

RI-Skripta 2025

Petar Pavlović

October 2025

Sadržaj

1 Uvod	4
1.1 Analitika	6
1.2 Odlučivanje	7
2 Paradigme u veštačkoj inteligenciji	9
2.1 Veštačke neuronske mreže (ANN)	10
2.2 Evolutivna izračunavanja (EC)	10
2.2.1 Genetsko algoritmi (GA)	11
2.2.2 Genetsko programiranje (GP)	11
2.2.3 Evolutivno programiranje (EP)	12
2.2.4 Koevolucija	12
2.3 Inteligencija grupa (Swarm Intelligence - SI)	12
2.3.1 PSO	12
2.4 Rasplinuti (fuzzy) sistemi	12
3 Optimizacioni algoritmi	14
3.1 Uvod u optimizacije	15
3.2 Optimizacioni algoritmi	16
3.3 Korist optimalnih algoritama	18
4 Linearno programiranje (LP)	20
4.1 Pogodan region (Feasible Region)	20
4.2 Geometrijski princip rešavanja	20
4.3 Simplex metod	23
4.3.1 Primer primene simplex algoritma – Farmer	24
4.3.2 Dualni problem	26
5 Celobrojno programiranje (ILP)	27
5.1 Pogodan region ILP (ILP Feasable region)	27
5.2 Geometrijski princip rešavanja ILP	27
5.2.1 Problem ranca	28
6 Nelinearno programiranje (NLP)	30
6.1 Polovljenje intervala (Bisection Method)	30
6.1.1 Primer polovljenja intervala:	33
6.2 Njutnov metod	34
6.2.1 Primer njutnog metoda	35
6.3 Gradijentni spust	36
6.3.1 Gradijentni spust sa momentumom	37
6.3.2 Nestorov gradijentni spust	38
6.3.3 Adam	38
6.4 Njutnov metod za više promenljivih (Multiple variable NM)	40

7 Metaheuristike	42
7.1 Trajectory methods (S - Metaheuristics)	42
7.1.1 Primer za VNS	45
7.2 Population-based methods (P - Metaheuristica)	46
7.2.1 Primer jata ptica	49
8 Evolutivna izračunavanja (EC)	51
8.1 Osnovni pojmovi	51
8.2 Genetski algoritmi (GA)	54
8.2.1 Problem trgovačkog putnika (TSP)	57
8.3 Genetsko programiranje (GP)	59
8.3.1 Primer primene GP	61
9 Inteligencija grupa (SI - Swarm Intelligence)	62
9.1 Particle Swarm Optimization (PSO)	63
9.2 Ant Colony Optimization (ACO)	63
9.2.1 Primer TSP problema rešen ACO algoritmom	64
9.3 Artificial Bee Colony (ABC)	66
9.3.1 Primer primene ABC algoritma	67
10 S - Metaheuristike	70
10.1 Simulated Annealing (SA)	70
10.1.1 Primer primene SA	70
10.2 Tabu Search (TS)	72
10.2.1 Primer primene TS	73

1 Uvod

Nalazimo se u vremenu velike količine podataka, sve te podatke čoveku je protrebbno mnogo vremena da obradi, a sam proces pronalaženja nekog obrazac, ili mesta za poboljšanje je težak posao koji zahteva velike količine matematičke analize...

Danas imamo razne sisteme koji te podatke, uz smernice, brže obrade i sami nađu rešenja koja mogu biti pritom mnogo kvalitetnija od onih nađenih od strane čoveka.

Sam naziv ovog predmeta "Računarska inteligencija", kao i oblast kojoj pripadaju sistemi koje obrađujemo na ovom predmetu "Veštačka inteligencija", pominju **inteligenciju** u svojim nazivima što stvara pogrešnu ideju.

Mada se kaže veštačka inteligencija, inteligencija nije stvarno ono što je u pozadini ovih sistema... Naravno zavisi kako se na inteligenciju gleda, ali ono što najbolje predstavlja **inteligenciju**, kakvu mi ljudi posedujemo, jeste sposobnost da iz starog znanja gradimo novo znanje u vidu ideja, umetnosti i sl... Iz ove ideje je i sama oblast nastala, "posmatramo ljude kao mašine" i pokušamo da to rekreiramo. Postoje sistemi koji automatski dokazuju teoreme i taj dokaz predstavljaju kao što bi čovek to uradio. Problem nastaje kod skalabilnosti zadataka zbog čega se praktikuje drugačiji pristup.

Sami principi rada mnogih algoritama koje obrađujemo zasnovani su na **prirodnim pojavama** - kao što su kretanje mrvava (ACO), organizovanost pčela (ABC), interakcija čestica (PSO) ili čak i sam koncept evolucija inspirišu ove algoritme. Sam princip rada ovih algoritama (**metaheuristika**) svodi se na istraživanje prostora rešenja, odnosno pronalaženju što optimalnijeg rešenja, koristeći heuristike kao "smernice".

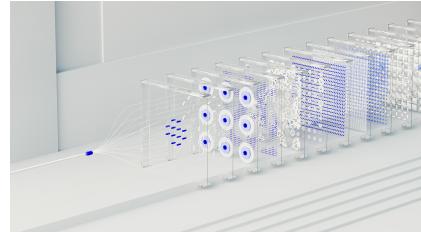
Mašinsko učenje, inspirisano radom neurona, svodi se na izgradnju modela koji su velike kompleksne funkcije koje posmatrajući podatke podešavaju težine parametara i konstante unutar svakog neurona. Ovi modeli se kasnije koriste za rešavanje problema sličnim onima nad kojima su konstruisani. Za razliku od algoritama koji koriste logiku tj. heuristiku kao glavni način pronalaženja rešenja, sistemi



Slika 1: Primeri ponašanja u prirodi.

kao što je mašinsko učenje koriste statistiku koju su stekli posmatranjem velikog broja podataka.

Za rad ovakvih sistema potrebno je dobro analizirati podatake, njihovu kompleksnost, način na koji ćemo rešiti probleme i naravno alate koje koristimo. Svaki problem možemo rešiti na više načina, sa više algoritama, jedino je bitno naći onaj pristup koji najviše odgovara problemu i daje najbolje rezultate.



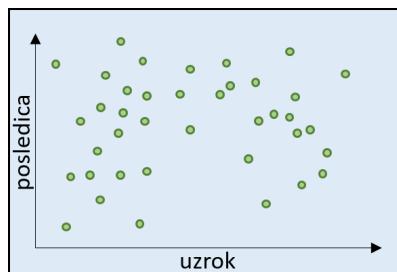
Slika 2: Primer neuronske mreže

1.1 Analitika

Analitika podataka predstavlja **istraživanje, interpretaciju i komunikaciju** između podataka u cilju pronalaženja šablonu u podacima (prilikom istraživanja koristimo statistiku). Ono što analitika obuhvata jeste istraživanje starih podataka radi pronalaženja **trendova** među njima, i kroz te trendove možemo videti uticaj naših odluke ili dešavanja. Želimo da popravimo rezultate tako što učimo nad starim događajima.

Pre svega želimo da znamo da opišemo podatke, jasno je da je onda jedn od glavnih pristupa analiziranje podataka, odnosno njihova vizualizacija. Vizualizacija podataka može da nam pomogne da vidimo zavisnosti podataka, njihove povezаности, kao i da uočimo njihove statističke osobine. Ovaj proces nazivmo **deskriptivna** analitika.

Deskriptivna analitika koristi **deskriptivne alate**. Ovi alati omogućavaju analizu onoga što se već dogodilo ili što se trenutno dešava. Izveštavaju i daju uvid u prošle i trenutne performanse sistema ili poslovanja. Primeri uključuju analize prodaje u prethodnim kvartalima, podatke o prometu na web stranicama, izveštaje o korisničkom zadovoljstvu...



Grafički prikaz podataka

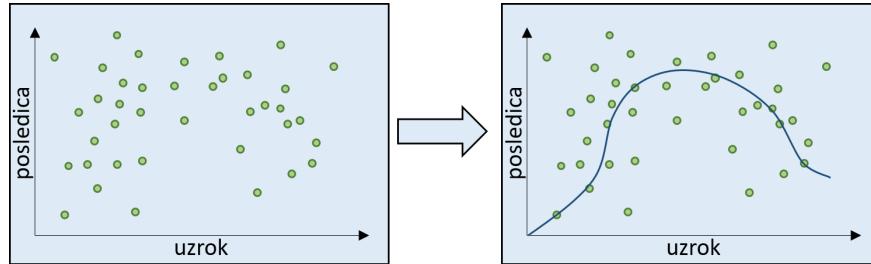
uzrok	posledica
0.13865437	0.81848312
0.27557319	0.08494506
0.31100051	0.38395513
0.48309504	0.55211393
0.50600021	0.27376461
0.66200004	0.50083444
0.74290901	0.64219453
0.89489006	0.70111111
0.92767033	0.82124814
...	...

Tabelarni prikaz podataka

Kada znamo kako podaci izgledaju, to znanje možemo koristiti za predviđanje. Kako podaci predstavljaju neke podatke u prošlosti, računamo da će i ubuduće podaci poštovati sličan trend. Koristeći se tim znanjem možemo praviti modele koji opisuju te podatke, uče nad njima. Za pravljenje modela možemo koristimo linearnu regresiju, ML modele (ANN), CART... Dobijene modele koristimo za predikciju budućih ponašanja, odakle i potiče ime **prediktivna** analitika.

Poput deskriptivne i prediktivne analize ima svoje alate - **prediktivne alate**. Koriste statističke tehnike i algoritme mašinskog učenja za predviđanje budućih događaja ili ponašanja na osnovu istorijskih podataka npr. predviđanje potražnje za proizvodima, prognoze vremenskih uslova, predviđanje kvarova na mašinama... Mana ove analitike jeste u tome da kako podaci stare preciznost samih modela opada, pa je neophodno modele ažurirati na nekoliko godina.

Dakle sada znamo kako podaci izgledaju, kako se ponašaju i imamo alat za predviđanje rešenja, naravno do na grešku. Znajući sve ovo možemo da optimizujemo kvalitet naše proizvodnje, uštedu potrošnje... Ovo postižemo dizajniranjem alata koji može da na osnovu predikcije oformi predlog, odnosno preskripciju za



Slika 4: Uspešno opisujemo podatke

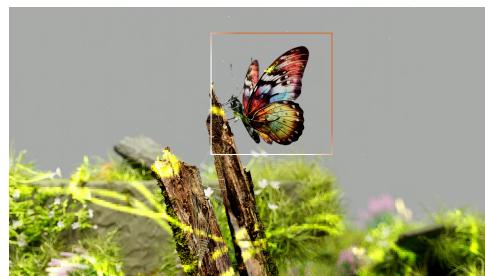
poboljšanje kako bi dobili najbolje rezultate. Ovi alati rade po principu simulacija, nelinearnog, linearog i celobrojnog programiranja... Ova analitika, kao doktor, nam **preskriptuje** lek za poboljšenje rešenja, zdravlja.

Konačno, **preskriptivni alati** idu korak dalje od predikcije jer ne samo da govore šta će se dogoditi, već i sugerisu šta treba preduzeti kako bi se postigli željeni rezultati. Oni koriste optimizacione algoritme i simulacije kako bi preporučili najbolje akcije ili odluke, npr. optimizacija zaliha u trgovini ili kreiranje efikasnih ruta za dostavu.

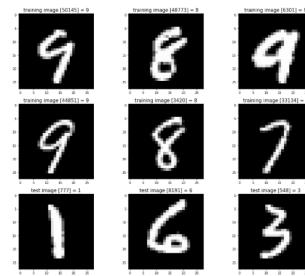
1.2 Odlučivanje

Cilj odljučivanja je dolazak do nekog rezultata ili zaključka.

Nisu sve odluke iste, pre svega razlikuju se po tome da li su rutinske ili nove. Rutiski problemi su, kako sama reč kaže, poznati ili **struktuirani** pa za njih već postoje **programirana** rešenja. Ovakve probleme nazivamo **struktuirani problemi**, a njihove odluke **programirane odluke**. Primer za ovakve probleme je bilo šta što se prethodno rešavalo, dakle prepoznavanje lica, prepoznavanje reči, predviđanje cena nekretnina...



Prepoznavanje leptira na slici



Prepoznavanje brojeva

Novi problemi su nešto sa čime se prethodno nismo sreli, samim tim su kompleksniji ili nejasni..., odnosno **nestruktuirani**. Ovakve probleme odlikuju, ko bi rekao, **neprogramirana** rešenja. Ovakve probleme nazivamo **nestruktui-**

rani problemi, a njihove odluke nazivamo **neprogramiranim odlukama**. Za ovakve probleme primer je bilo šta što je nejasno ili preopširno. Zamislimo da klijent od nas traži da modelujemo AI koji treba da radi "sve"... Šta znači sve? To definiše ove probleme.

Odluke pravimo u svakom slučaju, bitno je samo da smo svesni svog neznanja, odnosno znanja. U situacijama kada možemo da predvidimo posledice i gde možemo napraviti tačne odluke kažemo da smo **sigurni**, odnosno možemo se uzdati u **sigurnost** unapred određenih pravila i predviđenih posledica. Sa druge strane kada nemamo dovoljno informacija, dobijemo loše definisan problem, nismo sigurni, postoji **rizik** prilikom donošenja odluka tj. **nesigurnost** u posledice, te onda možemo samo definisati verovatnoće posledica.

Zaključak je da kada pravimo bilo koji model moramo imati neki način da predvidimo posledice odluke koju ćemo napraviti.

2 Paradigme u veštačkoj inteligenciji

Paradigme veštačke inteligencije možemo podeliti na one koje se koriste različitim heuristikama u cilju pronalaženja rezultata i one paradigmе koje se koriste statistikama.

Oblast veštačke inteligencije je počela kao nauka zasnovana na algoritmima koji oponašaju neke pojave u svetu oko nas. Pa tako svako otkriće u oblasti je bilo pronalaženje nove heuristike tj. algoritma koji bi proširoio načine rešavanja problema.

Danas kada se kaže veštačka inteligencija uglavnom se misli na **velike jezičke modele (LLM - Large Language Models)**, od kojih je najpoznatiji ChatGPT (OpenAI). Po pisanju ove skripte veliku popularnost stiče i DeepSeek zbog ubedljivo manje uloženih novčanih sredstava za izgradnju samog modela, kao i činjenice da je open-source.

Ovakvi modeli rade po principu statistika, odnosno posmatrajući velike količine odabranih podataka grade svoje "znanje", što im omogućava "razumevanj" teksta kao i generisanje teksta. Kompanija OpenAI je izbacila i AI model Sora koji je trenutno vodeći u generisanju video zapisa.

Ovakvi modeli, mada funkcionišu, ne doprinose preterano razvoju ove oblasti. Za razliku od prethodno aktuelnih paradigmа koje su radile po principu algoritama, svakim nastankom novog LLM modela, ne nastaje nikakvo novo otkriće, pošto njihov kvalitet zavisi isključivo od količine i kvaliteta podataka koje su dostupne modelu prilikom njegovog razvića. [1]



ChatGPT



Deepseek

Na ovom predmetu bavimo se i paradigmama koje su zasnovane na logici, posmatranju sveta oko nas kao ali i onim paradigmama koje su zasnovane na statistici i podacima.

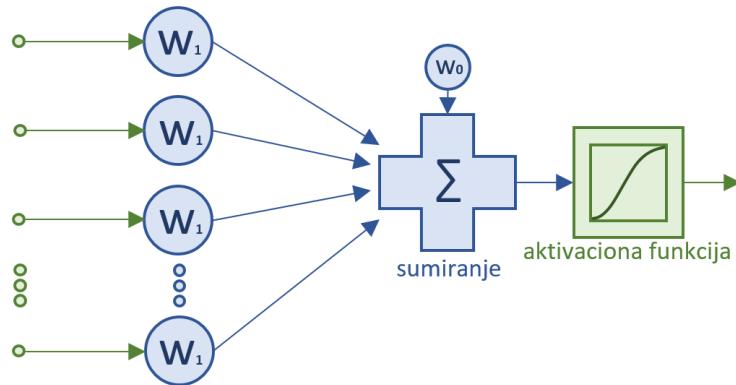
Kao u gore navedenom tekstu, trenutno najpoznatija paradiigma su **Veštačke neuronske mreže (ANN)**. Pored ANN-a bavićemo se i **Evolutivnim izračunavanjima (EC)**, **Inteligencijama grupa (SI)**, **Veštačkim imuni sistemima (AIS)** i **Rasplinutim sistemima (Fuzzy systems)**.

U narednim sekcijama ćemo malo opisati algoritme ali o istim će više biti rečeno kasnije.

2.1 Veštačke neuronske mreže (ANN)

Ove mreže oponašaju način na koji ljudski mozak obrađuje informacije, koristeći veštačke neurone koji obrađuju podatke kroz slojeve. Svaki neuron prima ulazne podatke (npr. podatke iz okruženja ili drugih neurona), množi ih odgovarajućim težinama ($w_1, w_2, w_3 \dots$) i primjenjuje aktivacionu funkciju kako bi odlučio da li će poslati signal i sa kojom snagom. ANN se široko koriste u oblastima kao što su prepoznavanje zvuka (npr. u glasovnim asistentima), prepoznavanje oblika (npr. u računarskoj viziji), i kontrola robota. Spadaju u paradigme statistike i podataka.

Ovi koncepti i alati su ključni za razumevanje savremenih tehnika veštačke inteligencije i mašinskog učenja, pogotovo u kontekstu njihovih praktičnih primena i sposobnosti.



Slika 7: Primer veštačkog neurona

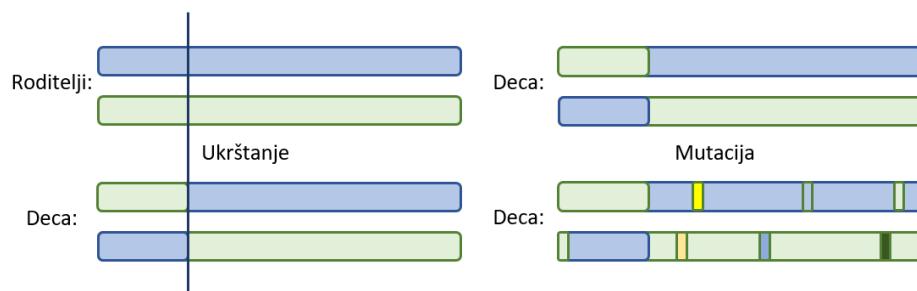
2.2 Evolutivna izračunavanja (EC)

Evolutivna izračunavanja su metoda optimizacije zasnovana na simulaciji prirodne selekcije, gde jedinke koje su bolje prilagođene okruženju "preživljavaju" i prenose svoje "gene" (karakteristike) na sledeće generacije. Oni koji se lošije prilagode ne prenose svoje gene, što omogućava da populacija u celini postane bolja vremenom. Rekombinacija gena preživelih jedinki stvara nova rešenja, simulirajući prirodni evolutivni proces. Na ovaj način, evolutivni algoritmi "uče" kroz generacije kako da dođu do optimalnog rešenja za zadati problem.

Svako evolutivno izračunavanje radi sa skupom potencijalnih rešenja koje formiraju **populaciju**, kvalitet svakog rešenja, jedinke, se izračunava **fitnes funkcijom**.

2.2.1 Genetsko algoritmi (GA)

Genetski algoritmi su najpoznatija forma evolutivnih izračunavanja i funkcionišu tako što predstavljaju jedinke kao nizove binarnih vrednosti (0 i 1), poznate kao "hromozomi". Svaki hromozom predstavlja jedan mogući način rešavanja problema. Algoritam zatim koristi operacije kako bi stvorio nove jedinke u svakoj generaciji. Ove nove jedinke se zatim testiraju pomoću fitnes funkcije, a one koje daju bolje rezultate imaju veće šanse da budu zadržane i dalje evoluiraju.

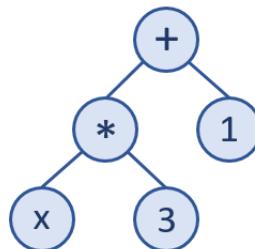


Slika 8: Primeri rekombinacije i mutacije

2.2.2 Genetsko programiranje (GP)

Genetsko programiranje je posebna vrsta evolutivnog algoritma koja ne evoluira nizove brojeva ili binarne kodove kao u genetskim algoritmima, već evoluira **programe**. Svaka jedinka je računski program koji se optimizuje kroz generacije. Cilj je stvoriti program koji na najbolji način rešava zadati problem. Kroz generacije, GP koristi rekombinaciju, mutacije i selekciju kako bi kreirao sve efikasnije programe.

Može se koristiti za automatizovano generisanje koda, kreiranje strategija ili predikcionih modela.



Slika 9: Primer jednog rešenja GP populacije

2.2.3 Evolutivno programiranje (EP)

Evolutivno programiranje se razlikuje od genetskih algoritama po tome što se ne fokusira na rekombinaciju gena, već se promene vrše na individualnim jedinkama kroz stohastičke mutacije. U EP, populacija je skup jedinki koja se optimizuju kroz promenu pojedinih karakteristika tih jedinki.

2.2.4 Koevolucija

Koevolucija opisuje proces u kojem različite populacije evoluiraju u međusobnoj interakciji. Ovo znači da evolucija jedne populacije zavisi od evolucije druge. Primer su predatori i plen u prirodi: kako plen postaje brži i spretniji, tako predatori moraju evoluirati da bi ih i dalje hvatali. U kontekstu algoritama, to može uključivati "parazite" ili ometajuće faktore koji usporavaju ili komplikuju evolucijski proces, čineći zadatak težim za optimizaciju.

2.3 Inteligencija grupa (Swarm Intelligence - SI)

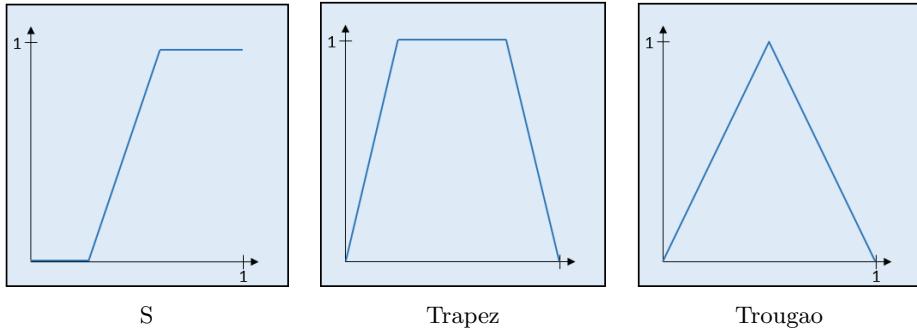
Odnosi se na sposobnost grupa organizama, poput ptica, mrava ili pčela, da pokažu složeno ponašanje kroz jednostavne individualne interakcije. Ovi organizmi funkcionišu prema pravilima kolektivnog ponašanja, gde svaka jedinka donosi jednostavne odluke, ali zajedno postižu inteligentne rezultate bez centralizovane kontrole. U domenu veštačke inteligencije, SI se koristi za optimizaciju rešenja problema za koji je SI prikladan, imitirajući ovu vrstu grupne koordinacije.

