Optimalno punjenje kontejnera pomoću evolucijskih algoritama

Dujmović, Nikolina

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Nikolina Dujmović

OPTIMALNO PUNJENJE KONTEJNERA POMOĆU EVOLUCIJSKIH ALGORITAMA

Diplomski rad

Voditelj rada:
doc. dr. sc. Nela Bosner

Zagreb, Rujan, 2021

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mojim roditeljima i bratu koji su zajedno sa mnom proživjeli sve godine mojeg studiranja

Sadržaj

Uvod

Transport robe u kontejnerima je logistička osnova današnje svjetske ekonomije i važan problem kod upravljanja tim transportom je optimalno punjenje kontejnera. Pod tim se podrazumijeva da se određen broj različitih objekata, različitih dimenzija, smjesti u jedan ili više kontejnera tako da se iskoristi što više prostora unutar kontejnera, odnosno da se upotrijebi što manji broj kontejnera za transport tih objekata. U principu, ovaj problem je NP-težak pa ćemo za njegovo rješavanje u ovoj radnji ponuditi evolucijske algoritme. Na početku ćemo prikazati osnovne alate evolucijskih algoritama koji će biti potrebni za rješavanje danog problema te opisati nekoliko varijanti evolucijskih algoritama. Na kraju bi se predloženi algoritmi ilustrirali programima izrađenima u MATLAB-u, koji će biti primijenjeni na konkretnom problemu.

Poglavlje 1

Osnovni pojmovi i definicije evolucijskih algoritama

Prirodno se nameće pitanje kako se evolucija i rješavanje problema optimizacije povezuju. Prilikom promatranja evolucije između ostalog u obzir uzimamo okoliš, populaciju, jedinku te sposobnost za preživljavanje ili razmnožavanje. To možemo povezati s rješavanjem problema tako da problem promatramo kao okoliš, skup mogućih rješenja kao populaciju, moguće rješenje kao jedinku te sposobnost za preživljavanje ili razmnožavanje kao kvalitetu rješenja. Sada problem optimizacije možemo promatrati kao traženje najjače jedinke koja zadovoljava određene uvjete.

Kroz godine razvile su se 4 struje evolucijskog računanja. Nije naodmet napomenuti da je ovo poprilično mladi smjer rješavanja problema. 1960-ih su se razvile prve varijante evolucijskih algoritma i to evolucijsko programiranje, genetski algoritmi te evolucijska strategija. Kasnije, 1990-ih pojavilo se još i genetsko programiranje. U poglavlju 2 ćemo opisati po čemu je koja varijanta posebna. U ovom poglavlju ćemo navesti zajedničku ideju te pojmove koji su nam bitni za sve varijante.

1.1 Glavna ideja evolucijskih algoritama

Sve varijante evolucijskih algoritama imaju istu ideju. Promatramo populaciju u nekom okruženju s ograničenim sredstvima. Borba za ta sredstva uzrokoje prirodnu selekciju, koja uzrokuje povećanje sposobnosti cijele populacije. Prilikom kreiranja nove generacije populacije funkcijom vrednovanja tražimo najsposobnije jedinke. Zatim varijacijskim operatorima (rekombinacija i mutacija) kreiramo nove jedinke te iz nove povećane populacije funkcijom vrednovanja dobivamo populaciju početne veličine s novim jedinkama. Ovaj proces iteriramo dok ne dobijemo dovoljno dobrog kandidata.

Slijedi i pseudokod gore opisanog procesa:

```
inicijaliziraj populaciju slučajnim odabirom
ocijeni svaku jedniku iz populacije
while određeni uvjet zadovoljen do
    odaberi roditelje
    primijeni varijacijske operatore na roditeljima
    ocijeni nove jedinke
    odaberi jedinke za sljedeću generaciju
end while
```

1.2 Komponente evolucijskih algoritama

Kako je već navedeno, postoji nekoliko varijanti evolucijskih algoritama. One se najčešće razlikuju po reprezentaciji rješenja, tj. strukturama koje koristimo za opis jedinki. Uz to, potrebno je i precizno definirati brojne komponente, procedure i operatore.

Najvažnije komponente evolucijskih algoritama su: reprezentacija jedinki, funkcija vrednovanja, populacija, način odabira roditelja, varijacijski operatori te odabir nove populacije. Ako želimo da algoritam stane u nekom trenutku dodatno moramo definirati i uvjet zaustavljanja.

Reprezentacija

Uz jedinke vežemo dva različita tipa prikaza, to su fenotip i genotip. U svijetu biologije fenotip bi predstavljao vanjski izgled jedinke, dok bi genotip bio genetska struktura koja čini taj izgled. U svijetu računarstva, mogli bismo za primjer uzeti binarni zapis brojeva gdje je genotip 0110, a fenotip bi tada bio broj 6. Valja napomenuti da se selekcija vrši na temelju fenotipskih karakteristika, što se vidi u jednadžbi 1.1, a genotip nam govori kako je taj fenotip nastao.

Kvalitetna reprezentacija ili definicija jedinki nam može uvelike pomoći pri određivanju koju varijantu evolucijskog algoritma ćemo koristiti. Ako se za rješavanje danog problema odlučimo za reprezentaciju stringovima, najadekvatniji će nam biti genetski algoritmi. Ako se odlučimo za reprezentaciju realnim vektorima, poslužit će nam evolucijske strategije. Dok će nam reprezentacija stablima biti najbolja za genetsko programiranje.

Dodatno, moramo voditi računa o kakvoj je točno reprezentaciji riječ jer imamo dva različita značenja. Ako govorimo o mapiranju iz fenotipa u genotip pričamo o kodiranju, a ako govorimo o obratnom mapiranju (iz genotipa u fenotip) pričamo o dekodiranju. Uglavnom ćemo koristiti pojam kodiranja, a iz samog problema će biti jasno o kojem mapiranju govorimo.

Funkcija vrednovanja

Funkcija vrednovanja predstavlja potrebe koje populacija treba usvojiti i definira što to znači napredak. Kod problema optimizacije, ova funkcija se može poistovjetiti s funkcijom cilja.

