // Generiraj slucajno pocetno rjesenje
        Jedinka pocetno;
        generate(pocetno.gen.begin(), pocetno.gen.end(), [&]() { return distrib(gen) % 2;
    });

        std::cout << "\n>>> Nasumicni restart " << (i + 1)
            << " | pocetak = " << pocetno.uBroj() << std::endl;

        // Lokalna pretraga iz trenutnog pocetnog rjesenja
        Jedinka lokalno = PenjanjeUzBrijeg(pocetno, optimum);

        if (lokalno.dobrota < najboljeRjesenje.dobrota)
            najboljeRjesenje = lokalno;

        std::cout << ">>> Kraj restarta " << (i + 1) << ": " << lokalno.uBroj()
            << " (dobrota = " << lokalno.dobrota << ")\n";
    }
    return najboljeRjesenje;
}

int main() {
    unsigned int optimum = 175;
    int pocetnoRjesenjeBroj = 53;
    Jedinka pocetnoRjesenje(brojUBitove(pocetnoRjesenjeBroj));
```

```
    std::cout << "pocetak = " << pocetnoRjesenje.uBroj() << std::endl;

    // ako penjanje uz brijeg nije rezultiralo globalnim optimumom
    if (PenjanjeUzBrijeg(pocetnoRjesenje, optimum).uBroj() != optimum) {
        // penjanje uz brijeg s nasumicnim ponovnim pokretanjem (maks. 5 poziva)
        Jedinka rezultat = PenjanjeUzBrijegRestart(optimum, 5);
        std::cout << "\n>>> Najbolje globalno rjesenje = " << rezultat.uBroj()
            << ", Dobrota: " << rezultat.dobrota << std::endl;
    }
    return 0;
}
```

9.25 Simulirano kaljenje (binarna reprezentacija broja)

Referenca: Poglavlje 6.4

```
#include <iostream>
#include <vector>
#include <random>

class Jedinka {
public:
    std::vector<bool> gen = std::vector<bool>(8);
    unsigned int dobrota = UINT8_MAX;

    Jedinka(const std::vector<bool>& _gen = {});
    void evaluiraj(unsigned int optimum);
    unsigned int uBroj() const;
    void flipBit(int i);
};

Jedinka::Jedinka(const std::vector<bool>& _gen) {
    if (_gen.empty())
        gen = std::vector<bool>(8);
    else
        gen = _gen;
}

void Jedinka::flipBit(int i) {
    if (i >= 0 && i < gen.size()) {
        gen[i] = !gen[i];
    }
}

void Jedinka::evaluiraj(unsigned int optimum) {
    unsigned int broj = uBroj();
    dobrota = (broj < optimum) ? (optimum - broj) : (broj - optimum);
}

unsigned int Jedinka::uBroj() const {
    unsigned int sum = 0;
    for (unsigned int i = 0; i < gen.size(); i++)
        sum += gen[i] * (1U << (gen.size() - i - 1));
    return sum;
}

std::vector<bool> brojUBitove(unsigned int broj, size_t duljina = 8) {
    std::vector<bool> bitovi(duljina, 0);
    for (int i = 0; i < duljina; ++i)
        bitovi[duljina - i - 1] = (broj >> i) & 1;
    return bitovi;
}

// Funkcija za generiranje slucajnog susjeda
Jedinka generirajSusjeda(const Jedinka& original, std::mt19937& rng) {
    Jedinka novi = original;
```

```
    std::uniform_int_distribution<int> distribucija(0, original.gen.size() - 1);
    int pozicija = distribucija(rng);
    novi.flipBit(pozicija);
    return novi;
}

Jedinka SimuliranoKaljenje(
    Jedinka r,                  // pocetno rjesenje
    unsigned int optimum,       // ciljna vrijednost
    double T0,                  // pocetna temperatura
    double Tmin,                // minimalna temperatura
    double alpha,               // faktor hladenja
    int maksIteracija           // maksimalni broj iteracija
) {
    std::random_device rd;
    std::mt19937 rng(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    double T = T0;
    r.evaluiraj(optimum);
    Jedinka rNajbolji = r;

    int iter = 0;
    while (T > Tmin && iter < maksIteracija && rNajbolji.uBroj() != optimum) {
        Jedinka rSusjed = generirajSusjeda(r, rng);
        rSusjed.evaluiraj(optimum);

        int delta = rSusjed.dobrota - r.dobrota;

        if (delta < 0) { // ako je susjedno rjesenje bolje (blize optimumu)
            r = rSusjed;
            if (r.dobrota < rNajbolji.dobrota)
                rNajbolji = r;
        }
        else { // susjedno rjesenje je losije
            double vjerojatnost = std::exp(-delta / T);
            if (dist(rng) < vjerojatnost)
                r = rSusjed;
        }

        std::cout << "Iteracija: " << iter + 1
            << ", Broj: " << r.uBroj()
            << ", Dobrota: " << r.dobrota
            << ", Temp: " << T << std::endl;

        T *= alpha;
        ++iter;
    }
    return rNajbolji;
}

int main() {
    unsigned int optimum = 175;
    Jedinka pocetnoRjesenje(brojUBitove(53));

    // Parametri SA algoritma
    double T0 = 100.0;
    double Tmin = 0.01;
    double alpha = 0.95;
    int maksIteracija = 1000;

    Jedinka rezultat = SimuliranoKaljenje(
        pocetnoRjesenje, optimum, T0, Tmin, alpha, maksIteracija
    );

    std::cout << "Najbolji broj: " << rezultat.uBroj()
        << ", Dobrota: " << rezultat.dobrota << std::endl;
    return 0;
}
```

9.26 Tabu pretraživanje s pamćenjem rješenja (problem trgovačkog putnika)

Referenca: Poglavlje 6.5

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <limits>

class Jedinka {
public:
    std::vector<int> put;
    int dobrota = INT_MAX;

    Jedinka(const std::vector<int>& _put = {});
    void evaluiraj(const std::vector<std::vector<int>>& matricaUdaljenosti);
    void zamijeniGradove(int i, int j);
};

Jedinka::Jedinka(const std::vector<int>& _put) {
    if (_put.empty())
        put = { 0, 1, 2, 3 }; // default ruta
    else
        put = _put;
}

void Jedinka::zamijeniGradove(int i, int j) {
    if (i >= 0 && j >= 0 && i < put.size() && j < put.size())
        std::swap(put[i], put[j]);
}

void Jedinka::evaluiraj(const std::vector<std::vector<int>>& matricaUdaljenosti) {
    int udaljenost = 0;
    for (size_t i = 0; i < put.size() - 1; ++i)
        udaljenost += matricaUdaljenosti[put[i]][put[i + 1]];
    udaljenost += matricaUdaljenosti[put.back()][put.front()];
    dobrota = udaljenost;
}

void ispisiPut(const Jedinka& jedinka) {
    for (int grad : jedinka.put)
        std::cout << grad << " -> ";
    std::cout << jedinka.put.front();
    std::cout << "\tDuljina puta: " << jedinka.dobrota << std::endl;
}

Jedinka TabuPretraga(Jedinka rjesenje, const std::vector<std::vector<int>>& matrica, int
    tabuDuljina = 5, int maksIteracija = 100) {
    rjesenje.evaluiraj(matrica);          // Evaluiraj pocetno rjesenje
    Jedinka najboljeRjesenje = rjesenje; // Inicijalno postavi kao najbolje
    std::list<Jedinka> tabuLista;  // Lista zabranjenih (nedavno posjecenih) rjesenja
    int iteracija = 0;

    while (iteracija < maksIteracija) {
        Jedinka najboljiKandidat;  // Trenutno najbolji susjed
        najboljiKandidat.dobrota = INT_MAX;
        bool nadjen = false;  // Pracenje je li pronaden kandidat izvan tabu liste

        // Generiranje i evaluacija svih susjeda
        for (size_t i = 0; i < rjesenje.put.size(); ++i) {
            for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
                Jedinka susjed = rjesenje;
                susjed.zamijeniGradove(i, j);  // Zamjena dvaju gradova = novi susjed
                susjed.evaluiraj(matrica);     // Izracunaj duljinu rute
```

```
                // Provjera nalazi li se susjed u tabu listi (usporeduje se put)
                auto it = std::find_if(tabuLista.begin(), tabuLista.end(),
                    [&](const Jedinka& e) { return e.put == susjed.put; });

                // Ako nije tabu i bolji je od dosadasnjeg kandidata - prihvati
                if (it == tabuLista.end() && susjed.dobrota < najboljiKandidat.dobrota) {
                    najboljiKandidat = susjed;
                    nadjen = true;
                }
            }
        }
        if (!nadjen) {
            // Ako su svi susjedi tabu - ignoriraj tabu listu i uzmi najbolji moguci susjed
            for (size_t i = 0; i < rjesenje.put.size(); ++i) {
                for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
                    Jedinka susjed = rjesenje;
                    susjed.zamijeniGradove(i, j);
                    susjed.evaluiraj(matrica);

                    if (susjed.dobrota < najboljiKandidat.dobrota)
                        najboljiKandidat = susjed;
                }
            }
        }
        rjesenje = najboljiKandidat; // Pomak na novo rjesenje

        // Azuriraj najbolje globalno rjesenje ako je novo bolje
        if (rjesenje.dobrota < najboljeRjesenje.dobrota) {
            najboljeRjesenje = rjesenje;
            std::cout << "Iteracija " << iteracija + 1 << " -> novo najbolje rjesenje: ";
            ispisiPut(rjesenje);
        }
        // Dodaj novo rjesenje u tabu listu
        tabuLista.push_back(rjesenje);

        // Odrzavanje duljine tabu liste (FIFO)
        if (tabuLista.size() > static_cast<size_t>(tabuDuljina))
            tabuLista.pop_front(); // Izbaci najstarije
        ++iteracija;
    }
    return najboljeRjesenje; // Vrati najbolje pronadeno rjesenje
}

int main() {
    std::vector<std::vector<int>> udaljenosti = {
        { 0, 10, 13, 20, 19 },
        { 10,  0, 23, 22, 19 },
        { 13, 23,  0, 24, 25 },
        { 20, 22, 24,  0,  5 },
        { 19, 19, 25,  5,  0 }
    };

    std::vector<int> pocetniPut = { 0, 1, 2, 4, 3 };
    Jedinka pocetnoRjesenje(pocetniPut);
    Jedinka rezultat = TabuPretraga(pocetnoRjesenje, udaljenosti);

    std::cout << "\nNajbolje pronadeno rjesenje: ";
    ispisiPut(rezultat);
    return 0;
}
```

9.27 Tabu pretraživanje s pamćenjem poteza (problem trgovačkog putnika)

Referenca: Poglavlje 6.5

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <limits>

class Jedinka {
public:
    std::vector<int> put;
    int dobrota = INT_MAX;

    Jedinka(const std::vector<int>& _put = {});
    void evaluiraj(const std::vector<std::vector<int>>& matricaUdaljenosti);
    void zamijeniGradove(int i, int j);
};

Jedinka::Jedinka(const std::vector<int>& _put) {
    if (_put.empty())
        put = { 0, 1, 2, 3 }; // default ruta
    else
        put = _put;
}

void Jedinka::zamijeniGradove(int i, int j) {
    if (i >= 0 && j >= 0 && i < put.size() && j < put.size())
        std::swap(put[i], put[j]);
}

void Jedinka::evaluiraj(const std::vector<std::vector<int>>& matricaUdaljenosti) {
    int udaljenost = 0;
    for (size_t i = 0; i < put.size() - 1; ++i)
        udaljenost += matricaUdaljenosti[put[i]][put[i + 1]];
    udaljenost += matricaUdaljenosti[put.back()][put.front()];
    dobrota = udaljenost;
}

void ispisiPut(const Jedinka& jedinka) {
    for (int grad : jedinka.put)
        std::cout << grad << " -> ";
    std::cout << jedinka.put.front();
    std::cout << "\tDuljina puta: " << jedinka.dobrota << std::endl;
}

Jedinka TabuPretraga(Jedinka rjesenje, const std::vector<std::vector<int>>& matrica, int
    tabuDuljina = 5, int maksIteracija = 100) {
    rjesenje.evaluiraj(matrica);
    Jedinka najboljeRjesenje = rjesenje;

    // Umjesto pamcenja cijelog rjesenja, pamte se samo POTEZI (zamijenjeni indeksi)
    std::list<std::pair<int, int>> tabuLista;

    int iteracija = 0;
    while (iteracija < maksIteracija) {
        Jedinka najboljiKandidat;
        najboljiKandidat.dobrota = INT_MAX;
        int najboljiI = -1, najboljiJ = -1; // Sprema se najbolji potez

        // Lokalna pretraga: pokusaj svih mogucih zamjena gradova
        for (size_t i = 0; i < rjesenje.put.size(); ++i) {
            for (size_t j = i + 1; j < rjesenje.put.size(); ++j) {
                Jedinka susjed = rjesenje;
                susjed.zamijeniGradove(i, j);
```

```
                susjed.evaluiraj(matrica);

                // Provjera je li POTEZ (i,j) u tabu listi
                bool jeTabu = std::find(tabuLista.begin(), tabuLista.end(), std::make_pair(
    i, j)) != tabuLista.end() ||
                    std::find(tabuLista.begin(), tabuLista.end(), std::make_pair(j, i)) !=
    tabuLista.end();

                // ASPIRACIJSKI KRITERIJ: dozvoljava tabu potez
                // ako vodi do boljeg globalnog rjesenja
                if (!jeTabu || susjed.dobrota < najboljeRjesenje.dobrota) {
                    if (susjed.dobrota < najboljiKandidat.dobrota) {
                        najboljiKandidat = susjed;
                        najboljiI = i;
                        najboljiJ = j;
                    }
                }
            }
        }
        // Azuriraj trenutno rjesenje
        rjesenje = najboljiKandidat;

        // Azuriraj globalno najbolje rjesenje
        if (rjesenje.dobrota < najboljeRjesenje.dobrota) {
            najboljeRjesenje = rjesenje;
            std::cout << "Iteracija " << iteracija + 1 << " -> novo najbolje rjesenje: ";
            ispisiPut(rjesenje);
        }
        // Dodaj odabrani potez u tabu listu
        if (najboljiI != -1 && najboljiJ != -1)
            tabuLista.push_back({ najboljiI, najboljiJ });