2.3.1 PSO

PSO je metoda optimizacije koja se zasniva na kolektivnom ponašanju jata ptica. U ovom algoritmu, ptice (ili čestice) se kreću kroz višedimenzionalni prostor podataka, tražeći optimalno rešenje problema. Svaka čestica u PSO algoritmu prilagođava svoju poziciju na osnovu sopstvenog iskustva i iskustva drugih čestica iz jata. Kako se ptice kreću kroz prostor, privlače jedna drugu ka boljim rešenjima, pa su veće šanse da će se kretati ka optimalnom rešenju. Na sličan način funkcionišu i algoritmi zasnovani na ponašanju mrava (ACO - Ant Colony Optimization), gde mravi tragaju za optimalnim rešenjem problema zasnovanim na lokalnim interakcijama i ponašanju grupe.

2.4 Rasplinuti (fuzzy) sistemi

Ovo su sistemi koji mogu raditi sa nejasnim, nepotpunim ili dvostrukim informacijama. Umesto da koriste striktna pravila kao klasični sistemi, rasplinuti sistemi omogućavaju "sive zone", gde se odluke donose na osnovu verovatnoće ili približnih vrednosti. Logika nije strogo binarna (0 ili 1), već vrednosti mogu biti između 0.0 i 1.0. U ovim sistemima, odluke se donose na osnovu stepena pripadnosti određenim skupovima, što omogućava fino podešavanje odluka i ponašanja u situacijama koje nisu strogo definisane.



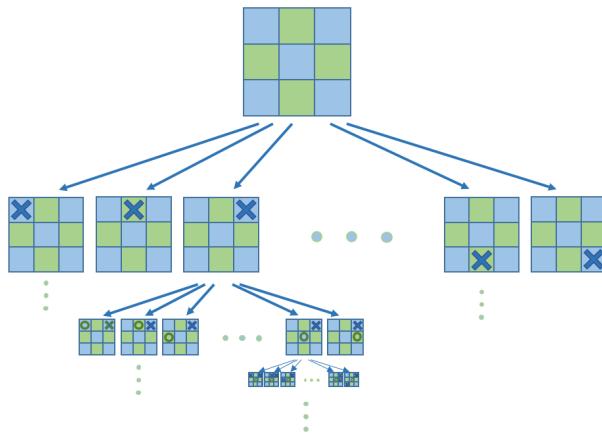
Slika 10: Neke od funkcija korišćenih u fuzzy sistemima

Fazi logika dozvoljava rad sa nepreciznim, nejasnim ili nepotpunim informacijama, simulirajući ljudsko rasuđivanje u takvim uslovima. Koriste se u oblasti kontrole, kao što su pametni termostati ili sistemi za navigaciju.

3 Optimizacioni algoritmi

Veštačka inteligencija se najviše koristi kao alat za pronalaženje rešenja onih problema koji imaju velike prostore pretrage, odnosno problemi koji poseduju veliki broj rešenja.

Ako posmatramo praktičnu primenu, jedan od najboljih primera jeste igra šaha. Poznata **minimax** teorema koju je formulisao Džon fon Nojman u radu na temu teorije strateških igara, je korisna za pronalaženje najboljeg poteza u datom stanju table. Problem kod ovog pristupa jeste da minimax pronalazi najbolji potez tako što pretražuje stablo svih mogućih ishoda igre... Ovo jeste korisno, a pre svega moguće izvesti za igre kao što je igra XO koje sadrže malo stanja. Znajući da šah sadrži 64 polja i 32 figure porpilično je jasno da simulacija ovakve



Slika 11: Pretraga mogućih stanja igre XO.

igre ima stablo ogromnih dimenzija, nemoguće je izvesti usled velike vremenske složenosti. U ovakvim slučajevima koriste se optimizacione metode koje mogu da aproksimiraju ishod igre nakon **n** poteza na osnovu različitih statistika i tabela, i time daju najbolji potez.

Sve probleme koje formiramo, kao i prethodni primer šaha, želimo da optimizujemo. Kada kažemo da želimo da optimizujemo problem, mislimo na pronaženje njegovog minimuma odnosno maksimumu, tražimo najbolje rešenje našeg problema, najbolji potez u igri.

Pa možemo reći da je optimizacioni algoritam specifična vrsta algoritama koji pripadaju grupi pretrage, a njihov glavni cilj je da pronađu optimalno rešenje za datu ciljnju funkciju, koja može biti minimizacija ili maksimizacija, uz određeni skup ograničenja.

3.1 Uvod u optimizacije

Prilikom definisanja bilo kog problema, potrebno je definisati **prostor rešenja** (feasible region), odnosno moramo definisati koja rešenja su dozvoljena koristeći **skup ograničenja**, definišemo zavisnosti između nezavisnih promenljivih.

Prethodno navedeni skup rešenja možemo definisati na različite načine. Neka prva ideja, prirodna, jeste da prosto **odbacimo** rešenja, za vreme pretrage, koja ne ispunjavaju unapred definisana ograničenja. Recimo ako tražimo koliko maksimalno možemo zaraditi izradom stolova ili stolica, rešenje koje nam daje pola stola ili pola stolice nije najoptimalnije, zar ne? Ovo, mada rešava problem, sama akcija odbacivanja rešenja može dovesti do toga da izgubimo neko potencijalno dobro rešenje.

Možda malo bolji način jeste da sačekamo da se generiše skup rešenja, **smanjivanje na bez ograničenja**, a zatim biramo ona rešenja koja ispunjavaju naša ograničenja. Međutim ponovo možemo postaviti pitanje: Šta ako su odbačena rešenja uz malo izmena mogla biti potencijalna najbolja?

Iz prethodnog pitanja intuitivno možemo da dođemo do ideje **popravljanja** rešenja. Mada neko rešenje ne ispunjava ograničenja, recimo tražimo minimum funkcije u nekom domenu i izađemo iz navedenog domena, ne želimo prosto da ga odbacimo već popravimo. Ideja ovde može da bude da ga prosto vratimo nazad u domen, recimo neku ivičnu vrednost ili izvršimo pomeraj u smeru domena. A što dopuštamo da se rešenja koja nam ne ispunjavaju ograničenja pojave kao rešenja?

Ukoliko koristimo **popravljanje** rešenja pre pronalaženja novog rešenja, dakle za vreme pretrage, možemo popraviti rešenja pre nego što se ona dese. Dakle ako vidimo da će dodavanjem vektora na neko prethodno rešenje dati rešenje van domena u kom tražimo minimum, prosto obrnemo smer ili oslabimo intenzitet vektora. Primetimo da ovim procesom **održavamo dopustivost** rešenja.

Još jedan način da potencijalno zaobiđemo nedopustiva rešenja jeste **dodeljivanjem penala**. Tehnika koju ne možemo uvek da izvedemo, generisanje uvek dopustivih rešenja ili **uređivanje dopustivih rešenja**. Kao programeri svesni smo ograničenja, koja mora da ispunjava svako rešenje, zašto onda nebismo samo uvek generisali dopustiva? Ovo je korisno za probleme u kojima imamo diskretan prostor rešenja, međutim za probleme koji su kontinualni je malo teže iskoristiti i više liči na održavanje dopustivosti. Ovde se može ponovo pojaviti isti problem gde ne pređemo na rešenje koje je možda pogrešno, ali iz kod bi kasnije možda došli do optimalnog rešenja.

U primerima navedenim u prethodnom pasosu vidimo da problemi mogu imati različite prostore rešenja. Rad sa problemima koji imaju rešenja u beskonačnim prostorima, kao što su funkcije, nazivamo **kontinualnim problemima**. Dok oni problemi čija se rešenja predstavljaju kombinatornim formulama nazivamo i **diskretnim problemima**, najkraći put u grafu.

Definisanje problema i prostor rešenja je pola posla, sledeći deo jeste osmisli funkciiju kojom možemo da računamo kvalitet pronađenog rešenja.

Ovakva funkcija se naziva i **funkcija cilja** ili **fitness funkcija**, ona mapira prostor rešenja \mathbb{S} u realne brojeve \mathbb{R} . Pa funkciju možemo definisati kao $f : \mathbb{S} \rightarrow \mathbb{R}$,

onda ako je $x \in \mathbb{R}$ što je vrednost fitness funkcije manja, to je rešenje optimalnije. Minimum i maksimum ovakve funkcije može da se svede na isto, pošto važi da je maksimum od f isto što i minimum od $-f$. Fitness funkcija nam pomaže kao alata za poređenje rešenja, kako bi se konstantno kretali ka nama optimalnom rešenju.

Sama fitness funkcija ne mora da bude jedna, već može da bude skup funkcija, kriterijuma. Ovakva optimizacija naziva se i **višeciljna optimizacija**. Uglavnom u složenijim situacijama kao što je ekonomija i transportni problemi, odnosno problemi u kojima se razmatraju različiti faktori, poput troškova, vremena, kapaciteta i kvaliteta usluge.

Način na koji možemo da rešimo probleme sa više funkcija cilja, jeste da **agregiramo**, sumiramo, više ciljeva u jednu jedinstvenu funkciju cilja. Ovaj pristup uključuje odmerivanje, dodeljivanje važnosti određenim funkcijama cilja koje se zatim sabiraju i daju neku vrednost. Proces u kom dodeljujemo važnosti naziva se **ponderisanje**, a ceo pristup **pravljenje ponderisanih proseka (agregacija)**. Da bi ovaj proces bio uspešan neophodno je dobro poznavanje oblasti problema.

Drugi pristup jeste da poboljšavamo pojedine funkcije. Ideja je da pravimo rešenja u kojima poboljšanje bilo koje funkcije cilja kvari kvalitet druge. Ovakva rešenja se nazivaju i **pareto-optimalna**^[3]¹ rešenja. Skup ovakvih rešenja naziva se **Pareti front (Pareto kriva ili površ)** koja prikazuje odnose funkcije (optimalne), odakle se biraju ona rešenja koja nam najviše odgovaraju.

3.2 Optimizacioni algoritmi

Konačno kada znamo kako da definišemo prostor rešenja kao i kako da kvantifikujemo kvalitet pojedinih rešenja, možemo se posvetiti samim algoritmima. Optimizacioni algoritmi nam omogućavaju da nađemo što bolja rešenja postavljenih problema.

Svi algoritmi se razlikuju po principu (**Metaheuristici**²) nalaženja sledećeg rešenja, ali rade po identičnom principu.

Algoritam 1 Opšti pseudokod optimizacionih algoritama

```
while nije ispunjen određeni cilj ili kriterijum do
    Izračunaj vrednost
    Izračunaj pravac i smer pretrage
    Izračunaj dužinu koraka pretrage
    Predi u naredno rešenje
end while
```

¹**Pareto-optimalna rešenja** ime dobijaju po Pareto principu koji navodi da u mnogim ishodima, otprilike 80% posledica dolazi iz 20% uzroka.

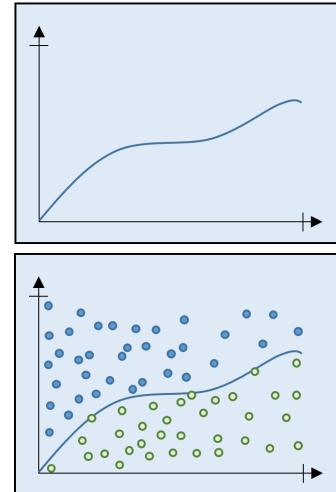
²**Metaheuristike** predstavljaju skup opštih metoda koje pružaju okvir za rešavanje optimizacionih problema, često na osnovu stohastičnih, odnosno nasumičnih, procesa.

Kao što vidimo u datom pseudokodu ponavljamo neke akcije dok se ne ispunji cilj, kriterijum. Kriterijum može biti vreme obrade, broj iteracija ili neka predefinisana greška... Na početku svake iteracije prvo se izračunava vrednost funkcije cilja za trenutno rešenje, na osnovu koje se određuje pravac u kojem će se nastaviti pretraga. Zatim se definiše koliko daleko se ide u odabranom pravcu i na osnovu prethodnih koraka, prelazi se na sledeće rešenje u pretrazi. Bitno je znati da ovi algoritmi neće uvek vratiti najbolje rešenje **globalni minimum**, odnosno **globalni maksimum**. Najrealnije je očekivati da pronađu vrednosti kao što su **lokalni minimum**, odnosno **lokalni maksimum**. U mnogim slučajevima, algoritmi optimizacije mogu naići na lokalne minimume koji nisu globalni, što može otežati postizanje optimalnog rešenja.

Zaključujemo da je jedna od osobina po kojim se razlikuju ove metode njihova tačnost. Iz ove osobine možemo izdvojiti dve vrste rešenja na osnovu njihove **tačnosti**:

- **Egzaktno rešavanje:** Ovim pristupom se dolazi do garantovano optimalnog rešenja za problem. Koriste se algoritmi koji pružaju tačan odgovor, međutim je vremenski zahtevno za velike probleme.
- **Približno rešavanje:** Približni algoritmi, često bazirani na (meta)heuristikama, pružaju rešenja koja nisu nužno optimalna, ali su dovoljno dobra u praktici. Ovi pristupi su korisni kada je problem previše složen za egzaktno rešavanje u razumnom vremenu (NP ili eksponencijalne složenosti).

Jasno je da ove algoritme koristimo u cilju skraćivanja pretrage velikih prostora rešenja. Odakle se izdvaja još jedna osobina, odnosno **pristup pretrazi**. Za vreme pretraga koristimo se predefinisanim pravilima na osnovu kojih gradimo rešenja ili znamo u kom smeru nastaviti pretragu. Primer može biti pretraga minimuma funkcije koristeći se gradijentom te funkcije, na osnovu čega znamo u kom smjeru je lokalni minimum (više o ovom kasnije). Ovaj pristup naziva se **determinističkim** i zavisi od predefinisanih pravila, u istim uslovima dolazi uvek do istih rešenja. Deterministički pristup se ogleda u osobini **intenzifikacije (eksploracija, eng. exploitation)**, time što se fokusira na pretragu oko trenutnog dobrog rešenja. Možemo uočiti problem lokalnih minimuma, gde ukoliko koristimo intenzifikaciju a upali smo u lokalni minimum iz istog se nećemo izvući, u većini slučajeva. Baš iz tog razloga uvodimo nasumičnost u algoritme, **stohastički** pristup (poput Monte Karlo³). Osobina ovakvog pristupa jeste da ponekad ne prate strogo definisana pravila.



Slika 12: Monte Karlo

Baš iz tog razloga uvodimo nasumičnost u algoritme, **stohastički** pristup (poput Monte Karlo³). Osobina ovakvog pristupa jeste da ponekad ne prate strogo definisana pravila.

³Monte Karlo metoda računa površinu ispod krive izborom nasumičnih tačaka. Unutar

Ovo je pomalo kontraintuitivno, zašto smo pisali pravila ako ih ne poštujemo? Odgovor leži u problemu prethodnog pristupa. Postoji *šansa* da ako ne ođemo u optimalnom smeru slučajno nađemo na rešenje iz kog dolazimo u novi lokalni minimum, potencijalno bolji. Odnosno, možemo se izvući iz lokalnih minimuma u nadi da pronađemo bolje rešenje van oblasti prethodno pronađenih dobrih rešenja. Proces koji smo opisali se naziva **diverzifikacija (istraživanje, eng. exploration)**, povremeno skreće sa optimalnog puta kako bi istražio nova rešenja, uveo nove mogućnosti.

Ova dva pristupa, mada definisana odvojeno uvek se pojavljuju zajedno radi optimizacije pretrage. Neće se uvek pojavljivati ista količina intenzifikacije i diverzifikacije u svakom algoritmu.

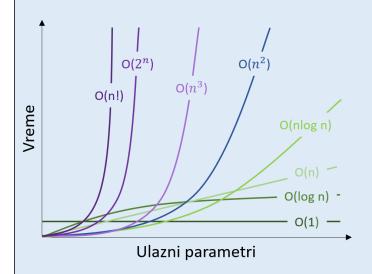
Još jedna razila jeste u **širini pretrage**. Kao što i sama reč kaže dele se po širini prostora koji posmatraju prilikom pretrage. Pa se dele na one koji posmatraju samo svoju okolinu, tj. **lokalne pretrage**, i one koji posmatraju neki veći podprostor rešenja, tj. **globalne pretrage**.

3.3 Korist optimalnih algoritama

Kada dođe do problema visoke vremenske složenosti, želimo da se koristimo optimizacionim algoritmima. Oni omogućavaju da vreme pretrage velikih prostora rešenja skrate, time što ne proveravaju svako rešenje već, koristeći se eksploracijom i eksploracijom, *pametnije* prolaze prostor rešenja i zaustavljaju se u vremenu koje ne prelazi više stotina godina :). Vremenska složenost algoritma opisuje kako se vreme izvršavanja algoritma menja sa veličinom ulaznih podataka. Oznaka vremenske složenosti izražava se sa **O notacijom**. **O notacija** se koristi za izražavanje gornje granice složenosti algoritma i omogućava klasifikaciju algoritma prema njihovoj efikasnosti.

Na osnovu vremenske složenosti, problemi koji su najproblematičniji su:

- **Eksponencijalni problemi** - Vreme izvršavanja raste eksponencijalno sa veličinom ulaznih podataka. Primer eksponencijalnog problema je „pronalazak svih razapinjajućih stabala u potpunom grafu sa n čvorova“ ili „traženje najboljeg poteza u igri šaha“. Eksponencijalni problemi su računski vrlo zahtevni i obično nisu rešivi u razumnoj vremenu za velike ulaze.
- **Faktorijalni problemi** - Slično, vreme izvršavanja raste faktorijalno sa veličinom ulaza.



Slika 13: Vremenske složenosti

određenog pravougaonika, izračuna se procenat tačaka ispod krive i znajući površ pravougaonika možemo procentualno odrediti površinu ispod krive.

Zanimljiva priča koja može da nam malo približi veličinu problema eksponencijalne prirode jeste priča o **pirinču i šahovskoj tabli**.



1	2	4	8	16	32	64	128
256	512	1024	2048	4096	8192	16384	32768
65.5k	131k	262k	524k	1.05M	2.10M	4.19M	8.39M
16.8M	33.6M	67.1M	134M	268M	537M	1.07G	2.15G
4.29G	8.59G	17.2G	34.4G	68.7G	137G	275G	550G
1.1T	2.2T	4.4T	8.8T	17.6T	35.2T	70.4T	141T
281T	563T	1.13P	2.25P	4.5P	9.01P	18P	36P
72.1P	144P	288P	576P	1.15E	2.31E	4.61E	9.22E

Slika 14: Ilustracija pirinča i šahovske table

Priča pominje izumitelja koji od kralja traži da dobije jedan pirinač za prvu ćeliju na šahovskoj tabli, dva za drugu, četiri za treću, osam za četvrtu... Kako kralj nije razumeo veličinu ovog problema, misleći da je zahtev mali odlučio je da ispuni želje izumitelja. Vrlo brzo saznaje da u kraljevstvu nema dovoljno pirinča.

Ovaj problem je, kako smo već pomenuли, eksponencijalne prirode $O(2^{n-1})$. Kada posmatramo formulu vidimo da za niske brojeve kao što su 1, 2, 3, 4... do 15 (ne znam ko bi brojao toliku količinu pirinča), dok dođemo do 20-te ćelije pričamo o milionima zrnavlja. Ako bi nastavili i došli do 64-te ćelije morali bi da imamo 9 kvantiliona zrnavlja (10^{18})... Zbir svih ovih ćelija je otprilike 18.4 kvantiliona zrnavlja koliko nije proizvedeno u celoj istoriji svetra.

Sada možda malo jasnije vidimo zašto su nam tehnike kao što su optimizacije potrebne za rešavanje problema ovakve složenosti.

Treba pomenuti i nedeterminističke polinomske probleme. **NP problemi** su problemi za koje je moguće proveriti rešenje u polinomijalnom vremenu, ali nije jasno da li ih možemo i rešiti u polinomijalnom vremenu za svaki slučaj. Za takve probleme nije poznato da li se mogu tačno rešiti u polinomijalnom vremenu.

NP-potpuni problemi su specifični NP problemi za koje važi da, ako možemo napraviti algoritam koji ih rešava u polinomijalnom vremenu (P složenosti), svi NP problemi mogu se rešiti u polinomijalnom vremenu (ovo najverovatnije nije moguce, neki smatraju da su klase P i NP odvojene klase). NP-potpuni problemi su istovremeno najteži problemi u NP klasi i sinonim za NP-teške probleme.

4 Linearno programiranje (LP)

U sekciji uvoda u optimizacije (3.1) naveli smo obavezno definisanje prostora rešenja. U ovoj sekciji se bavimo tehnikom optimizacije kojom pronalazimo optimalno rešenje koristeći više ograničenja.

Linearno programiranje (LP) predstavlja optimizaciju linearne funkcije sa ograničenjima koja su izražena kao nejednakosti. Na primer, farmer ima ograničene resurse za sadnju povrća, linearno programiranje može mu pomoći da maksimizuje profit unutar tih ograničenja.

Samo linearno programiranje je uhvatilo maha tek nakon Drugog svetskog rata. Mada, pomenuto od strane poznatih matematičara od kojih je jedan **Žozefa Furijea** (poznat po Furjeovom redu), njihovo doprinos je svega po jedan rad. Najveći doprinos razvoju dali su **Vasilij Leontijev**, u radu gde je modelirao celu ekonomiju koristeći linearno programiranje, dok je **Džon fon Nojman** uočio da njegova **minimax teorema** iz teorije igara već koristi principe linearног programiranja i uspešno pokazao da svaki LP problem ima svoj **dualni problem** (više o ovome kasnije).

Kada je LP steklo popularnost olakšano je predstavljanje nekih problema iz realnog sveta matematičkim formulama, a način rešavanja ovih problema pokazuje tačan redosled akcija da se postigne najbolji ishod. [2]

4.1 Pogodan region (Feasible Region)

Naveli smo da se sve predstavlja matematičkim formulama, kako onda definišemo probleme? Za početak definišimo prostor rešenja ili **pogodan region** (eng. **feasible region**).

Pogodan region predstavlja prostor rešenja dobijen presecima ograničenja, koji čine prostor u kojem pretražujemo optimalna rešenja. Kod problema sa više promenljivih, ovaj prostor se nalazi u višedimenzionalnom prostoru. Minimalna ili maksimalna vrednost ciljne funkcije će se sigurno naći u nekoj od graničnih tačaka ovog regionala, kao što su teme, ivice ili stranice.

Znajući kako da definišemo pogodni region, jasno je da rešavanje ovih problema možemo predstaviti geometrijski. Naravno ovo je korisno isključivo radi vizualizacije i samo u nižim dimenzijama.

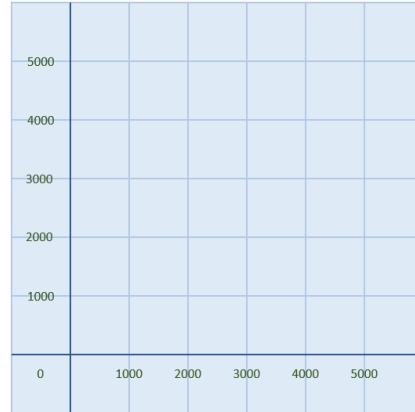
4.2 Geometrijski princip rešavanja

Iscrtamo prostor, ograničimo ga funkcijama i zatim definišemo linearnu funkciju koja predstavlja cilj.

Za pronađenje optimalnog rešenja linearna funkcija (npr. $ax + by = c$) se "pomeranjem" kroz pogodan region (menjanjem vrednosti konstante **c**) postavlja u najvišu ili najnižu tačku gde dotiče region. Kada funkcija prestane da se pomera unutar ovog prostora, dostigli smo optimalno rešenje.