Jedna vrijednost funkcije vrednovanja ne znači nužno da je prisutan samo jedan fenotip, isto tako jedan fenotip ne garantira prisutnost samo jednog genotipa. Ali, ako je prisutan jedan genotip to povlači da imamo samo jedan fenotip te samo jednu vrijednost funkcije vrendovanja. Matematičkim rječnikom bi to bilo: Neka su x i y neki fenotipovi, X i Y odgovarajući genotipovi te f funkcija vrednovanja. Gornju izjavu možemo zapisati na sljedeći način:

$$\begin{gathered} f(x) = f(y) \nRightarrow x = y, x = y \nRightarrow X = Y \\ X = Y \Rightarrow x = y \Rightarrow f(x) = f(y) \end{gathered} \tag{1.1}$$

Populacija

Populacija je multiset genotipa (u multisetu dopuštamo postojanje dva ista elementa). Uz broj jedinki populaciju mogu određivati i neki drugi uvjeti kao što je npr. udaljenost između dvije susjedne jedinke. Za evolucijske algoritme gotovo uvijek koristimo konstantnu veličinu populacije te se ona označava s μ. Funkcije vrednovanja primijenjujemo na cijelu populaciju, dok varijacijske operatore primjenjujemo na jednu ili više odabranih jedinki.

Način odabira roditelja

Način odabira roditelja zajedno s odabirom nove populacije je odgovoran za poboljšanja u populaciji. Jedinka postaje roditelj ako je prošao odabir po vrijednosti funkcije vrednovanja. Način odabira roditelja je vjerojatnosni u ovisnosti o vrijednosti funkcije vrednovanja. Jedinkama s nižim vrijednostima se uglavnom daje mala vjerojatnost jer bi u protivnom populacija mogla zaglaviti u lokalnom optimumu. Imamo nekoliko mogućnosti za odabir roditelja:

- odabir proporcionalan podobnosti , eng. *fitness proportional selection* (FPS) - vjerojatnost da će jedinka i biti izabrana jednaka je kvocijentu njene vrijednosti fukncije vrednovanja i zbroja vrijednosti funkcija vrednovanja svih jedinki, $p_i = \frac{f(i)}{sum_{j \in P} f(j)}$
- odabir rangiranjem - populacija se rangira na temelju sposobnosti preživljavanja te im se dodjeljuje vjerojatnost na temelju ranga, a ne vrijednosti funkcije vrednovanja
- algoritam baziran na ruletu - rupe u koje kuglica može pasti unutar ruleta predstavljaju vjerojatnosti odabira jedinke

- turnir - najčešće se koristi za velike populacije te populacije kod kojih nije lako odrediti sposobnost preživljavanja. Možemo usporediti bilo koje dvije jedinke i odabiremo onu koja ima veću sposobnost preživljavanja

Varijacijski operatori

Zadaća varijacijskih operatora je da iz starih jedinki kreiraju nove. Razlikujemo dva varijacijska operatora: mutaciju i rekombinaciju.

Mutacija je unaran operator. Iz jedne jedinke kreira jednu novu promijenjenu jedinku. Nije svaka promjena na jedinki mutacija, za mutaciju je bitna slučajnost, tj. promjena se generira slučajno bez određenog uvjeta.

Rekombinacija ili križanje spaja genetsku strukturu dva roditelja u dva ili više potomaka. Najčešće uzimamo da imamo točno dva potomka. Kao i kod mutacije odabir dijelova genetske strukture roditelja te na koji način će se oni spojiti je slučajan.

Odabir nove populacije

Odabir nove populacije događa se nakon kreiranja nove generacije. Za odabir nove populacije koristimo funkciju vrednovanja te neke dodatne uvjete koji nam garantiraju napredak. Odabir nove populacije je deterministički, dok je odabir roditelja stohastički. Kroz godine razvilo se nekoliko strategija:

- zamjena na temelju godina - svaka jedinka može postojati u populaciji određen broj generacija
- zamjena najgorih jedinki
- elitizam - najsposobnija jedinka se čuva u populaciji što duže
- $(\mu + \lambda)$ odabir - roditelji i potomci se rangiraju po sposobnosti, a u populaciji ostaje najboljih μ jedinki
- (μ, λ) odabir - koristi se kada se generira $\lambda > \mu$ potomaka i baziran je na starosti i sposobnosti. Komponenta starosti garantira da će jedinke preživjeti samo jednu generaciju, a komponenta sposobnosti generira rangiranu populaciju na temelju koje se odabire μ najboljih

Ostale komponente

Uz prethodno navedene komponente u evolucijskim algoritmima nerijetko se pojavljuju i inicijalizacija te uvjet zaustavljanja.

Inicijalizacija populacije se uglavnom radi slučajnim odabirom. Za neke posebne probleme mogu se koristiti heuristike kako bismo dobili što bolju populaciju, ali to ne utječe na sam algoritam.

Uvjet zaustavljanja definiramo ako zadani problem ima poznati optimum. Tada program možemo zaustaviti kada se taj optimum postigne ili kad dobiveno rješenje bude dovoljno blizu optimuma. Kako su evolucijski algoritmi stohastički ne možemo uvijek biti sigurni da ćemo do takvog rješenja i doći. Iz tog razloga znaju se koristiti sljedeći uvjeti:

- prijeđeno je maksimalno dopušteno CPU vrijeme
- ukupni broj evaluacija je prešao zadani limit
- poboljšanje populacije stagnira određeno vrijeme
- raznolikost populacije je pala ispod određene vrijednosti

Poglavlje 2

Pregled evolucijskih algoritama

U prethodnom poglavlju smo ukratko opisali pojedine komponente evolucijskih algoritama, a u ovom poglavlju ćemo malo detaljnije raspisati različite reprezentacije podataka te pripadne varijacijske operatore radi lakšeg razumijevanja različitih varijanti evolucijskih algoritama.

2.1 Reprezentacije i pripadni varijacijski operatori

Binarna reprezentacija

Kod binarne reprezentacije vrijednostima gena (alelima) pridružujemo vrijednosti 0 ili 1 te sukladno zadanom problemu određujemo dužinu stringa koji nam je potreban za adekvatnu reprezentaciju populacije. Binarnu reprezentaciju koristimo kod problema naprtnjače, gdje imamo indeksiran popis stvari koje je potrebno staviti u naprtnjaču. Ako nam se na i-tom mjestu nalazi 0, i-tu stvar s popisa nismo stavili u naprtnjaču, a ako se nalazi 1 znači da smo i-tu stvar spremili u naprtnjaču.

Uz binarnu reprezentaciju najčešće koristimo sljedeće varijacijske operatore. Kao mutaciju uzimamo tzv. *bit-flip*, nasumično biramo gene kojima ćemo promijeniti vrijednost iz 0 u 1 ili obratno. Za operator rekombinacije imamo više različitih mogućnosti.