        // Odrzavaj duljinu tabu liste (FIFO)
        if (tabuLista.size() > static_cast<size_t>(tabuDuljina))
            tabuLista.pop_front();
        ++iteracija;
    }
    return najboljeRjesenje;
}

int main() {
    std::vector<std::vector<int>> udaljenosti = {
        { 0, 10, 13, 20, 19 },
        { 10,  0, 23, 22, 19 },
        { 13, 23,  0, 24, 25 },
        { 20, 22, 24,  0,  5 },
        { 19, 19, 25,  5,  0 }
    };

    std::vector<int> pocetniPut = { 0, 1, 2, 4, 3 };
    Jedinka pocetnoRjesenje(pocetniPut);
    Jedinka rezultat = TabuPretraga(pocetnoRjesenje, udaljenosti);

    std::cout << "\nNajbolje pronadeno rjesenje: ";
    ispisiPut(rezultat);
    return 0;
}
```

9.28 Genetski algoritam (binarna reprezentacija broja)

Referenca: Poglavlje 6.6

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
```

```
// Klasa koja predstavlja jedinku (kromosom) u populaciji
class Jedinka {
public:
    std::vector<bool> geni = std::vector<bool>(32); // 32-bitni genetski kod
    unsigned int dobrota{ std::numeric_limits<int>::max() }; // evaluacijska vrijednost
    void evaluiraj(unsigned int ciljanaVrijednost); // izracun dobrote
    unsigned int uBroj() const; // konverzija gena u broj
};

void Jedinka::evaluiraj(unsigned int ciljanaVrijednost) {
    unsigned int broj = uBroj();
    dobrota = (broj < ciljanaVrijednost) ? ciljanaVrijednost - broj : broj -
    ciljanaVrijednost;
}

unsigned int Jedinka::uBroj() const {
    unsigned int suma = 0;
    for (unsigned int i = 0; i < geni.size(); i++)
        suma += geni[i] * (1U << (geni.size() - i - 1));
    return suma;
}

// Klasa koja predstavlja populaciju jedinki
class Populacija {
public:
    std::vector<Jedinka> jedinke;
    void generirajSlucajnu(int velicina, std::mt19937& generator); // inicijalna populacija
    void evaluiraj(unsigned int ciljanaVrijednost); // evaluacija svih jedinki
    void dohvatiNajboljuDobrotu(unsigned int* najbolja, unsigned int* najblizaVrijednost);
    // dohvat najboljeg
    void primijeniElitizam(Populacija* prethodna, int postotakElite); // prijenos elite iz
    prethodne populacije
};

void Populacija::generirajSlucajnu(int velicina, std::mt19937& generator) {
    std::uniform_int_distribution<> distrib(0, 1);
    jedinke.clear();
    for (int i = 0; i < velicina; i++) {
        Jedinka j;
        for (unsigned int g = 0; g < j.geni.size(); g++)
            j.geni[g] = distrib(generator); // nasumicno generiranje gena (0 ili 1)
        jedinke.push_back(j);
    }
}

void Populacija::evaluiraj(unsigned int ciljanaVrijednost) {
    for (auto& j : jedinke)
        j.evaluiraj(ciljanaVrijednost); // evaluacija svakog clana
}

void Populacija::dohvatiNajboljuDobrotu(unsigned int* najbolja, unsigned int*
    najblizaVrijednost) {
    *najbolja = jedinke[0].dobrota;
    *najblizaVrijednost = jedinke[0].uBroj();
    for (unsigned int i = 1; i < jedinke.size(); i++) {
        if (jedinke[i].dobrota < *najbolja) {
            *najbolja = jedinke[i].dobrota;
            *najblizaVrijednost = jedinke[i].uBroj();
        }
    }
}

void Populacija::primijeniElitizam(Populacija* prethodna, int postotakElite) {
    // Sortiraj trenutnu i prethodnu populaciju po dobroti
    sort(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
    sort(prethodna->jedinke.begin(), prethodna->jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
```

```
    // Prebaci postotak elite iz prethodne generacije u trenutnu (elitizam)
    int brojElite = static_cast<int>(ceil((postotakElite * jedinke.size()) / 100.0));
    for (int i = 0; i < brojElite; i++)
        jedinke[jedinke.size() - i - 1] = prethodna->jedinke[i];

    // Ponovno sortiranje nakon zamjene
    sort(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

// Funkcija za formatirani ispis broja u binarnom obliku
std::string brojUBinarni(int n, int minBitova) {
    std::string rezultat;
    for (int i = minBitova - 1; i >= 0; --i) {
        if ((minBitova - i - 1) % 4 == 0) rezultat += '.';
        rezultat += ((n >> i) & 1) ? '1' : '0';
    }
    return rezultat.substr(1); // ukloni pocetnu tocku
}

// Glavna klasa koja upravlja izvrsavanjem genetskog algoritma
class GenetskiAlgoritam {
public:
    unsigned int ciljanaVrijednost;
    int velicinaPopulacije = 250;
    int velicinaTurnira = 2;
    int postotakKrizanja = 70;
    int postotakMutacije = 10;
    int postotakElite = 10;
    unsigned int maksimalanBrojGeneracija = 2500;

    GenetskiAlgoritam(unsigned int cilj) : ciljanaVrijednost(cilj) {}
    Populacija turnirskaSelekcija(const Populacija& P0, std::mt19937& generator);
    Populacija krizanje(const Populacija& P1, std::mt19937& generator);
    void mutacija(Populacija* P, std::mt19937& generator);
    void pokreni(std::mt19937& generator); // pokretanje cijelog algoritma
};

// Turnirska selekcija: biranje najboljeg iz podskupine
Populacija GenetskiAlgoritam::turnirskaSelekcija(const Populacija& P0, std::mt19937&
    generator) {
    Populacija P1;
    std::uniform_int_distribution<> distrib(0, P0.jedinke.size() - 1);

    for (unsigned int i = 0; i < P0.jedinke.size(); i++) {
        std::vector<int> natjecatelji;
        for (int j = 0; j < velicinaTurnira; j++)
            natjecatelji.push_back(distrib(generator));

        int pobjednik = natjecatelji[0];
        for (int j = 1; j < natjecatelji.size(); j++) {
            if (P0.jedinke[natjecatelji[j]].dobrota < P0.jedinke[pobjednik].dobrota)
                pobjednik = natjecatelji[j];
        }
        P1.jedinke.push_back(P0.jedinke[pobjednik]);
    }
    return P1;
}

// Uniformno krizanje: kombiniranje gena nasumicno
Populacija GenetskiAlgoritam::krizanje(const Populacija& P1, std::mt19937& generator) {
    Populacija P2 = P1;
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::uniform_int_distribution<> distribBit(0, 1);
    std::vector<int> kandidati;

    for (int i = 0; i < P1.jedinke.size(); ++i)
        if (distribVjerojatnost(generator) <= postotakKrizanja)
            kandidati.push_back(i);
```

```
    if (kandidati.size() < 2) return P2;
    if (kandidati.size() % 2 != 0) kandidati.pop_back();
    shuffle(kandidati.begin(), kandidati.end(), generator);  // nasumicno mijesanje

    for (size_t i = 0; i < kandidati.size(); i += 2) {
        int i1 = kandidati[i], i2 = kandidati[i + 1];
        const Jedinka& r1 = P1.jedinke[i1], & r2 = P1.jedinke[i2];
        Jedinka p1, p2;

        for (size_t g = 0; g < r1.geni.size(); ++g) {
            int b1 = distribBit(generator);
            int b2 = distribBit(generator);
            p1.geni[g] = (b1 == 0) ? r1.geni[g] : r2.geni[g];
            p2.geni[g] = (b2 == 0) ? r1.geni[g] : r2.geni[g];
        }
        P2.jedinke[i1] = p1;
        P2.jedinke[i2] = p2;
    }
    return P2;
}

// Mutacija: promjena pojedinih gena s odredenom vjerojatnoscu
void GenetskiAlgoritam::mutacija(Populacija* P, std::mt19937& generator) {
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::uniform_int_distribution<> distribBit(0, 1);

    for (auto& jed : P->jedinke) {
        for (size_t i = 0; i < jed.geni.size(); ++i) {
            if (distribVjerojatnost(generator) <= postotakMutacije)
                jed.geni[i] = distribBit(generator);
        }
    }
}

// Glavna metoda za pokretanje genetskog algoritma
void GenetskiAlgoritam::pokreni(std::mt19937& generator) {
    std::vector<Populacija> generacije;
    Populacija trenutna;
    trenutna.generirajSlucajnu(velicinaPopulacije, generator);
    trenutna.evaluiraj(ciljanaVrijednost);

    unsigned int najboljaDobrota, trenutnaDobrota, najblizaVrijednost;
    trenutna.dohvatiNajboljuDobrotu(&trenutnaDobrota, &najblizaVrijednost);
    najboljaDobrota = trenutnaDobrota;
    generacije.push_back(trenutna);

    std::cout << "Ulazna vrijednost: " << ciljanaVrijednost << "\t\t(" << brojUBinarni(
    ciljanaVrijednost, 32) << ")\n";
    std::cout << "Generacija 1:\tdobrota = " << trenutnaDobrota << ",\tvrijednost: " <<
    najblizaVrijednost << "\t(" << brojUBinarni(najblizaVrijednost, 32) << ")\n";

    do {
        Populacija P1 = turnirskaSelekcija(generacije.back(), generator);
        Populacija P2 = krizanje(P1, generator);
        mutacija(&P2, generator);
        P2.primijeniElitizam(&generacije.back(), postotakElite);
        P2.evaluiraj(ciljanaVrijednost);
        generacije.push_back(P2);

        generacije.back().dohvatiNajboljuDobrotu(&trenutnaDobrota, &najblizaVrijednost);
        if (trenutnaDobrota < najboljaDobrota) {
            najboljaDobrota = trenutnaDobrota;
            std::cout << "Generacija " << generacije.size() << ":\tdobrota = " <<
    najboljaDobrota << ",\tvrijednost: " << najblizaVrijednost << "\t(" << brojUBinarni(
    najblizaVrijednost, 32) << ")\n";
        }
        if (najboljaDobrota == 0) {
            std::cout << "Rjesenje je pronadeno u generaciji: " << generacije.size() << "\n
    ";
            break;
```

```
        }
    } while (najboljaDobrota != 0 && generacije.size() <= maksimalanBrojGeneracija);
}

int main() {
    std::random_device rd;
    int sjeme = rd();
    std::mt19937 generator(sjeme); // ili npr. 1687280607
    GenetskiAlgoritam ga(1234567890); // trazena vrijednost

    std::cout << "Sjeme: " << sjeme << "\n";
    ga.pokreni(generator);
    return 0;
}
```

9.29 Memetski algoritam (binarna reprezentacija broja), pohlepna zamjena

Referenca: Poglavlje 6.6.1

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>

// Klasa koja predstavlja jedinku (kromosom) u populaciji
class Jedinka {
public:
    std::vector<bool> geni = std::vector<bool>(32); // 32-bitni genetski kod
    unsigned int dobrota{ std::numeric_limits<int>::max() }; // evaluacijska vrijednost (
    sto niza, to bolja)
    void evaluiraj(unsigned int ciljanaVrijednost); // izracun dobrote jedinke u odnosu na
    ciljnu vrijednost
    unsigned int uBroj() const; // konverzija gena u broj (dekodiranje binarnog zapisa)
};

void Jedinka::evaluiraj(unsigned int ciljanaVrijednost) {
    unsigned int broj = uBroj();
    dobrota = (broj < ciljanaVrijednost) ? ciljanaVrijednost - broj : broj -
    ciljanaVrijednost;
}

unsigned int Jedinka::uBroj() const {
    unsigned int suma = 0;
    for (unsigned int i = 0; i < geni.size(); i++)
        suma += geni[i] * (1U << (geni.size() - i - 1));
    return suma;
}

// Klasa koja predstavlja populaciju jedinki
class Populacija {
public:
    std::vector<Jedinka> jedinke;
    void generirajSlucajnu(int velicina, std::mt19937& generator); // nasumicno
    inicijaliziranje populacije
    void evaluiraj(unsigned int ciljanaVrijednost); // evaluacija svih jedinki u populaciji
    void dohvatiNajboljuDobrotu(unsigned int* najbolja, unsigned int* najblizaVrijednost);
    // dohvat najbolje jedinke
    void primijeniElitizam(Populacija* prethodna, int postotakElite); // prijenos elite iz
    prethodne generacije
};

void Populacija::generirajSlucajnu(int velicina, std::mt19937& generator) {
    std::uniform_int_distribution<> distrib(0, 1);
```

```
    jedinke.clear();
    for (int i = 0; i < velicina; i++) {
        Jedinka j;
        for (unsigned int g = 0; g < j.geni.size(); g++)
            j.geni[g] = distrib(generator); // generiranje svakog bita gena
        jedinke.push_back(j);
    }
}

void Populacija::evaluiraj(unsigned int ciljanaVrijednost) {
    for (auto& j : jedinke)
        j.evaluiraj(ciljanaVrijednost);
}

void Populacija::dohvatiNajboljuDobrotu(unsigned int* najbolja, unsigned int*
    najblizaVrijednost) {
    *najbolja = jedinke[0].dobrota;
    *najblizaVrijednost = jedinke[0].uBroj();
    for (unsigned int i = 1; i < jedinke.size(); i++) {
        if (jedinke[i].dobrota < *najbolja) {
            *najbolja = jedinke[i].dobrota;
            *najblizaVrijednost = jedinke[i].uBroj();
        }
    }
}

void Populacija::primijeniElitizam(Populacija* prethodna, int postotakElite) {
    std::sort(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
    std::sort(prethodna->jedinke.begin(), prethodna->jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });

    int brojElite = static_cast<int>(ceil((postotakElite * jedinke.size()) / 100.0));
    for (int i = 0; i < brojElite; i++)
        jedinke[jedinke.size() - i - 1] = prethodna->jedinke[i];

    std::sort(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

std::string brojUBinarni(int n, int minBitova) {
    std::string rezultat;
    for (int i = minBitova - 1; i >= 0; --i) {
        if ((minBitova - i - 1) % 4 == 0) rezultat += '.';
        rezultat += ((n >> i) & 1) ? '1' : '0';
    }
    return rezultat.substr(1);
}

class MemetskiAlgoritam {
public:
    unsigned int ciljanaVrijednost;
    int velicinaPopulacije = 250;
    int velicinaTurnira = 2;
    int postotakKrizanja = 70;
    int postotakMutacije = 10;
    int postotakElite = 10;
    int brojZaLokalnuPretragu = 10; // 10 najboljih jedinki
    unsigned int maksimalanBrojGeneracija = 2500;

    MemetskiAlgoritam(unsigned int cilj) : ciljanaVrijednost(cilj) {}
    Populacija turnirskaSelekcija(const Populacija& P0, std::mt19937& generator);
    Populacija krizanje(const Populacija& P1, std::mt19937& generator);
    void mutacija(Populacija* P, std::mt19937& generator);
    void lokalnaPretraga(Populacija* P);
    void pokreni(std::mt19937& generator);
};
```

```
Populacija MemetskiAlgoritam::turnirskaSelekcija(const Populacija& P0, std::mt19937&
    generator) {
    Populacija P1;
    std::uniform_int_distribution<> distrib(0, P0.jedinke.size() - 1);
    for (unsigned int i = 0; i < P0.jedinke.size(); i++) {
        std::vector<int> natjecatelji;
        for (int j = 0; j < velicinaTurnira; j++)
            natjecatelji.push_back(distrib(generator));
        int pobjednik = natjecatelji[0];
        for (int j = 1; j < natjecatelji.size(); j++) {
            if (P0.jedinke[natjecatelji[j]].dobrota < P0.jedinke[pobjednik].dobrota)
                pobjednik = natjecatelji[j];
        }
        P1.jedinke.push_back(P0.jedinke[pobjednik]);
    }
    return P1;
}