Kako bi bio što jednostavniji geometrijski prikaz, držaćemo se jednostavnog problema farmera. Problem je definisan u uvodu:

Farmer ima ograničene resurse za sadnju povrća, linearno programiranje može mu pomoći da maksimizuje profit unutar tih ograničenja. Farmer može posaditi maksimalno 4t šargarepa i 3t krompira, ali ima 5t đubriva (može posaditi 5t povrća). Kako da zaradi najviše?



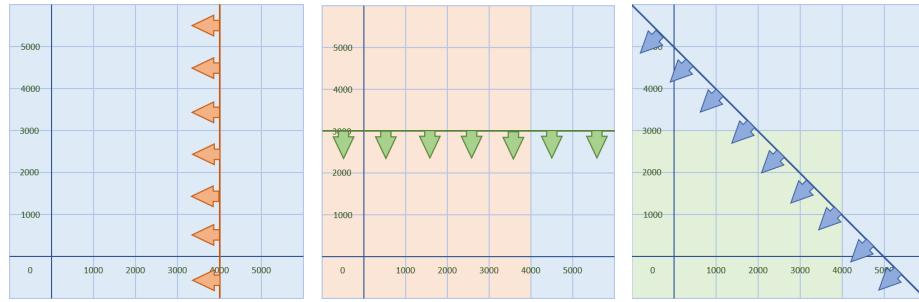
Slika 15: Neograničen prostor

Vidimo da ne može zasaditi više od 4t šargarepa, a ne više od 3t krompira. Ovo su jednostavne nejednakosti:

$$\begin{aligned}x &\leq 4000 \\y &\leq 3000\end{aligned}$$

Kada posmatramo ograničenje djubriva vidimo da je to ustvari ograničenje baš šargarepa i krompira, odnosno njihovog zbira. Pa:

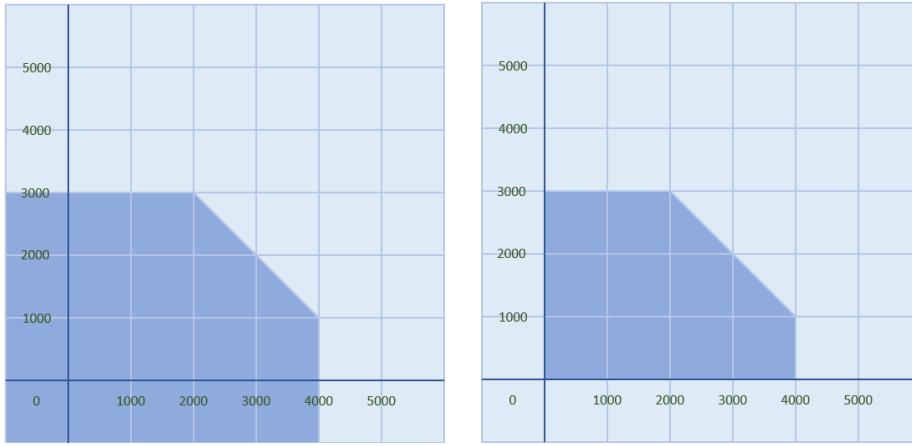
$$x + y \leq 5000$$



Ograničenja šargarepa Ograničenje krompira Ograničenje đubrivom

Slika 16: Definisanje pogodnog regiona

Na slici vidimo da je zapravo pogodan region bas presek ove tri boje. Racunajući samo ono u pozitivnom delu, ne može se posaditi negativan broj povrća...

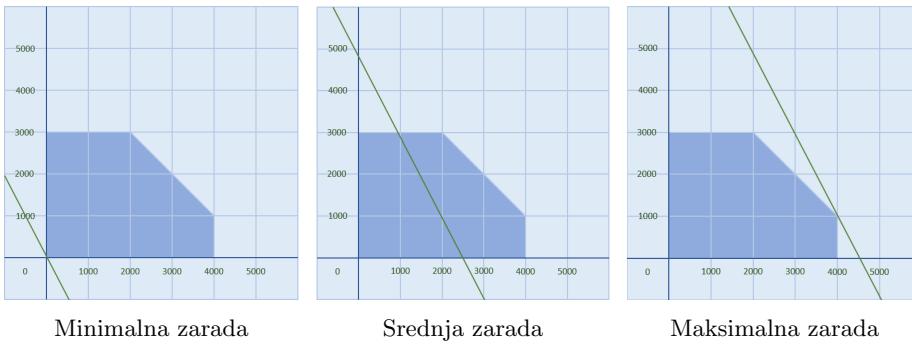


Slika 17: Definisanje pogodnog regionala

Računamo da sve proivedeno će biti prodato. Neka je cena šargarepa a , a cena krompira b . Odavde funkcija koja predstavlja zaradu farmera izgleda:

$$ax + by = c$$

Pomeranjem funkcije $ax + by = c$, uvećavanjem konstante c nalazimo maksimum. Minimalno rešenje za $c = 0$, a maksimum za $c = 8000$. Na ovom primeru se primećeju osobina kojusmo naglasili na početku. Kada pomeramo funkciju maksimum će sigurno biti teme pogodne oblasti ili čak cela ivica, stranica... (zavisi od dimenzije). U ovom primeru to je baš jedno od temena.



Slika 18: Neka rešenja

Međutim ukoliko je problem složeniji, ima više promenljivih i ograničenja, geometrijski pristup nije efikasan. U takvim slučajevima se koristi **Simplex metod**.

4.3 Simplex metod

Simplex metod optimizuje linearu funkciju pomerajući se po ivicama pogodne oblasti, od temena do temena. Na osnovu svojstva da se optimum nalazi u temenima (ili na ivicama). Simplex postupno menja vrednosti promenljivih, identifikujući onu promenljivu koja će poboljšati vrednost funkcije. Promenljive koje se menjaju nazivamo "popuštene" (loose ili non-basic), dok su one koje ostaju nepromenjene "zategnute" (tight ili basic).

Simplex algoritma:

- Definišemo linearu funkciju tj. funkciju cilja (promenljive se predstavljaju sa x i dodeljenim indeksom) i ograničenja.

Funkcija cilja:

$$f(x_1, x_2, \dots, x_n) = ax_1 + bx_2 + \dots + nx_n$$

Ograničenja:

$$\begin{aligned} x_1 &\geq k_1 \\ x_2 &\leq k_2 \\ &\dots \\ x_n &\geq k_n \end{aligned}$$

- Svakom ograničenju dodeljujemo novu promenljivu, čime svodimo nejednakosti na jednakosti.

$$s_1, s_2, \dots \geq 0$$

Na početku sve s-promenljive su popuštene (loose), i stoje same sa leve strane jednačine.

$$\begin{aligned} s_1 &= k_1 - x_1 \\ s_2 &= k_2 - x_2 \\ &\dots \\ s_n &= k_n - x_n \end{aligned}$$

- Ponavljamo sledeće korake:

- (a) Biramo x-promenljivu sa najvećim koeficijentom u funkciji cilja i "popuštamo" je. Ona promenljiva sa većim koeficijentom dovodi rešenje brže do njenog maksimuma, zbog čega se i bira.
- (b) Biramo koju s-promenljivu ćemo "zategnuti" na osnovu razmera konstanti u ograničenjima onih funkcija kod kojih se pojavljuje popuštena x-promenljiva. Biramo onaj s čija jednačina sadrži manju konstantu. Razlog su ograničenja, moramo poštovati da su s-promenljive pozitivne. U primeru će biti napomenuto.

- (c) Ubacujemo dobijenu vrednost promenljive u funkciju i ponavljamo proces dok svi koeficijenti uz promenljive ne postanu negativni.

4.3.1 Primer primene simplex algoritma – Farmer

Farmer može posaditi šargarepu (4 tone) i krompir (3 tone), ima 5 tona đubriva (može posaditi maksimalno 5 tona povrća). Krompir donosi 1.2 evra po kilogramu, dok šargarepa donosi 1.7 evra. Cilj je maksimizacija profita:

1. Definišemo funkciju cilja:

$$f(x_1, x_2) = 1.7 * x_1 + 1.2 * x_2$$

I ograničenja:

$$\begin{aligned} x_1 &\leq 4000 \\ x_2 &\leq 3000 \\ x_1 + x_2 &\leq 5000 \end{aligned}$$

2. Tight: x_1, x_2 :

$$\begin{aligned} x_1 + s_1 &= 4000 \\ x_2 + s_2 &= 3000 \\ x_1 + x_2 + s_3 &= 5000 \end{aligned}$$

S-promenljive su popuštene pa:

$$\begin{aligned} s_1 &= 4000 - x_1 \\ s_2 &= 3000 - x_2 \\ s_3 &= 5000 - x_1 - x_2 \end{aligned}$$

Početna postavka problema je:

$$\max(1.7 * x_1 + 1.2 * x_2)$$

$$s_1 = 4000 - x_1 \tag{1}$$

$$s_2 = 3000 - x_2 \tag{2}$$

$$s_3 = 5000 - x_1 - x_2 \tag{3}$$

3. (a) U formuli $\max(1.7 * x_1 + 1.2 * x_2)$, najveći koeficijent ima x_1 , pa prelazi u loose skup, posmatramo u kojim fomrulama se pojavljuje.

- (b) Vidimo da se x_1 pojavljuje u 1. i 3., upoređujemo razmere $-4000 \geq -5000$, dakle s_1 prelazi u tight skup.
 Recimo da uzmemo veću konstantu 5000, ovu konstantu možemo zameniti umesto popuštene x-promenljive tj. x_1 , čime dolazi do dobijanja negativne vrednosti s_1 . Ovo ne poštuje ograničenja da su s-promenljive pozitivne.
- (c) Raspoređujemo opet tight i loose elemente na predodređene strane jednačine, odnosno:

$$x_1 = 4000 - s_1$$

Zamenimo je u sve ostale:

$$\max(-1.7 * s_1 + 1.2 * x_2 + 6800)$$

$$x_1 = 4000 - s_1$$

$$s_2 = 3000 - x_2$$

$$s_3 = 1000 + s_1 - x_2$$

Kako idalje imamo pozitivne koeficijente u funkciji ponavljamo 3. korak.
 Sada je najveći koeficijent uz x_2 on prelazi u skup loose. Upoređujemo razmere $-3000 \leq -1000$, prebacujemo s_3 u tight i dobijemo

$$x_2 = 1000 - s_1 - s_3$$

Zamenom u ostale funkcije:

$$\max(-2.9 * s_1 - 1.2 * s_3 + 8000)$$

$$x_1 = 4000 - s_1$$

$$s_2 = 2000 - s_1 + s_3$$

$$x_2 = 1000 + s_1 - s_3$$

Kako su obe konstante negative znamo da ako nastavimo algoritam brojevi će se množiti sa negativnom konstantom i oduzimaće se od trenutne konstante umutar funkcije koju maksimizujemo (8000), pa ćemo naći samo manja rešenja.

Da bi našli traženu maksimalnu vrednost dovoljno je da zamenimo s_1 i s_3 sa 0, dobijamo da je **zarada = 8000**. Ako nas zanimaju vrednosti x_1 i x_2 dovoljno je da zamenimo s_1 i s_3 sa 0 u ograničenjima koje smo dobili, $x_1 = 4000$ i $x_2 = 1000$.

Što je i tačno, pogledati maksimalno rešenje geometrijskim pristupom 4.2.

Kod linearnih programa, nije poželjno koristiti prevelike ili premale brojeve za izražavanje vrednosti. Najbolje je da su veličine u sličnom opsegu, odnosno da

su sve veličine u približno istog reda veličine. Ne želimo da šargarepe merimo u tonama a krompir u miligramima.

Postoji i prošireni pristup poznat kao **Celobrojno linearne programiranje (Integer Linear Programming tj. ILP)**, koji dozvoljava samo celobrojne vrednosti, što može biti korisno u praktičnim situacijama gde su delimična rešenja nemoguća.

4.3.2 Dualni problem

Bitno je napomenuti da nema svaki problem jedno rešenje i uvek postoji **dualni problem**, koji smo pomenuili u uvodu. Od prethodno postavljenog problema pravimo novi koji nam daje gornje ograničenje rešenja (koliko maksimalno para možemo da dobijemo a da su uslovi ispunjeni), rešenje dualnog problema će dati isto rešenje kao prvo postavljen problem.

Za gore navedena ograničenja tražimo takvo y_1 , y_2 i y_3 da kada pomnože redom ograničenja dobijemo vrednosti koje u zbiru nam govore da tražena maksimalna vrednost ne može biti veća od dobijene.

Moraju biti nenegativni, i posmatrajući ograničenja predstavljamo x_1 preko:

$$y_1 + y_3 \geq 1.2$$

pošto je to baš vrednost koja стоји поред x_1 .

Slično za x_2 pravimo novo ograničenje:

$$y_2 + y_3 \geq 1.7$$

Ovo su nam nova ograničenja a funkciju više ne želimo da maksimizujemo već minimizujemo. Funkcija menja oblik:

$$\min(3000 * y_1 + 4000 * y_2 + 5000 * y_3)$$

5 Celobrojno programiranje (ILP)

Već napomenuta vrsta problema kada ne želimo delimična rešenja... Šta ako se bavimo izradom stolica i stolova, koristeći gore LP način možemo da dobijemo rešenje koje nam donosi maksimalan profit ali izradom pola stolice.... Ovo nema smisla, zbog toga postoji **celobrojno programiranje (Integer linear programming)** koje ograničava da su rešenja u skupu prirodnih brojeva.

5.1 Pogodan region ILP (ILP Feasible region)

Slično kao kod LP-a, pravljenje pogodnog regiona se ponovo ograničava funkcijama ali ima dodatno ograničenje celobrojnih rešenja. Gledamo ovo kao dodatno ograničenje u svakom zadatku.

Ovi regioni izgledaju kao skup tačaka, od kojih svaka tačka predstavlja neko rešenje.

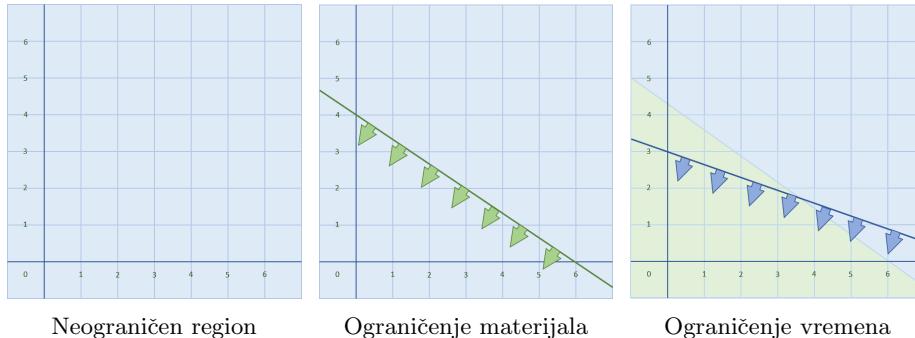
5.2 Geometrijski princip rešavanja ILP

Funkcija cilja se definiše isto kao u LP-u. Radi prikazivanja korisnosti ILP pristupa koristimo malo drugačiji primer, mada je funkcija cilja poprilično slična... Rešavamo problem stolica i stolova. Znamo da stolice vrede 20, stolovi 50 (zane-marimo valutu). Izrada stolica troši 10 jedinica drveta, a stolova 15. Vremenski stolice se prave 2 sata a stolovi 5 sati. Koja je maksimalna zarada ako imamo 15 sati i 60 jedinica drveta za izradu.

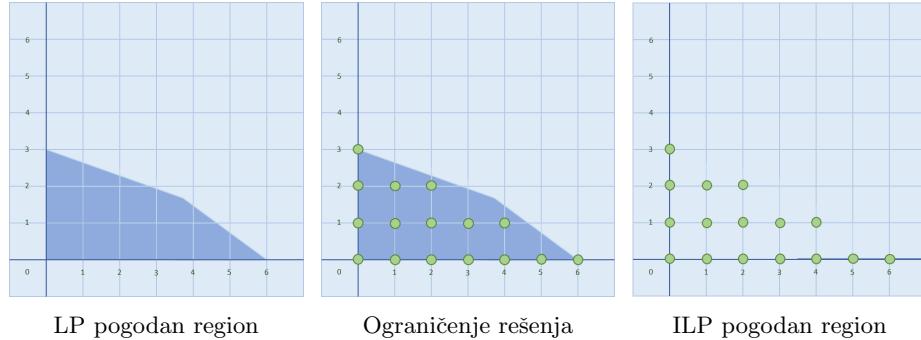
Predstavimo funkciju koju želimo da maksimizujemo kao $f(x_1, x_2) = c$, gde je c konstanta, i samo prevlačimo funkciju po regionu dok ne dotaknemo poslednu pogodnu tačku. U našem primeru ova funkcija je $20 * x_1 + 50 * x_2 = c$

Prvo moramo definisati pogodni region našeg problema. Slično LP-u prvo definišemo prostor koristeći nejednakosti. U ovom primeru to je ograničenje materijala i ograničenje vremena, redom:

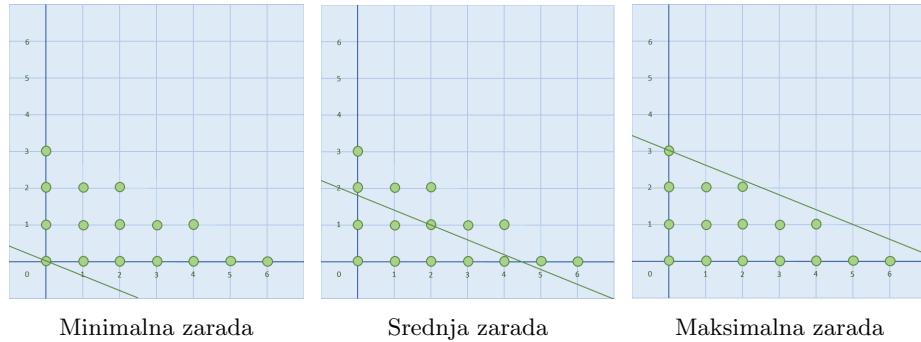
$$10 * x_1 + 15 * x_2 \leq 60 \\ 2 * x_1 + 5 * x_2 \leq 15$$



Na dobijeni LP pogodni region primenjujemo ograničenje celobrojnih rešenja i dobijamo:



Tačke predstavljaju sva dopuštena rešenja. Najbolje rešenje se pronađe preveličanjem funkcije cilja dok je u pogodnom regionu. Najbolje je ono koje je poslednje dodirivala.



Na osnovu Grafika vidimo da najveću zasadu imamo ako proizvedemo samo stolove, zasadu je 150.

5.2.1 Problem ranca

Imamo 8 različitih objekata svaki ima svoju težinu (4, 2, 8, 3, 7, 5, 9, 6) i vrednost (19, 17, 30, 13, 25, 29, 23, 10). Imamo ranac koji može da nosi maksimalno 17kg. Naš zadatak je da nađemo maksimalnu zasadu, ali da ne pređemo ograničenje težine koju ranac može da nosi.

Za ovaj problem možemo da koristimo binarne promenljive (jesmo li uzeli ili ne objekat). Postavka zadatka je sledeća:

- Ograničavamo svaku promenljivu da može biti 1 ili 0 koristeći sledeća dva ograničenja.

$$0 \leq o_1, o_2 \leq 1$$

Slično zapisujemo za svaki objekat.

$$0 \leq o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8 \leq 1$$

2. Pišemo ostala ograničenja kao što bi inače pisali u LP. Ograničenje težine:

$$4 * o_1 + 2 * o_2 + 8 * o_3 + 3 * o_4 + 7 * o_5 + 5 * o_6 + 9 * o_7 + 6 * o_8 \leq 17$$

3. I konačno funkcija koju želimo da maksimizujemo.

$$\max(19 * o_1 + 17 * o_2 + 30 * o_3 + 13 * o_4 + 25 * o_5 + 29 * o_6 + 23 * o_7 + 10 * o_8)$$

Radi lepšeg ispisa možemo sve ove promenljive definisati vektorima.
Problem je teško vizualizovati, zbog višedimenzione prirode problema.

6 Nelinearno programiranje (NLP)

Kada se bavimo problemima koji se definiše kao nelinearne funkcije, ili ograničenja pogodnog regiona su definisana kao nelinearne funkcije, tada se bavimo **nelinearnim programiranjem**. Sa druge strane sam princip zadavanja problema i njihovih ograničenja je isti kao kod LP i ILP, jer sam solver rešava ovo umesto nas. Problem kod ovih sistema jesu lokalni optimumi, jer sam NLP solver može da se zaustavi na lokalnom optimumu. Početna pozicija utiče na rešenje pa nekad može vratiti rešenje koje je lošije ili bolje od prethodnog pokretanja koda.

Kada NLP solver nadje rešenje to rešenje ne mora biti optimalno!
Načini za rešavanje ovakvih probelma može biti analitički. Koristeći se znanjem izvoda možemo pronaći minimum/maximum neke funkcije. Koliko god analitički način bio dobar on ne može da radi uvek. Neke funkcije mogu biti mnogo kompleksnije za rešavanje. U tim situacijama koristimo se numeričkim metodama.

6.1 Polovljenje intervala (Bisection Method)

Jedan od osnovnih algoritama pronalaženja nule neke nepoznate funkcije jeste **Polovljenje intervala**. Ko god se bavio programiranjem čuo je za **binarnu pretragu**. Polovljenje intervala ima identičan proces pretrage prostora rešenja. Ono što je prirodno da pitamo kada koristimo binarnu pretragu jeste: "Da li je skup na kom pretražujemo sortiran?". Jasno je ovde ne moramo postaviti ovo pitanje, prostor je sortiran... Ali moramo odrediti koji je to prostor rešenja $[x_l, x_r]$ na kom tražimo nulu funkcije. Najjednostavniji način, ukoliko nula postoji, jeste da tražimo onaj deo funkcije gde znamo da funkcija sigurno preseca x-osu. Dakle rešenje funkcije ima drugačiji znak na oba kraja domena!

$$f(x_l)f(x_r) < 0 \quad (1)$$

Ovakav prostor želimo da održavamo kroz celu pretragu.

Kad znamo da se nula nalazi sigurno u našem intervalu, prepostavimo da je ona baš u centru domena.

$$x_0 = \frac{x_l + x_r}{2} \quad (2)$$

Pravljenje intervala koji odmah ima nulu funkcije baš na sredini intervala koji smo odabrali je redak pa moramo da znamo kako tačno da nastavimo pretragu kada nismo našli nulu funkcije... ipak, ukoliko čitalac uspe iz prve da slučajno nađe takav interval, predlažem da odmah ode i kupi loto listić. Šala na stranu, napomenuto je da želimo da održimo domen koji poštuje formulu (1). Logično je, posmatramo znak funkcije u centru domena x_0 i biramo neki od domena $[x_l, x_0]$ ili $[x_0, x_r]$.

Sada imamo sve što nam je potrebno da odredimo samu nulu funkcije, stajemo onda kada je formula (1) = 0.