- križanje u jednoj točki: nasumično odaberemo mjesto u stringu koje je strogo manje od duljine samog stringa na kojem ćemo odrezati oba roditelja. Potomke ćemo dobiti tako da početak jednog roditelja spojimo s krajem drugom roditelja
- križanje u n točaka: nasumično odaberemo n mjesta u stringu gdje će se stringovi oba roditelja odrezati. Potomke dobivamo tako da naizmjence spajamo dijelove roditelja
- uniformno križanje: svaki gen u stringu roditelja se tretira neovisno o drugom te se nasumično odabire što će se od kojeg roditelja naslijediti

Cjelobrojna reprezentacija

Kod cjelobrojne reprezentacije možemo imati dva različita shvaćanja vrijednosti. Vrijednost možemo shvaćati kao poziciju pa tada govorimo o ordinalnim vrijednostima ili možemo shvaćati kao brojčanu vrijednost te je tada riječ o kardinalnim vrijednostima. Kod cjelobrojne reprezentacije češće koristimo operator mutacije.

Kod operatora mutacije imamo dvije glavne forme i obje mutiraju gen neovisno o drugom s nekom definiranom vjerojatnošću.

- slučajni razmještaj: na određenom položaju se nasumično bira nova vrijednost iz dozvoljenog skupa. Ova mutacija je najprikladnija za kardinalne vrijednosti.
- creep mutacija: na određenoj poziciji pribraja neku vrijednost s vjerojatnošću p. Uglavnom je riječ o minimalnim izmjenama, ali je potrebno dodati parametre koji kontroliraju distribuciju za određivanje pribrojnika.

Rekombinacije kod cjelobrojnih reprezentacija koristimo ako imamo konačno velik broj mogućnosti i tada koristimo rekombinacije kao i kod binarne reprezentacije.

Reprezentacija realnim ili floating-point vrijednostima

Koristimo vektor realnih brojeva dužine k, gdje je k broj gena u genotipu jedinke. Operator mutacije u ovom slučaju zamjenjuje vrijednost nekog gena s nekom od vrijednosti iz zadanog intervala koji je primjenjiv na svaki gen. U ovisnosti o vjerojatnosnoj distribuciji razlikujemo:

- uniformna mutacija: nove vrijednosti gena se uniformno slučajno određuju iz zadanog intervala. Ovaj operator je analogan *bit-flip* operatoru kod binarne reprezentacije.
- neuniformna mutacija: na vrijednost gena (alel) x_i dodajemo vrijednost koju dobijemo slučajnim odabirom iz Gaussove distribucije s očekivanjem 0, standardnu devijaciju σ određuje korisnik te se po potrebi prilagođava tako da vrijednost upadne u zadani interval. Ova distribucija je opisana u 2.1. Otprilike dvije trećine dobivenih brojeva će nam dati poprilično male promjene vrijednosti gena, ali postoji i ne-nul vjerojatnost da će promjena biti velika jer rep distribucije ne poprima vrijednost 0. Zbog toga je vrijednost σ parametar koji određuje za koliko će se prosječno promjeniti vrijednost gena u mutaciji te se σ najčešće zove i korak mutacije.

$$p(\Delta x_i) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(\Delta x_i - \xi)^2}{2\sigma^2}} \tag{2.1}$$

Uz navedene mutacije, možemo koristiti još i samoprilagođavajuću mutaciju u kojoj se prilagođava veličina koraka mutacije kroz varijacije. Veličina koraka se "spaja" s jedinkom te kroz procese varijacije i odabira koristimo par (x, σ) kao jednu jedinku. U ovom slučaju se radi dvostruka evaluacija, prvo se evaluira nova jednika, a zatim i sposobnost kreiranja nove jedinke. Razlikujemo tri vrste ove mutacije: nepovezana mutacija s jedinstvenom veličinom koraka, nepovezana mutacija s n veličina koraka te povezana mutacija. Za operator rekombinacije koriste se neki od sljedeća tri tipa:

- diskretna rekombinacija: vrijednost na i-tom genu potomka odgovara jednoj od vrijednosti na i-tom genu roditelja
- aritmetička rekombinacija: vrijednost gena na potomku će biti aritmetička sredina vrijednosti gena na roditelju. Kod aritmetičke rekombinacije razlikujemo nekoliko različitih varijanti:
  - jednostavna: odaberemo točku k, za prvog potomka uzmemo prvih k vrijednosti prvog roditelja, a ostale vrijednosti popunimo konveksnom kombinacijom vrijednosti gena roditelja. Drugog potomka analogno dobijemo iz drugog roditelja.
  - jedna: na genu k napravimo konveksnu kombinaciju, ostale vrijednosti ostaju kao i kod roditelja
  - potpuna: svaka vrijednost gena je konveksna kombinacija vrijednosti odgovarajućih gena roditelja
- miješana rekombinacija: aritmetički dobivena vrijednost gena potomka je dovoljno blizu vrijednosti gena roditelja, ali može biti malo veća ili malo manja. Na ovaj način potomke dobivamo izvan prostora kojeg razapinju roditelji

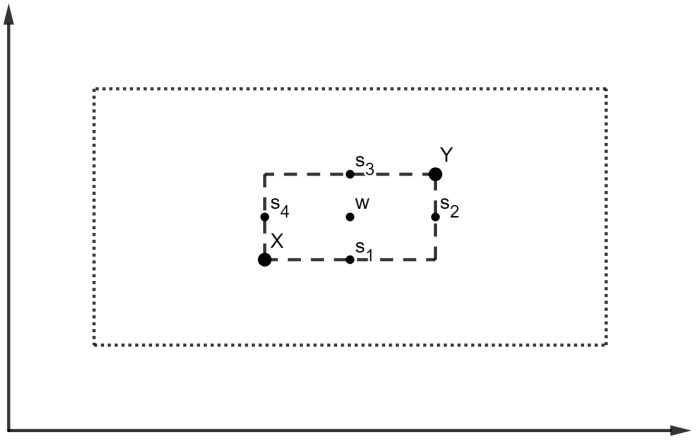
Aritmetičke i miješanu rekombinaciju možemo prikazati i na sljedeći način. Neka su $X = (x_1, ..., x_n)$ i $Y = (y_1, ..., y_n)$ roditelji, $\alpha \in [\ 0, 1]$ slučajno odabrana vrijednost i k slučajno odabran alel.

Za jednostavnu aritmetičku rekombinaciju prvi potomak će izgledati $(x_1, ..., x_k, \alpha \cdot y_{k+1} + (1 - \alpha) \cdot x_{k+1}, ..., \alpha \cdot y_n + (1 - \alpha) \cdot x_n$. Drugog potomka dobijemo zamjenom x i y.

Za jednu aritmetičku rekombinaciju prvi potomak će izgledati $(x_1, ..., x_{k-1}, \alpha \cdot y_k + (1 - \alpha) \cdot x_k, x_{k+1}, ..., x_n)$, a drugog potomka ponovno dobijemo zamjenom x i y.