Populacija MemetskiAlgoritam::krizanje(const Populacija& P1, std::mt19937& generator) {
    Populacija P2 = P1;
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::uniform_int_distribution<> distribBit(0, 1);
    std::vector<int> kandidati;
    for (int i = 0; i < P1.jedinke.size(); ++i)
        if (distribVjerojatnost(generator) <= postotakKrizanja)
            kandidati.push_back(i);
    if (kandidati.size() < 2) return P2;
    if (kandidati.size() % 2 != 0) kandidati.pop_back();
    std::shuffle(kandidati.begin(), kandidati.end(), generator);
    for (size_t i = 0; i < kandidati.size(); i += 2) {
        int i1 = kandidati[i], i2 = kandidati[i + 1];
        const Jedinka& r1 = P1.jedinke[i1], & r2 = P1.jedinke[i2];
        Jedinka p1, p2;
        for (size_t g = 0; g < r1.geni.size(); ++g) {
            int b1 = distribBit(generator);
            int b2 = distribBit(generator);
            p1.geni[g] = (b1 == 0) ? r1.geni[g] : r2.geni[g];
            p2.geni[g] = (b2 == 0) ? r1.geni[g] : r2.geni[g];
        }
        P2.jedinke[i1] = p1;
        P2.jedinke[i2] = p2;
    }
    return P2;
}

void MemetskiAlgoritam::mutacija(Populacija* P, std::mt19937& generator) {
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::uniform_int_distribution<> distribBit(0, 1);
    for (auto& jed : P->jedinke) {
        for (size_t i = 0; i < jed.geni.size(); ++i) {
            if (distribVjerojatnost(generator) <= postotakMutacije)
                jed.geni[i] = distribBit(generator);
        }
    }
}

// Lokalna pretraga se primjenjuje nad predefiniranim brojem najboljih jedinki
void MemetskiAlgoritam::lokalnaPretraga(Populacija* P) {
    // Sortiranje populacije po dobroti - najbolje jedinke dolaze na pocetak
    std::sort(P->jedinke.begin(), P->jedinke.end(), [](const Jedinka& a, const Jedinka& b)
    {
        return a.dobrota < b.dobrota;
        });

    // Iteracija kroz elitne jedinke koje ce biti lokalno poboljsane
    for (int i = 0; i < brojZaLokalnuPretragu; ++i) {
        Jedinka najbolji = P->jedinke[i]; // Pocetna jedinka koju pokusavamo poboljsati

        // Generiranje svih susjeda promjenom svakog pojedinog gena
        for (size_t g = 0; g < najbolji.geni.size(); ++g) {
```

```
            Jedinka susjed = najbolji;
            susjed.geni[g] = !susjed.geni[g]; // Invertiraj g-ti gen
            susjed.evaluiraj(ciljanaVrijednost); // Evaluiraj dobrotu susjeda

            // Ako je susjed kvalitetniji (ima manju dobrotu), zadrzi ga kao novog
    najboljeg
            if (susjed.dobrota < najbolji.dobrota)
                najbolji = susjed;
        }
        // Ako je pronaden bolji susjed, zamijeni izvornu jedinku u populaciji
        P->jedinke[i] = najbolji;
    }
}

void MemetskiAlgoritam::pokreni(std::mt19937& generator) {
    std::vector<Populacija> generacije;
    Populacija trenutna;
    trenutna.generirajSlucajnu(velicinaPopulacije, generator);
    trenutna.evaluiraj(ciljanaVrijednost);
    unsigned int najboljaDobrota, trenutnaDobrota, najblizaVrijednost;
    trenutna.dohvatiNajboljuDobrotu(&trenutnaDobrota, &najblizaVrijednost);
    najboljaDobrota = trenutnaDobrota;
    generacije.push_back(trenutna);
    std::cout << "Ulazna vrijednost: " << ciljanaVrijednost << "\t\t(" << brojUBinarni(
    ciljanaVrijednost, 32) << ")\n";
    std::cout << "Generacija 1:\tdobrota = " << trenutnaDobrota << ",\tvrijednost: " <<
    najblizaVrijednost << "\t(" << brojUBinarni(najblizaVrijednost, 32) << ")\n";
    do {
        Populacija P1 = turnirskaSelekcija(generacije.back(), generator);
        Populacija P2 = krizanje(P1, generator);
        mutacija(&P2, generator);
        lokalnaPretraga(&P2); // !!!
        P2.primijeniElitizam(&generacije.back(), postotakElite);
        P2.evaluiraj(ciljanaVrijednost);
        generacije.push_back(P2);
        generacije.back().dohvatiNajboljuDobrotu(&trenutnaDobrota, &najblizaVrijednost);
        if (trenutnaDobrota < najboljaDobrota) {
            najboljaDobrota = trenutnaDobrota;
            std::cout << "Generacija " << generacije.size() << ":\tdobrota = " <<
    najboljaDobrota << ",\tvrijednost: " << najblizaVrijednost << "\t(" << brojUBinarni(
    najblizaVrijednost, 32) << ")\n";
        }
        if (najboljaDobrota == 0) {
            std::cout << "Rjesenje je pronadeno u generaciji: " << generacije.size() << "\n
    ";
            break;
        }
    } while (najboljaDobrota != 0 && generacije.size() <= maksimalanBrojGeneracija);
}

int main() {
    std::random_device rd;
    int sjeme = rd();
    std::mt19937 generator(rd()); // ili npr. 1687280607
    unsigned int ciljnaVrijednost = 1234567890;
    MemetskiAlgoritam algoritam(ciljnaVrijednost);
    std::cout << "Sjeme: " << sjeme << "\n";
    algoritam.pokreni(generator);
    return 0;
}
```

9.30 Memetski algoritam (binarna reprezentacija broja), višestruka zamjena

Referenca: Poglavlje 6.6.2

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>

// Klasa koja predstavlja jedinku (kromosom) u populaciji
class Jedinka {
public:
    std::vector<bool> geni = std::vector<bool>(32); // 32-bitni genetski kod
    unsigned int dobrota{ std::numeric_limits<int>::max() }; // evaluacijska vrijednost (
    sto niza, to bolja)
    void evaluiraj(unsigned int ciljanaVrijednost); // izracun dobrote jedinke u odnosu na
    ciljnu vrijednost
    unsigned int uBroj() const; // konverzija gena u broj (dekodiranje binarnog zapisa)
};

void Jedinka::evaluiraj(unsigned int ciljanaVrijednost) {
    unsigned int broj = uBroj();
    dobrota = (broj < ciljanaVrijednost) ? ciljanaVrijednost - broj : broj -
    ciljanaVrijednost;
}

unsigned int Jedinka::uBroj() const {
    unsigned int suma = 0;
    for (unsigned int i = 0; i < geni.size(); i++)
        suma += geni[i] * (1U << (geni.size() - i - 1));
    return suma;
}

// Klasa koja predstavlja populaciju jedinki
class Populacija {
public:
    std::vector<Jedinka> jedinke;
    void generirajSlucajnu(int velicina, std::mt19937& generator); // nasumicno
    inicijaliziranje populacije
    void evaluiraj(unsigned int ciljanaVrijednost); // evaluacija svih jedinki u populaciji
    void dohvatiNajboljuDobrotu(unsigned int* najbolja, unsigned int* najblizaVrijednost);
    // dohvat najbolje jedinke
    void primijeniElitizam(Populacija* prethodna, int postotakElite); // prijenos elite iz
    prethodne generacije
};

void Populacija::generirajSlucajnu(int velicina, std::mt19937& generator) {
    std::uniform_int_distribution<> distrib(0, 1);
    jedinke.clear();
    for (int i = 0; i < velicina; i++) {
        Jedinka j;
        for (unsigned int g = 0; g < j.geni.size(); g++)
            j.geni[g] = distrib(generator); // generiranje svakog bita gena
        jedinke.push_back(j);
    }
}

void Populacija::evaluiraj(unsigned int ciljanaVrijednost) {
    for (auto& j : jedinke)
        j.evaluiraj(ciljanaVrijednost);
}

void Populacija::dohvatiNajboljuDobrotu(unsigned int* najbolja, unsigned int*
    najblizaVrijednost) {
    *najbolja = jedinke[0].dobrota;
    *najblizaVrijednost = jedinke[0].uBroj();
```

```
    for (unsigned int i = 1; i < jedinke.size(); i++) {
        if (jedinke[i].dobrota < *najbolja) {
            *najbolja = jedinke[i].dobrota;
            *najblizaVrijednost = jedinke[i].uBroj();
        }
    }
}

void Populacija::primijeniElitizam(Populacija* prethodna, int postotakElite) {
    std::sort(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
    std::sort(prethodna->jedinke.begin(), prethodna->jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });

    int brojElite = static_cast<int>(ceil((postotakElite * jedinke.size()) / 100.0));
    for (int i = 0; i < brojElite; i++)
        jedinke[jedinke.size() - i - 1] = prethodna->jedinke[i];

    std::sort(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

std::string brojUBinarni(int n, int minBitova) {
    std::string rezultat;
    for (int i = minBitova - 1; i >= 0; --i) {
        if ((minBitova - i - 1) % 4 == 0) rezultat += '.';
        rezultat += ((n >> i) & 1) ? '1' : '0';
    }
    return rezultat.substr(1);
}

class MemetskiAlgoritam {
public:
    unsigned int ciljanaVrijednost;
    int velicinaPopulacije = 250;
    int velicinaTurnira = 2;
    int postotakKrizanja = 70;
    int postotakMutacije = 10;
    int postotakElite = 10;
    int brojZaLokalnuPretragu = 10; // 10 najboljih jedinki
    unsigned int maksimalanBrojGeneracija = 2500;

    MemetskiAlgoritam(unsigned int cilj) : ciljanaVrijednost(cilj) {}
    Populacija turnirskaSelekcija(const Populacija& P0, std::mt19937& generator);
    Populacija krizanje(const Populacija& P1, std::mt19937& generator);
    void mutacija(Populacija* P, std::mt19937& generator);
    void lokalnaPretraga(Populacija* P, std::mt19937& generator);
    void pokreni(std::mt19937& generator);
};


Populacija MemetskiAlgoritam::turnirskaSelekcija(const Populacija& P0, std::mt19937&
    generator) {
    Populacija P1;
    std::uniform_int_distribution<> distrib(0, P0.jedinke.size() - 1);
    for (unsigned int i = 0; i < P0.jedinke.size(); i++) {
        std::vector<int> natjecatelji;
        for (int j = 0; j < velicinaTurnira; j++)
            natjecatelji.push_back(distrib(generator));
        int pobjednik = natjecatelji[0];
        for (int j = 1; j < natjecatelji.size(); j++) {
            if (P0.jedinke[natjecatelji[j]].dobrota < P0.jedinke[pobjednik].dobrota)
                pobjednik = natjecatelji[j];
        }
        P1.jedinke.push_back(P0.jedinke[pobjednik]);
    }
    return P1;
}

Populacija MemetskiAlgoritam::krizanje(const Populacija& P1, std::mt19937& generator) {
```

```
    Populacija P2 = P1;
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::uniform_int_distribution<> distribBit(0, 1);
    std::vector<int> kandidati;
    for (int i = 0; i < P1.jedinke.size(); ++i)
        if (distribVjerojatnost(generator) <= postotakKrizanja)
            kandidati.push_back(i);
    if (kandidati.size() < 2) return P2;
    if (kandidati.size() % 2 != 0) kandidati.pop_back();
    std::shuffle(kandidati.begin(), kandidati.end(), generator);
    for (size_t i = 0; i < kandidati.size(); i += 2) {
        int i1 = kandidati[i], i2 = kandidati[i + 1];
        const Jedinka& r1 = P1.jedinke[i1], & r2 = P1.jedinke[i2];
        Jedinka p1, p2;
        for (size_t g = 0; g < r1.geni.size(); ++g) {
            int b1 = distribBit(generator);
            int b2 = distribBit(generator);
            p1.geni[g] = (b1 == 0) ? r1.geni[g] : r2.geni[g];
            p2.geni[g] = (b2 == 0) ? r1.geni[g] : r2.geni[g];
        }
        P2.jedinke[i1] = p1;
        P2.jedinke[i2] = p2;
    }
    return P2;
}

void MemetskiAlgoritam::mutacija(Populacija* P, std::mt19937& generator) {
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::uniform_int_distribution<> distribBit(0, 1);
    for (auto& jed : P->jedinke) {
        for (size_t i = 0; i < jed.geni.size(); ++i) {
            if (distribVjerojatnost(generator) <= postotakMutacije)
                jed.geni[i] = distribBit(generator);
        }
    }
}

// Lokalna pretraga se primjenjuje nad predefiniranim brojem najboljih jedinki
void MemetskiAlgoritam::lokalnaPretraga(Populacija* P, std::mt19937& generator) {
    // Sortiranje populacije po dobroti (fitnessu) - najbolje jedinke dolaze na pocetak
    std::sort(P->jedinke.begin(), P->jedinke.end(), [](const Jedinka& a, const Jedinka& b)
    {
        return a.dobrota < b.dobrota;
    });
    std::vector<Jedinka> susjedi;
    std::uniform_int_distribution<> distribGen(0, 31);
    std::uniform_real_distribution<> distribVjerojatnost(0.0, 1.0);

    // Generiraj susjede za najbolji dio populacije
    for (int i = 0; i < brojZaLokalnuPretragu; ++i) {
        Jedinka original = P->jedinke[i];
        Jedinka kandidat = original;
        for (size_t g = 0; g < kandidat.geni.size(); ++g) {
            if (distribVjerojatnost(generator) <= 0.5)
                kandidat.geni[g] = !kandidat.geni[g];
        }
        kandidat.evaluiraj(ciljanaVrijednost);
        susjedi.push_back(kandidat);
    }

    // Sortiraj generirane susjede po dobroti
    std::sort(susjedi.begin(), susjedi.end(), [](const Jedinka& a, const Jedinka& b) {
        return a.dobrota < b.dobrota;
    });

    // Zamijeni najlosije jedinke u populaciji generiranim susjedima
    for (int i = 0; i < susjedi.size(); ++i)
        P->jedinke[P->jedinke.size() - 1 - i] = susjedi[i];
}
```

```
void MemetskiAlgoritam::pokreni(std::mt19937& generator) {
    std::vector<Populacija> generacije;
    Populacija trenutna;
    trenutna.generirajSlucajnu(velicinaPopulacije, generator);
    trenutna.evaluiraj(ciljanaVrijednost);
    unsigned int najboljaDobrota, trenutnaDobrota, najblizaVrijednost;
    trenutna.dohvatiNajboljuDobrotu(&trenutnaDobrota, &najblizaVrijednost);
    najboljaDobrota = trenutnaDobrota;
    generacije.push_back(trenutna);
    std::cout << "Ulazna vrijednost: " << ciljanaVrijednost << "\t\t(" << brojUBinarni(
    ciljanaVrijednost, 32) << ")\n";
    std::cout << "Generacija 1:\tdobrota = " << trenutnaDobrota << ",\tvrijednost: " <<
    najblizaVrijednost << "\t(" << brojUBinarni(najblizaVrijednost, 32) << ")\n";
    do {
        Populacija P1 = turnirskaSelekcija(generacije.back(), generator);
        Populacija P2 = krizanje(P1, generator);
        mutacija(&P2, generator);
        lokalnaPretraga(&P2, generator); // !!!
        P2.primijeniElitizam(&generacije.back(), postotakElite);
        P2.evaluiraj(ciljanaVrijednost);
        generacije.push_back(P2);
        generacije.back().dohvatiNajboljuDobrotu(&trenutnaDobrota, &najblizaVrijednost);
        if (trenutnaDobrota < najboljaDobrota) {
            najboljaDobrota = trenutnaDobrota;
            std::cout << "Generacija " << generacije.size() << ":\tdobrota = " <<
    najboljaDobrota << ",\tvrijednost: " << najblizaVrijednost << "\t(" << brojUBinarni(
    najblizaVrijednost, 32) << ")\n";
        }
        if (najboljaDobrota == 0) {
            std::cout << "Rjesenje je pronadeno u generaciji: " << generacije.size() << "\n
    ";
            break;
        }
    } while (najboljaDobrota != 0 && generacije.size() <= maksimalanBrojGeneracija);
}

int main() {
    std::random_device rd;
    int sjeme = rd();
    std::mt19937 generator(sjeme); // ili npr. 1687280607
    unsigned int ciljnaVrijednost = 1234567890;
    MemetskiAlgoritam algoritam(ciljnaVrijednost);
    algoritam.brojZaLokalnuPretragu = 20;
    std::cout << "Sjeme: " << sjeme << "\n";
    algoritam.pokreni(generator);
    return 0;
}
```