Alternativa i mnogo jeftinija vremenski jeste korišćenje aproksimirane greške

$|\epsilon_a|$. Rećićemo da prihvatamo samo grešku koja je ispod nekog procenta i radimo algoritam sve dok greška nije manja ili jednaka izabranom procentu.

$$|\epsilon_a| = \left| \frac{x_0^{new} - x_0^{old}}{x_0^{new}} \right|$$

Možemo primetiti da je ideja algoritma da interval u kom se nalazi nula konstantno smanjujemo, ali pritom da samu nulu ne izgubimo, ceo interval konvergira ka rešenju. Odavde zaključujemo:

$$a_0 \leq x_0 \leq b_0$$

$$a_1 \leq x_0 \leq b_1$$

$$a_2 \leq x_0 \leq b_2$$

...

$$a_n \leq x_0 \leq b_n$$

Iz niza intervala možemo izgraditi sledeće:

$$a_0 \leq a_1 \leq a_2 \leq \dots \leq a_n \leq x_0 \leq b_n \leq \dots \leq b_2 \leq b_1 \leq b_0$$

Vidimo da imamo dva opadajuća niza sa obe strane x_0 (neka je x_0 prava nula funkcije), samim tim konvergiraju ka x_0 pa možemo da zaključimo da važi:

$$\lim_{n \rightarrow \infty} b_n = x_0$$

Isto važi za niz a_n .

Odavde možemo izvesti još jednu formulu pomoću koje računamo potreban broj iteracija da bi greška rešenja bila manja od nekog zadatog ϵ . Ono što znamo iz formule za odabir sledećeg intervala jeste da je svaki novi niz pola dužine starog intervala. Iz ovog zapažanja možemo da zaključimo sledeće (sa x^* označavamo približnu vrednost nule):

$$|x^* - x_0| \leq \frac{1}{2} |b_n - a_n|$$

$$\frac{1}{2} |b_n - a_n| = \frac{1}{2^{n+1}} |b_0 - a_0|$$

Želimo da ograničimo da razlika x_0 i x^* bude manje od nekog ϵ :

$$|x^* - x_0| \leq \frac{1}{2^{n+1}} |b_0 - a_0| < \epsilon$$

$$b_0 - a_0 < 2^{n+1} \epsilon$$

$$\frac{b_0 - a_0}{\epsilon} < 2^{n+1}$$

$$\ln \frac{b_0 - a_0}{\epsilon} < (n + 1) \ln 2$$

$$\frac{\ln \frac{b_0 - a_0}{\epsilon}}{\ln 2} - 1 < n$$

Iz konačne formule možemo unapred znati koliko nam je potrebno iteracija da bi dobili rezultat koji je tačnosti $|x^* - x_0| < \epsilon$:

$$\frac{\ln \frac{b_0 - a_0}{\epsilon}}{\ln 2} - 1 < n$$

Dakle koraci **metode polovljenja intervala** su sledeći:

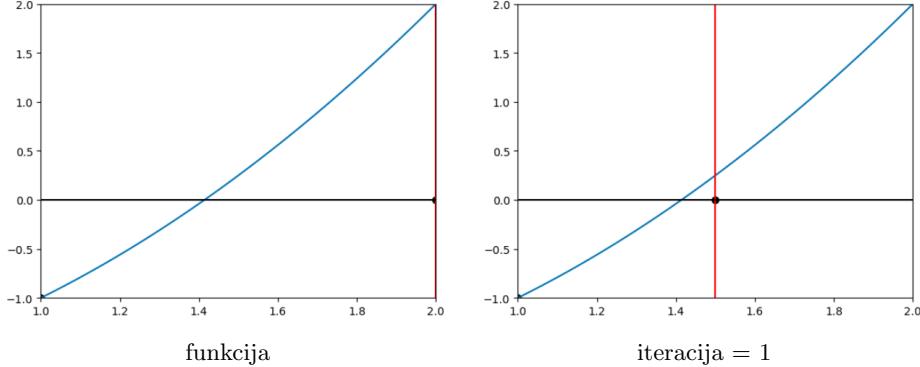
1. Biramo dve tačke na intervalu za koje važi $f(x_l)f(x_r) < 0$. To je naš interval $[x_l, x_r]$
2. Nula funkcije je na sredini ovog intervala odnosno $x_0 = \frac{x_l + x_r}{2}$
3. Posmatrajući sada intervale $[x_l, x_0]$ i $[x_0, x_r]$ koristeći formulu iz 1. koraka i određujemo novi interval.
4. Ponavljamo onoliko puta koliko smo izračunali da je potrebno za traženu grešku ϵ , koristeći formulu:

$$\frac{\ln \frac{b_0 - a_0}{\epsilon}}{\ln 2} - 1 < n$$

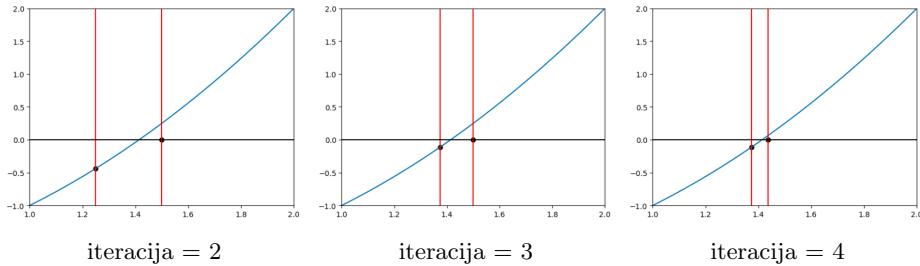
6.1.1 Primer polovljenja intervala:

Želimo da nađemo nulu funkcije $f(x) = x^2 - 2$, poprilično je jasno da je nula $x_0 = \sqrt{2}$ mi se pravimo da ovo ne znamo :).

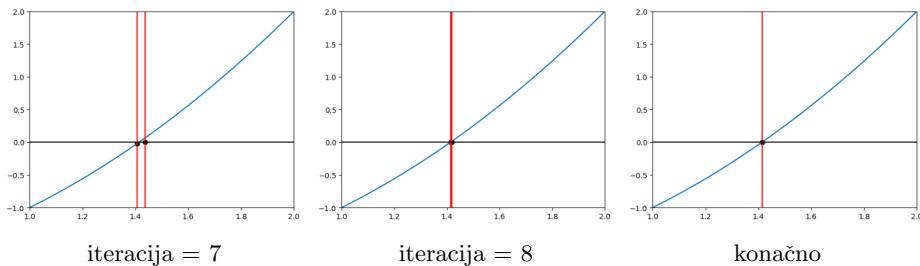
Započinjemo algoritam postavljanjem domena $[1, 2]$.



U prvoj iteraciji smo prepovoljili interval i odabrali novu tačku kao nulu funkcije, međutim kao što se vidi na slici ta nula idalje nije dovoljno blizu, odnosno greška je prevelika.



Ono što se primećuje sa ovih slika jeste da domen relativno brzo konvergira ka rešenju. Ova brzina se održava na početku međutim vrlo brzo se usporava:



Po testiranju koda ovog algoritma, bilo je potrebno ukupno 36 iteracija da se postigne željena tačnost rešenja ($\epsilon = 10^{-12}$). Posmatranjem slika možemo

primetiti da je razlika dužine intervala u prvih 8 iteracija mnogo značajnija od onoga postignuto nakon 8 iteracija.

Zaključak je da je mana algoritma njegova brzina, što smo bliži nuli funkcije rešenje sporije konvergira.

6.2 Njutnov metod

Polovljenje intervala možemo unaprediti pre svega time što koristimo i neka znanja o funkciji liju nulu tražimo... Primetimo da u polovljenju mi nismo koristili znanje kao što je brzina promene funkcije, što nam može pomoći da se brže približimo nuli.

Naravno, potrebno je pronaći domen u kom se nalazi rešenje. Domen $[x_l, x_r]$ nalazimo isto kao i kod polovljenja intervala, mora da važi:

$$f(x_l)f(x_r) < 0$$

Pomenuli smo "brzinu promene funkcije", mada je bolje reći nagib funkcije. Poznato je da je promena funkcije ustvari prvi izvod funkcije (f'). Kako će se ovaj algoritam koristiti tom promenom moramo utvrditi da je funkcija diferencijabilna.

Pitanje je kako da iskoritimo ovu promenu? Kako je promena u isto vreme i nagib funkcije... šta ako izgradimo tangentu na tu tačku pod datim uglom? Ta tangenta će preseći x-osi u nekom momentu, ali samo ako $f'(x) \neq 0$, ako jeste onda se funkcija ne menja i tangenta nema nagib da preseće x-osi (horizontalna je). Sa druge strane ako je nagib dobar neophodno je da se on ne menja prebrzo (f''), inače aproksimacija ne valja i previše daleko skačemo prilikom promene x-a. Možemo reći da prvi i drugi izvod moraju biti stabilni:

$$\operatorname{sgn}(f') = \text{const}$$

$$\operatorname{sgn}(f'') = \text{const}$$

Neophodno je da važi sledeća formula:

$$f(x_0)f''(x_0) > 0$$

Ukoliko je vrednost funkcije pozitivan u trenutnom rešenju onda funkcija mora biti konveksna, ako je rešenje negativno funkcija mora biti konkavna.

Kako odrediti formulu svakog sledećeg x-a? Krenimo od aproksimacije funkcije:

$$f(x_n + (x^* - x_n)) = f(x_n) + \frac{f'(x_n)}{1!}(x^* - x_n) + o(1)$$

Kako važi $f(x_n + (x^* - x_n)) = f(x^*) = 0$, dobijamo:

$$0 = f(x_n) + f'(x_n)(x^* - x_n)$$

$$f(x_n) = -f'(x_n)(x^* - x_n)$$

$$-\frac{f(x_n)}{f'(x_n)} = x^* - x_n$$

Konačno:

$$x^* = x_n - \frac{f(x_n)}{f'(x_n)}$$

Što i jeste formula tangente! Do iste formule možemo doći i geometrijski. Još jednom, algoritam radi tako što u svakoj iteraciji gradi tangentu na trenutno tačku $(x_i, f(x_i))$ na krivi (linearno aproksimira funkciju u dатој таčки), posmatramo ono x gde je vrednost funkcije tangente 0, čime dobijamo novo x.

Dakle **koraci njutnovog metoda** su sledeći:

1. Pre svega funkcija mora biti diferencijabilna
2. Nađemo interval $[x_l, x_r]$ koji ispunjava $f(x_l)f(x_r) < 0$
3. Proverimo da li funkcija na izabranom intervalu zadovoljava:

$$\operatorname{sgn}(f') = \text{const}$$

$$\operatorname{sgn}(f'') = \text{const}$$

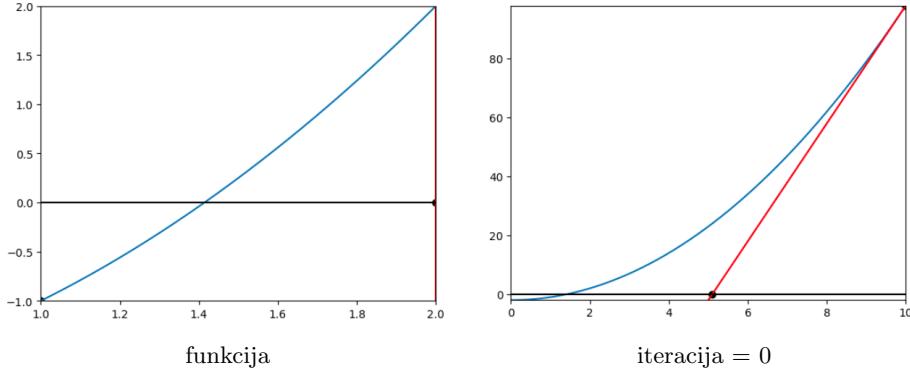
4. Mora da važi $f(x_0)f''(x_0) > 0$
5. Kada su sve prethodne tačke ispunjene pronalazimo rešenje koristeći sledeću formulu u svakoj iteraciji:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

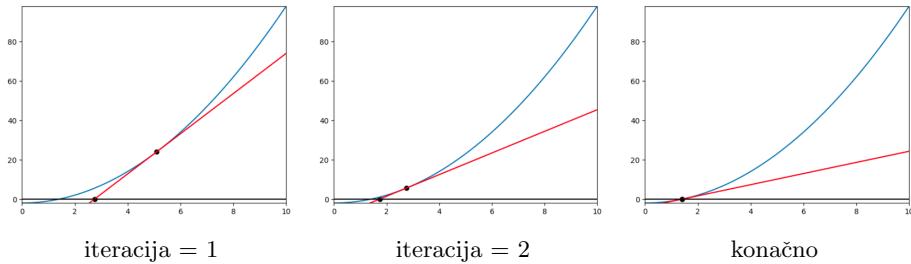
6.2.1 Primer njutnog metoda

Koristimo isti primer kao za **BM**, $f(x) = x^2 - 2$. Tražena greška biće ista kao kod **BM** primera ($\epsilon = 10^{-12}$).

U nultoj iteraciji imamo funkciju i odabранo početno x_0 :



Koristimo novo rešenje $x_0 = x_1$ koje smo dobili i nastavljamo građenje tangenti na krivi, odnosno u tački $(x_1, f(x_1))$:



Po pokretanju ovog koda sa traženom greškom istom kao u primeru testiranja **BM** algoritma, rešenje je dobijeno u 7 iteracija.

Kako možemo ovaj algoritam iskoristiti za pronalaženje **minimuma funkcije**? Formulu koju smo koristili koristi se za pronalaženje nule funkcije, koja je gradila tangentu (koja je takođe linearna aproksimacija funkcije čiju nulu tražimo) koristimo tu tangentu da vidimo kada seče x- osu i tu poziciju koristimo kao novu nulu funkcije.

Kako inače tražimo maximum ili minimum funkcije? Tako što uzimamo izvod te funkcije i pokušavamo da je izjednačimo sa nulom. Možemo da uočimo da nalik formuli koju smo koristili za nalaženje nule možemo napraviti i formulu za pronalaženje minimuma/maksimuma (ustvari gradimo kvadratnu aproksimaciju funkcije). Formula se dobija na identičan način, razlika je što razvijamo izvod funkcije:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Uzimamo one vrednosti koje su manje od prethodne najbolje vrednosti, ostatak algoritma je isti kao u algoritmu za pronalženje nule funkcije.

6.3 Gradijentni spust

U slučaju funkcija sa više promenljivih možemo koristiti **gradijentni spust** ili malo izmenjenu njutnovu metodu koju ćemo kasnije objasniti. Ideja prilikom istraživanja prostora jeste da nađemo njegov minimum/maksimum, odnosno da idemo u onom smeru u kom se funkcija najbrže menja. Gradijent nam daje smer u kom funkcija najbrže raste. Odavde možemo zaključiti da ako želimo da nađemo minimum neke funkcije, dovoljno je da gledamo suprotan smer u odnosu na gradijent.

Algoritam se koristi gradijentom u svakoj iteraciji, uz dodatnu funkciju koja nam omogućava da algoritam zaustavimo. Uslov zaustavljanja je najčešće onda kada je razlika prethodne i sadašnje vrednosti funkcije rešenja manja od neke konstante vrednosti.

Funkcija $\alpha(x_i)$, **brzina učenja** (eng. **learning rate**) računa:

$$\frac{df(x_i + \alpha_i * \text{descentVector})}{d\alpha_i} = 0$$

Algoritam 2 Pseudokod gradijentnog spusta

```
generišemo rešenje  $x_i$ 
i = 0
while nije ispunjen određeni cilj ili kriterijum do
    descentVector = -gradientF( $x_i$ )
     $x_{i+1} = x_i + alpha(i)*descentVector$ 
    i = i + 1
end while
```

Minimum se traži korišćenjem nekih od tehnika koje se bave jednom promenljivom kao što je **NM** ili **BM**. Još jedno rešenje za $alpha(i)$ jeste da kroz iteracije za iteraciju i vrati $alpha(i) = const/i$.

Ovaj algoritam ima par problema. Logičan je da će se skoro sigurno zaglaviti u lokalnom minimumu, ali to je osobina većine ovih algoritama. Bitnija mana jeste pronalaženje vrednosti u specifičnim slučajevima. Jedan od njih jeste kada funkcija sadrži "plato". U ovom slučaju izvod funkcije je 0 pa automatski nema nikakav pomeraj... Primetan problem kod funkcija u 3 dimenzije jeste i brdo. U ovom slučaju gradijent ide u dobrom smeru ali stalno silazi sa brad, maši dobru vrednost.

Iz ovih razloga i boljih rešenja funkcije brzine učenja pravimo slične algoritme sa par dodataka.

6.3.1 Gradijentni spust sa momentumom

Već napomenut problem koji se javlja kada koristimo gradijentni spust jeste problem **platoa**. U ovom slučaju funkcija je konstantna, ravna, na nekom delu i gradijent je 0 samim tim rešenje se neće više pomerati.

Gradijentni spust sa momentumom može da delimično reši ovaj problem, time što zadržava momentum prethodnog gradijenta. Čak i kad je novi gradijent 0 nastavlja se istraživanje prostora rešenja. Međutim u slučaju da krećemo iz položaja gde je gradijent 0 problem idalje ostaje.

Dakle ideja je da uzimamo u obzir šta se dešavalo pre trenutno izračunatog gradijenta. Prenos informacija prethodnih gradijenata postižemo dodavanjem **inercije**. Prethodno množili smo gradijent sa alfa radi kontrolisanja koraka kroz iteracije, kako sada imamo i inerciju želimo i nju da smanjujemo vremenom (ne želimo da inercija sa početka ima istu snagu nakon n iteracija).

Algoritam 3 Pseudokod gradijentnog spussta sa momentumom

```
generišemo rešenje  $x_i$ 
i = 0
while nije ispunjen određeni cilj ili kriterijum do
    inertia = beta*inertia + alpha*gradientF( $x_i$ )
     $x_{i+1} = x_i - inertia$ 
    i = i + 1
end while
```

6.3.2 Nestorov gradijentni spust

Nestorov gradijentni spust sličan je gradijentnom sa momentumom, ali računa jedan korak unapred. Dakle ideja je da umesto da računamo prvo gradijent u trenutnoj tački i dodamo na to inerciju. Kako inerciju svakako dodajemo, što ne bi odmah izračunali gradijent u tački pomerenoj za inerciju i onda na to dodali inerciju? Ovim ubrzavamo konvergiranje ka rešenju.

Algoritam 4 Pseudokod nestorovog gradijentnog spusta

```
generišemo rešenje  $x_i$ 
i = 0
while nije ispunjen određeni cilj ili kriterijum do
    inertia = beta*inertia + alpha*gradientF( $x_i - beta*inertia$ )
     $x_{i+1} = x_i - inertia$ 
    i = i + 1
end while
```

6.3.3 Adam

Primetili smo da prethodne dve varijante gradijentnog spusta imaju ideju da koriste prethodno znanje radi boljeg istraživanja prostora u budućim iteracijama. **Adam** produbljuje ovu ideju.

Adam je napravljen tako da se vremenom prilagođava na funkciju i time zna kad da uspori a kad da ubrza, smanji ili uveća, korak. Postižemo ovo koristeći prvi i drugi momenat. Prvi momenat je očekivanje od x ($E(x)$), a drugi momenat je očekivanje od x^2 (slično varijansi $E(x^2)$).

Idejno prvi momenat (m) akumulira gradijente, ako se gradijent sve vreme kreće na desno onda će i prvi momenat da uzima to u obzir i povećavaće korak kroz iteracije. Drugi momenat (v) posmatra promene gradijenta. U ovom primeru kako je gradijent sve vreme isao na desno drugi momenat zaključuje da nema velike promene u gradijentu (primer kada bi se drugi momenat uvećavao bio bi kada bi konstantno preskakali minimum).

Kako nas zanimaju stari događaji, takođe vidimo da prvi i drugi momenat koriste neki prosek da bi odredili svoje vrednosti moramo osmisliti način računanja proseka koji stalno dobija nove vrednosti. Kada inače računamo prosek to ra-

dimo tako što sabremo ceo niz vrednosti i podelimo sa brojem članova niza... Ovde nemamo ceo niz već u svakoj iteraciji imamo prethodni prosek i novi član niza:

$$\begin{aligned} avg_n &= \frac{\sum_{i=0}^{n-1} x_i}{n} (n-1) + x_n \\ avg_n &= \frac{avg_{n-1}(n-1) + x_n}{n} \end{aligned}$$

U našem slučaju stariji podaci su nam manje bitni pa želimo da **eksponencijalno** računamo naš prosek, a koliko uzimamo u obzir prethodni prosek određujemo korišćenjem parametra α :

$$avg_n = \alpha * avg_{n-1} + (1 - \alpha) * avg_{n-1}$$

Prvi i drugi momentum računamo kao eksponencijalni pokretni prosek:

$$\begin{aligned} m &= \beta_1 m + (1 - \beta_1) grad \\ v &= \beta_2 v + (1 - \beta_2) grad^2 \end{aligned}$$

Kako prvi momenat raste tako korak treba da raste ("prosek" gradijenata), a kako drugi momenat raste tako korak treba da opada (ne želimo prevelike skokove ako smo na brdu).

Želimo da su naše ocene m i v nepristrasne. Želimo da:

$$E(grad) = E(m)$$

Pravimo niz m -ova:

$$\begin{aligned} m_0 &= 0 \\ m_1 &= \beta_1 m_0 + (1 - \beta_1) grad_1 = (1 - \beta_1) grad_1 \\ m_2 &= \beta_1 m_1 + (1 - \beta_1) grad_2 = \beta_1(1 - \beta_1) grad_1 + (1 - \beta_1) grad_2 \\ m_3 &= \beta_1^2(1 - \beta_1) grad_1 + \beta_1(1 - \beta_1) grad_2 + (1 - \beta_1) grad_3 \end{aligned}$$

Primetimo ponašanje, i sređivanjem formule:

$$m_2 = (1 - \beta_1) \sum_{i=1}^2 (\beta_1^{2-i} g_i)$$

Uopštenje:

$$m_n = (1 - \beta_1) \sum_{i=1}^n (\beta_1^{n-i} g_i)$$

Očekivanje za m_n :

$$E(m_n) = E(g_n)(1 - \beta_1) \sum_{i=1}^n \beta_1^{n-i} + error$$

Kako nije svuda isti gradijent moramo da dodamo grešku. Očigledno očekivanja nisu jednaka znači nije nepristrasno. Želimo da napravimo da bude nepristrasno, popravimo m_n :

$$(1 - \beta_1) \sum_{i=1}^n \beta_1^{n-i} = 1 - \beta_1^n$$

$$\hat{m}_n = \frac{m_n}{1 - \beta_1^n}$$

Ovim smo namestili da ocena bude nepristrasna. Isto radimo i sa drugim momentom.

Algoritam 5 Pseudokod Adam-a

```

generišemo rešenje  $x_i$ 
m, v, i = 1
while nije ispunjen određeni cilj ili kriterijum do
    descentVector = -gradientF( $x_i$ )
    m = beta1*m + (1 - beta1)*descentVector
    v = beta2*v + (1 - beta2)*descentVector**2
    mhat = m / (1-beta1**i)
    vhat = v / (1-beta2**i)
     $x_{i+1} = x_i + alpha(x_i)*(mhat/(\sqrt{vhat}) + delta))$ 
    i = i + 1
end while

```

Kako drugi momentum može da bude 0 dodajemo delta koje je jako sitno (npr. 0.000001), ali nas obezbeđuje da ne delimo sa nulom.