Kod potpune aritmetičke rekombinacije svaki gen prvog potomka je oblika $\alpha \cdot x_i + (1 - \alpha) \cdot y_i$, dok su za drugog potomka x i y zamijenjeni.

Kod miješane rekombinacije imamo parametar $\gamma = (1-2\alpha)u-\alpha$, gdje je u neki slučajno uniformno odabran broj iz intervala $[\ 0, 1]$. Tada je svaki gen potomka oblika $(1-\gamma)x_i+\gamma y_i$. Primjetimo da vrijedi $\gamma \in [\ -2, 1]$.



Slika 2.1: Prikaz mogućih vrijednosti potomaka dobivenih različitim aritmetičkim rekombinacijama

Na slici 2.1 su prikazani mogući potomci za roditelje X i Y. $\{s_1, s_2, s_3, s_4\}$ su 4 moguća potomka dobivena jednom aritmetičkom rekombinacijom, gdje je $\alpha = 0.5$. w je potomak dobiven iz potpune aritmetičke rekombinacije s $\alpha = 0.5$, a unutarnji pravokutnik daje sve moguće vrijednosti potomaka uz variranje α. Vanjski pravokutnik prikazuje sve moguće potomke dobivene miješanom rekombinacijom uz $\alpha = 0.5$

Reprezentacija permutacijama

Ovu reprezentaciju koristimo kod dvije klase problema, jedan ovisi o redu kada će se nešto dogoditi, a drugi o povezanosti između različitih entiteta. Kod permutacija, mutacijom ne mijenjamo vrijednost pojedinih gena već im zamijenjujemo mjesta. Razlikujemo nekoliko mutacija:

- mutacija zamjenom: slučajnim odabirom pronađemo dva gena te im zamijenimo vrijednosti
- mutacija umetanjem: slučajnim odabirom pronađemo dva gena i drugog prebacimo neposredno prije prvog
- mutacija miješanjem: određeni dio kromosoma se odabere slučajno i u tom dijelu se vrijednosti izmiješaju

- mutacija inverzijom: određeni dio kromosoma (početak i kraj) se odabere slučajno i u tom dijelu se vrijednosti obrnu.

Kod permutacija, rekombinacije se malo kompliciraju, pogotovo ako je riječ o problemu koji ovisi o povezanosti između različitih entiteta. Navest ćemo tri različite varijante rekombinacije:

- djelomično mapirano križanje (PMX) prikladan za probleme koji ovise o povezanosti
- edge križanje za koje je potrebno definirati tablicu rubova u kojoj se nalaze podaci koji element je povezan s kojim u oba roditelja
- slijedno ili redno križanje, slično je PMX-u te se najčešće koristi za probleme bazirane na redoslijedu
- cikličko križanje dijeli elemente u cikluse te ih spaja

Reprezentacija stablima

Stabla prikazuju izraze u danoj formalnoj sintaksi. Kako bismo što bolje odradili reprezentaciju stablima potrebno je definirati set funkcija koje su dozvoljene samo kao unutarnji čvorovi te terminal skup koji su dozvoljeni kao listovi. Ovu reprezentaciju najčešće koristimo kod genetskog programiranja.

Operator mutacije kod stabala nausmično odabire čvor i podstablo kojem je taj čvor korijen te ga zamjenjuje s nasumično generiranim stablom. Trenutni algoritmi genetskog programiranja koriste malu, ali pozitivnu vjerojatnost mutiranja. Operator rekombinacije nausmično odabere dva čvora u stablu te zamijeni podstabla kojima su ti čvorovi korijeni.

2.2 Varijante evolucijskih algoritama

Genetski algoritmi

Genetski algoritam je najpoznatija varijanta evolucijskih algoritama te se vrlo često njime započinje poučavanje o evolucijskim algoritmima. Njegova najpoznatija inačica je svakako jednostavni genetski algoritma ili SGA (*simple genetic algorithm*). Ukratko, genetski algoritam uzme populaciju s μ jedinki, odabirom roditelja kreira intermediarnu populaciju koja dopušta duplikate. Nasumično se kreiraju parovi za rekombinaciju koja se izvodi s vjerojatnošću p_c te potomci zauzmu mjesto roditelja. Na svakom potomku se potom primjeni mutacija s vjerojatnošću p_m. Dobivamo potpuno novu generaciju s jednakim brojem

jedinki. Ovaj jednostavan genetski algoritam može se smatrati vinskom mušicom evolucijskih algoritama, kako je navedeno u [3]. U tablici 2.1 definirane su osnovne komponente jednostavnog genetskog algoritma.

Kasnijim analizama se utvrdilo da se dodavanjem faktora kao što su elitizam i negeneracijski modeli, postiže brža konvergencija. Dodatno, došlo se do zaključka da binarna reprezentacija nije prikladna za sve vrste problema koji su se javljali.

Reprezentacija	bit string
Rekombinacija	križanje u jednoj točki
Mutacija	bit-flip
Odabir roditelja	FPS implementiran u rulet algoritam
Odabir preživjelih	zamjena na temelju godina

Tablica 2.1: Skica jednostavnog genetskog algoritma

Evolucijske strategije

Evolucijske strategije izumili su početkom 1960-ih Rechenberg i Schwefel. Najranija evolucijska strategija su bili jednostvni dvoslojni algoritmi zapisani kao (1 + 1) ES koji su djelovali na vektorskom prostoru. Potomci su kreirani dodavanjem nasumično odabranog broja svakoj komponenti vektora roditelja. Postoji i alternativna $(1, 1)$ ES koja uvijek zamjenjuje roditelja s potomkom i tako zaboravlja prethodno rješenje. Nasumični brojevi se uzimaju iz Gaussove distribucije s očekivanjem 0 i standardnom devijacijom σ, gdje σ označava korak mutacije. 1970-ih predstavljene su višečlane evolucijske strategije, koje u svojem nazivu imaju broj jedinki populacije μ te broj jedinki koje nastaju u jednom ciklusu λ. Te nove višečlane evolucijske strategije, $(\mu + \lambda)$ ES i (μ, λ) ES dovele su do razvoja elegantnije kontrole koraka mutacije te samoprilagodbe bitnih parametara, vrlo korisnog svojstva u evolucijskom računanju. Ukratko, samoprilagodba znači da su neki parametri evolucijskog algoritma podložni promjenama kroz cikluse, tj. podložni su primjeni varijacijskih operatora. Moderne evolucijske strategije gotovo uvijek samoprilagođavaju korak mutacije. Kratak pregled osnovnih komponenti dan je u tablici 2.2.
Od varijacijskih operatora koji se koriste u evolucijskim strategijama valja spomenuti diskretnu rekombinaciju te rekombinaciju srednje vrijednosti. Što se tiče odabira preživjelih puno češće se koristi (μ, λ) odabir jer potomci u potpunosti zamjene roditelje u jednom ciklusu, dok kod $(\mu + \lambda)$ odabira postoji mogućnost da neka lošija rješenja opstanu dosta dugo u populaciji.