9.31 Genetski algoritam (minimizacija funkcije $|x-5|+\sin(3x)$, $x \in [0,10]$)

Referenca: Poglavlje 7.1

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <random>

// Jedinka s realnim genom
class Jedinka {
public:
    double x; // stvarna vrijednost, u intervalu [0, 10]
    double dobrota; // f(x)
```

```
    void evaluiraj() {
        dobrota = std::abs(x - 5.0) + std::sin(3 * x); // |x-5| + sin(3x)
    }
};

// Populacija jedinki
class Populacija {
public:
    std::vector<Jedinka> jedinke;

    void generirajSlucajnu(int velicina, std::mt19937& generator, double minX, double maxX)
    ;
    void evaluiraj();
    void dohvatiNajboljuJedinku(Jedinka& najbolja);
};

void Populacija::generirajSlucajnu(int velicina, std::mt19937& generator, double minX,
    double maxX) {
    std::uniform_real_distribution<> distrib(minX, maxX);
    jedinke.clear();
    for (int i = 0; i < velicina; i++) {
        Jedinka j;
        j.x = distrib(generator);
        jedinke.push_back(j);
    }
}

void Populacija::evaluiraj() {
    for (auto& j : jedinke)
        j.evaluiraj();
}

void Populacija::dohvatiNajboljuJedinku(Jedinka& najbolja) {
    najbolja = *std::min_element(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

// Genetski algoritam
class GenetskiAlgoritam {
public:
    int velicinaPopulacije = 100;
    int velicinaTurnira = 3;
    int postotakKrizanja = 80;
    int postotakMutacije = 20;
    unsigned int maksimalanBrojGeneracija = 200;
    double minX = 0.0;
    double maxX = 10.0;

    Populacija turnirskaSelekcija(const Populacija& P0, std::mt19937& generator);
    Populacija aritmetickoKrizanje(const Populacija& P1, std::mt19937& generator);
    void mutacija(Populacija* P, std::mt19937& generator);
    void pokreni(std::mt19937& generator);
};

Populacija GenetskiAlgoritam::turnirskaSelekcija(const Populacija& P0, std::mt19937&
    generator) {
    Populacija P1;
    std::uniform_int_distribution<> distrib(0, P0.jedinke.size() - 1);

    for (int i = 0; i < velicinaPopulacije; i++) {
        Jedinka najbolja = P0.jedinke[distrib(generator)];
        for (int j = 1; j < velicinaTurnira; j++) {
            Jedinka kandidat = P0.jedinke[distrib(generator)];
            if (kandidat.dobrota < najbolja.dobrota)
                najbolja = kandidat;
        }
        P1.jedinke.push_back(najbolja);
    }
    return P1;
}
```

```
Populacija GenetskiAlgoritam::aritmetickoKrizanje(const Populacija& P1, std::mt19937&
    generator) {
    Populacija P2 = P1;
    std::uniform_real_distribution<> alpha(0.0, 1.0);
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);

    for (size_t i = 0; i + 1 < P1.jedinke.size(); i += 2) {
        if (distribVjerojatnost(generator) <= postotakKrizanja) {
            double a = alpha(generator);
            double x1 = P1.jedinke[i].x;
            double x2 = P1.jedinke[i + 1].x;
            P2.jedinke[i].x = a * x1 + (1 - a) * x2;
            P2.jedinke[i + 1].x = (1 - a) * x1 + a * x2;
        }
    }
    return P2;
}

void GenetskiAlgoritam::mutacija(Populacija* P, std::mt19937& generator) {
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    std::normal_distribution<> distribMutacija(0.0, 0.1); // Gauss

    for (auto& jed : P->jedinke) {
        if (distribVjerojatnost(generator) <= postotakMutacije) {
            jed.x += distribMutacija(generator);
            // Ogranici na [minX, maxX]
            if (jed.x < minX) jed.x = minX;
            if (jed.x > maxX) jed.x = maxX;
        }
    }
}

void GenetskiAlgoritam::pokreni(std::mt19937& generator) {
    Populacija trenutna;
    trenutna.generirajSlucajnu(velicinaPopulacije, generator, minX, maxX);
    trenutna.evaluiraj();

    Jedinka najbolja;
    trenutna.dohvatiNajboljuJedinku(najbolja);
    std::cout << "Generacija 1: f(x)=" << najbolja.dobrota << ", x=" << najbolja.x << "\n";

    for (unsigned int gen = 2; gen <= maksimalanBrojGeneracija; ++gen) {
        Populacija P1 = turnirskaSelekcija(trenutna, generator);
        Populacija P2 = aritmetickoKrizanje(P1, generator);
        mutacija(&P2, generator);
        P2.evaluiraj();
        trenutna = P2;

        Jedinka trenutniNajbolji;
        trenutna.dohvatiNajboljuJedinku(trenutniNajbolji);
        if (trenutniNajbolji.dobrota < najbolja.dobrota) {
            najbolja = trenutniNajbolji;
            std::cout << "Generacija " << gen
                << ": f(x)=" << std::fixed
                << std::setprecision(std::numeric_limits<double>::digits10 + 1)
                << najbolja.dobrota
                << ", x=" << najbolja.x << "\n";
        }
    }
}

int main() {
    std::random_device rd;
    std::mt19937 generator(rd());
    GenetskiAlgoritam ga;
    ga.pokreni(generator);
    return 0;
}
```

9.32 Genetski algoritam (problem osam kraljica)

Referenca: Poglavlje 7.2

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>
#include <iomanip>

// Jedinka: predstavlja jedno rjesenje - permutaciju stupaca za svaki redak
class Jedinka {
public:
    std::vector<int> pozicije; // pozicije[i] = stupac kraljice u retku i
    int dobrota;               // broj konflikata po dijagonalama (sto manje, to bolje)

    Jedinka() : pozicije(8), dobrota(0) {}
    // Funkcija evaluacije jedinke - broji parove kraljica koje se napadaju po dijagonalama
    void evaluiraj();
};

void Jedinka::evaluiraj() {
    dobrota = 0;
    for (int i = 0; i < 8; ++i)
        for (int j = i + 1; j < 8; ++j)
            // Kraljice se napadaju ako su na istoj dijagonali:
            if (std::abs(i - j) == std::abs(pozicije[i] - pozicije[j]))
                ++dobrota; // broj konflikata se povecava
}

// Populacija: skup jedinki
class Populacija {
public:
    std::vector<Jedinka> jedinke;
    void generirajSlucajnu(int velicina, std::mt19937& generator);
    void evaluiraj();
    void dohvatiNajboljuJedinku(Jedinka& najbolja);
};

void Populacija::generirajSlucajnu(int velicina, std::mt19937& generator) {
    jedinke.clear();
    for (int i = 0; i < velicina; i++) {
        Jedinka j;
        for (int k = 0; k < 8; ++k) j.pozicije[k] = k;
        std::shuffle(j.pozicije.begin(), j.pozicije.end(), generator); // nasumicna
    permutacija
        jedinke.push_back(j);
    }
}

void Populacija::evaluiraj() {
    for (auto& j : jedinke)
        j.evaluiraj();
}

void Populacija::dohvatiNajboljuJedinku(Jedinka& najbolja) {
    // Trazi jedinku s najmanjim brojem konflikata (najbolju)
    najbolja = *std::min_element(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

// Genetski algoritam
class GenetskiAlgoritam {
public:
    int velicinaPopulacije = 100;       // broj jedinki po generaciji
    int velicinaTurnira = 3;            // broj kandidata u turnirskoj selekciji
    int postotakKrizanja = 80;          // vjerojatnost krizanja (u %)
    int postotakMutacije = 20;          // vjerojatnost mutacije (u %)
```

```
    unsigned int maksimalanBrojGeneracija = 500;

    Populacija turnirskaSelekcija(const Populacija& P0, std::mt19937& generator);
    Populacija krizanje(const Populacija& P1, std::mt19937& generator);
    void mutacija(Populacija* P, std::mt19937& generator);
    void pokreni(std::mt19937& generator);
};

Populacija GenetskiAlgoritam::turnirskaSelekcija(const Populacija& P0, std::mt19937&
    generator) {
    Populacija P1;
    std::uniform_int_distribution<> distrib(0, P0.jedinke.size() - 1);
    for (int i = 0; i < velicinaPopulacije; i++) {
        Jedinka najbolja = P0.jedinke[distrib(generator)];
        for (int j = 1; j < velicinaTurnira; j++) {
            Jedinka kandidat = P0.jedinke[distrib(generator)];
            if (kandidat.dobrota < najbolja.dobrota)
                najbolja = kandidat;
        }
        P1.jedinke.push_back(najbolja); // dodaj najboljeg u novu populaciju
    }
    return P1;
}

// PMX (Partially Mapped Crossover): krizanje pogodno za permutacije
Populacija GenetskiAlgoritam::krizanje(const Populacija& P1, std::mt19937& generator) {
    Populacija P2 = P1; // kopiraj roditeljsku populaciju
    std::uniform_int_distribution<> distPozicija(0, 7);
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);

    for (size_t i = 0; i + 1 < P1.jedinke.size(); i += 2) {
        if (distribVjerojatnost(generator) <= postotakKrizanja) {
            int a = distPozicija(generator);
            int b = distPozicija(generator);
            if (a > b) std::swap(a, b); // osiguraj da je a < b

            // Kopiraj roditeljske permutacije
            std::vector<int> dijete1 = P1.jedinke[i].pozicije;
            std::vector<int> dijete2 = P1.jedinke[i + 1].pozicije;

            // Mape za rjesavanje konflikata (duplikata)
            std::vector<int> mapa1(8, -1), mapa2(8, -1);
            for (int k = a; k <= b; ++k) {
                std::swap(dijete1[k], dijete2[k]);
                mapa1[dijete1[k]] = dijete2[k];
                mapa2[dijete2[k]] = dijete1[k];
            }

            // Funkcija za popravak permutacije izvan segmenta
            auto popravi = [](std::vector<int>& dijete, const std::vector<int>& mapa, int a
    , int b) {
                for (int k = 0; k < 8; ++k) {
                    if (k >= a && k <= b) continue; // preskoci segment
                    int val = dijete[k];
                    while (std::find(dijete.begin() + a, dijete.begin() + b + 1, val) !=
    dijete.begin() + b + 1) {
                        val = mapa[val]; // rekurzivno popravi vrijednosti koje su u
    konfliktu
                    }
                    dijete[k] = val;
                }
                };

            // Popravi obje jedinke
            popravi(dijete1, mapa1, a, b);
            popravi(dijete2, mapa2, a, b);

            // Spremi rezultate u novu populaciju
            P2.jedinke[i].pozicije = dijete1;
            P2.jedinke[i + 1].pozicije = dijete2;
```

```
        }
    }
    return P2;
}

// Mutacija: zamjena dvaju elemenata u permutaciji
void GenetskiAlgoritam::mutacija(Populacija* P, std::mt19937& generator) {
    std::uniform_int_distribution<> distPozicija(0, 7);
    std::uniform_int_distribution<> distribVjerojatnost(1, 100);
    for (auto& jed : P->jedinke) {
        if (distribVjerojatnost(generator) <= postotakMutacije) {
            int i = distPozicija(generator);
            int j = distPozicija(generator);
            std::swap(jed.pozicije[i], jed.pozicije[j]); // permutacija
        }
    }
}

// Pokretanje GA petlje
void GenetskiAlgoritam::pokreni(std::mt19937& generator) {
    Populacija trenutna;
    trenutna.generirajSlucajnu(velicinaPopulacije, generator); // pocetna populacija
    trenutna.evaluiraj();

    Jedinka najbolja;
    trenutna.dohvatiNajboljuJedinku(najbolja); // najbolji iz generacije 1
    std::cout << "Generacija 1: konflikata = " << najbolja.dobrota << "\n";

    for (unsigned int gen = 2; gen <= maksimalanBrojGeneracija && najbolja.dobrota > 0; ++
    gen) {
        Populacija P1 = turnirskaSelekcija(trenutna, generator); // selekcija roditelja
        Populacija P2 = krizanje(P1, generator);                 // krizanje
        mutacija(&P2, generator);                                // mutacija
        P2.evaluiraj();                                          // evaluacija
        trenutna = P2;

        Jedinka trenutniNajbolji;
        trenutna.dohvatiNajboljuJedinku(trenutniNajbolji);
        if (trenutniNajbolji.dobrota < najbolja.dobrota) {
            najbolja = trenutniNajbolji;
            std::cout << "Generacija " << gen << ": konflikata = " << najbolja.dobrota << "
    \n";
        }
    }

    // Ispis konacnog rjesenja u stilu sahovske notacije
    std::cout << "Najbolje rjesenje: ";
    for (int i = 0; i < 8; ++i)
        std::cout << (char)('A' + i) << najbolja.pozicije[i] + 1
        << ((i == 7) ? "" : ",");
}

int main() {
    std::random_device rd;
    std::mt19937 generator(rd());
    GenetskiAlgoritam ga;
    ga.pokreni(generator);
    return 0;
}
```