6.4 Njutnov metod za više promenljivih (Multiple variable NM)

Kao i u njutnovoj metodi sa jednom promenljivom, koristimo Tejlorov razvoj za $f(x_k + \Delta x)$, gde je $\Delta x = x_{k+1} - x_k$ nepoznata promenljiva, njutnov korak. Pitamo se u kom smeru treba da se pomerimo od tačke x_k da bi našli sledeće bolje rešenje u njenoj blizini.

$$f(x_k + \Delta x) \approx f(x_k) + \nabla f(x_k)\Delta x + \Delta x H(x_k)\Delta x$$

⁴ Nas zanima vrednost Δx pa tražimo optimalnu vrednost preko:

$$\frac{\partial f(x_k + \Delta x)}{\partial \Delta x} = 0$$

⁴ ∇f , čita se **nabla** i predstavlja **gradijent**

⁵ H predstavlja **Hesijan matricu**, kvadratna matrica parcijalnih izvoda drugog reda

Kako su sve vrednosti konstante sem Δx izvod je:

$$\begin{aligned}\frac{\partial(f(x_k) + \nabla f(x_k)\Delta x + \Delta x H(x_k)\Delta x)}{\partial \Delta x} &= 0 \\ 0 + \nabla f(x_k) + H(x_k)\Delta x &= 0 \\ H(x_k)\Delta x &= -\nabla f(x_k) \\ \Delta x &= -H(x_k)^{-1}\nabla f(x_k)\end{aligned}$$

Dobili smo formulu za njutnov korak, kako je on $\Delta x = x_{k+1} - x_k$ dobijamo iterativnu formulu:

$$x_{k+1} = x_k - H(x_k)^{-1}\nabla f(x_k)$$

Zaustavljanje slično kao u prethodnom algoritmu.

Algoritam 6 Pseudokod MVNM

```
generišemo rešenje  $x_i$ 
while nije ispunjen određeni cilj ili kriterijum do
    izračunah  $H(x_i)$  i  $\nabla f(x_i)$ 
     $x_{i+1} = x_i - H(x_i)^{-1}\nabla f(x_i)$ 
end while
```

Kako se ovde koristi Hesijan, matrica, za kompleksnije funkcije je bolje koristiti gradijenti spust.

7 Metaheuristike

Heuristika je metod koji "navodi" algoritam ka potencijalno dobrom rešenju, bez garancije da će pronaći optimalno rešenje. Ovi pristupi često služe za rešavanje problema gde je kompletan pretraga prostora rešenja nepraktična zbog vremenskih ili računarskih ograničenja.

Metaheuristike (meta - apstrakcija) su apstrahovane heuristike koje se primenjuju na širi spektar problema, pružajući okvir za izgradnju heuristika za rešavanje problema slične prirode. One koriste apstraktne principe koji mogu da se prilagode specifičnostima konkretnog problema i time "navedu" pretragu u dobrom smeru.

Primeri metaheuristika su: Genetski algoritmi (GA), Simulirano kaljenje (Simulated annealing), Optimizacija kolonije mrava (Ant colony optimization ACO), Optimizacija grupe čestica (Partical swarm optimization PSO), Variable neighborhood search (VNS) ...

Kako metaheuristike nemaju ideju o kvalitetu rešenja uglavnom se zaustavljaju na osnovu nekih spoljašnjih faktora (kao što su vreme rada algoritma ili broj iteracija). Kako nisu rešenja egzaktnog problema uglavnom se koriste u kombinaciji sa egzaktnim problemima. Relativno je nova oblast koja nije bazirana ni na jednom dokazu ili teoremi.

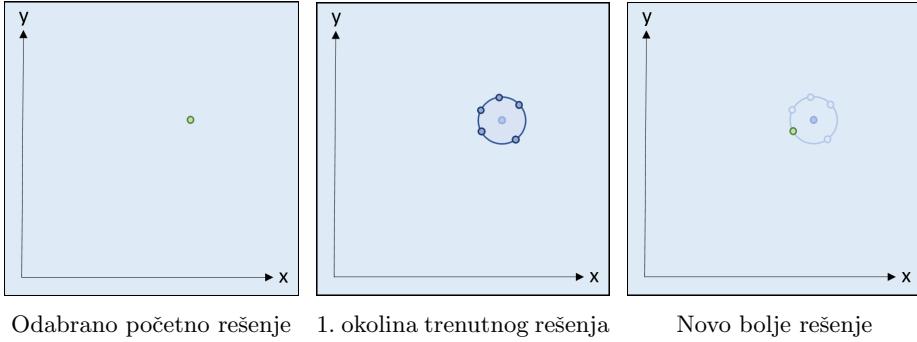
Ako se prisetimo 9. oblasti, tada smo napomenuli da se genetski algoritmi bave populacijama. Ali ne moramo uvek imati **populacije** rešenja, možemo unaprediti jedno **singularno** rešenje, npr. simulirano kaljenje. Odavde možemo napraviti podelu onih metaheuristika koje rade sa većim brojem rešenja tj. **P-Metaheuristike** (**P** - Population) i onih koje rade nad jednim rešenjem tj. **S-Metaheuristike** (**S** - Single).

7.1 Trajectory methods (S - Metaheuristics)

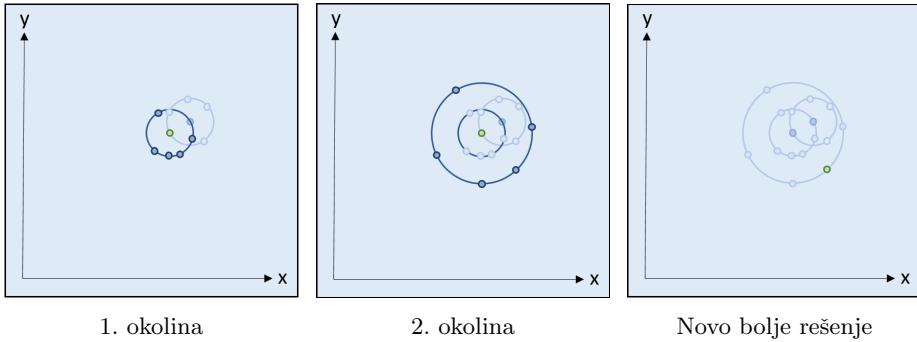
U ovu grupu spadaju sve one metaheuristike koje se bave jednim rešenjem koje unapređuju vremenom.

Jedan od poznatih metaheuristika ovog tipa jeste **Variable neighborhood search (VNS)**. Osmišljena od strane **Nenada Mladenovića** i Pierra Hansena. Ideja je intuitivna i može se lepo zamisliti. Posmatramo neku trodimenzionu funkciju, i odaberemo neku tačku na njoj. Sledće rešenje će biti ono koje u njegovoj okolini ima bolju vrednost, uglavnom bi to bilo ono rešenje za koje funkcija vraća manju vrednost.

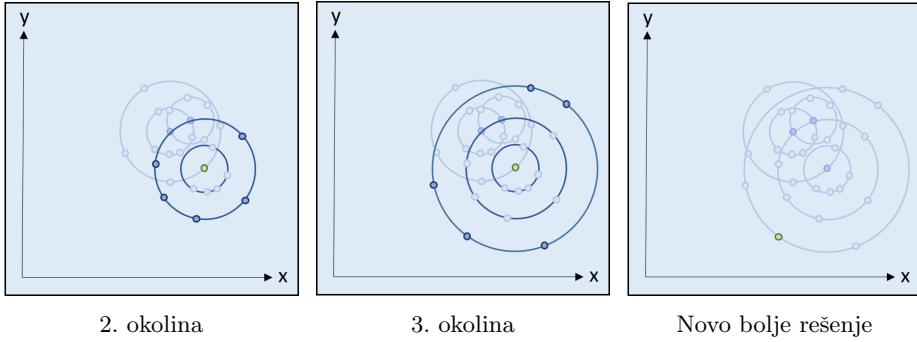
Kako se posmatra okolina rešenja? U situacijama kao što je ova, kontinualni problemi (3.1 Uvod u optimizacije), rešenje može biti bilo koja tačka na kružnici oko trenutnog rešenja... Kako ovo ima beskonačno mnogo rešenja, biramo unapred odabran broj nasumično odabranih tačaka na toj kružnici. Ova operacija se naziva **shake** (promućkati). Radi stvaranje bolje intuicije, koristan način za vizualizaciju jeste da pravimo talase oko našeg rešenja i proveravamo tačke tih talasa u potrazi za boljim rešenjem.



Našli smo novo bolje rešenje u prvoj okolini našeg početnog rešenja, pa će ono postati baš to novo pronađeno rešenje. Dalje tražimo sledeće bolje rešenje po istom principu.



Ovde primećujemo specijalnu osobinu VNS-a. Unapred je određeno, pre početka algoritma, koliko okolina svako rešenje ima. Neophodno je ograničiti jer inače algoritam ne ume da se zaustavi. Ukoliko bolje rešenje nije pronađeno u prvoj okolini rešenja prelazi se na drugu okolinu. U drugoj okolini ponovo po istom principu biramo nekoliko tačaka potencijalnih boljih rešenja i proveravamo za svaku njegov kvalitet. Ovaj primer je veštački osmišljen, ali za cilj ima da pokaže kako VNS koristeći svoje okoline može da se **izvuče iz lokalno optimuma**. U ovom slučaju da smo ograničili okoline na samo jednu, što pritom ne bi imalo smisla, ne bi uspeli da se izvučemo iz lokalnog optimuma i algoritam bi se zaustavio pre pronalaska rešenja koje smo mi našli... zaglavio bi se u lokalnom minimumu.



Da utvrdimo, na dalje nastavljamo i ponovo smo proverili prvu okolinu ali nismo našli bolje rešenje pa smo proverili drugu okolinu. Kako ni u drugoj okolini nismo naišli na bolje rešenje prelazimo na treću. U trećoj nailazimona bolje rešenje i uzimamo ga kao najbolje.

Kao što je već implicitirano, ovaj proces ponavljamo sve dok se ne dogodi da smo proverili sve okoline i nismo našli bolje rešenje ili smo dostigli željeni kvalitet rešenja.

Kod problema diskretnog prirode proces je malo drugačiji, ali isključivo po **shake** operaciji. Kod ovih problema okolina se definiše kao izmena jednog, dva, tri,... n delova nekog rešenja. Primer pronalaženja najkraćeg puta od 3 čvora u potpuno povezanom grafu. Put se definiše kao čvorovi koje prelazimo. Onda bi prva okolina rešenja bila sve kombinacije kada izmenimo po jedan čvor.

Neka je trenutno rešenje [ADF], slova su imena čvorova od A-F. Rešenja prve okoline bi bila sledeća [BDF], [CDF], [EDF], [ABF], [ACF], ... [ADC], [ADE], slično za drugu i treću okolinu. Idemo do okoline tri zbog činjenice da imamo samo tri čvora tj. za okoline > 3 dobijamo isti skup rešenja kao za okolinu 3. Algoritam se svodi na sledeće korake:

1. Izaberi neko pseudo nasumično rešenje
2. Odredi njegov kvalitet
3. Vršimo **Shake** operaciju koja od našeg sadašnjeg rešenja pravi nova rešenja u njegovoj okolini.
 - (a) prvo se predstavljaju sva rešenja dobijena izmenom jednog parametra prethodnog rešenja
 - (b) ukoliko nije nađeno bolje rešenje povećavamo broj parametara koji smeju da se izmene i ponavljamo prethodni korak
 - (c) ukoliko smo našli bolje rešenje, ili više nemamo parametre koje možemo da izmenimo, prekidamo postupak
4. Sada bolje dobijeno rešenje, ukoliko je takvo nađeno, postavljamo kao novo optimalno rešenje i ovaj postupak ponavljamo dok ne istekne broj

iteracija, neko vremensko ograničenje ili smo prošli sve okoline trenutnog rešenja bez da nađemo novo bolje.

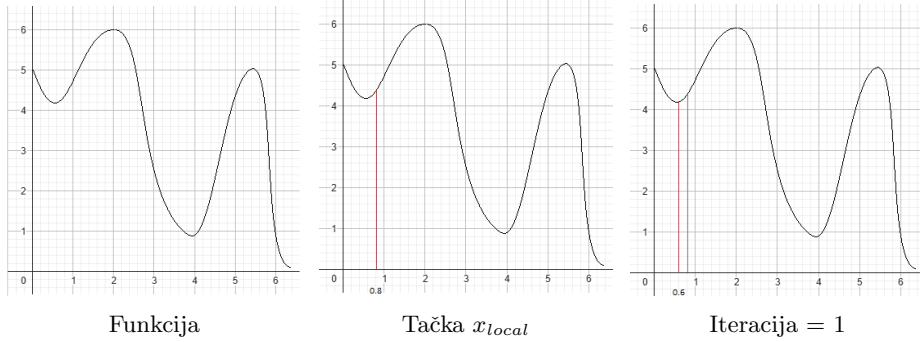
7.1.1 Primer za VNS

Ovim primerom ćemo pokazati kako rešavanje VNS-a radi u dve dimenzije. Tražimo x za koje funkcija dostiže minimum. Kako se ovde bavimo jednom promenljivom shake funkcija ce biti malo drugačija, ali ideja je slična. Funkciju **Shake** možemo predstaviti kao funkciju koja pravi od našeg trenutnog rešenja x_{local} dva rešenja x_{left} i x_{right} koji se izračunavaju kao:

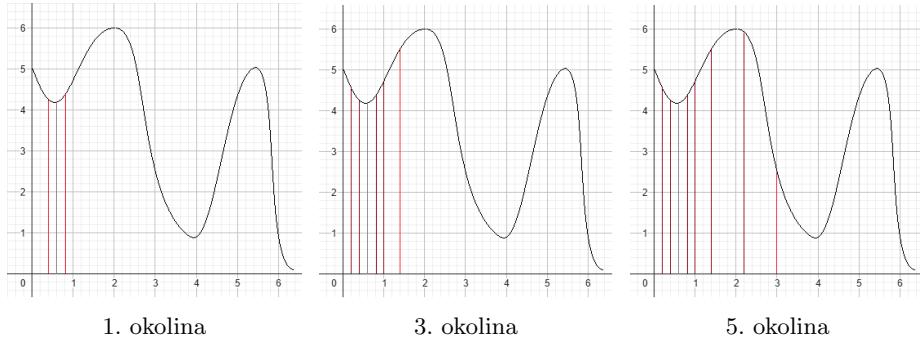
$$x_{left} = x_{local} - value$$

$$x_{right} = x_{local} + value$$

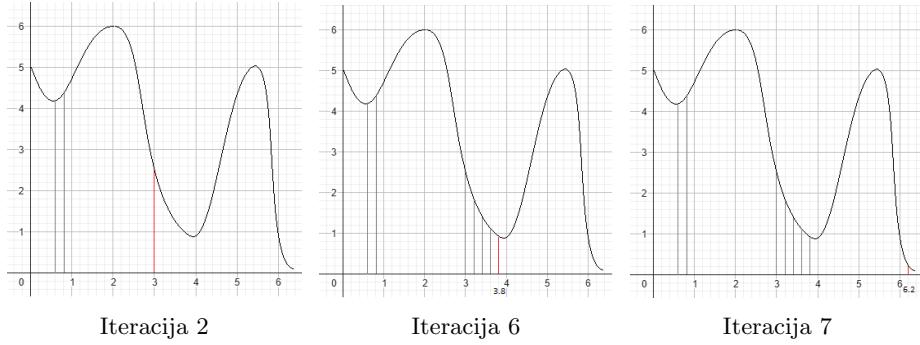
Gde je $value$ vrednost koja za svako novo shake-ovanje ima redom vrednosti 0.2, 0.4, 0.8, 1.6, 2.4. Izaberimo neku nasumičnu tačku $x_{local} = 0.8$:



Prve okoline su nam $x_{left} = 0.6$, $x_{right} = 1.0$, pronalazimo da je vrednost funkcije u tački x_{left} manja nego u trenutnoj x_{local} tako da uzimamo to rešenje kao novi optimum. Sada posmatramo okolinu nove tačke $x_{local} = 0.6$, u okolini ove tačke prve dve vrednosti dobijene shake-om $x_{left} = 0.4$, $x_{right} = 0.8$ ni jedna nije novi optimum. Ponavljamo shake, $x_{left} = 0.2$, $x_{right} = 1.0$ ponovo nijedno rešenje nije bolje, a prvo poboljšanje se dešava za $value = 2.4$ kada dobijamo da je $x_{right} = 3.0$:



Dalje se isti postupak ponavlja dok ne dođemo do konačnog rešenja kada više ne možemo da poboljšamo rešenje, ponestalo nam je iteracija ili okolina za trenutno rešenje.



U ovom primeru imali smo sreće prilikom odabira svake sledeće okoline, ali da nismo dodali okolinu 2.4 ovaj algoritam bi stao već nakon druge tačke. Još jedan primer problema sa diskrenim prostorom rešenja je problem ranca. Jedina razlika bi bila u predstavljanju rešenja. Rešenje bi bio niz bitova (odabranu stvar predstavljamo kao 1, a neodabranu kao 0), a funkcija shake-ovanja bi bila implementirana kao sve kombinacije kada negiramo 1, 2, 3...n bitova gde je n broj objekata koje možemo odabrati. Na osnovu toga da problem koji nemam tolike sličnosti sa drugim problemom a da pritom možemo iskoristiti VNS dokazujemo da je algoritam zaista metaheuristika.

7.2 Population-based methods (P - Metaheuristika)

U ovu grupu spadaju sve one metaheuristike koje se bave unapređivanjem više rešenja njihovim simultanim evoluiranjem. Primer ovakvih algoritama je algoritam **jata ptica (Bird flocking PSO)**.

Zamislimo jato prica ili jato riba. Kad god smo ih posmatrali videli bi da većina ptica prati jednu pticu na početku jata. Sa druge strane kada vidimo jato riba vidimo da se sve kreću unutar jata ali retko kajače.

Možemo ovako organizovati i neko naše veštačko jato. Napravimo stotinu instanci jedinki sa **pozicijama** p_{pos} gde svaka predstavlja rešenje našeg problema. Svaka jedinka se kreće ka onoj jedinki koja ima najbolje rešenje, **globalno** p_{global} .



Jato ptica



Jato riba

Ako postavimo ovako problem ako ocrtamo ovo ponašanje dobćemo nešto što liči na kretanje ptica ili riba u jatu... nastaje jedan problem a to je - šta ako postoji bolje rešenje van prostora u kom su postavljene inicijalne jedinke? Pa ništa nećemo ga naći :).

Ovako postavljeno rešenje će vrlo brzo samo konvergirati ka najboljoj jedinki i istraživanje prostora je minimalno. Najbolja jedinka ni ne vrši istraživanje prostora, ako se ne nađe bolja jedinka prilikom pretrage onda ta nova prestaje istraživanje.

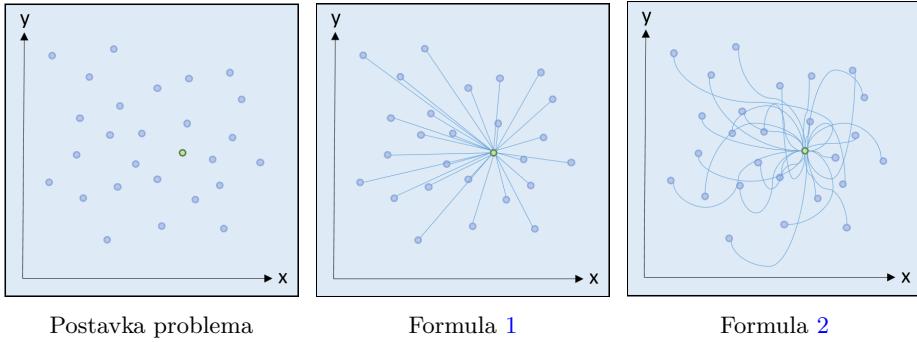
Ova osobina je ustvari osobina eksploatacije prostora rešenja. Mada se tačke kreću i istražuju prostor $p_{pos} = p_{pos} + v$ najviše što će se istražiti jeste linija između svake tačke i najboljeg rešenja:

$$v = \alpha * (p_{global} - p_{pos}) \quad (1)$$

Množimo sa α koji je neka vrednost između 0 i 1, da se ne bi direktno prebačili u tačk p_{global} . Ovo nije dovoljno i pospešujemo eksploataciju dodavanjem **inercije** v_i . Na ovaj način svaka jedinka ima neki pravac kretanja ali ih pritom najbolje rešenje vuče ka sebi. Ovim dobijamo malo kompleksnije $p_{pos} = p_{pos} + v_i$ ponašanje i bolju eksploataciju prostora:

$$v_i = \alpha * v_i + (1 - \alpha) * (p_{global} - p_{pos}) \quad (2)$$

Sada kada imamo i inerciju želimo da novu poziciju računamo koji deo inercije i koji deo kretanja ka globalnom najboljem čuvamo, dobijeni vektor je inercija za sledeću iteraciju. Ovim dodajemo i eksploraciju, jer dodatni početni nasumični vektor inercije omogućava da se istraži prostor van jata...



Postavka problema

Formula 1

Formula 2

Ovo idalje vodi svaku jedinku ka globalnom maksimumu i ne istražuje dovoljno prostor oko svojih lokalnih rešenja. Da bi pospešili istraživanje lokalnih rešenja uvodimo i kretanje svake jedinke ka njihovoј zapamćenoj najboljoj poziciji p_{local} :

$$v_i = \alpha * v_i + (1 - \alpha) * ((p_{local} - p_{pos}) + (p_{global} - p_{pos})) \quad (3)$$

Zapamtimo da tokom kretanja jata stalno proveravamo da li je neko od jedinki našlo novo globalno bolje rešenje, ukoliko jeste uzimamo to novo rešenje. U isto vreme kada neka jedinka nađe lokalno bolje rešenje od onog koje čuva potrebno je da ažurira svoje lokalno najbolje.

Radi pospešivanja nasumičnog ponašanja svake jedinke uvodimo dodatne pseudonasumične promenljive r_{global} i r_{local} , vrednosti između 0 i 1.

Dodajemo i promenljive koje korisnik može da podešava pre samog pokretanja programa. Promenljive nazivamo c_{global} i c_{local} , predstavljaju koliko želimo da uticaja na kretanje jedinke ima respektivno globalni vektor u formuli

$v_{global} = p_{global} - p_{pos}$ tj. lokalno vektor u formuli $v_{local} = p_{local} - p_{pos}$.

Konačna formula:

$$v_i = \alpha * v_i + (1 - \alpha) * (r_{local} * c_{local} * v_{local} + r_{global} * c_{global} * v_{global}) \quad (4)$$

Vrednost α utiče najviše na količinu intenzifikacije i diverzifikacije. Ako postavimo da je $\alpha = 1$ onda grupa divergira, ako postavimo da je 0 onda se gubi znanje od prethodnom kretanju. Najbolje je koristiti $0 < \alpha < 1$, kroz iteracije stara kretanja se uzimaju u obzir ali ona starija nemaju toliko udela koliko novija.