Reprezentacija	vektori s realnim vrijednostima
Rekombinacija	diskretna ili srednja vrijednost
Mutacija	Gaussova perturbacija
Odabir roditelja	uniformno nasumično
Odabir preživjelih	deterministično elitistička zamjena s (μ, λ) ili $(\mu + \lambda)$
Posebnost	samoprilagođavanje veličine mutacijskog koraka

Tablica 2.2: Skica jednostavne evolucijske strategije

Evolucijsko programiranje

Evolucijsko programiranje je razvio Fogel et al. u 1960-ih kako bi simulirao evoluciju kao proces učenja s ciljem kreiranja umjetne inteligencije. Ovdje se za inteligenciju uzimala sposobnost prilagođavanja ponašanja sustava kako bi zadovoljio određene ciljeve. Klasično evolucijsko programiranje je koristilo konačna stanja kao jedinke, danas se puno češće koriste vektori s realnim vrijednostima, ali izbor reprezentacije pa zatim i mutacije ovisi o samom problemu. U evolucijskom programiranju svaka jedinka prikazuje jednu vrstu, stoga nema rekombinacije. Za razliku od evolucijskih strategija, ovdje svaki roditelj generira točno jednog potomka, a novu generaciju dobivamo ”borbom” između roditelja i potomaka. U 2.3 možemo vidjeti kako izgleda jedan od jednostavnijih primjera evolucijskog programiranja. Teško ih je sve opisati jednim prikazom jer se sve komponente prilagođavaju zadanom problemu.

Reprezentacija	vektori s realnim vrijednostima
Rekombinacija	nema
Mutacija	Gaussova perturbacija
Odabir roditelja	deterministički (potomak nastaje iz jednog roditelja mutacijom)
Odabir preživjelih	vjerojatnosno $(\mu + \lambda)$
Posebnost	samoprilagođavanje veličine mutacijskog koraka

Tablica 2.3: Skica jednostavnog evolucijskog programiranja

Genetsko programiranje

Za razliku od dosadašnjih varijanti evolucijskih algoritama, genetsko programiranje se najčešće koristi za strojno učenje te koristi reprezentaciju stablima. Genetsko programiranje se uglavnom koristi kako bi se pronašli modeli koji najbolje odgovaraju zadanom problemu, naravno uz uvođenje maksimizacije ovakvi problemi se mogu svesti na probleme optimizacije. Kod genetskog programiranja, potomka se najčešće dobije mutacijom

ili rekombinacijom. S time da se u knjizi [4] preporuča da genetsko programiranje radi bez mutacije, dok se u knjizi [2] preporuča da se u 5% slučajeva mutacija ipak izvrši.

Reprezentacija	stabla
Rekombinacija	zamjena podstabala
Mutacija	nasumična promjena u stablu
Odabir roditelja	FPS
Odabir preživjelih	zamjena na temelju godina

Tablica 2.4: Skica jednostavnog genetskog programiranja

Kada spominjemo evolucijske algoritme najčešće mislimo na jednu od prethodne četiri navedene. Dodajmo još nekoliko mlađih varijanti evolucijskih algoritama.

Diferencijalna evolucija

Ovo je poprilično mladi član evolucijskih algoritama, ali ne i manje važan. Storn i Price su 1995. objavili rad [5] gdje su opisali glavne ideje "novog heurističkog pristupa minimiziranju moguće nelinearnih i nediferencijalnih neprekidnih prostornih funkcija" (eng. *"new heuristic approach for minimizing possibly nonlinear and nondifferentiable continuous space functions"*). Novitet u ovoj varijanti je diferencijalna mutacija. Ukratko, ako se promatra prostor $\mathbb{R}^n$, nova jedinka x' se dobiva dodavanjem vektora perturbacije p, tj. $x' = x + p$, gdje je vektor perturbacije skalirani vektor razlike dva nasumično odabrana člana populacije, $p = F \cdot (y - z)$, a faktor skaliranja $F > 0$ je realni broj koji kontrolira brzinu evolucije populacije. Drugi varijacijski operator je standardno uniformno križanje.

Reprezentacija	vektori s realnim vrijednostima
Rekombinacija	uniformno križanje
Mutacija	diferencijalna mutacija
Odabir roditelja	uniforman nasumičan odabir tri potrebna vektora
Odabir preživjelih	deterministično elitistička zamjena (roditelj vs. potomak)

Tablica 2.5: Skica jednostavnog algoritma diferencijalne evolucije

Optimizacija roja čestica (PSO)

Ova varijanta se poprilično razlikuje od dosadašnjih jer je nadahnuta društvenom ponašanju riba ili ptica u jatu. PSO se bazira na prostornom shvaćanju čestica s položajem i brzinom.

Kao i kod diferencijalne evolucije, PSO ima posebne varijacijske operatore, tj. ne koristi rekombinaciju, a mutacija je definirana preko zbrajanja vektora. No PSO se razlikuje od ostalih po tome što svaka jedinka sa sobom ima svoj vektor perturbacije, tj. promatramo članove (x, p). Potomka (x', p') se dobije tako da se izračuna novi vektor perturbacije p' te se pomoću njega dobije x' kao $x' = x + p'$.

Reprezentacija	vektori s realnim vrijednostima
Rekombinacija	nema
Mutacija	dodavanje vektora brzine
Odabir roditelja	deterministično (svaki roditelj daje jednog potomka mutiranjem)
Odabir preživjelih	zamjena na temelju godina

Tablica 2.6: Skica jednostavnog PSO-a

Možemo spomenuti još neke varijante: tabu-pretraga, memetski algoritmi, inkrementalno učenje na temelju populacije, *scatter* pretraga, kulturološki algoritmi, algoritmi bazirani na ponašanju mrava u koloniji.

Većina prethodnih varijanti se koristi za probleme optimizacije, no neke mlađe varijante se mogu koristiti i za strojno učenje. Jedan takav primjer je klasifikacijski sustav učenja (LCS). To je evolucijski pristup kreiranju modela koji se temelji na korištenju skupa pravila kako bi se reprezentiralo znanje. Najčešće se koriste kada je potrebno razviti sustav koji će odgovoriti na trenutno stanje okoliša tako da predloži odgovor koji će na neki način makisimizirati nagradu za okoliš u budućnosti. LCS je kombinacija klasifikacijskog sustava i algoritma učenja. Klasifikacijski sustav donosi skup pravila, gdje svako pravilo za pojedini ulazni podatak određuje neku akciju. Algoritam učenja se odnosi na neki evolucijski algoritam, gdje svaka jedinka predstavlja ili pojedino pravilo ili cijeli skup pravila. Zbog tih razlika u reprezentaciji jedinki imamo dva različita pristupa LCS-u, a to su redom Michiganski pristup te Pittsburški pristup.