9.33 Genetski algoritam (problem ruksaka)

Referenca: Poglavlje 7.3

```
#include <iostream>
#include <vector>
```

```
#include <random>
#include <algorithm>
#include <string>
#include <iomanip>

// Struktura koja opisuje jedan predmet
struct Predmet {
    std::string naziv;
    double vrijednost;  // izrazena u eurima
    double tezina;      // tezina u kg
};

// Jedinka predstavlja jedno rjesenje (binarni niz)
struct Jedinka {
    std::vector<int> geni;
    double dobrota;

    Jedinka(size_t broj_gena) : geni(broj_gena, 0), dobrota(0.0) {}
    void evaluiraj(const std::vector<Predmet>& predmeti, double kapacitet_ruksaka);
};

// Klasa genetskog algoritma
class GA {
public:
    int velPop = 50;
    double vjKrizanja = 0.7;
    double vjMutacije = 0.01;
    int brojGeneracija = 100;
    int brojElitnih = 1;
    double kapacitet_ruksaka; // u kg
    std::vector<Predmet> predmeti;
    std::mt19937 gen;

    GA(double kapacitet, const std::vector<Predmet>& predmeti)
        : kapacitet_ruksaka(kapacitet), predmeti(predmeti), gen(std::random_device{}()) {}
    void pokreni();
    std::vector<Jedinka> generirajPopulaciju();
    void evaluiraj(std::vector<Jedinka>& populacija);
    std::vector<Jedinka> selekcija(const std::vector<Jedinka>& populacija);
    std::vector<Jedinka> krizanje(const std::vector<Jedinka>& roditelji);
    void mutacija(std::vector<Jedinka>& populacija);
    void dodajElitneJedinke(std::vector<Jedinka>& izvor, std::vector<Jedinka>& odrediste,
    int brojElitnih);
    Jedinka dohvatiNajboljuJedinku(const std::vector<Jedinka>& populacija);
    void ispisiRjesenje(const Jedinka& najbolja);
};

void Jedinka::evaluiraj(const std::vector<Predmet>& predmeti, double kapacitet_ruksaka) {
    double ukupnaTezina = 0.0;
    double ukupnaVrijednost = 0.0;

    for (size_t i = 0; i < predmeti.size(); ++i) {
        if (geni[i]) { // Ako je i-ti predmet u ruksaku
            ukupnaTezina += predmeti[i].tezina;
            ukupnaVrijednost += predmeti[i].vrijednost;
        }
    }
    // Ukupna tezina ne smije preci kapacitet ruksaka!
    dobrota = (ukupnaTezina <= kapacitet_ruksaka) ? ukupnaVrijednost : 0.0;
}

std::vector<Jedinka> GA::generirajPopulaciju() {
    std::vector<Jedinka> P;
    std::uniform_int_distribution<> d(0, 1);
    for (int i = 0; i < velPop; ++i) {
        Jedinka jed(predmeti.size());
        for (int& bit : jed.geni)
            bit = d(gen);
        P.push_back(jed);
    }
```

```
    return P;
}

void GA::evaluiraj(std::vector<Jedinka>& populacija) {
    for (auto& j : populacija)
        j.evaluiraj(predmeti, kapacitet_ruksaka);
}

// Stohasticka univerzalna selekcija (SUS)
std::vector<Jedinka> GA::selekcija(const std::vector<Jedinka>& populacija) {
    std::vector<Jedinka> nova;

    // Izracunaj ukupnu dobrotu svih jedinki
    double sumaDobrota = 0.0;
    for (const auto& j : populacija)
        sumaDobrota += j.dobrota;

    // Ako su sve jedinke lose (nula dobrote), vrati originalnu populaciju
    if (sumaDobrota == 0.0)
        return populacija;

    // Izgradi kumulativnu raspodjelu vjerojatnosti
    std::vector<double> kumulativna;
    double akumulirano = 0.0;
    for (const auto& j : populacija) {
        akumulirano += j.dobrota / sumaDobrota; // normalizirana vrijednost
        kumulativna.push_back(akumulirano);     // granica za odabir jedinki
    }
    // Odaberi ravnomjerno rasporedene tocke (strelice)
    double start = std::uniform_real_distribution<>(0.0, 1.0 / velPop)(gen);

    for (int i = 0; i < velPop; ++i) {
        double tocka = start + i * (1.0 / velPop); // strelica u [0,1]

        // Pronadi jedinku koju ova strelica "pogodi"
        int indeks = 0;
        for (size_t j = 0; j < kumulativna.size(); ++j) {
            if (tocka <= kumulativna[j]) {
                indeks = j;
                break; // prva granica koju strelica probije je pobjednik
            }
        }
        // Dodaj odabranu jedinku u novu populaciju
        nova.push_back(populacija[indeks]);
    }
    return nova;
}

// Krizanje u jednoj tocki prekida
std::vector<Jedinka> GA::krizanje(const std::vector<Jedinka>& roditelji) {
    std::vector<Jedinka> potomci = roditelji;
    std::uniform_real_distribution<> dist(0.0, 1.0);
    std::uniform_int_distribution<> prekid(1, predmeti.size() - 1);

    // Krizaju se parovi roditelja (trenutni i sljedeci)
    for (size_t i = 0; i + 1 < roditelji.size(); i += 2) {
        if (dist(gen) <= vjKrizanja) {
            int k = prekid(gen); // nasumicna tocka prekida
            for (size_t j = k; j < predmeti.size(); ++j)
                std::swap(potomci[i].geni[j], potomci[i + 1].geni[j]);
        }
    }
    return potomci;
}

// Uniformna mutacija
void GA::mutacija(std::vector<Jedinka>& populacija) {
    std::uniform_real_distribution<> dist(0.0, 1.0);
    for (auto& j : populacija)
        for (int& bit : j.geni)
```

```
            if (dist(gen) <= vjMutacije)
                bit = 1 - bit; // Inverzija bita
}

// Elitizam
void GA::dodajElitneJedinke(std::vector<Jedinka>& izvor, std::vector<Jedinka>& odrediste,
    int brojElitnih) {
    std::sort(izvor.begin(), izvor.end(), [](const Jedinka& a, const Jedinka& b) {
        return a.dobrota > b.dobrota;
        });
    std::sort(odrediste.begin(), odrediste.end(), [](const Jedinka& a, const Jedinka& b) {
        return a.dobrota > b.dobrota;
        });
    for (int i = 0; i < brojElitnih; ++i)
        odrediste[odrediste.size() - i - 1] = izvor[i];
}

Jedinka GA::dohvatiNajboljuJedinku(const std::vector<Jedinka>& populacija) {
    return *std::max_element(populacija.begin(), populacija.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

void GA::ispisiRjesenje(const Jedinka& najbolja) {
    double ukupnaTezina = 0.0;
    double ukupnaVrijednost = 0.0;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << std::left << std::setw(15) << "Naziv" << std::setw(17) << "Vrijednost (EUR
    )" << "Tezina (kg)\n";
    std::cout << "-----------------------------------------------\n";

    for (size_t i = 0; i < predmeti.size(); ++i) {
        if (najbolja.geni[i]) {
            std::cout << std::setw(15) << predmeti[i].naziv
                << std::setw(17) << predmeti[i].vrijednost
                << predmeti[i].tezina << "\n";
            ukupnaVrijednost += predmeti[i].vrijednost;
            ukupnaTezina += predmeti[i].tezina;
        }
    }
    std::cout << "-----------------------------------------------\n";
    std::cout << "Ukupna vrijednost: " << ukupnaVrijednost << " EUR\n";
    std::cout << "Ukupna tezina:     " << ukupnaTezina << " kg\n";
    std::cout << "Kapacitet ruksaka: " << kapacitet_ruksaka << " kg\n";
}

void GA::pokreni() {
    std::vector<Jedinka> populacija = generirajPopulaciju();
    evaluiraj(populacija);

    for (int genBroj = 0; genBroj < brojGeneracija; ++genBroj) {
        std::vector<Jedinka> P1 = selekcija(populacija);
        std::vector<Jedinka> P2 = krizanje(P1);
        mutacija(P2);
        evaluiraj(P2);
        dodajElitneJedinke(populacija, P2, brojElitnih);
        Jedinka najbolja = dohvatiNajboljuJedinku(P2);
        std::cout << "Generacija " << genBroj + 1 << ": najbolja dobrota = " << najbolja.
    dobrota << " EUR\n";
        populacija = P2;
    }
    Jedinka najbolja = dohvatiNajboljuJedinku(populacija);
    ispisiRjesenje(najbolja);
}

int main() {
    // popis mogucih predmeta u ruksaku
    std::vector<Predmet> predmeti = {
        {"Knjiga",     12.99, 1.2}, // naziv, cijena (EUR), tezina (kg)
        {"Boca vode",   1.50, 1.5},
        {"Laptop",    599.00, 2.5},
```

```
        {"Hrana",      20.00, 1.8},
        {"Kabanica",   49.99, 1.0},
        {"Fotoaparat", 320.00, 2.0},
        {"Upaljac",     2.00, 0.2},
        {"Jakna",      89.99, 1.4},
        {"Tablete",    15.50, 0.3},
        {"Baterije",    6.99, 0.5}
    };
    double kapacitet_ruksaka = 8.5; // u kg

    GA algoritam(kapacitet_ruksaka, predmeti);
    algoritam.pokreni();
    return 0;
}
```

9.34 Genetski algoritam (optimizacija školskog rasporeda)

Referenca: Poglavlje 7.4

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <random>
#include <algorithm>
#include <set>
#include <fstream>

struct Predmet {
    std::string naziv;
    std::string profesor;
};

struct Termin {
    int dan;
    int sat;    // pocetni sat
    int ucionica;
};

struct Jedinka {
    std::vector<Termin> raspored;
    int dobrota = 0;
    Jedinka(int broj_predmeta) : raspored(broj_predmeta) {}
};

class GA {
public:
    std::vector<Predmet> predmeti;   // svi predmeti
    int brojUcionica;                // broj dostupnih ucionica
    int brojDana = 5;                // broj dana nastave (pon-pet)
    int velPop = 100;                // velicina populacije
    int brojGeneracija = 200;        // maksimalni broj generacija
    double vjMutacije = 0.1;         // vjerojatnost mutacije
    int brojElitnih = 1;             // broj elitnih jedinki koje se prenose dalje
    int pocetniSat = 8;              // pocetak nastave (u satima)
    int zavrsniSat = 12;             // kraj nastave (u satima)
    int trajanjePredavanja = 1;      // trajanje jednog predavanja (u satima)
    std::mt19937 gen;                // generator slucajnih brojeva

    GA(const std::vector<Predmet>& predmeti, int brojUcionica, int brojDana = 5)
        : predmeti(predmeti), brojUcionica(brojUcionica), brojDana(brojDana), gen(std::
    random_device{}()) {
    }
    void pokreni();
    Jedinka generirajJedinku();
    void evaluiraj(Jedinka& jedinka);
```

```
    std::vector<Jedinka> selekcija(const std::vector<Jedinka>& populacija);
    std::vector<Jedinka> krizanje(const std::vector<Jedinka>& roditelji);
    void mutacija(Jedinka& jedinka);
    Jedinka dohvatiNajbolju(const std::vector<Jedinka>& populacija);
    void dodajElitu(std::vector<Jedinka>& staraPop, std::vector<Jedinka>& novaPop, int
    brojElitnih);
    void ispisiRaspored(const Jedinka& najbolja); // ispis konacnog rasporeda
    void ispisiExcelRaspored(const Jedinka& jedinka); // ispis rasporeda u CSV
    std::string danUTekst(int danIndex); // dan kao tekst
};

// Pretvara indeks dana u tekstualni naziv dana
std::string GA::danUTekst(int danIndex) {
    static const char* dani[] = { "Pon", "Uto", "Sri", "Cet", "Pet", "Sub" };
    return (danIndex >= 0 && danIndex < 6) ? dani[danIndex] : "???";
}

// Generira nasumicnu jedinku (raspored)
Jedinka GA::generirajJedinku() {
    std::uniform_int_distribution<> dan(0, brojDana - 1);
    std::uniform_int_distribution<> sat(pocetniSat, zavrsniSat - trajanjePredavanja);
    std::uniform_int_distribution<> ucionica(0, brojUcionica - 1);

    Jedinka jed(predmeti.size());
    for (auto& t : jed.raspored) {
        t.dan = dan(gen);
        t.sat = sat(gen);
        t.ucionica = ucionica(gen);
    }
    return jed;
}

// Evaluira jedinku i racuna broj penalizacija (sukoba)
void GA::evaluiraj(Jedinka& jedinka) {
    std::set<std::tuple<int, int, int>> zauzeti; // zauzeti termini po danu, satu i
    ucionici
    std::map<std::string, std::set<std::pair<int, int>>> profesorTermini; // termini po
    profesorima
    int penal = 0;

    for (size_t i = 0; i < predmeti.size(); ++i) {
        const auto& predmet = predmeti[i];
        const auto& t = jedinka.raspored[i];

        for (int d = 0; d < trajanjePredavanja; ++d) {
            int sat = t.sat + d;
            if (sat >= zavrsniSat) { ++penal; continue; } // termin izlazi iz vremenskog
    okvira
            if (!zauzeti.insert({ t.dan, sat, t.ucionica }).second) ++penal; // ucionica
    zauzeta
            if (!profesorTermini[predmet.profesor].insert({ t.dan, sat }).second) ++penal;
    // profesor zauzet
        }
    }
    jedinka.dobrota = penal; // broj sukoba
}

// Turnirska selekcija: biraju se bolji izmedu nasumicnih parova
std::vector<Jedinka> GA::selekcija(const std::vector<Jedinka>& populacija) {
    std::vector<Jedinka> nova;
    std::uniform_int_distribution<> indeks(0, populacija.size() - 1);
    for (int i = 0; i < velPop; ++i) {
        const Jedinka& a = populacija[indeks(gen)];
        const Jedinka& b = populacija[indeks(gen)];
        nova.push_back((a.dobrota < b.dobrota) ? a : b);
    }
    return nova;
}
```

```
// Uniformno ponderirano krizanje: svaki gen ima 50% sanse da se naslijedi od svakog
    roditelja
std::vector<Jedinka> GA::krizanje(const std::vector<Jedinka>& roditelji) {
    std::vector<Jedinka> potomci;
    size_t n = roditelji.size();
    std::uniform_real_distribution<> dist(0.0, 1.0);

    for (size_t i = 0; i + 1 < n; i += 2) {
        size_t broj_predmeta = roditelji[i].raspored.size();
        Jedinka dijete(broj_predmeta);
        for (size_t j = 0; j < broj_predmeta; ++j) {
            // Za svaki gen (termin) nasumicno biraj iz jednog od roditelja
            dijete.raspored[j] = (dist(gen) < 0.5) ? roditelji[i].raspored[j]
                : roditelji[i + 1].raspored[j];
        }
        potomci.push_back(dijete);
    }