Algoritam 7 Pseudokod jata ptica

```

generisi populaciju ( $P(n)$ )
globalMin = min( $P(n)$ )
while nije ispunjen određeni cilj ili kriterijum do
    for p in ( $P(n)$ ) do
        velocity_update()
        position_update()
    end for
end while

```

7.2.1 Primer jata ptica

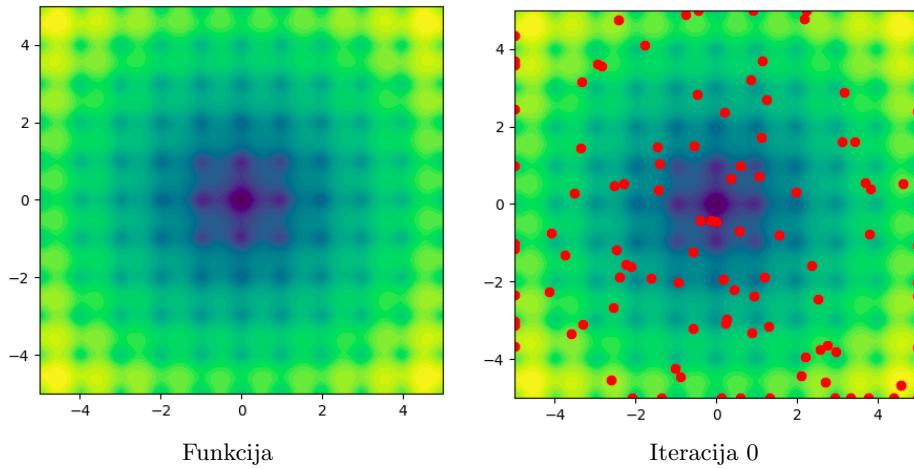
Tražimo minimum funkcije:

$$f(x, y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(\cos 2\pi x + \cos 2\pi y)} + e + 20$$

poznata i kao "Ackley function", tražimo rešenje na domenu $-5 \leq x, y \leq 5$. Uzimamo ovakvu funkciju pošto se ova vrsta algoritma uglavnom koristi za probleme sa **kontinualnim** domenom.

Prvo što smo rekli da želimo jeste da rasporedimo n ptica pseudo nasumično po prostoru rešenja (domenu), dakle biramo n pozicija $\{(x, y) \mid -5 \leq x, y \leq 5\}$. Dalje svakoj ptici izračunamo njenu lokalnu najbolju poziciju, njen vektor kretajna (inercija) i odredimo koje je to globalno najbolje rešenje (najbolje od svih lokalnih rešenja grupe).

Hladnije boje na grafu predstavljaju niske vrednosti a toplice predstavljaju više vrednosti.



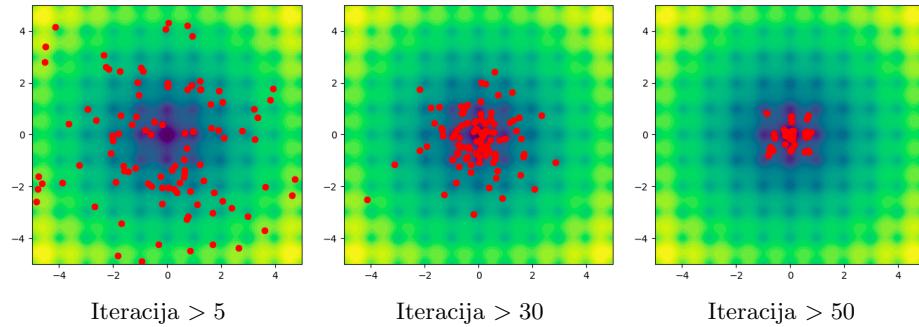
Već u prvoj iteraciji želimo adekvatno da pomerimo pticu na njenu sledeću poziciju. Imamo sledeće vrednosti **najbolju poziciju ptice** (p_{local}), **najbolju poziciju grupe** (p_{global}) i **inerciju ptice** (v_i) kao i **trenutnu poziciju ptice** (p_{pos}). Novu poziciju dobijamo kao zbir vektora inercije (v_i), vektora $v_{global} = p_{global} - p_{pos}$ i vektora $v_{local} = p_{local} - p_{pos}$.

Kako vektor ka globalnoj najboljoj poziciji ima verovatno najveću vrednost vrednost lokalnog rešenja ptice, biće zanemareno i time gubimo na osobini **intenzivikacije**. Da bi izbegli ovo želimo da množimo svaki vektor nekom konstantom koju možemo da menjamo, pa bi novu poziciju računali na sledeći način $v_i = c_i v_i + c_{local} v_{local} + c_{global} v_{global}$.

Kako bi još više poboljšali naše istraživanje domena želimo da uvedemo stohastičnosti, odabirom dva broja između 0 i 1 (r_{local} i r_{global}), i time omogućimo

diverzifikaciju. Koristimo formulu (4).

Konačno ono što moramo da proverimo posle svake iteracije jeste da li je neko od novih lokalnih pozicija novo globalno najbolje rešenje.



Primetimo da se vremenom sve tačke skupljaju u jednu tačku, ta tačka je zapravo minimum naše funkcije.

8 Evolutivna izračunavanja (EC)

Kako i samo ime kaže **evolutivna izračunavanja (EC)** su inspirisana evolucijom. Samim tim inspirisana su nekim istorijskim idejama o evoluciji.

Lamarkova teorija predlaže da se karakteristike koje je jedan organizam stekao tokom života mogu naslediti narednim generacijama... Međutim većina eksperimentirala koja su pokušali da dokažu valjanost ove teoreme bili su neuspešni.

Druga teorija koja se i danas prihvata jeste **Darvinova**. Darwin tvrdi da se evolucija odvija kroz prirodnu **selekciju** gde samo najprilagođenije jedinke opstaju. Ovo omogućava njihovim karakteristikama da se propagiraju kroz **populaciju** prilikom **ukrštanja**.

Danas postoji ideja o još jednoj osobini evolucije, a to je genetska **mutacija**. Prilikom mutacije dolazi do izmene gena što dovodi do pojave novih osobina nekih jedinki. Do iste, može doći zbog greške u replikaciji gena ili usled spoljašnjih faktora kao što je radijacija.

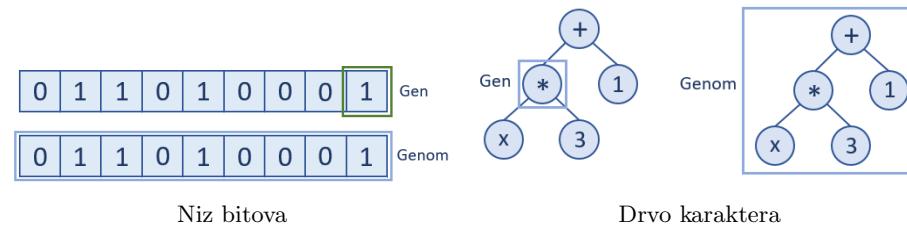
Možda je već dobijena ideja kako inspirisani evolucijom možemo da pronađemo bolja rešenja nekih problema. Nalik evoluciji, **evolutivno izračunavanje** predstavlja proces poboljšavanja organizama ili sistema kroz takmičarsko okruženje. Najuspešnije jedinke (rešenja) prilagođavaju i prenose svoje karakteristike na buduće generacije. Poznati i kao **Evolutivni algoritmi (EA)** pripadaju **P-Metaheuristici**.

Pregled ovih principa i njihova primena u inženjerskim problemima detaljno su predstavljeni u radu [5].

8.1 Osnovni pojmovi

U prirodi svaka jedinka je definisana na osnovu koda zapisanog u hromozomima. Hromozomi su sačinjeni od DNK lanca koji sadrži informacije o izgradnji proteina koji omogućavaju izgradnju tog organizma. Drugi naziv za DNK je **genom**, a specijalni delovi genoma se nazivaju **geni**. Svaki gen jedna osobina.

Ako želimo da implementiramo našu simulaciju evolucije, naši genomi će predstavljati rešenje našeg probelma. Znajući ovo različiti problemi mogu imati drugačije definisane genome. Genomi mogu biti definisani kao nizovi, liste, drveta... u zavisnosti od pribliznog problema. Sami geni genoma zavise od probelma pa mogu biti bitovi, relani brojevi, prirodni brojevi, karakteri, čvorovi...



Postoji i opcija gde se celi ili realni brojevi mogu predstaviti **Grejovim kodiranjem**. Prilikom kodiranja sa realnim brojevima ukoliko ih želimo predstaviti

grejovim kodom moramo prihvati gubitak tačnosti. Što veću tačnost želimo to su hromozomi duži a time evolucija sporija. Međutim danas je prihvaćeno kodiranje direktno odabranim tipom.

Kada imamo definisane genome moramo smisliti kako da ih "rangiramo". Ovaj proces je ustvari određivanje **fitness funkcije**. Fitness funkcija na osnovu genoma vraća neku vrednost, bolje vrednosti karakterišu bolje genome. *Bolje* zavisi od konteksta, npr. ako tražimo minimum funkcije želimo da fitness funkcija daje male vrednosti za bolje genome.

Ako znamo da kvantifikujemo kvalitet genoma možemo ih sortirati, ovaj proces se naziva **selekcija**. Proces u kom biramo roditelje naredne generacije i u cilju nam je da veću šansu za ukrštanje imaju one bolje jedinke. Malo kontra intuitivno jeste da ponekad želimo da se ukrste dve jednike lošeg kvaliteta ili da bar jedan roditelj ima lošiji kvalitet. Na ovaj način postižemo da možda uzmemos neki gen (dobra osobina) loše jedinke koji može da poboljša kvalitet rešenja.

Postoje dva osnovna principa selekcije:

1. Turnir

Ideja ovog pristupa jeste da izaberemo neki deo populacije a zatim jedinke tog izabranog dela rangiramo prema fitness-u i izaberemo dve najbolje jedinke.

U zavisnosti od veličine izabranog broja jedinki za turnir (k), koji može biti jednak ili manji od veličine populacije (n), imamo sledeće ishode:

- $k = 1$: praktično je nasumičan odabir jedinki za crossover
- $k < n$: najbolji pristup, pošto i one dobre i one loše jedinke imaju šansu da budu izabrane.
- $k = n$: u ovom slučaju uvek će biti izabrane one jedinke koje su najbolje u celoj populaciji, pa će se uniformna populacija steći jako brzo.

Primetimo da što je veće k to je **selekcioni pritisak**⁶ veći.

2. Rulet

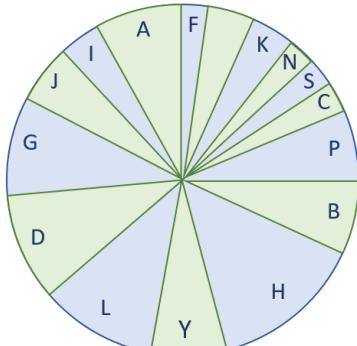
Zamišljamo ruletski točak gde je svaka pozicija u koju loptica može da upadne jedna jedinka. Veličina pozicije za svaku jedinku je proporcionalna fitness vrednosti jedinke, ovo predstavlja verovatnoću njenog odabira. Kako verovatnoća svih mogućih događaja mora da bude jednaka 1, moramo normalizovati sve verovatnoće. Pa u populaciji sa n jedinki, za svaku jedinku računamo njenu verovatnoću:

$$p_x = \frac{f_x}{\sum_{i=1}^n f_i}$$

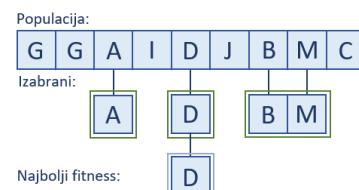
Ovakav način selekcije nije najbolji, jer kako one jedinke koje imaju bolji fitness imaju veće šanse da budu odabранe, one lošije jedinke će ređe biti

⁶**Selekcioni pritisak** - predstavlja koliko vremena je potrebno da najbolje jedinke izgrade uniformnu populaciju. Pa bi najniži selekcioni pritisak bio da izmešamo populaciju i nasumično biramo jedinke.

odabrane (šanse jako male). Možda baš te lošije jedinke imaju neki gen koji nam je potreban u nekom drugom genomu da bi dobili bolje rešenje. Jedna od modifikacija ruletske selekcije je **Stohastičko univerzalno uzorkovanje (SUS)**.



Ruletska selekcija



Turnirska selekcija

Konačno ostaje da odredimo kako će se vršiti ukrštanje (crossover) i mutacija genoma. Ukrštanje predstavlja razmenu gena dva roditelja, kako mi radimo sa nizovima, drvetima... ovo može biti zamena pojedinih delova niza, zamena podrveta ili čak samo čvorova drveta...
Mutacija se svodi na nasumičnu izmenu nekog dela genoma. Biramo neki gen i zamenimo ga nekom novom vrednošću.
Dodatno će biti rečeno u narednim sekcijama.

Sad možemo i osmisliti kako bi tačno radio ovakav program. Na početku generišemo našu populaciju, a zatim simuliramo evoluciju koristći selekciju, ukrštanje i mutaciju. Proces ponavljamo predodređeni broj iteracija zovemo ih i **epohe**.

Algoritam 8 Pseudokod EA

```

Generiši inicijalnu populaciju  $P_0(n)$ 
 $t = 0$ 
while nije ispunjen određeni cilj ili kriterijum do
    Izračunaj fitness za svaku jedinku u  $P_t(n)$ 
    Koristeći crossover napravi novu generaciju  $P_{t+1}(n)$ 
    Predi u sledeću generaciju  $t = t + 1$ 
end while

```

Neki od algoritama zasnovani na EA su:

1. **Genetski algoritmi GA** - genetski kod u obliku niza bitova.
2. **Genetsko programiranje GP** - genetski kod u obliku stabla.
3. **Evolutivno programiranje EP** - koristi samo mutaciju prilikom pravljenja sledeće generacije.

4. **Evolutivne strategije ES** - gradi novu privremenu populaciju (drugačije veličine u odnosu na prethodnu generaciju), nad njom vrši ukrštanje i mutaciju i tek onda rangira sva rešenja i na osnovu njih gradi populaciju P_{t+1} . Rešenja u obliku niza realnih brojeva.
5. **Diferencijalna evolucija DE** - ne koristi optimizaciju gradijenta, pa funkcije ne moraju biti neprekidne ili diferencijabilne, već za svaku jedinu bira približne tri tačke a , b i c nasumično, koje ne smeju biti iste, i ukrštanjem gradi novu tačku po formuli $x_{new} = a + F \times (v - c)$.
6. **Kulturna evolucija (CA)** - pored populacije sadrži dodatnu komponentu **prostor verovanja (belief space)**, ta "verovanja" jedinke prihvataju čime utiču na populaciju.
7. **Koevolucija** - rešenja ograničenja nekim drugim populacijama i njihovim ponašanjima.

8.2 Genetski algoritmi (GA)

Jedan od osnovnih algoritama EA jeste genetsko programiranje, takođe glavni u diskretnom domenu. Mada su spori, ovi algoritmi su odlični za rešavanje kombinaornih problema. U osnovnom algoritmu GA tj. **SGA (Simple GA)**, selekcija roditelja se vrši tako što razbacamo jedinke, čime svake dve jedinke imaju podjednake šanse da budu izabrane za ukrštanje. Do ukrštanja može i ne mora doći, ukoliko ne dođe do ukrštanja jedinke se samo prepisuju

Algoritam 9 Pseudokod SGA

```

Generiši inicijalnu populaciju  $P_0(n)$ 
 $t = 0$ 
while nije ispunjen određeni cilj ili kriterijum do
    Gradimo populaciju  $P_{t+1}(n)$  izvršavanjem crossovera i mutacije
    Pređi u sledeću generaciju  $t = t + 1$ 
end while

```

Jednostavni genetski algoritmi koriste takozvani **genracijski model (Generation GA - GGA)**. Svaka jedinka preživi tačno jednu generaciju, odnosno cela populacija se zameni novom populacijom tj. potopmcima.

Malo kompleksniji mada u nekim slučajevima bolji način jeste korišćenje **genracijskog jaza**. **Generacijski jaz** predstavlja proces gde određujemo koji deo populacije želimo da menjamo. Ako je odnos stare i nove generacije manji od 1, samo se deo populacije menja (ako je odnos nove i stare generacije 1 to je GGA). Ovaj model se koristi u **Evolutivnim strategijama ES**.

Postoji i **model stabilnog stanja (Steady-State GA - SSGA)** gde generišemo jedno dete.

Primećujemo da se u druga dva modela dešava da samo kopiramo jedinke iz stare generacije. Postoji još jedan princip po kom se čuva najboljih **m** jedinki tj. princip **elitizma**. Ideja je da čuvamo one jedinke koje su kvalitetne, imaju

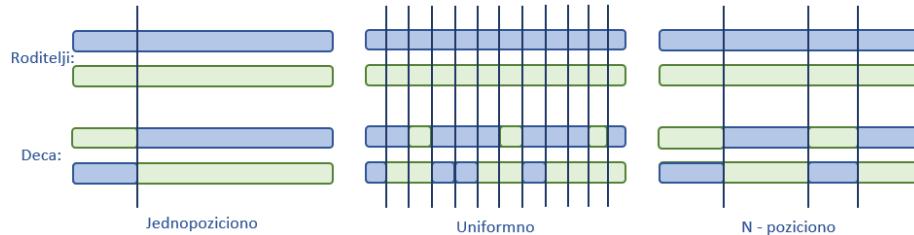
dobar fitness, njihovim direktnim prepisivanjem. Ne želimo da dozvolimo da one bolje jedinke nadvaladaju, dakle želimo da **m** bude neki mali deo.

Napomenuli smo da ćemo pričati o drugim primerima ukrštanja. Pre svega treba razumeti zašto se ukrštanje vrši po principima optimizacionih algoritama...

Ideja ukrštanja u smislu pronalaženja rešenja predstavlja intezifikaciju. Drugim rečima, ukrštanje je istraživanje prostora rešenja koja mogu nastati rekombinovanjem već postojećih rešenja.

Ukrštanje (Crossover) može biti drugačiji u zavisnosti od problema, neki principi:

1. **Uniformno** - svaki gen jendog roditelja ima jednake šanse da se swapuje sa genom na istoj poziciji drugog roditelja.
2. **Jednopoziciono** - bira jednu poziciju u hromozomu, pa se levi deo hromozoma prvog roditelja swapuje sa levim delom drugog roditelja.
3. **n - Poziciono** - bira više pozicija i deli oba gena na $n+1$ celinu nakon čega radi "cik-cak" swap delova gena.



Slika 39: Ukrštanja

U slučaju realnog kodiranja koristimo **aritmetičko ukrštanje**:

$$z_i = \alpha * x_i + (1 - \alpha) * y_i$$

$$0 \leq \alpha \leq 1$$

Alfa možemo da biramo u svakoj generaciji, da postavimo na početku ili da je menjamo kroz generacije nekom formulom. **Jednostvano** (jednopoziciono) aritmetičko ukrštanje je kada izaberemo poziciju $k = i$ u genomu i odatle vršimo ukrštanje samog narednog gena, **celovito** (uniformno) je kada je $k = 0$ odnosno svaki gen genoma ukrštamo.

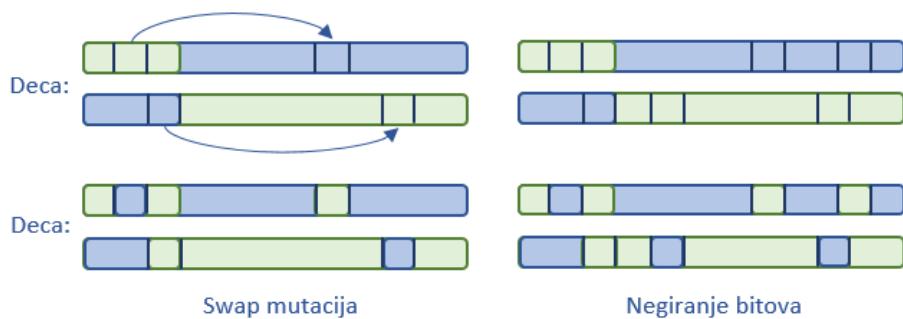
Mana ukrštanja je da ukoliko je optimalno rešenje "van dometa" trenutnog prostora rešenja nikada ga nećemo naći. Kako je "van dometa"?... Određene osobine tog rešenja ne postoje ni u jednom od rešenja koja se nalaze u trenutnoj populaciji (nastaje lokalni minimum).

Rešenje za ovo predstavljaju mutacije. Mutacija prolazi svaki gen i sa nekom verovatnoćom može ga izmeniti. Ideja mutacije služi za diverzifikaciju, odnosno

uvodenje novih osobina (rešenja) koje nisu mogle biti dobijene drugaćije. Drugim rečima, mutacija pokušava da proširi prostor rešenja koja imamo do sada. Mana mutacije je njena destruktivnost. Kako može da proširi skup rešenja tako može da ga umanji ili ukoliko je previše učestala da ne dozvoli populaciji da konvergira ka bilo kom minimumu. Za genome kodirane bitovima to bi bila negacija ili swapovanje gena.

Kao i kod ukrštanja i mutacija ima više, neke od njih:

1. **Umetanje** - izaberemo neka dva gena i drugi umećemo tako da stoji odmah posle prvog, svaki ostali geni se pomeraju.
2. **Zamena** - izaberemo dva gena i swapujemo ih.
3. **Inverzija** - izaberemo segment hromozoma nad kojom vršimo inverziju.
4. **Mešanje** - izaberemo segment hromozoma i izmešamo gene unutar segmenta.



Slika 40: Mutacije

Kada kodiramo realnim brojevima, neke od čestih mutacija su:

1. **Dodavanje šuma (Gaussian mutation)**

Na broj dodajemo šum dobijen iz standardne normalne raspodele. Računamo novo x na sledeći način:

$$x' = x + N(0, \sigma)$$

2. **Uniformna mutacija**

Svakom genu je dodeljen opseg i nova vrednost mu se dodeljuje uniformno iz tog opsega. Računamo novo x na sledeći način:

$$x' = U(a, b)$$

3. **Menjanje gena za korak**

8.2.1 Problem trgovackog putnika (TSP)

Specifični problemi jesu oni zasnovani na **permutacijama**, jedan takav je problem trgovackog putnika (TSP).

Trgovac treba da obide sve gradove i da se vrati u početni grad (napravi ciklus), ali sme da poseti svaki grad jednom (neuključujući početni). Za 30 gradova $30!$ rešenja...

Kako trgovac treba da obide sve gradove znamo da jedino po čemu se rešenje može menjati jeste po redosledu obidjenih gradova.

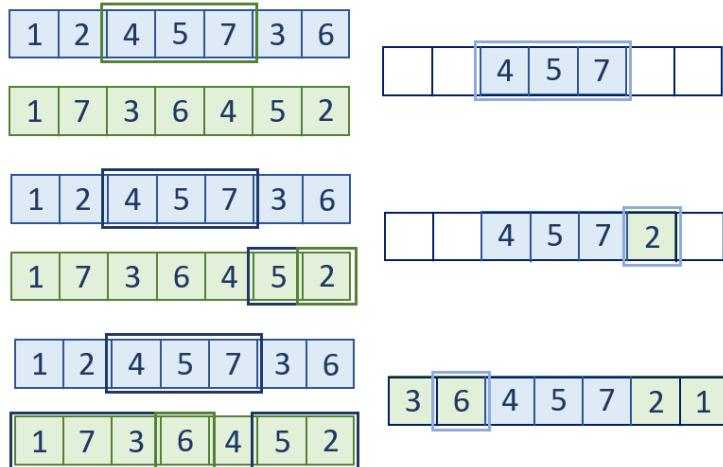
Kodiranje ovakvog rešenja je najjednostavnije kodirati nizom celih brojeva, gde svaki broj predstavlja neki čvor (može i karakterima). Redosled ovih brojeva predstavlja redosled posećivanja.

Pošto moramo da posetimo sve gradove znamo da standardno ukrštanje neće raditi u našem slučaju, jer može da napravi rešenje koje ponavlja gradove i briše druge... Slično i mutacija može da zameni neka dva grada gde pritom ne proverava da li postoji novodobijeni ciklus ili može da zameni jedan grad nekim drugim čime dolazi do ponavljanja grada.