Poglavlje 3

Predstavljanje problema punjenja kontejnera

3.1 Postavljanje problema

Problem promatramo u tri dimenzije te se svodi na to kako u jedan ili više kontejnera posložiti različite predmete ili pakete uz uštedu što više prostora, tj. da što manje prostora ostane neiskorišteno. Dodatno, zadana količina paketa mora biti posložena tako da se koristi najmanji mogući broj kontejnera. Ovaj problem je u načelu NP-težak, što je dobar pokazatelj da bi evolucijski algoritmi mogli biti korisna metoda za rješavanje.

Za početak, navedimo nekoliko definicija preuzetih iz [6].

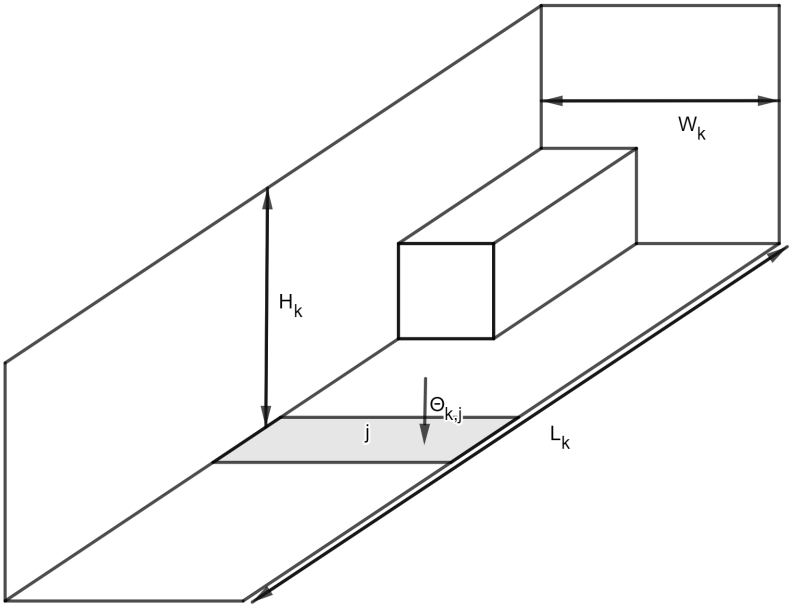
Definicija 3.1.1. *Paket definiramo*

- *kao kvadar i s visinom h_i, širinom w_i, dubinom l_i te težinom γ_i*
- *kao različiti kvadri s istim svojstvima*

Definicija 3.1.2. *Kontejner k se definira preko svoje dimenzije (visina H_k, širina W_k, dubina L_k) kao i maksimalno moguće opterećenje $\Theta_{k,j}$ za svaki poprečni presjek j fiksne širine.*

Na slici 3.1 možemo vidjeti kako izgleda kontejner iz gornje definicije.

Definicija 3.1.3. *Za skup paketa P i opis kontejnera izračunava se broj potrebnih kontejnera C, gdje je $|C| = m$ i plan odlaganja $S(P, C)$ koji dijeli pakete između m kontejnera, $P = P_1 \dot{\cup} ... \dot{\cup} P_m$, i svakom paketu $i \in P$ dodjeljuje položaj (x_i, y_i, z_i) za lijevi donji stražnji kut paketa u kontejneru. Paketi moraju biti složeni u kontejner bez preklapanja. Dopuštene su rotacije paketa u kontejneru.*



Slika 3.1: Slika kontejnera

Na temelju danih definicija, cilj optimizacije može se formulirati kao gustoća pakiranja ili iskorištavanje volumena, što stvara potrebu za što kompaktnijim utovarom kontejnera. Funkcija cilja se tada može prikazati kao:

$$f(S(P,C)) = \frac{\sum_{i \in P} l_i \cdot w_i \cdot h_i}{\sum_{k \in C(k \leq m)} L_k \cdot W_k \cdot H_k} \tag{3.1}$$

Za probleme iz primjene jedan kriterij nije dovoljan za adekvatno opisivanje željenog rješenja. Dva dodatna kriterija, koja ćemo samo navesti su ravnoteža svih kontejnera $f_{bal}(S(P,C))$ i razmatranje najvećeg mogućeg opterećenja po presjeku $f_{over}(S(P,C))$. Za ovaj potonji nam je potreban poprečni presjek iz definicije 3.1.2. Funkcija vrednovanja za ovaj problem formulirana je kao linearna kombinacija tri kriterija cilja:

$$f(P) = \alpha_1 \cdot f(P) + \alpha_2 \cdot f_{bal}(P) + \alpha_3 \cdot f_{over}(P) \tag{3.2}$$

Uz to, prilikom optimizacije u obzir se mogu uzeti i sljedeći rubni uvjeti

- stabilno postavljanje svakog paketa

- razmatranje maksimalnog kapaciteta paketa ako je natovaren drugim paketima
- za svaki paket specificirati smije li se on postaviti na drugi paket, smije li se drugi paket postaviti na njega te smije li se okretati

3.2 Oblikovanje problema u smislu evolucijskih algoritama

Ako bismo ovaj algoritam formulirali tako da radimo direktno s koordinatama paketa u kontejneru, očekivali bismo loše rezultate iz dva razloga

- često bi došlo do preklapanja paketa, u tom slučaju morali bismo odbaciti neke pakete, što nije učinkovito ili bi operatori trebali uzimati u obzir već postavljene pakete, što dovodi do složenijeg sustava slaganja
- optimalni plan odlaganja obično se temelji na činjenici da se paketi postavljaju jedan do drugoga kako bi preostala slobodna površina bila što veća. Ovo se također treba uzeti u obzir prilikom oblikovanja operatora.

Predstavimo nekoliko mogućih načina rješavanja ovog problema. Kako se za ovaj problem mogu postaviti razni uvjeti, tako ćemo imati i različite načine implemetacije, u ovisnosti o uvjetima koje želimo ispuniti.

Binarna reprezentacija

Krenimo s binarnom reprezentacijom. Za n paketa koje treba posložiti u m kontejnera uzmemo $m \times n$ matricu, koja u jednom stupcu ima točno jednu vrijednost različitu od nule (paket možemo staviti u samo jedan kontejner), ali jedan redak može imati više ne-nul vrijednosti. Za $n = 5$ paketa i $m = 2$ kontejnera imamo prikaz:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Ovo nam prikazuje da se u prvom kontejneru nalaze prvi, drugi i četvrti paket, a u drugom kontejneru treći i peti paket.