    // Ako postoji neparan broj roditelja, zadnji ide u novu generaciju nepromijenjen
    if (n % 2 == 1)
        potomci.push_back(roditelji.back());
    return potomci;
}

// Mutira gene pojedine jedinke s odredenom vjerojatnoscu
void GA::mutacija(Jedinka& jedinka) {
    std::uniform_real_distribution<> dist(0.0, 1.0);
    std::uniform_int_distribution<> dan(0, brojDana - 1);
    std::uniform_int_distribution<> sat(pocetniSat, zavrsniSat - trajanjePredavanja);
    std::uniform_int_distribution<> ucionica(0, brojUcionica - 1);

    for (auto& t : jedinka.raspored)
        if (dist(gen) <= vjMutacije) {
            t.dan = dan(gen);
            t.sat = sat(gen);
            t.ucionica = ucionica(gen);
        }
}

// Vraca jedinku s najvecom dobrotom
Jedinka GA::dohvatiNajbolju(const std::vector<Jedinka>& populacija) {
    return *std::min_element(populacija.begin(), populacija.end(),
        [](const Jedinka& a, const Jedinka& b) {
            return a.dobrota < b.dobrota;
        });
}

// Kopira najbolje jedinke iz prethodne generacije u novu (elitizam)
void GA::dodajElitu(std::vector<Jedinka>& staraPop, std::vector<Jedinka>& novaPop, int
    brojElitnih) {
    std::partial_sort(staraPop.begin(), staraPop.begin() + brojElitnih, staraPop.end(),
        [](const Jedinka& a, const Jedinka& b) {
            return a.dobrota < b.dobrota;
        });

    for (int i = 0; i < brojElitnih && i < (int)novaPop.size(); ++i)
        novaPop[novaPop.size() - 1 - i] = staraPop[i];
}

// Ispisuje konacni raspored jedinke
void GA::ispisiRaspored(const Jedinka& najbolja) {
    for (size_t i = 0; i < predmeti.size(); ++i) {
        const auto& p = predmeti[i];
        const auto& t = najbolja.raspored[i];
        std::cout << p.naziv << " (" << p.profesor << ") -> "
            << danUTekst(t.dan) << " " << t.sat << "-" << (t.sat + trajanjePredavanja)
            << "h, ucionica " << t.ucionica + 1 << "\n";
    }
}
```

```
// Ispis rasporeda u pivot formatu u CSV datoteku
void GA::ispisiExcelRaspored(const Jedinka& jedinka) {
    std::map<int, std::map<int, std::vector<std::string>>> tablica;

    for (size_t i = 0; i < predmeti.size(); ++i) {
        const auto& p = predmeti[i];
        const auto& t = jedinka.raspored[i];

        for (int d = 0; d < trajanjePredavanja; ++d) {
            int sat = t.sat + d;
            std::string opis = p.naziv + " (" + p.profesor + ") [uc. " + std::to_string(t.
    ucionica + 1) + "]";
            tablica[sat][t.dan].push_back(opis);
        }
    }
    std::ofstream out("raspored_pivot.csv");
    out << "Sat";
    for (int d = 0; d < brojDana; ++d)
        out << ";" << danUTekst(d);
    out << "\n";

    for (const auto& [sat, poDanima] : tablica) {
        std::string vrijeme = std::to_string(sat) + ":00-" + std::to_string(sat + 1) + ":00
    ";
        out << vrijeme;
        for (int d = 0; d < brojDana; ++d) {
            std::string sadrzaj = "-";
            if (poDanima.count(d)) {
                const auto& aktivnosti = poDanima.at(d);
                sadrzaj = "";
                for (const auto& a : aktivnosti) {
                    if (!sadrzaj.empty()) sadrzaj += " | ";
                    sadrzaj += a;
                }
            }
            out << ";" << sadrzaj;
        }
        out << "\n";
    }
    out.close();
    std::cout << "\nDatoteka 'raspored_pivot.csv' je spremljena u radni direktorij.\n";
}

void GA::pokreni() {
    std::vector<Jedinka> populacija;
    for (int i = 0; i < velPop; ++i) {
        Jedinka jed = generirajJedinku();
        evaluiraj(jed);
        populacija.push_back(jed);
    }
    bool nasaoOptimalni = false;
    for (int g = 0; g < brojGeneracija && !nasaoOptimalni; ++g) {
        auto roditelji = selekcija(populacija);
        auto djeca = krizanje(roditelji);
        for (auto& dijete : djeca) {
            mutacija(dijete);
            evaluiraj(dijete);
            if (!dijete.dobrota) {
                nasaoOptimalni = true; // ako je bez penalizacija
                break;
            }
        }
        dodajElitu(populacija, djeca, brojElitnih);
        populacija = djeca;
        std::cout << "Generacija " << g + 1
            << ": najbolja dobrota = " << dohvatiNajbolju(populacija).dobrota << "\n";
    }
    ispisiRaspored(dohvatiNajbolju(populacija));
    ispisiExcelRaspored(dohvatiNajbolju(populacija));
}
```

```
int main() {
    // popis predmeta
    std::vector<Predmet> predmeti = {
        {"Predmet 1",  "Prof. 1"}, {"Predmet 2",  "Prof. 1"}, {"Predmet 3",  "Prof. 1"},
        {"Predmet 4",  "Prof. 2"}, {"Predmet 5",  "Prof. 2"}, {"Predmet 6",  "Prof. 2"},
        {"Predmet 7",  "Prof. 3"}, {"Predmet 8",  "Prof. 3"}, {"Predmet 9",  "Prof. 3"},
        {"Predmet 10", "Prof. 3"}, {"Predmet 11", "Prof. 4"}, {"Predmet 12", "Prof. 4"},
        {"Predmet 13", "Prof. 4"}, {"Predmet 14", "Prof. 4"}, {"Predmet 15", "Prof. 4"},
        {"Predmet 16", "Prof. 5"}, {"Predmet 17", "Prof. 5"}, {"Predmet 18", "Prof. 5"},
        {"Predmet 19", "Prof. 5"}, {"Predmet 20", "Prof. 5"}
    };
    int brojUcionica = 2;   // dvije ucionice na raspolaganju
    int brojDana = 3;       // nastava traje 3 dana (pon-sri)

    GA ga(predmeti, brojUcionica, brojDana);
    ga.pokreni();
    return 0;
}
```

9.35 Genetsko programiranje (simbolička regresija)

Referenca: Poglavlje 7.5

```
#include <iostream>
#include <iomanip>
#include <random>
#include <string>

// Cvor stabla izraza
class Cvor {
public:
    enum class Tip { KONST, VAR, OP } tip;
    double vrijednost = 0.0;
    char op = '?';
    std::shared_ptr<Cvor> lijevi = nullptr, desni = nullptr;

    Cvor(Tip t) : tip(t) {}

    // Rekurzivno evaluira izraz za danu vrijednost x
    double evaluiraj(double x) const {
        if (tip == Tip::KONST) return vrijednost;
        if (tip == Tip::VAR) return x;
        double l = lijevi->evaluiraj(x);
        double d = desni->evaluiraj(x);
        switch (op) {
        case '+': return l + d;
        case '-': return l - d;
        case '*': return l * d;
        default: return 0;
        }
    }

    // Vraca duboku kopiju stabla (koristi se za krizanje i mutaciju)
    std::shared_ptr<Cvor> kopiraj() const {
        auto novi = std::make_shared<Cvor>(tip);
        novi->vrijednost = vrijednost;
        novi->op = op;
        if (lijevi) novi->lijevi = lijevi->kopiraj();
        if (desni) novi->desni = desni->kopiraj();
        return novi;
    }

    // inorder prelazak stabla
    std::string toString() const {
        if (tip == Tip::KONST) return std::to_string(vrijednost);
```

```
        if (tip == Tip::VAR) return "x";
        return "(" + lijevi->toString() + " " + op + " " + desni->toString() + ")";
    }
};

// Jedinka: stablo izraza + dobrota
class Jedinka {
public:
    std::shared_ptr<Cvor> korijen;
    double dobrota;

    // Racuna dobrotu jedinke kao sumu kvadrata pogresaka na uzorcima
    void evaluiraj(const std::vector<std::pair<double, double>>& uzorci) {
        double suma = 0.0;
        for (auto& [x, y] : uzorci) {
            double fx = korijen->evaluiraj(x);
            // Ako rezultat izraza nije broj (NaN) ili je beskonacan (inf),
            // tretirajmo ga kao vrlo losu jedinku tako da mu dobrota bude velika.
            // Time sprjecavamo da "numericki opasne" jedinke opstanu u populaciji.
            if (std::isnan(fx) || std::isinf(fx)) fx = 1e6;
            suma += std::pow(fx - y, 2);
        }
        dobrota = suma;
    }
};

// Populacija
class Populacija {
public:
    std::vector<Jedinka> jedinke;

    void generirajSlucajnu(int velicina, std::mt19937& generator, const std::vector<char>&
    operatori);
    void evaluiraj(const std::vector<std::pair<double, double>>& uzorci);
    void dohvatiNajboljuJedinku(Jedinka& najbolja);
};

// Funkcija koja generira novo slucajno stablo izraza (rekurzivno)
std::shared_ptr<Cvor> generirajStablo(std::mt19937& gen,
    const std::vector<char>& operatori,
    int dubina = 0)
{
    std::uniform_real_distribution<> konst(0.0, 5.0); // ...za slucajne konstante
    std::uniform_int_distribution<> opDis(0, operatori.size() - 1);  // ...za odabir
    operatora

    // Baza rekurzije ili nasumicna odluka da stablo ne raste dalje:
    if (dubina > 2 || rand() % 3 == 0) {
        // Odluka hoce li biti varijabla (x) ili konstanta
        if (rand() % 2 == 0) {
            // Stvara cvor varijable x
            auto n = std::make_shared<Cvor>(Cvor::Tip::VAR);
            return n;
        }
        else {
            // Stvara cvor s nasumicnom konstantom iz [0, 5]
            auto n = std::make_shared<Cvor>(Cvor::Tip::KONST);
            n->vrijednost = konst(gen);
            return n;
        }
    }
    else {
        // Inace, generira se binarni operator cvor
        auto n = std::make_shared<Cvor>(Cvor::Tip::OP);
        n->op = operatori[opDis(gen)];  // Odabir operatora
        n->lijevi = generirajStablo(gen, operatori, dubina + 1);
        n->desni = generirajStablo(gen, operatori, dubina + 1);
        return n;
    }
}
```

```
// Generira novu populaciju jedinki sa slucajnim stablima izraza
void Populacija::generirajSlucajnu(int velicina, std::mt19937& generator, const std::vector
    <char>& operatori) {
    jedinke.clear();
    for (int i = 0; i < velicina; i++) {
        Jedinka j;
        j.korijen = generirajStablo(generator, operatori);
        jedinke.push_back(j);
    }
}

// Evaluira sve jedinke u populaciji na temelju zadatih uzoraka
void Populacija::evaluiraj(const std::vector<std::pair<double, double>>& uzorci) {
    for (auto& j : jedinke)
        j.evaluiraj(uzorci);
}

// Dohvaca jedinku s najmanjom pogreskom iz populacije
void Populacija::dohvatiNajboljuJedinku(Jedinka& najbolja) {
    najbolja = *std::min_element(jedinke.begin(), jedinke.end(),
        [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
}

// Algoritam genetskog programiranja
class GP {
public:
    int velicinaPopulacije = 100;
    int brojGeneracija = 100;
    int velicinaTurnira = 2;
    double vjerojatnostMutacije = 0.2;
    std::vector<std::pair<double, double>> uzorci;

    void pokreni(std::mt19937& generator,
        const std::vector<std::pair<double, double>>& ulazniUzorci,
        const std::vector<char>& operatori);

    std::vector<Jedinka> selekcija(const Populacija& pop, std::mt19937& gen);
    void mutiraj(Jedinka& j, std::mt19937& gen, const std::vector<char>& operatori);
    Jedinka krizaj(const Jedinka& a, const Jedinka& b, std::mt19937& gen);
};

// Vraca slucajno odabrano podstablo iz danog stabla
std::shared_ptr<Cvor> nasumicnoPodstablo(std::shared_ptr<Cvor> t, std::vector<std::
    shared_ptr<Cvor>>& lista) {
    if (!t) return nullptr;
    lista.push_back(t);
    if (t->lijevi) nasumicnoPodstablo(t->lijevi, lista);
    if (t->desni) nasumicnoPodstablo(t->desni, lista);
    return lista[rand() % lista.size()];
}

// Turnirska selekcija
std::vector<Jedinka> GP::selekcija(const Populacija& pop, std::mt19937& gen) {
    std::vector<Jedinka> selektirane;
    for (int i = 0; i < velicinaPopulacije; ++i) {
        std::uniform_int_distribution<> dist(0, pop.jedinke.size() - 1);
        Jedinka najbolji = pop.jedinke[dist(gen)];
        for (int j = 1; j < velicinaTurnira; ++j) {
            Jedinka kandidat = pop.jedinke[dist(gen)];
            if (kandidat.dobrota < najbolji.dobrota)
                najbolji = kandidat;
        }
        selektirane.push_back(najbolji);
    }
    return selektirane;
}

// Kriza dva roditelja tako da nasumicno zamijeni podstablo iz a podstablom iz b
Jedinka GP::krizaj(const Jedinka& a, const Jedinka& b, std::mt19937& gen) {
```

```
    Jedinka dijete;
    dijete.korijen = a.korijen->kopiraj();
    std::vector<std::shared_ptr<Cvor>> pod;
    auto u = nasumicnoPodstablo(b.korijen, pod);
    std::vector<std::shared_ptr<Cvor>> mijenjaj;
    auto zamijeni = nasumicnoPodstablo(dijete.korijen, mijenjaj);
    *zamijeni = *u->kopiraj();
    return dijete;
}

// Mutira jedinku tako da zamijeni jedno podstablo novim slucajnim stablom
void GP::mutiraj(Jedinka& j, std::mt19937& gen, const std::vector<char>& operatori) {
    std::uniform_real_distribution<> dis(0.0, 1.0);
    if (dis(gen) <= vjerojatnostMutacije) {
        std::vector<std::shared_ptr<Cvor>> lista;
        auto cilj = nasumicnoPodstablo(j.korijen, lista);
        *cilj = *generirajStablo(gen, operatori);
    }
}

// Glavna petlja algoritma
void GP::pokreni(std::mt19937& generator,
    const std::vector<std::pair<double, double>>& ulazniUzorci,
    const std::vector<char>& operatori)
{
    Populacija pop;
    pop.generirajSlucajnu(velicinaPopulacije, generator, operatori);
    pop.evaluiraj(ulazniUzorci);

    for (int g = 1; g <= brojGeneracija; ++g) {
        std::vector<Jedinka> P1 = selekcija(pop, generator);

        std::vector<Jedinka> P2;
        while ((int)P2.size() < velicinaPopulacije) {
            int i = rand() % P1.size();
            int j = rand() % P1.size();
            Jedinka dijete = krizaj(P1[i], P1[j], generator);
            mutiraj(dijete, generator, operatori);
            dijete.evaluiraj(ulazniUzorci);
            P2.push_back(dijete);
        }