Moramo da osmislimo novi način ukrštanja, dok mutaciju totalno izbacujemo. Recimo dva principa **ukrštanja kod permutacija** su:

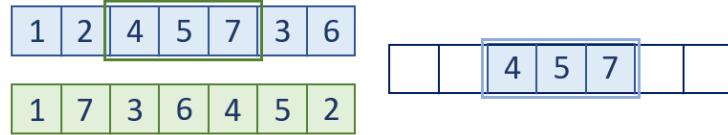
1. Ukrštanje prvog reda

Odaberemo segment hromozoma prvog roditelja, iskopiramo dati segment u prvo dete, nastavljamo desno od kopiranog segmenta i upisujemo sve one brojeve koji se nisu još pojavili u prvom detetu, sve dok se ne popune sve pozicije. Slično i za drugo dete.



2. Delimično ukrštanje (PMX)

Odaberemo segment hromozoma **prvog** (I) roditelja. Gledajući početak ovog segmenta idemo redom kroz isti segment **drugog** (II) roditelja i posmatramo preslikavanja, $f : II \rightarrow I$. U isto vreme gledamo gde se ta slika pojavljuje u drugom roditelju, $g : I \rightarrow II$...



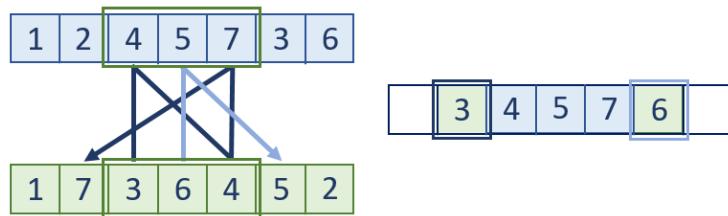
Na zadatom primeru vidimo sledeća preslikavanja:

$$\begin{array}{ll} f(3_{II}) = g(4_I) & g(4_I) = f(4_{II}) \\ f(6_{II}) = g(5_I) & g(5_I) = 5_{II} \\ f(4_{II}) = g(7_I) & g(7_I) = 7_{II} \end{array}$$

Npr. kako je:

$$f(3_{II}) = g(4_I) = f(4_{II}) = g(7_I) = 7_{II}$$

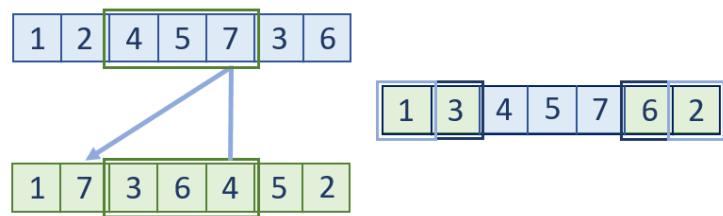
zaključujemo da se trojka slika na poziciju sedmice drugog roditelja. Isto raditmo za 6 i 4.



Postupak ponavljamo dok ne izađemo iz segmenta drugog roditelja iz kog preslikavamo.

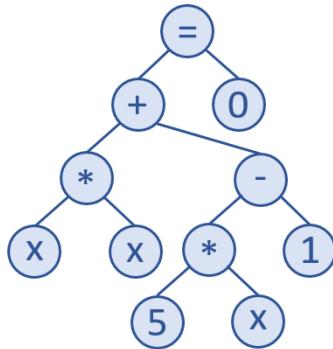
U ovom primeru se desilo i da pokušamo da preslikamo 4, međutim kako je 4 već prepisan u dete zanemarujemo ga (takođe na poziciji gde piše da se slika postoji 3).

Nakon preslikavanja slobodne pozicije popunjavamo prepisivanjem redom onih brojeva drugog roditelja koji se već nisu pojavili u detetu. Slično za drugo dete.



8.3 Genetsko programiranje (GP)

Osmišljen od strane **John R. Koza**, ovaj algoritam je inspirisan idejom da ako se računaru daju ulazni i izlazni parametri on bi trebalo da zna da napravi program koji rešava dati problem. Koristi princip GA, generacijska evolucija rešenja, odnosno **genetsko programiranje (GP)** je nadovezan na GA. [4] Idejno ovaj algoritam radi nad stablima, odnosno **Apstraktnim Sintaksnim Stablima (AST)**. Svaki program može se zapisati u sintaksnom stablu, primer može da bude neka jednačina $x^2 + 5x - 1 = 0$ drugačije zapisana i kao $x * x + 5x - 1 = 0$, njeni sintaksni stablo bi izgledalo:



Slika 45: Primer AST-a

Mozemo primetiti da se konstante i promenljive nalaze na samim listovima **terminalima** ovih stabala, dok se **funkcije** (operacije) nalaze u unutrašnjim čvorovima stabla. Ovako bi izgledalo i stablo bilo kog drugog programa.

Znajući ovo evolucija programa bi se zasnivala kao i kod GA na crossover-u i mutacijama. Prilikom mutacija nije dozvoljeno da list mutiramo tako da on postane neka od funkcija, operatora i slično...

Mada je Koza zamišljaо da ovi programi sami grade programe na osnovu ulaznih parametara...ovo nije realno i mi kao programeri moramo da definišemo pre svega **terminale** i naravno **funkcije** sa kojima GP može da gradi stabla.

Definisanje ovih funkcija i terminala vuče sa sobom potpuno definisanje njihovih ponašanja. U kakvim oblicima su sintaksno i semantički valjni.

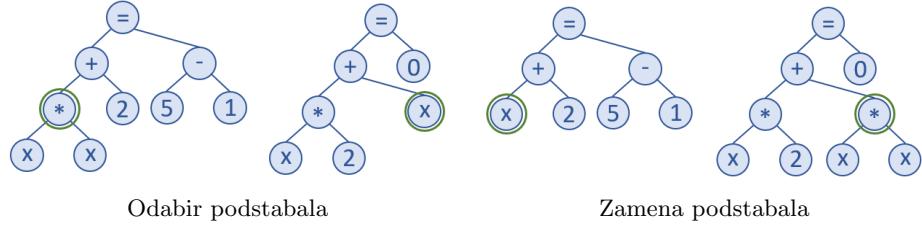
GP-u moramo odrediti **parametre kontrolisanja izvršavanja**. U ove parametre spadaju veličina populacije, šansa za mutaciju, elitizam, veličina greške, broj iteracija odnosno kada se zaustavlja... Takođe bitno, odmah nakon načina zaustavljanja, jeste veličina stabla njegova širina i dubina. Ukoliko se ne ograniči moguće je dobiti stabla koja su redundantno kompleksna npr.:

$$1 * (2/2) * 1 * 1 * (1 * 1 * 1 * 1 * 1 * 1 * x * x + 5 * 1 * 1 * 1 * x - 1) = 0$$

Tolikok duboko stablo, a kad se pogleda ništa ne dobrinosi problemu njegova množenja sa jedan... Bitniji problem koji nastaje ovim jeste vreme za izvršavanje operacija ukrštanja i mutacije. Da bi osigurali da ovakva stabla ne opstaju u populaciji kvarimo im fitness funkciju ili ih totalno brišemo.

Kvalitet programa definišemo sa fitnes funkcijom čija nam vrednost, kao i u svakom od EC algoritama... Uvodimo dodatne osobine ovoj funkciji, a to je da uzima u obzir da li se kod uopšte kompjajlira ili koliko grešaka kompjajler prijavljuje.

Ukrštanja se izvode poprilično intuitivno, prostim zamenom podstabla. Slično kao kod GA, uzmemu dva roditelja i biramo dve pozicije u roditeljima zatim podstabla krenuvši od tih pozicija swap-ujemo (**jednopoziciono ukrštanje**). Dugačije se ova operacija nad stablima zove i **Headless chicken crossover**.

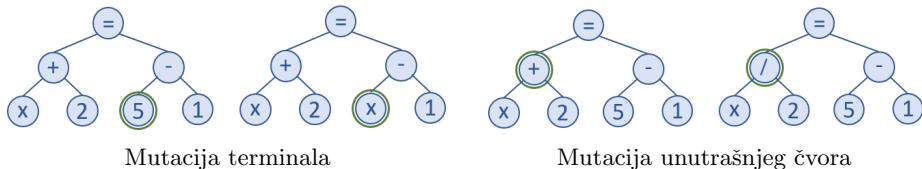


Slika 46: Primer ukrštanja

Sledeće su mutacije. Moramo osmisliti pogodnu mutaciju tako da poštuje osobinu odabranog čvora. Nalik GA prolazimo svaki gen i imamo šansu da ga mutiramo. Prilikom mutacije moramo voditi računa o tome da li je čvor terminal (promenljiva ili konstanta) ili unutrašnji čvor (funkcija tj. operator). Dakle, moramo GP mora voditi računa. U slučaju **mutacije terminala** imamo dve mogućnosti. Prva je da je dati čvor konstanta, u tom slučaju konstantu možemo uvećati ili umanjiti, zameniti drugom konstantom ili zameniti promenljivom... Ako je ipak terminal promenljiva onda slično možemo je zameniti drugom promenljivom ili konstantom.

Ako se vrši **mutacija unutrašnjih čvorova**, kako su ovo funkcije možemo ih zameniti drugim funkcijama. Bitno je pomenuti da moramo voditi računa o sitaksnoj pravilnosti prilikom odabiranja druge funkcije. Ukoliko smanjujemo ili uvećavamo n-arnost funkcije u datom čvoru neophodno je prilagoditi podstablo tako da popunjava ulazne parametre odabrane funkcije.

Još jedna moguća mutacija je poznata kao **mutacija umetanjem novog stabla(Grow)**. Ideja jeste da od odabranog dela stabla brišemo celo podstablo i umesto njega gradimo novo podstablo.



Slika 47: Primer mutacija

Primetimo da ovaj algoritam, mada u teoriji može da gradi program sam proces je za ozbiljnije programe prekompleksan. Mora se voditi računa o veli-

kom broju sintaksnih i semantičkih pravila, što kompleksniji program to je šire i dublje stablo.

Ipak jesu primenljivi, a neki primeri su GP algortimi koji prave **regexe**, **aritmetičke formule**, **logičke formule**, neki laksi programi...

8.3.1 Primer primene GP

Recimo da imamo program koji rešava zadatke iz poznate [Moj broj](#) igre. U ovoj igri igraču je data neka tražena vrednost **b** kao i niz od 6 brojeva **nums**, od igrača se traži da koristeći osnovne operacije (+, -, , /), nađe traženi broj **b** tako da se ni jedan broj iz **nums** ne ponavlja. (igrač ne mora da iskoristi sve brojeve)

U ovom primeru, prepostavimo da smo implementirali sve operacije kao binarno stablo, primetimo da nam nije optimalno da GP pravi preduboka stabla, ili stabla koja imaju više od 11 čvorova. Ovo nije magičan broj... znamo da ako stablo ima **n** listova na poslednjem nivou onda je maksimalan broj čvorova nivoa iznad $n/2$, nivoa iznad $n/4$ 1. Dakle, formula popunjenoog binarnog stabla bi bila $2n - 1$. Samim tim ne bi želeli da nam GP pravi bilo koja stabla čiji je ukupan broj čvorova veći od 11.

9 Inteligencija grupa (SI - Swarm Intelligence)

Inteligencija grupa predstavlja proces u kojem jedinke istražuju svoja lokalna rešenja dok istovremeno komuniciraju i sarađuju sa ostalim članovima grupe. Kroz ovu interakciju dolazi do globalnog ponašanja, što omogućava grupi da kolektivno pronađe optimalno rešenje problema ili se prilagodi promenljivim uslovima u okruženju.

Jedna od bitnih osobina ovih algoritama je da su dinamički, odnosno ukoliko se prostor menja tokom rada algoritma, sposobni su da se prilagode i nađu novo rešenje. Primer ovakvog ponašanja je ako postavimo kamen na put koji je oformljen ka hrani od strane mrava, brzo će naći novi optimalni put ka istoj toj hrani.

Ovi algoritmi pripadaju **P - Metaheuristikama**, i često se koriste na problemima **kontinualne** prirode. Osobina računara da brojeve čuvaju u ograničenoj memoriji, **No Free Lunch (NFL)** teorema tvrdi da su svi **kontinualni** problemi rešavani na računarima ustvari **diskretni** problemi.

Razlika ovih algoritama od EA jeste u činjenici da jedinke ovde rade zajedno, tj. kreću se zajedno u smeru rešenja.

Kao što smo već rekli glavna osobina ovih algoritama jeste komunikacija između jednici. Prvo kako ovo možemo da simuliramo jeste tako što svaka jedinka komunicira sa svakom drugom na "ličnom" nivou. Ovakav pristup se zove i **direktan**. sa druge strane moguća je **indirektna** komunikacija. Postiže se tako što svaka jedinka komunicira sa okruženjem, npr. okruženje pamti najbolje rešenje i toj informaciji pristupa svaka jednika.

Neki od primera su:

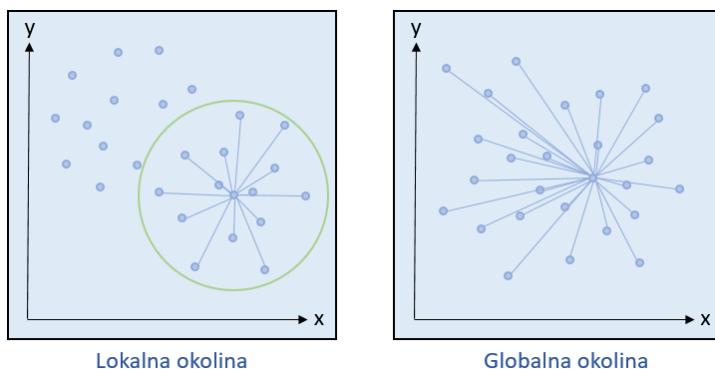
1. **Particle Swarm Optimization (PSO)** - svaka čestica računa novu inerciju poštovaći najbolje rešenje grupe + njeno najbolje rešenje + kao i njenu staru inerciju.
2. **Ant colony optimization (ACO)** - svaki mrav ostavlja feromone prilikom traženja hrane, jačina feromona predstavlja kvalitet rešenja koji drugi mravi mogu i ne moraju da prate.
3. **Artificial Bee Colony (ABC)** - inspirisan grupama pčela u potrazi za nektarom, postoje tri vrste pčela **izviđači** (diverzifikacija), **zaposleni** (traže lokalna rešenja) i **posmatrači** (intenzivikacija, biraju gde će se vršiti istraživanje).
4. **Bacterial foraging optimization (BFO)** - bakterije se razmnožavaju u onim delovima gde je najbolje rešenje, u svakoj generaciji se odstranjuje polovina rešenja.

9.1 Particle Swarm Optimization (PSO)

U osnovnom PSO (Particle Swarm Optimization) algoritmu imamo grupu čestica, tako nazvanih zbog osobine da svaka jedinka ima brzinu i ubrzanje, u svakoj iteraciji se računa nova pozicija svake čestice. Svaka čestica čuva **svoju najbolju poziciju čestice** (p_{local}), **inerciju čestice** (v_i) kao i **trenutnu poziciju čestice** (p_{pos}). Dok se globalno pamti i po potrebi menja najbolja pozicija grupe (p_{global}).

Jedan ovakav algoritam smo pretsavili u poglavlju 7.2. Gde je opisan takozvani **Global best (gbest)** pristup rešavanja ovih problema. Rešava problem tako što dozvoljava svakoj jednici da pristupi svim informacijama globalno. Svaka jedinka zna najbolje rešenje kao i poziciju svakog drugog rešenja.

Suprotno **Local best (lbest)** imaju pristup samo informacijama u svojoj "okolini". Okolina se može definisati kao udaljenost drugih jedinika od trenutne... Samim tim kako nemaju uvid u globalno stanje sistema algoritam sporije konvergira kao rešenju.



9.2 Ant Colony Optimization (ACO)

Inspirisan ponašanjem kolonije mrava. Kada pronađu neku hranu mravi traže najoptimalniji, tj. najkraći put do te hrane.

Prilikom kretanja kroz odabrani put svaki mrav iza sebe ostavlja feromone, koji vremenom isparavaju. Drugi mravi imaju opciju da prate one puteve čiji su feromoni jači. U povratku mrava nazad u mravinjak dodatno postavljaju feromone na isti taj put. Samim tim, ako je put kraći, mrav će češće da prelazi put i feromoni će biti snažniji, neće stići da ispare do kraja, što privlači druge mrave. Ovakovo ponašanje možemo da iskoristimo u problemima sa grafovima, najbolji primer je **Traveling Salesman Problem (TSP)** gde tražimo najkraći put, a da pritom posetimo sve gradove.

U fazi inicijalizacije, **svakom mravu** je dodeljen **drugačiji čvor** kao početni, kako bi se izbeglo favoritizovanje rešenja na osnovu heuristike. [6]

Iz prethodnog opisa jasno je da pre svega mrav mora da pamti **svoj početni čvor** i kada traži rešenje uvek kreće iz tog čvora. Na kraju sake iteracije neop-

hodno je da mrav zna koji je put prešao da bi znao kuda treba da se vrati, mora da pamti **trenutno rešenje (niz čvorova)**. Svaki mrav postavlja feromone a snagu svojih feromona određuje na osnovu **dužine puta** koju mora da zna i **snage feromona (Q)**.

Jedino što okruženje mora da pamti jeste **količina feromona na svim granama**.

Algoritam 10 Pseudokod ACO

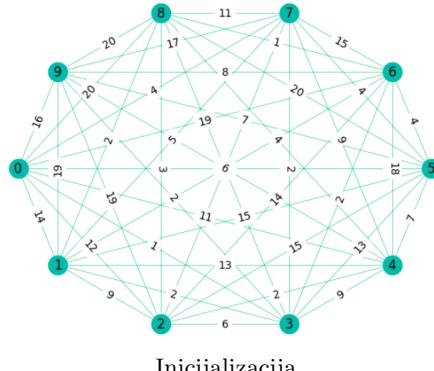
```

Generiši inicijalnu populaciju  $P_0(n)$ 
Inicijalizuj feromone na svim granama  $pheromones(m)$ 
while nije ispunjen određeni cilj ili kriterijum do
    for ant in  $P(n)$  do
        resetSolution(ant)
        findSolution(ant)
    end for
    updatePheromones( $pheromones(m)$ )
end while

```

9.2.1 Primer TSP problema rešen ACO algoritmom

TSP sa 10 gradova. U prvom delu koda inicijalizujemo svakog mrava, mrava ima onoliko koliko ima čvorova (bar je to praksa) i svaki mrav kreće iz drugačijeg čvora. Inicijalizujemo i količinu feromona na svim granama:



Nakon inicijalizacije, započinjemo petlju:

1. **resetSolution()** - nakon svake iteracije je potrebno da se resetuje rešenje svakog mrava, kako bi moglo da se nađe novo.
2. **findSolution()** - mrav traži novo rešenje, odnosno novi put na osnovu feromona i odabrane heuristike.
Svaki mrav bira sledeći čvor dok nije posetio sve čvorove. Gledajući čvor na kom se trenutno nalazi (i - čvor), mrav posmatra sve one koji su susedni

i neposećeni ($j = susedi(i)$), bira put koji će preći na osnovu formule:

$$p_{i,j} = \frac{\tau_{i,j}^\alpha n_{i,j}^\beta}{\sum_{k=susedi(i)}^n \tau_{i,k}^\alpha n_{i,k}^\beta}$$

U ovoj formuli:

- (a) $\tau_{i,j}$ - predstavlja količinu feromona na grani (i, j) u globalnoj promenljivoj $pheromones(m)$.
- (b) $n_{i,j}$ - predstavlja odabranu heuristiku koja će se koristiti za računanje udaljenosti čvora i od j . Kod nas u primeru koristimo heuristiku:

$$n_{i,j} = \frac{1}{distance[i][j]}$$

- (c) α - stepen koji odlučuje koliko će feromoni uticati na verovatnoću odabira tog čvora. Uglavnom uzima vrednost između [1, 2].
- (d) β - stepen koji odlučuje koliko će heuristika za optimalnost dužine puta uticati na verovatnoću odabirata tog čvora. Uglavnom uzima vrednost između [2, 5].

U odnosu na verovatnoće dobijene prethodnom formulom za sve susede, mrav stohastički bira sledeći čvor. Nakon odabira čvora, on se dodaje na rešenje i dužina puta se povećava za pređeni put.

3. **updatePheromones()** - na kraju svake iteracije $pheromones(m)$ se ažurira sledećom formulom:

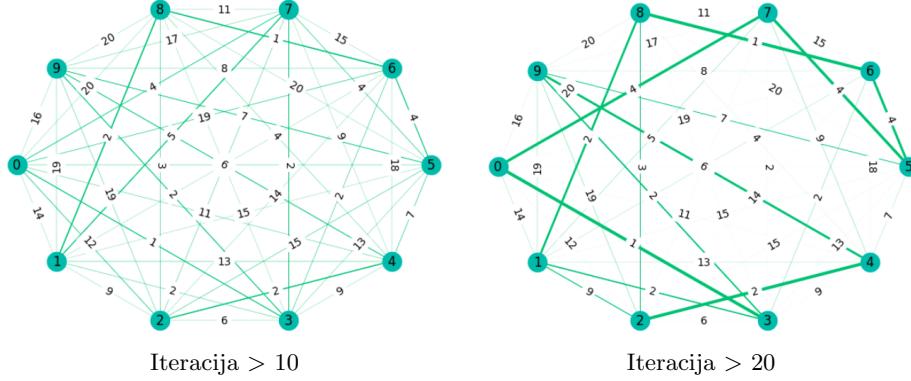
$$pheromones(i, j) = (1 - p)pheromones(i, j) + \sum_{k=1}^n \Delta\tau_{i,j}^k$$

U prethodnoj formuli:

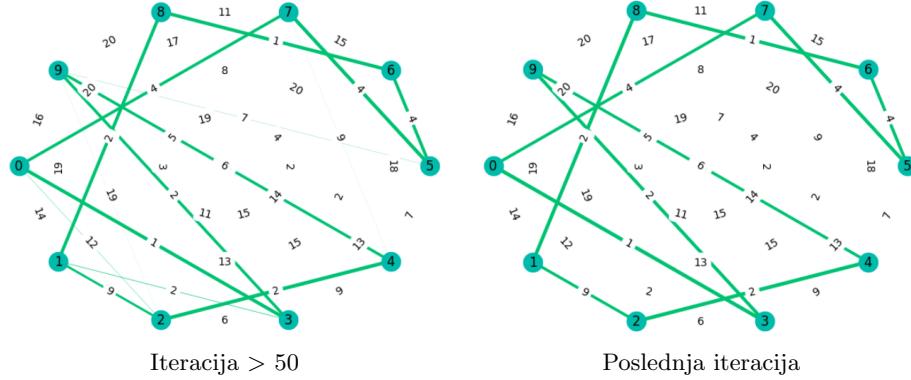
- (a) **(1 - p)** - koliko će feromoni posle svake iteracije da isparavaju, **p** je predefinisano pre početka algoritma, vrednost između [0, 1]. Na ovaj način stari feromoni imaju manji uticaj na odabir sledećeg čvora čime samo rešenje vremenom konvergira.
- (b) $\Delta\tau_{i,j}^k$ - količina feromona koji mrav k ostavlja na grani (i, j) , definisana sledećom formulom:

$$\Delta\tau_{i,j}^k = \frac{Q}{pathLen(k)}$$

Q je predefinisana konstanta, uglavnom broj u intervalu [1, 100]. Delimo sa ukupnom dužinom puta koji je mrav k prešao, radi postizanja prethodno navedene politike da kraći put ima snažniji miris feromona.