U slučaju da je $m = 1$, imamo problem naprtnjače u 3 dimenzije. Tada nam jedinka predstavlja "popis" paketa koji se nalaze u kontejneru, tj. paketi s rednim brojem indeksa na kojima imamo vrijednost 1 se nalaze u kontejneru.

Varijacijske operatore koje ćemo koristiti primijenjujemo na pojedini redak matrice. Za operator mutacije koristimo bit-flip na nasumično odabranih j mjesta, dok za operator rekombinacije koristimo križanje u k točaka. Nakon svake primjene operatora moramo

provjeriti je li nam nova jedinka prihvatljiva, tj. stanu li svi paketi u kontejner kojem su dodijeljeni te da se jedan paket ne nalazi u više kontejnera.

Možemo postaviti različite funkcije vrednovanja pa tako možemo maksimizirati vrijednost paketa u kontejneru, tj. profit ili možemo maksimizirati volumen koji paketi zauzimaju u kontejneru.

Odabir roditelja koji koristimo je kombinacija nasumičnog odabira i FPS-a, a za odabir preživjelih $(\mu + \lambda)$ odabir.

Cjelobrojna reprezentacija

Kod ove reprezentacije dozvolit ćemo mogućnost ponavljanja vrijednosti. Za n paketa koje treba posložiti u m kontejnera uzmemo cjelobrojni vektor duljine n, gdje će vrijednosti svake komponente biti broj od 1 do m, u ovisnosti koji paket je smješten u koji kontejner. Npr. za $n = 5$ paketa i $m = 2$ kontejnera, vektor $(1, 1, 2, 1, 2)$ bi značio da se prvi, drugi i četvrti paket nalaze u prvom kontejneru, dok se treći i pet nalaze u drugom kontejneru.

Koristit ćemo operator slučajnog razmještaja koji na određenom položaju nasumično bira novu vrijednost iz dozvoljenog skupa, u ovom slučaju od 1 do m, a za rekombinaciju ćemo koristiti križanje u k točaka. Kao i kod prethodne reprezentacije, nakon primjene operatora moramo provjeriti je li razmještaj izvediv.

Kao i kod binarne reprezentacije možemo postaviti različite funkcije vrednovanja. Odabir roditelja je nasumičan, a za odabir preživjelih koristimo $(\mu + \lambda)$ odabir.

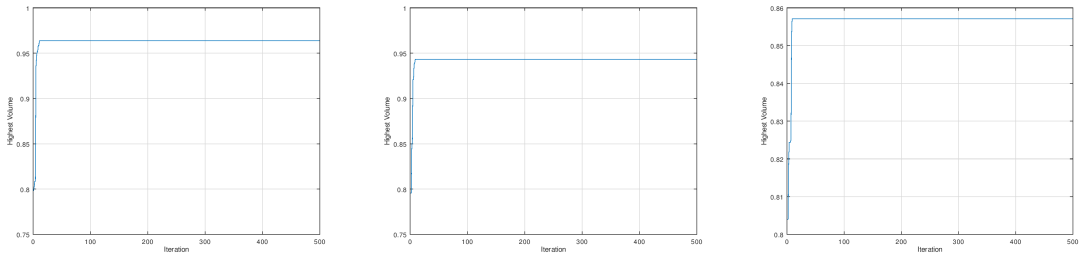
Poglavlje 4

Implementacija različitih verzija evolucijskih algoritama

Za binarnu reprezentaciju kreirali smo dvije verzije. Prva je problem naprtnjače u 3 dimenzije, a druga je s $m > 1$ kontejnera za utovar.

Kod problema naprtnjače uzeli smo pakete različitih dimenzija, ali (zasad) nismo pazili na položaj samih paketa unutar kontejnera. Za početak smo uzeli kontejner dimenzija $2 \times 2 \times 2.5$ metara. Ono što je potrebno naglasiti, ako se optimizira zauzeti volumen kontejnera potrebno je dimenzijama kontejnera prilagoditi broj paketa. Za prevelik broj paketa vrlo brzo kao maksimalnu vrijednost funkcije dobijemo broj veći od 1, što je nemoguće postići s jednim kontejnerom.

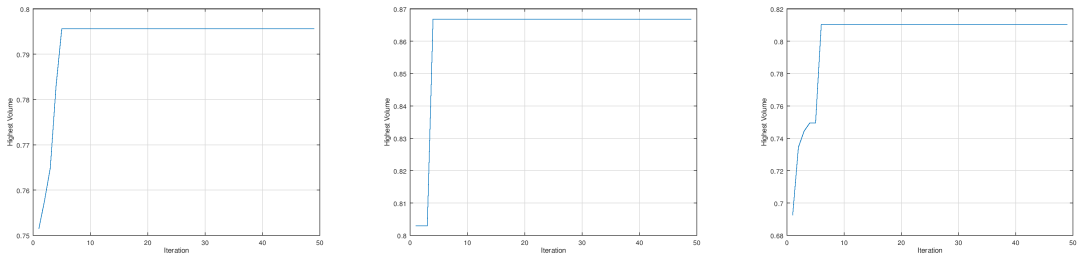
Maksimalan broj iteracija je postavljen na 50. U nekoliko pokretanja programa na nasumično generiranim populacijama smo utvrdili da se unutar 50 iteracija dosegne optimum populacije.



Slika 4.1: Prikaz postizanja optimalnog volumena za tri nasumično generirane populacije

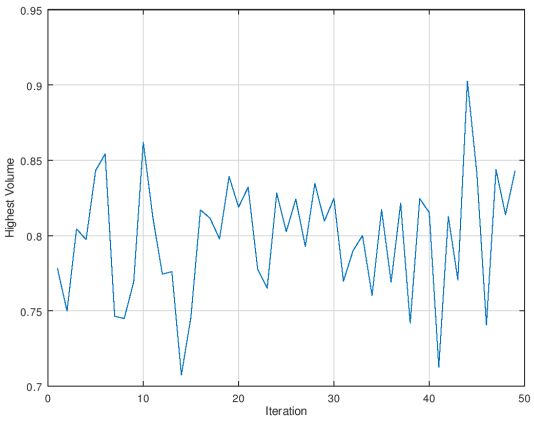
Prije početka samog evolucijskog algoritma provjeravamo početnu populaciju kako ne bismo zapeli u lokalnom optimumu. Ako dvije jednike imaju vrijednost funkcije vrendovanja unutar 10^{-3} razlike, drugu jedinku mijenjamo s novom nasumično generiranom

jedinkom. Ovaj postupak ponavljamo sve dok je broj zamjena veći od jedne trećine broja populacije. Dopuštamo da nam najviše trećina populacije bude slična.