        // Dohvati najbolju iz stare i najgoru iz nove generacije
        Jedinka elitna;
        pop.dohvatiNajboljuJedinku(elitna);
        auto najlosiji = std::max_element(P2.begin(), P2.end(),
            [](const Jedinka& a, const Jedinka& b) { return a.dobrota < b.dobrota; });
        *najlosiji = elitna;

        pop.jedinke = P2;
        Jedinka najbolja;
        pop.dohvatiNajboljuJedinku(najbolja);

        std::cout << "Generacija " << g << ": f(x)=" << std::fixed << std::setprecision(6)
            << najbolja.dobrota << " => " << najbolja.korijen->toString() << "\n";
    }
}

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    GP gp;

    // ulazne tocke (uzorci)
    std::vector<std::pair<double, double>> uzorci = {
        {-3, 4}, {-2, 1}, {-1, 0}, {0, 1}, {1, 4}, {2, 9}, {3, 16}
    };
    // podrzane matematicke operacije (u slucaju dodavanja
    // dodanih operatora nadopuniti evaluaciju cvora!)
    std::vector<char> operatori = { '+', '-', '*' };
```

```
    gp.pokreni(gen, uzorci, operatori);
    return 0;
}
```

9.36 Genetsko programiranje (klasifikacija zahtjeva za kredit)

Referenca: Poglavlje 7.6

```
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <random>
#include <algorithm>
#include <map>

// ======= Klasa za cvor stabla odlucivanja =======
class Cvor {
public:
    enum class Tip { VARIJABLA, KONST, IF, OP };      // tip cvora: varijabla, konstanta,
    uvjet (IF), operator
    enum class Operator { EQ, NEQ, AND, OR };         // logicki operatori

    Tip tip;
    int vrijednost = 0;              // samo za konstantne cvorove (0 ili 1)
    std::string varijabla;           // naziv varijable, npr. "A", "B", "C", "D"
    Operator op;                     // koristi se ako je cvor operator (OP)

    // Pokazivaci na podstabla, ovisno o tipu
    std::shared_ptr<Cvor> lijevi, desni, srednji;

    Cvor(Tip t) : tip(t) {}

    // Evaluacija izraza nad konkretnim primjerom iz skupa podataka
    int evaluiraj(const std::map<std::string, int>& primjer, int maxDepth = 1000) const {
        if (maxDepth <= 0)
            throw std::runtime_error("Prevelika dubina evaluacije (rekurzija?).");

        if (tip == Tip::VARIJABLA) return primjer.at(varijabla);
        if (tip == Tip::KONST) return vrijednost;

        if (tip == Tip::IF) {
            int uvjet = lijevi->evaluiraj(primjer, maxDepth - 1);
            return uvjet ? srednji->evaluiraj(primjer, maxDepth - 1)
                : desni->evaluiraj(primjer, maxDepth - 1);
        }
        if (tip == Tip::OP) {
            int l = lijevi->evaluiraj(primjer, maxDepth - 1);
            int d = desni->evaluiraj(primjer, maxDepth - 1);
            switch (op) {
            case Operator::EQ:  return l == d;
            case Operator::NEQ: return l != d;
            case Operator::AND: return l && d;
            case Operator::OR:  return l || d;
            }
        }
        return 0;
    }

    // Pretvara stablo u formatirani string (za prikaz)
    std::string tostring(int indent = 0) const {
        std::string pad(indent, ' ');
        if (tip == Tip::VARIJABLA) return pad + varijabla;
        if (tip == Tip::KONST) return pad + std::to_string(vrijednost);
```

```
        if (tip == Tip::OP) {
            std::string opstr = (op == Operator::EQ) ? "==" :
                (op == Operator::NEQ) ? "!=" :
                (op == Operator::AND) ? "&&" : "||";
            return pad + "(" + lijevi->tostring() + " " + opstr + " " + desni->tostring() +
     ")";
        }
        if (tip == Tip::IF) {
            std::string s = pad + "IF (" + lijevi->tostring() + ")\n";
            s += pad + "  THEN\n" + srednji->tostring(indent + 4) + "\n";
            s += pad + "  ELSE\n" + desni->tostring(indent + 4);
            return s;
        }
        return pad + "?";
    }

    // Rekurzivna duboka kopija cvora i podstabla
    std::shared_ptr<Cvor> kopiraj() const {
        auto novi = std::make_shared<Cvor>(tip);
        novi->vrijednost = vrijednost;
        novi->varijabla = varijabla;
        novi->op = op;
        if (lijevi)   novi->lijevi = lijevi->kopiraj();
        if (desni)    novi->desni = desni->kopiraj();
        if (srednji)  novi->srednji = srednji->kopiraj();
        return novi;
    }
};

// ======= Jedinka (kandidatno rjesenje) =======
struct Primjer {
    std::map<std::string, int> varijable; // unos: A, B, C, D
    int oznaka; // klasa (0 = odbiti, 1 = odobriti)
};

class Jedinka {
public:
    std::shared_ptr<Cvor> korijen; // stablo odlucivanja
    int dobrota = 0;               // broj tocnih klasifikacija

    void evaluiraj(const std::vector<Primjer>& skup) {
        dobrota = 0;
        for (const auto& p : skup)
            if (korijen->evaluiraj(p.varijable) == p.oznaka)
                dobrota++;
    }
};

class GP {
public:
    int velicina = 50;                // broj jedinki u populaciji
    int generacije = 50;              // broj iteracija
    int turnir = 2;                   // velicina turnira za selekciju
    double vjerojatnostMutacije = 0.2; // vjerojatnost da ce mutacija biti primijenjena
    std::vector<std::string> varijable = { "A", "B", "C", "D" };
    std::mt19937 gen{ std::random_device{}() }; // generator slucajnih brojeva

    // Generira novo nasumicno stablo (rekurzivno)
    std::shared_ptr<Cvor> randomstablo(int dubina = 3) {
        if (dubina <= 0) {
            std::uniform_int_distribution<> coin(0, 1);
            if (coin(gen)) {
                auto c = std::make_shared<Cvor>(Cvor::Tip::KONST);
                c->vrijednost = coin(gen);
                return c;
            }
            else {
                auto c = std::make_shared<Cvor>(Cvor::Tip::VARIJABLA);
                c->varijabla = varijable[gen() % varijable.size()];
                return c;
```

```
            }
        }

        int r = gen() % 3;
        if (r == 0) {
            // IF cvor: ima uvjet i dvije grane
            auto c = std::make_shared<Cvor>(Cvor::Tip::IF);
            c->lijevi = randomLogicki(dubina - 1);
            c->srednji = randomstablo(dubina - 1);
            c->desni = randomstablo(dubina - 1);
            return c;
        }
        else {
            return randomLogicki(dubina);
        }
    }

    // Generira logicki izraz (bin. operator s dva podizraza)
    std::shared_ptr<Cvor> randomLogicki(int d) {
        if (d <= 0) return randomstablo(0);
        auto c = std::make_shared<Cvor>(Cvor::Tip::OP);
        c->op = static_cast<Cvor::Operator>(gen() % 4);
        c->lijevi = randomstablo(d - 1);
        c->desni = randomstablo(d - 1);
        return c;
    }

    // Pronalazi nasumicno podstablo (koristi se za krizanje i mutaciju)
    std::shared_ptr<Cvor> nasumicnoPodstablo(std::shared_ptr<Cvor> c, std::vector<std::
    shared_ptr<Cvor>>& lista) {
        if (!c) return nullptr;
        lista.push_back(c);
        if (c->lijevi)   nasumicnoPodstablo(c->lijevi, lista);
        if (c->desni)    nasumicnoPodstablo(c->desni, lista);
        if (c->srednji)  nasumicnoPodstablo(c->srednji, lista);
        return lista[gen() % lista.size()];
    }

    // Mutira podstablo unutar danog stabla (uz zadanu vjerojatnost)
    void mutiraj(std::shared_ptr<Cvor>& c, int dubina = 3) {
        std::uniform_real_distribution<> dis(0.0, 1.0);
        if (dis(gen) <= vjerojatnostMutacije) {
            std::vector<std::shared_ptr<Cvor>> lista;
            auto cilj = nasumicnoPodstablo(c, lista);
            *cilj = *randomstablo(dubina);
        }
    }

    // Krizanje: kopira jedno stablo, a u njega umece nasumicno podstablo drugog
    Jedinka krizaj(const Jedinka& a, const Jedinka& b) {
        Jedinka dijete;
        dijete.korijen = a.korijen->kopiraj();
        std::vector<std::shared_ptr<Cvor>> podA, podB;
        auto u = nasumicnoPodstablo(b.korijen, podB);
        auto cilj = nasumicnoPodstablo(dijete.korijen, podA);
        *cilj = *u->kopiraj();
        return dijete;
    }

    // Turnirska selekcija: iz nasumicnih kandidata biramo najbolju jedinku
    std::vector<Jedinka> selekcija(const std::vector<Jedinka>& populacija) {
        std::vector<Jedinka> odabrani;
        for (int i = 0; i < (int)populacija.size(); ++i) {
            Jedinka najbolji = populacija[gen() % populacija.size()];
            for (int j = 1; j < turnir; ++j) {
                Jedinka kandidat = populacija[gen() % populacija.size()];
                if (kandidat.dobrota > najbolji.dobrota)
                    najbolji = kandidat;
            }
            odabrani.push_back(najbolji);
```

```
        }
        return odabrani;
    }

    // Glavna petlja evolucije
    void pokreni(const std::vector<Primjer>& skup) {
        std::vector<Jedinka> populacija;

        // Inicijalna nasumicna populacija
        for (int i = 0; i < velicina; ++i) {
            Jedinka j;
            j.korijen = randomstablo();
            j.evaluiraj(skup);
            populacija.push_back(j);
        }
        // Evolucija kroz generacije
        for (int g = 1; g <= generacije && populacija[0].dobrota != skup.size(); ++g) {
            std::sort(populacija.begin(), populacija.end(),
                [](const Jedinka& a, const Jedinka& b) {
                    return a.dobrota > b.dobrota;
                });

            std::cout << "Generacija " << g << ": "
                << populacija[0].dobrota << "/" << skup.size() << "\n";

            if (g == generacije || populacija[0].dobrota == skup.size())
                std::cout << populacija[0].korijen->tostring() << "\n";

            std::vector<Jedinka> nova;
            nova.push_back(populacija[0]); // elitizam - najbolja jedinka ide direktno
    dalje

            std::vector<Jedinka> roditelji = selekcija(populacija);
            // Generiraj novu populaciju krizanjem i mutacijom
            while ((int)nova.size() < velicina) {
                int i = gen() % roditelji.size();
                int j = gen() % roditelji.size();
                Jedinka dijete = krizaj(roditelji[i], roditelji[j]);
                mutiraj(dijete.korijen);
                dijete.evaluiraj(skup);
                nova.push_back(dijete);
            }
            populacija = nova;
        }
    }
};