Primetimo da one grane koje su češće posećivane se više ističu.



Put sa najboljim rešenjem će se na kraju najvipe istaći.

9.3 Artificial Bee Colony (ABC)

Algoritam je zasnovan na ponašanju pčela prilikom traženja polena. Pčele komuniciraju plesom, kojim govore ostalim pčelama gde se nalazi više polena, odnosno gde je kvalitetnije rešenje.

Želimo da simuliramo tri ponašanja pčela. Prvo i logično ponašanje je kupljenje polena. Ove pčele nazivamo **zaposlene pčele (Employed bees)**. Kod nas kupljenje polena predstavlja pronalaženje lokalnog minimuma! Dok skuplja polen, pčela vrši ples i time govori drugim pčelama da dođu i skupljaju polen.

One pčele koje su blizu da "iscrpe" svoj resurs polena žele da nađu sledeći resurs. U ovom momentu postaju **posmatrači pčele (Onlooker bees)**. Gledajući ples drugih pčela biraju onaj resurs pčele koja ima najbolji ples (najbolje lokalno rešenje)! U ovom momentu mogu da izaberu da nastave iscrpljivanje svog resursa ili da pređu na neki bolji.

Šta ako pčela nema više koristi od svog resura? U ovom momentu ona mora da se skloni sa svog resursa i odaberi neki drugi, međutim druge pčele nemaju

primamljive resurse... Naša pčela bira da postane **izviđač pčela (Scout bees)**!. Ona pokušava da pronađe novi resurs polena koji nijedna druga pčela nije istražila.

Dakle ideja je sledeća, zaposlene pčele prikuljuju polen (poboljšava lokalno svoje rešenje , tj. eksploracija). Pčele posmatrači nadgledaju i traže one pčele koje su našle najviše polena i šalju druge pčele da rade na tim pozicijama (konvergencija rešenja). Na kraju pčele istraživači traže druge izvore polena (eksploracija).

Ove pčele su podeljene u tri grupe radi lakšeg zamišljanja problema. U praksi ne postoji tri grupe već tri faze kroz koje prolazi svaka pčela.

Algoritam 11 Pseudokod ABC

```

Generiši inicijalnu populaciju  $P_0(n)$ 
Nadji globalniMin = min( $P(n)$ )
while nije ispunjen određeni cilj ili kriterijum do
    for bee in  $P(n)$  do
        employedPhase(bee)
        onlookerPhase(bee)
        scoutPhase(bee)
    end for
end while

```

Da bi algoritam funkcionišao svaka pčela mora da zna pre svega svoju **poziciju (pos)** i svoju **najbolju poziciju (bestPos)**.

Kako bi znala kada da dalje u neku od drugih faza neophodan joj je brojač **neuspšeno poboljšanje (failedImprovement)**. Ova promenljiva meri koliko dugo pčela nije poboljšala svoje rešenje ukoliko ovaj broj prekorači predodređenu vrednost **strpljenja** pčela postaje izviđač i uzima novo nasumično rešenje.

Logično pčela mora da zna kako da se ponaša u svakoj fazi gde stupaju sledeće funkcije:

1. **employedPhase()** - pčela vrši lokalnu pretragu svoje okoline, ukoliko ne nađe bolje rešenje failedImprovement se povećava.
2. **onlookerPhase()** - pčela postaje posmatrač i ako naiđe na pčelu koja ima bolje rešenje od njenog najboljeg rešenja prebacuje se na tu poziciju.
3. **scoutPhase()** - pčela postaje izviđač onog momenta kada se pređe strpljenje.

9.3.1 Primer primene ABC algoritma

Tražimo minimum funkcije:

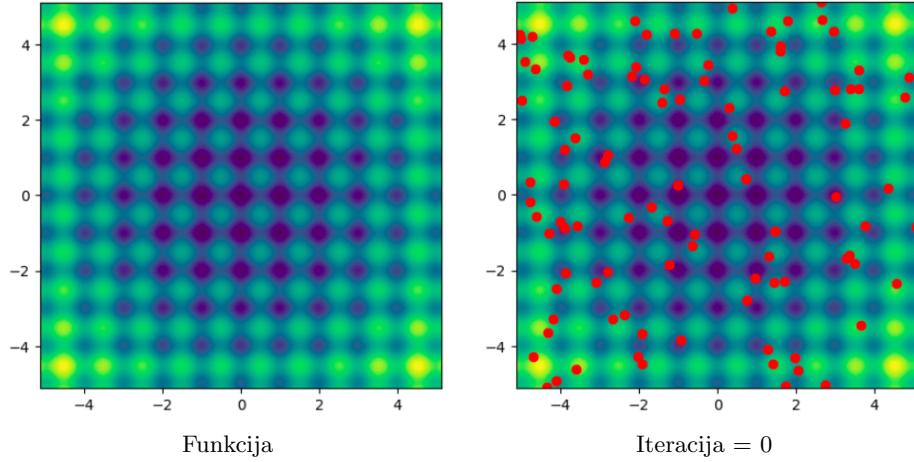
$$f(x, y) = 20 + (x^2 - 10 \cos(2\pi x) + y^2 - 10 \cos(2\pi y))$$

poznata i kao **"Rastrigin function"**, tražimo rešenje na $-5.12 \leq x, y \leq 5.12$.

Hladnije boje na grafu predstavljaju niže vrednosti a toplige predstavljaju više vrednosti.

U prvom koraku pseudokoda videli smo da postavljamo sve pčele na pseudo-nasumične pozicije, računamo redom fitness svake i uzimamo onu vrednost koja je globalno najbolja.

Pravimo n pčela na dopuštenom domenu, $(x, y) | -5 \leq x, y \leq 5$.



U svakoj iteraciji za svaku pčelu vršimo sve tri faze.

1. Employed Phase

Pčela istražuje okolinu svog trenutnog rešenja koristeći trenutno najbolje globalno rešenje pouplacije:

$$p_{new} = p_{old} + \phi_i(p_{localBest} - p_{random})$$

U ovoj formuli ϕ_i je vrednost između -1 i 1, koja uvodi stohastičnost u celu pretragu, a p_{random} predstavlja pseudo-nasumičnu poziciju neke od pčela iz populacije.

Ukoliko je novonađena pozicija kvalitetnija, pored izmene najboljeg lokalnog rešenja, proverava se da li je globalno bolja.

2. Onlooker Phase

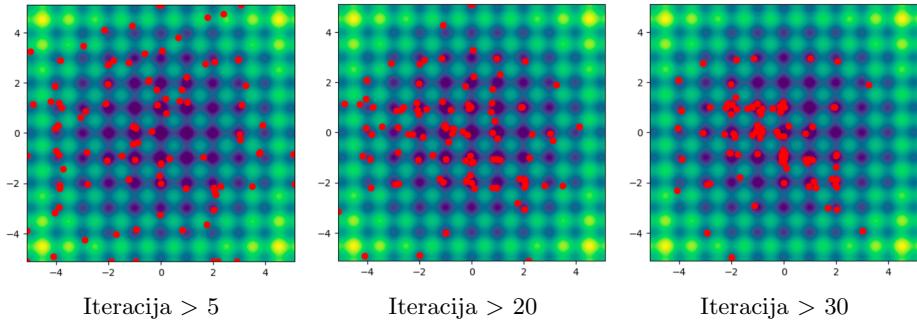
Pčela bira neku od pčela po principu ruleta, gde svaka od pčela ima šansu da bude izabrana na osnovu sledeće formule:

$$p_i = \frac{f_i}{\sum_{i=1}^n f_i}$$

Ukoliko je vrednost najboljeg rešenja izabrane pčele bolja od nabolje vrednosti pčele posmatrača, pčela posmatrač odbacuje sve resurse i prebacuje se na izabranu pčelu, inače se ne pomera.

3. Scout Phase

Ukoliko se kvalitet rešenja pčele nije poboljšalo predodređen broj iteracija pčela bira neku nasumičnu poziciju unutar domena.



10 S - Metaheuristike

Prethodno opisani kao metaheuristike koje se bave jednim rešenjem koje konstantno unapređuju.

U poglavlju 7.1 opisna je VNS metaheuristika, a ovde ćemo dodatno opisati **Simulirano kaljenje** (**Simulated Annealing**) i **Tabu pretragu** (**Tabu search**).

10.1 Simulated Annealing (SA)

U obradi čelika teknika **kaljenja** funkcioniše po principu zagrevanja i ponovnog hlađenja. Čestice prilikom zagrevanja dobijaju energiju što omogućava njihovo kretanje unutar kristalne rešetke, menjanje pozicije, prilikom hlađenja čestice gube energiju i polako zauzimaju novu poziciju. Ovaj proces, sporog hlađenja, dovodi do ojačavanja tvrdoće čelika.

Po uzoru na ovaj proces nastala je i optimizacija Simuliranog kaljenja. Mi ne radimo sa čelikom već funkcijama. U našem slučaju temperatura predstavlja šanse da rešenje stremi od "dobrog" rešenja i uzima neko lošije u cilju izbavljanja iz lokalnog minimuma.

Na početku algoritma "temperatura" sistema je podešena na 1 koja se vremenom smanjuje. Kako se temperatura smanjuje ta verovatnoća se smanjuje što simulira proces kaljenja.

Algoritam 12 Pseudokod SA

```
Generiši rešenje  $x_{local}$ 
Definišemo  $i = 0$ ,  $x_{best}$ 
while nije ispunjen određeni cilj ili kriterijum do
    if  $f(x_{i+1}) < f(x_{best})$  then
         $x_{best} = x_{i+1}$ 
         $x_{local} = x_{i+1}$ 
    else if  $random() < temp(i)$  then
         $x_{local} = x_{i+1}$ 
    end if
     $i = i + 1$ 
end while
```

10.1.1 Primer primene SA

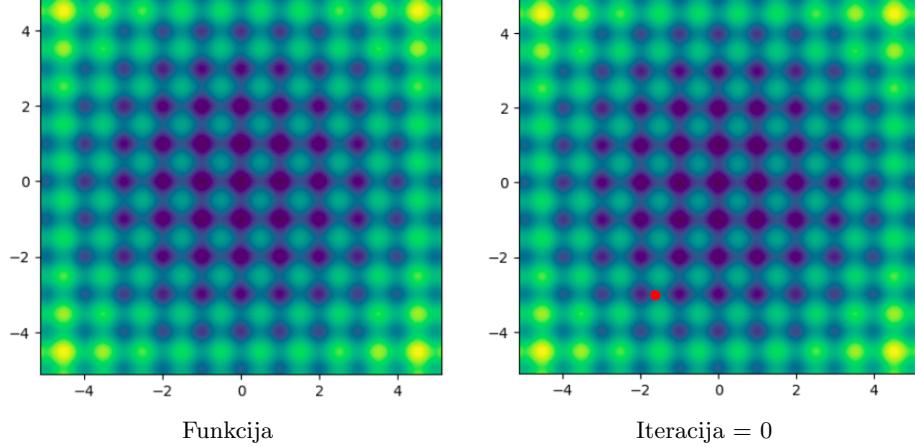
Tražimo minimum funkcije:

$$f(x, y) = 20 + (x^2 - 10 \cos(2\pi x) + y^2 - 10 \cos(2\pi y))$$

poznata i kao "**Rastrigin function**", tražimo rešenje na $-5.12 \leq x, y \leq 5.12$.

Hladnije boje na grafu predstavljaju niže vrednosti a toplije predstavljaju više vrednosti.

Na samom početku programa pravimo neko pseudo-nasumično rešenje, postavljamo to rešenje kao najbolje i brojač iteracija postavljamo na 0. U našem kodu kao uslov zaustavljanja biramo broj iteracija, mada bi možda u ovom primeru bilo bolje da posmatramo kada je razlika najboljeg i trenutnog novog rešenja $< \epsilon$. Na slikama crvena tačka predstavlja novo nađeno bolje rešenje, dok će crne tačke predstavljati rešenje koje je uzeto a koje je lošije nego najbolje.



U sledećem koraku biramo novu tačku u okolini sadašnje tačke koju posmatramo. Sam proces nalaženja nove tačke može se raditi na različite načine u zavisnosti od problema. U našem slučaju koristili smo pomeraj po uniformnoj raspodeli na intervalu $[-1, 1]$, odnosno:

$$x_{i+1,j} = x_{local,j} + U(-1, 1) \quad j \in [0, n]$$

gde je \mathbf{n} veličina prostora u kom stražimo rešenje.

Kada smo našli novo rešenje moramo da vidimo da li je ono bolje ili lošije od prethodnog najboljeg. U slučaju da je bolje čuvamo to rešenje kao najbolje povećavamo brojač $i = i+1$ zatim tražimo sledeće rešenje preko iste funkcije **move()**. U slučaju lošijeg rešenja koristimo **temperaturu** da bi odredili da li užimamo rešenje ili ne.

Temperatura može da bude bilo koja opadajuća funkcija:

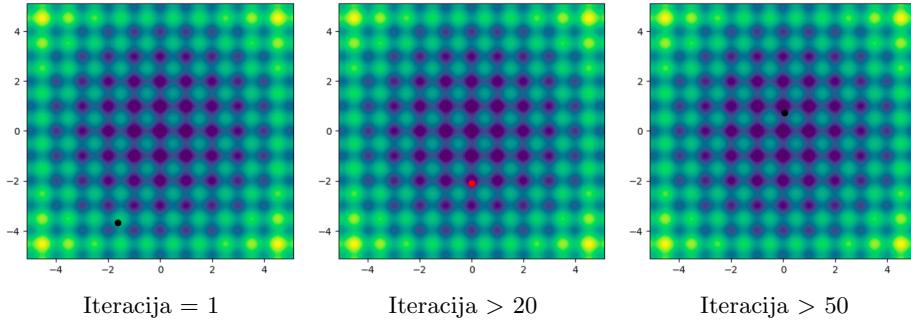
1. Logaritamska

$$temp(i) = T_0 * \frac{\ln(2)}{\ln(i+1)}$$

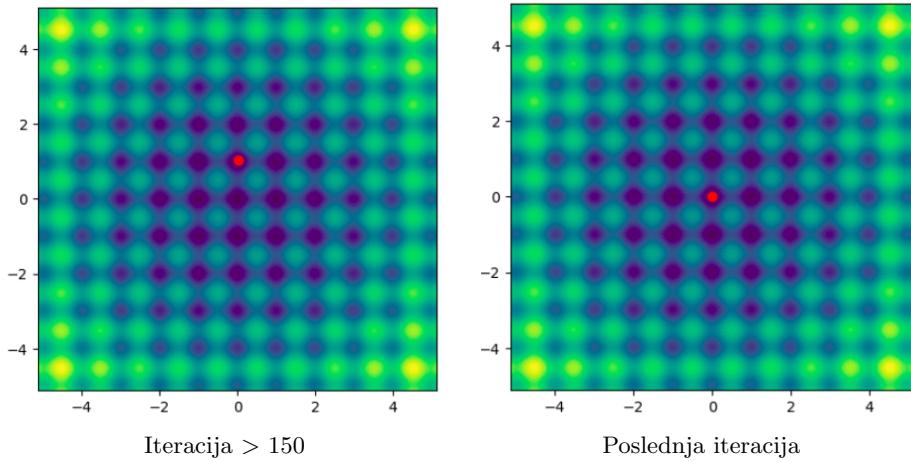
2. Brzo kaljenje

$$temp(i) = \frac{T_0}{i+1}$$

Ove funkcije na početku tolerišu sva lošija rešenja dok se kasnije, kako prolaze iteracije, ta tolerancija smanjuje. Ovaj princip u isto vreme na samom početku praktikuje eksploraciju a kasnije i eksploataciju prostora rešenja.



Vidimo da iako je lošije rešenje u često algoritam izabere to rešenje dok se to kasnije menja. Kasnije iteracije su strožije.



10.2 Tabu Search (TS)

Tabu pretraga temelji se na samom značenju reči "tabu", što u ovom kontekstu označava nešto zabranjeno. Termin potiče iz jezika naroda Tonge.

Slično ovoj definiciji, i algoritam tabu pretrage funkcioniše tako što nameće ograničenja koja sprečavaju izbor određenih rešenja tokom pretrage. Ključni aspekt ovog pristupa jeste korišćenje memorije, kratkoročne ili dugoročne. Ona pamti već posećena rešenja i na taj način sprečava njihovo ponavljanje. Ove strukture se nazivaju **tabu liste** i upravo one čine ovaj algoritam specifičnim.

Zabranom povratka na prethodno posećena rešenja algoritam je prisiljen da istražuje nove oblasti u prostoru rešenja. Na ovaj način povećava šanse za pronađenje boljeg rešenja. Međutim, zbog velikog broja potencijalnih rešenja, nije praktično pamtitи sva posećena. Umesto toga, u praksi se najčešće pamte samo ključne tačke ili se koristi ograničena memorija koja evidentira samo nekoliko poslednjih rešenja.

Važno zapažanje kod ovog algoritma jeste potreba da se u određenim trenucima prekrše tabu pravila. Ova praksa je poznata kao **kriterijum težnje (aspira-**

tion criteria). Odabirom rešenja koje je označeno kao tabu imamo šanse da odaberemo ono koje vodi ka značajno boljem od trenutnog najboljeg rešenja. Ovaj pristup osigurava fleksibilnost algoritma i pomaže u izbegavanju lokalnih minimuma.

Najvažniji deo tabu pretrage jeste definisanje načina pronalaženja susednih rešenja i odabira odgovarajućeg regiona za pretragu. Ove definicije su specifične za svaki problem i ključne su za uspešno funkcionisanje algoritma.

Algoritam 13 Pseudokod TS

Generiši rešenje x_{local}

Definišemo $T.append(x_{local})$, $x_{best} = x_{local}$

while nije ispunjen određeni cilj ili kriterijum **do**

if $f(x_{local}) < f(x_{best})$ **then**

$x_{best} = x_{local}$

end if

$T.append(x_{local})$

end while

10.2.1 Primer primene TS

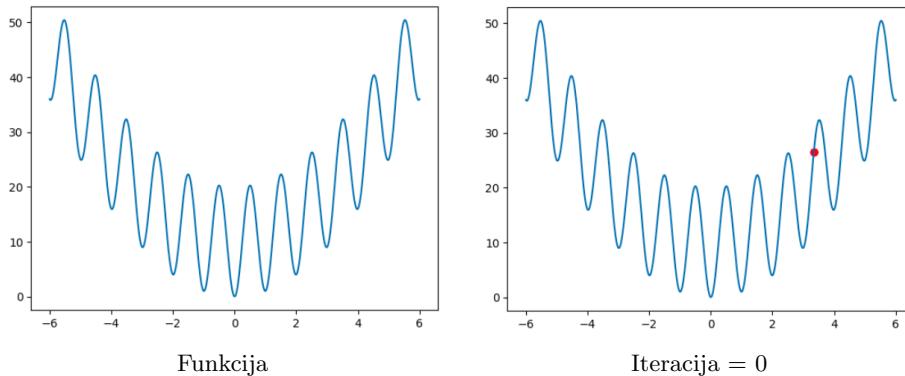
Tražimo minimum funkcije

$$f(x, y) = 10 + (x^2 - 10 \cos(2\pi x))$$

poznata i kao **"Rastrigin function"**, tražimo rešenje na $-5.12 \leq x \leq 5.12$.

Ljubičastom bojom predstavljamo trenutno lokalno rešenje, crvenom bojom najbolje nađeno rešenje i sivom prethodna rešenja.

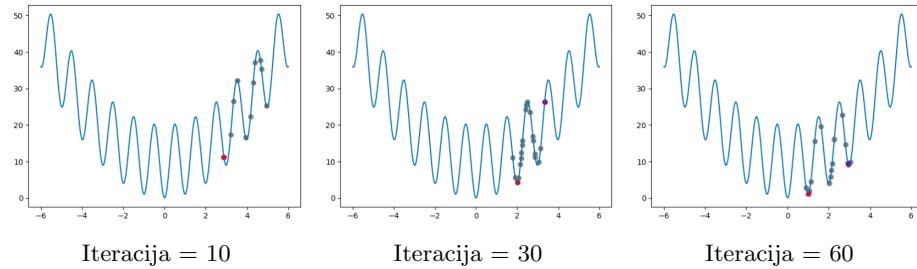
Po početku algoritma inicijalizujemo početno rešenje, što je i trenutno najbolje.



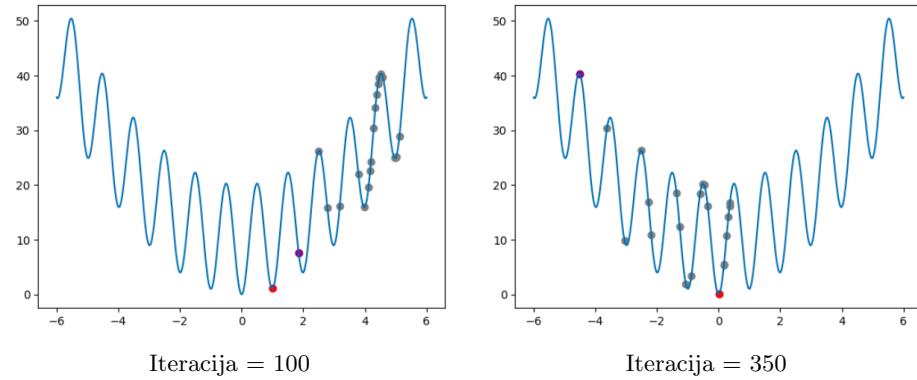
Na dalje tražimo sledeće rešenje koristeći formulu kao u primeru za SA:

$$x_{i+1} = x_{local} + U(-1, 1)$$

Ono što dodatno proveravamo jeste da li je ovo rešenje u listi T (tabu lista), ako jeste tražimo sledeće rešenje po istoj formuli. Možemo dodati dodatan uslov da iako rešenje nije u tabu listi ako je lošije od ostalih rešenja u tabu listi ipak ne bude izabранo. Kako ovde može doći do beskonačne petlje uvodimo dodatan uslov šanse da se ipak uzme tabu rešenje (aspiration criteria).



Vidimo da su tačke iz starijih iteracija obrisane. Ne želimo da čuvamo previše informacija, pa brišemo sve tačke koje su starije od \mathbf{m} iteracija.



Literatura

- [1] Nello Cristianini. “On the current paradigm in artificial intelligence”. U: *AI Communications* 27.1 (2014.), str. 37–43. DOI: [10.3233/AIC-130582](https://doi.org/10.3233/AIC-130582). URL: <https://doi.org/10.3233/AIC-130582>.
- [2] George B Dantzig. “Linear programming”. U: *Operations research* 50.1 (2002.), str. 42–47.
- [3] Kalyanmoy Deb i Jayavelmurugan Sundar. “Reference point based multi-objective optimization using evolutionary algorithms”. U: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 2006., str. 635–642.
- [4] John R Koza. “Genetic programming as a means for programming computers by natural selection”. U: *Statistics and computing* 4.2 (1994.), str. 87–112.
- [5] Adam Slowik i Halina Kwasnicka. “Evolutionary algorithms and their applications to engineering problems”. U: *Neural Computing and Applications* 32.16 (2020.), str. 12363–12379.
- [6] Thomas Stützle, Marco Dorigo i dr. “ACO algorithms for the traveling salesman problem”. U: *Evolutionary algorithms in engineering and computer science* 4 (1999.), str. 163–183.