int main() {
    // Skup primjera za treniranje/genetsko programiranje
    std::vector<Primjer> skup = {
        {{{"A",1},{"B",1},{"C",0},{"D",0}}, 1}, // idealan kandidat
        {{{"A",1},{"B",1},{"C",1},{"D",0}}, 1}, // ima dugove, ali ostalo pozitivno
        {{{"A",1},{"B",0},{"C",0},{"D",0}}, 1}, // zaposlen, bez dugova
        {{{"A",0},{"B",1},{"C",0},{"D",0}}, 1}, // dobra povijest, bez dugova
        {{{"A",0},{"B",0},{"C",1},{"D",1}}, 0}, // mlad, dugovi, bez posla/povijesti
        {{{"A",0},{"B",1},{"C",1},{"D",1}}, 0}, // samo povijest, ostalo lose
        {{{"A",1},{"B",0},{"C",1},{"D",1}}, 0}, // zaposlen, ali ima dugove, mlad
        {{{"A",0},{"B",0},{"C",0},{"D",1}}, 0}, // mlad, bez ikakvih prednosti
        {{{"A",0},{"B",0},{"C",1},{"D",0}}, 0}, // stariji, ali los profil
        {{{"A",1},{"B",1},{"C",0},{"D",1}}, 1}, // mlad, ali odlican profil
        {{{"A",1},{"B",0},{"C",0},{"D",1}}, 1}, // mlad, zaposlen, bez dugova
        {{{"A",0},{"B",1},{"C",0},{"D",1}}, 1}, // mlad, dobra povijest, bez dugova
        {{{"A",0},{"B",0},{"C",0},{"D",0}}, 0}  // stariji, ali bez ikakvih prednosti
    };
    GP gp;
    gp.pokreni(skup);
    return 0;
}
```

Popis slika

Popis algoritama

Popis primjera

Popis tablica

Bibliografija

[1] Željko Kovačević. "Automatic compiler/interpreter generation from programs for domain-specific languages using semantic inference". PhD thesis. Faculty of Electrical Engineering and Computer Science, Maribor, 2022.

[2] Thomas Bartz-Beielstein et al. "Circuit Design Using Evolutionary Algorithms". In: vol. 2. Feb. 2002, pp. 1904–1909. DOI: 10.1109/CEC.2002.1004534.

[3] Julian Miller et al. "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study". In: *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science* (Oct. 1999).

[4] Ana dos Santos Paulino, Jean-Christophe Nebel, and Francisco Flórez-Revuelta. "Evolutionary Algorithm for Dense Pixel Matching in Presence of Distortions". In: Apr. 2014. ISBN: 978-3-662-45522-7. DOI: 10.1007/978-3-662-45523-4_36.

[5] Christopher J. Solomon, Stuart J. Gibson, and Joseph J. Mist. "Interactive evolutionary generation of facial composites for locating suspects in criminal investigations". In: *Applied Soft Computing* 13.7 (2013), pp. 3298–3306. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2013.02.010. URL: https://www.sciencedirect.com/science/article/pii/S1568494613000641.

[6] Slimane Sefiane and Mohamed Benbouziane. "Portfolio Selection Using Genetic Algorithm". In: *Journal of Applied Finance & Banking* 2 (Jan. 2012), pp. 143–154.

[7] Michael White and Stuart Flockton. "Genetic Algorithms for Digital Signal Processing". In: Aug. 2000. ISBN: 978-3-540-58483-4. DOI: 10.1007/3-540-58483-8_22.

[8] Anna Ławrynowicz. "Genetic Algorithms for Solving Scheduling Problems in Manufacturing Systems". In: *Foundations of Management* 3.2 (2012), pp. 7–26. DOI: doi:10.2478/v10238-012-0039-2. URL: https://doi.org/10.2478/v10238-012-0039-2.

[9] Gregorius Satia Budhi, Kartika Gunadi, and Denny Alexander Wibowo. "Genetic Algorithm for Scheduling Courses". In: *Intelligence in the Era of Big Data*. Ed. by Rolly Intan et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 51–63. ISBN: 978-3-662-46742-8.

[10] Yongbeom Kwon and Juyong Lee. "MolFinder: an evolutionary algorithm for the global optimization of molecular properties and the extensive exploration of chemical space using SMILES". In: *Journal of Cheminformatics* 13.1 (Mar. 18, 2021), p. 24. ISSN: 1758-2946. DOI: 10.1186/s13321-021-00501-7. URL: https://doi.org/10.1186/s13321-021-00501-7.

[11] Poonam Garg. *Evolutionary Computation Algorithms for Cryptanalysis: A Study*. 2010. arXiv: 1006.5745 [cs.CR].

[12] Hendrawan Armanto et al. "Evolutionary Algorithm in Game – A Systematic Review". In: *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control* (May 2023). DOI: 10.22219/kinetik.v8i2.1714.

[13] K.A. De Jong. "An analysis of the behavior of a class of genetic adaptive systems". PhD thesis. Ann Arbor, MI, USA: University of Michigan, 1975.

[14] John H. Holland. “Genetic Algorithms and the Optimal Allocation of Trials”. In: *SIAM Journal on Computing* 2.2 (1973), pp. 88–105. DOI: 10.1137/0202009. eprint: https://doi.org/10.1137/0202009. URL: https://doi.org/10.1137/0202009.

[15] N. Krasnogor and J. Smith. “A tutorial for competent memetic algorithms: model, taxonomy, and design issues”. In: *Trans. Evol. Comp* 9.5 (Oct. 2005), pp. 474–488. ISSN: 1089-778X. DOI: 10.1109/TEVC.2005.850260. URL: https://doi.org/10.1109/TEVC.2005.850260.

[16] Wolfgang Banzhaf et al. “Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications”. In: Jan. 1998.

[17] John R. Koza. *Genetic Programming*. Cambridge, MA, USA: MIT Press, 1992.

[18] John R. Koza. *Genetic Programming II*. Cambridge, MA, USA: MIT Press, 1994.

[19] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992. ISBN: 978-0-262-11170-6.

[20] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution*. Wiley, 1966. ISBN: 9780471265160. URL: https://books.google.hr/books?id=75RQAAAAMAAJ.

[21] Hans-Georg Beyer. “Toward a Theory of Evolution Strategies: Self-Adaptation”. In: *Evolutionary Computation* 3.3 (1994). Gauss mutation, pp. 311–347. DOI: 10.1162/evco.1994.3.3.311.

[22] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata (Stuttgart). Frommann-Holzboog, 1973. ISBN: 9783772803741. URL: https://books.google.hr/books?id=-WAQAQAAMAAJ.

[23] Hans-Paul Schwefel. *Evolution and Optimum Seeking*. Jan. 1995. ISBN: 978-0-471-57148-3.

[24] Željko Kovačević et al. “Automatic compiler/interpreter generation from programs for Domain-Specific Languages: Code bloat problem and performance improvement”. In: *Journal of Computer Languages* 70 (2022), p. 101105. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2022.101105. URL: https://www.sciencedirect.com/science/article/pii/S2590118422000132.

[25] R. W. Hamming. “Error detecting and error correcting codes”. In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[26] Frank Gray. *Pulse Code Communication*. U.S. Patent 2,632,058. Filed November 13, 1947, and issued March 17, 1953. 1953.

[27] W. W. Rouse Ball. *Mathematical Recreations and Essays*. New York: Macmillan, 1960, pp. 165–171.

[28] Marjan Mernik et al. “Compiler/interpreter generator system LISA”. In: *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*. Jan. 2000, 10 pp.

[29] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262631857.

[30] OpenStax. *Calculus Volume 1*. OpenStax, Rice University, 2021. URL: https://openstax.org/books/calculus-volume-1/pages/4-1-maximum-and-minimum-values.

[31] LibreTexts. *Maxima and Minima*. LibreTexts, 2023. URL: https://math.libretexts.org/Bookshelves/Calculus/Calculus_Volume_1/04%3A_Applications_of_Derivatives/4.02%3A_Maxima_and_Minima.

[32] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[33] James E. Baker. “Reducing bias and inefficiency in the selection algorithm”. In: *Proceedings of the second international conference on genetic algorithms* (1987), pp. 14–21.

[34] Darrell Whitley. “The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best”. In: *Proceedings of the third international conference on genetic algorithms* 3 (1989), pp. 116–121.

[35] A. G Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Heidelberg, Germany: Springer, 2015.

[36] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[37] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Swap mutation. Addison-Wesley, 1989. ISBN: 978-0-201-15767-3. URL: https://www.worldcat.org/isbn/0201157675.

[38] Gilbert Syswerda. "Uniform crossover in genetic algorithms". In: *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989, pp. 2–9.

[39] Achini Herath and Dawn Wilkins. "Timetabling with Three-Parent Genetic Algorithm: a Preliminary Study". In: (May 2018). DOI: 10.11159/cdsr18.127.

[40] Francisco Herrera and Manuel Lozano. "Tackling real-coded genetic algorithms: Operators and tools for behavioral analysis". In: *Artificial Intelligence Review* 12.4 (1998), pp. 265–319.

[41] Kenneth A. De Jong and William M. Spears. "Heuristics based on genetic algorithms for solving large scale optimization problems". In: *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991, pp. 414–420.

[42] David E. Goldberg and Robert Lingle Jr. "Alleles, loci, and the traveling salesman problem". In: *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, 1985, pp. 154–159.

[43] Kenneth A. De Jong. "Cyclic crossover and mutation in genetic algorithms". In: *Proceedings of the 5th International Conference on Genetic Algorithms* (1993), pp. 2–9.

[44] Hernán Escalante, Efrén Cantu-Paz, and David E. Goldberg. "Geometrical crossover operators for genetic algorithms". In: *Proceedings of the 1994 IEEE International Conference on Evolutionary Computation*. IEEE, 1994, pp. 232–237.

[45] David E. Goldberg. "Genetic algorithms in optimization, machine learning, and artificial intelligence". In: *Proceedings of the 1st European Conference on Artificial Intelligence*. Elsevier, 1991, pp. 19–23.

[46] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Belmont, MA: Athena Scientific, 1997. ISBN: 1886528200.

[47] Lawrence Davis. "Applying adaptive algorithms to epistatic domains". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1985, pp. 162–164.

[48] Darrell Whitley. "The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best". In: *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989, pp. 116–121.

[49] Hans-Georg Beyer and Hans-Paul Schwefel. *Evolution Strategies: A Comprehensive Introduction*. Uniform mutation. Springer, 2002. ISBN: 978-3-540-42532-5. URL: https://doi.org/10.1007/978-3-662-04921-7.

[50] Wolfgang Banzhaf. "The "Molecular" Computer: Evolution and Natural Learning Processes in Genetic Programming". In: *Complex Systems* 4.1 (1990). Inversion mutation, pp. 1–16.

[51] Stephan Blum et al. "Adaptive Mutation Strategies for Evolutionary Algorithms". In: 2012. URL: https://api.semanticscholar.org/CorpusID:32320170.

[52] J.K. Patel and C.B. Read. *Handbook of the Normal Distribution*. Statistics, textbooks and monographs. M. Dekker, 1982. ISBN: 9780824715410. URL: https://books.google.hr/books?id=XDOdf0g0Yz4C.

[53] George Casella and Roger L. Berger. *Statistical Inference*. 2nd. Pacific Grove, CA: Duxbury Press, 2002. ISBN: 978-0-534-24312-8.

[54] Mandavilli Srinivas and Lalit M Patnaik. "Adaptive probabilities of crossover and mutation in genetic algorithms". In: *IEEE Transactions on Systems, Man, and Cybernetics* 24.4 (1994), pp. 656–667.

[55] Agoston E Eiben, Robert Hinterding, and Zbigniew Michalewicz. "Parameter control in evolutionary algorithms". In: *IEEE Transactions on evolutionary computation* 3.2 (1999), pp. 124–141.

[56] Haiming Du et al. “Elitism and Distance Strategy for Selection of Evolutionary Algorithms”. In: *IEEE Access* 6 (2018), pp. 44531–44541. DOI: 10.1109/ACCESS.2018.2861760.

[57] Hisao Ishibuchi, Noritaka Tsukamoto, and Yusuke Nojima. “Examining the Effect of Elitism in Cellular Genetic Algorithms Using Two Neighborhood Structures”. In: *Parallel Problem Solving from Nature – PPSN X*. Ed. by Günter Rudolph et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 458–467. ISBN: 978-3-540-87700-4.

[58] Enrique Alba and José M. Troya. “Parallel evolutionary algorithms: A survey”. In: *Proceedings of the 2000 Congress on Evolutionary Computation*. Vol. 1. IEEE. 2000, pp. 1–8.

[59] Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. USA: Kluwer Academic Publishers, 2000. ISBN: 0792372212.

[60] Erick Cantú-Paz. “A Survey of Parallel Genetic Algorithms”. In: 2000. URL: https://api.semanticscholar.org/CorpusID:14264381.

[61] Enrique Alba and Marco Tomassini. “Parallel genetic algorithms: From theory to applications”. In: *Evolutionary Computation* 6.2 (2002), pp. 143–169.

[62] Željko Kovačević. *Napredne tehnike programiranja u C++ Builderu*. Zagreb: Tehničko veleučilište u Zagrebu, 2020. ISBN: 978-953-7048-91-4.

[63] Wei-Neng Chen et al. *A Survey on Distributed Evolutionary Computation*. 2023. arXiv: 2304.05811 [cs.NE]. URL: https://arxiv.org/abs/2304.05811.

[64] G. Belletti, J. Troccaz, and L. DeLoura. *Parallel Genetic Algorithms: Theory and Applications*. Springer, 2012. ISBN: 978-1-4614-6435-8.

[65] D.E. Goldberg, W. Segrest, and J. Horn. “Asynchronous Parallel Genetic Algorithms”. In: *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2000)*. 2000, pp. 1075–1082. DOI: 10.1109/CEC.2000.870282.

[66] Matej Črepinšek et al. “Long Term Memory Assistance for Evolutionary Algorithms”. In: *Mathematics* 7.11 (2019). ISSN: 2227-7390. DOI: 10.3390/math7111129. URL: https://www.mdpi.com/2227-7390/7/11/1129.

[67] Paul R. Halmos. *Naive Set Theory*. Princeton, NJ: Van Nostrand, 1960.

[68] Stuart Russell and Peter Norvig. “Artificial Intelligence: A Modern Approach (3rd ed.)” In: (2009).

[69] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680.

[70] F. Glover. “Tabu Search—Part I”. In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206.

[71] P. Moscato. “On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms”. In: *Caltech Concurrent Computation Program* 826 (1989), p. 1989.

[72] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier, 2005.

[73] Christian Blum and Andrea Roli. “Metaheuristics in combinatorial optimization: Overview and conceptual comparison”. In: *ACM Computing Surveys (CSUR)* 35.3 (2003), pp. 268–308.

[74] Pablo Moscato. “Memetic algorithms: A short introduction”. In: *New Ideas in Optimization*. Ed. by D. Corne, M. Dorigo, and F. Glover. McGraw-Hill, 1999, pp. 219–234.

[75] Bart Selman. “Hill-climbing search”. In: *Encyclopedia of Cognitive Science*. Nature Publishing Group, 2003, pp. 578–579.

[76] Bart Selman, Henry A. Kautz, and Bram Cohen. “A random walk approach to hill-climbing search”. In: *AAAI* 92 (1992), pp. 10–15.

[77] P.J.M. Van Laarhoven and E.H.L. Aarts. “Simulated annealing”. In: *Simulated Annealing: Theory and Applications*. Springer, 1987, pp. 7–15.

[78] Ari Juels and Martin Wattenberg. “Stochastic Hillclimbing as a Baseline Method for Evaluating Genetic Algorithms”. In: *Advances in Genetic Programming*. Ed. by Kenneth E. Kinnear. MIT Press, 1995, pp. 267–283.

[79] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. See Chapter 3: Solving Problems by Searching, and discussion of Beam Search. Pearson, 2020. ISBN: 978-0134610993.

[80] Nicholas Metropolis et al. "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092. DOI: 10.1063/1.1699114.

[81] Stuart Geman and Donald Geman. "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (1984), pp. 721–741. DOI: 10.1109/TPAMI.1984.4767596.

[82] Fred Glover. "Future Paths for Integer Programming and Links to Artificial Intelligence". In: *Computers & Operations Research* 13.5 (1986), pp. 533–549. DOI: 10.1016/0305-0548(86)90048-1.

[83] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993. ISBN: 9780136175490.

[84] Richard Dawkins. *The Selfish Gene*. 1st. Oxford University Press, 1976. ISBN: 9780192860927.

[85] Ferrante Neri, Carlos Cotta, and Pablo Moscato. *Handbook of Memetic Algorithms*. Vol. 379. Studies in Computational Intelligence. Springer, 2012. DOI: 10.1007/978-3-642-23247-3.

[86] Yew-Soon Ong and Abhishek Gupta. "Evolutionary Multitasking: A Computer Science View of Cognitive Transfer". In: *IEEE Transactions on Evolutionary Computation* 21.5 (2016), pp. 770–784. DOI: 10.1109/TEVC.2016.2638011.

[87] Miha Ravber et al. "Inferring Absolutely Non-Circular Attribute Grammars with a Memetic Algorithm". In: *Applied Soft Computing* 100 (2021), p. 106956. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2020.106956. URL: https://www.sciencedirect.com/science/article/pii/S1568494620308942.

[88] Jordan Bell and Brett Stevens. "A Survey of Known Results and Research Areas for n-Queens". In: *Discrete Mathematics* 309.1 (2009), pp. 1–31.

[89] Bo Bernhardsson. "Explicit Solutions to the n-Queens Problem for All n". In: *ACM SIGART Bulletin* 2.2 (1991), pp. 7–11.

[90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Chichester: Wiley, 1990.

[91] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer Series in Operations Research. New York: Springer, 2004.

[92] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. 5th ed. Wiley, 2012. ISBN: 978-0470542811.

[93] Michael Schmidt and Hod Lipson. "Distilling free-form natural laws from experimental data". In: *Science* 324.5923 (2009), pp. 81–85.

[94] Andras Sobester, Alexander I.J. Forrester, and Andy J. Keane. "Engineering design applications of surrogate modelling". In: *Archives of Computational Methods in Engineering* 12.1 (2005), pp. 3–25.

[95] Sasa Vlasic et al. "Symbolic regression using genetic programming applied to modeling of complex systems". In: *Applied Soft Computing* 111 (2021), p. 107717.

[96] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN: 9780387310732.

[97] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539.

[98] Kalyanmoy Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[99] Rainer Storn and Kenneth Price. "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces". In: *Journal of Global Optimization* 11.4 (1997), pp. 341–359. DOI: 10.1023/A:1008202821328.

Ključne riječi