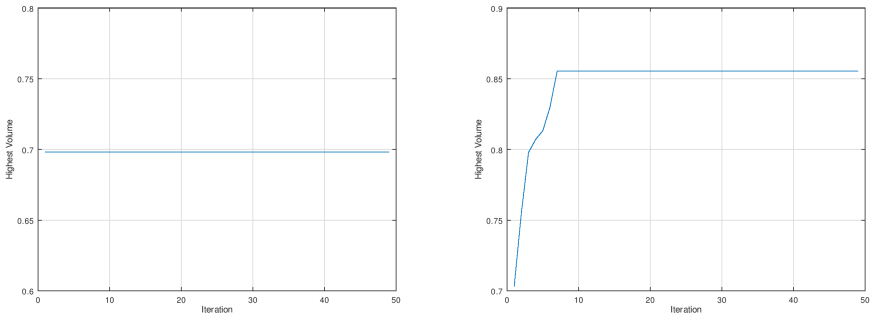
Slika 4.2: Prikaz postizanja optimalnog volumena u ovisnosti o provjeri populacije

Na slici 4.2 vidimo kako izgledaju maksimalne vrijednosti zauzetog volumena ako nemamo provjeru populacije (skroz lijevo), ako provjeru populacije napravimo samo na početku (sredina) te ako provjeru populacije radimo u svakom koraku(skroz desno). Možemo primjetiti da je maksimalna postignuta vrijednost u slučaju bez provjere populacije i s provjerom u svakom koraku nešto manja, tj. u jednom trenutku smo zapeli u lokalnom optimumu i nismo našli bolju vrijednost. Ovdje smo uzeli 100 paketa koje je potrebno smjestiti u kontejner dimenzija $2 \times 2 \times 2.5$ metra na istoj populaciji. Na slici 4.3 vidimo i kako se vrijednost zauzetog volumena ponaša ako u provjeri populacije zamjenimo najviše jednu trećinu populacije, tj. dopuštamo da nam dvije trećine broja populacije bude slična.



Slika 4.3: Prikaz postizanja optimalnog volumena s malim brojem promjena unutar provjere populacije

Treba pripaziti i kod odabira roditelja. Ako oba roditelja za operator rekombinacije i roditelja za operator mutacije biramo pomoću odabira proporcinalnom podobnosti (FPS) dobivamo nižu optimalnu vrijednost, ako jednog roditelja za operator rekombinacije biramo pomoću FPS-a, a ostale nasumičnim odabirom dobivamo nešto višu vrijednost, vidi sliku 4.4.



Slika 4.4: Razlika između odabira roditelja pomoću FPS-a i kombinacije FPS-a i nasumičnog odabira

Kod implementacije s binarnom reprezentacijom, ali s $m > 1$ kontejnera vrijedi sve što smo naveli za implementaciju problema naprtnjače u 3 dimenzije. Samo još treba pripaziti da se jedan paket ne nađe u više različitih kontejnera.

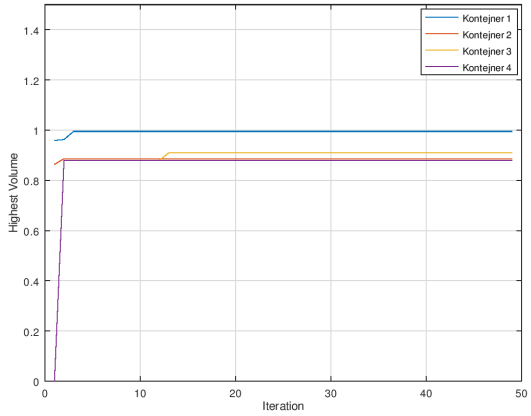
Za ovaj primjer, broj paketa koji treba smjestiti u kontejnere je bio 10, veličina populacije 50, a dimenzije kontejnera $1 \times 1 \times 0.5$. Kao rezultat dobiven je sljedeći raspored paketa po kontejnerima

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

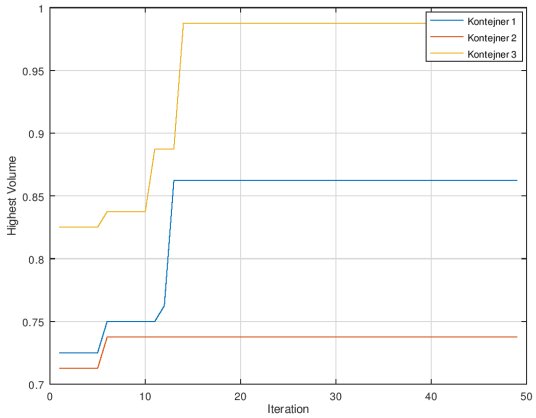
a na slici 4.5 vidimo kako se kroz iteracije postizala maksimalna vrijednost iskorištenog volumena.

Prepostavke i uvjeti koje smo iznijeli u prethodnom odlomku za binarnu preprezentaciju vrijede i kod cjelobrojne reprezentacije. Ono što je drukčije od prethodnih implementacija je to što smo za pojedinu jedinku računali m različitih vrijednosti zauzetog volumena, za svaki kontejner posebno te smo za rješenje uzeli onu jedinku koja ima najveće vrijednosti za svih m kontejnera. Ovdje možemo i popustiti uvjet te uzeti da je dovoljno da više od pola kontejnera dostigne maksimalnu vrijednost. Na slici 4.6 vidimo kako se kroz iteracije mijenjala maksimalna vrijednost zauzetog volumena po kontejnerima.

U implementacijama smo odlučili maksimizirati vrijednost zauzetog volumena, što možda ima smisla matematički, ali potrebno je provjeriti i je li taj razmještaj fizički moguć,



Slika 4.5: Prikaz postizanja optimalnog volumena za $m = 4$ kod binarne reprezentacije



Slika 4.6: Prikaz postizanja optimalnog volumena za $m = 3$ kod cjelobrojne reprezentacije

tj. moramo provjeriti mogu li se paketi složiti u kontejner bez preklapanja. Nakon generiranja potomaka provjeravamo koje jedinke daju smislene razmještaje. Sve one jedinke koje ne zadovoljavaju taj uvjet, mičemo iz populacije. Ako dođemo u situaciju da nakon provjere imamo manju populaciju od μ, nasumično generiramo nove jedinke koje provjeravamo tek u sljedećoj iteraciji. U razmještaju dopuštamo praznine, u stvarnom životu te se praznine mogu ispuniti nekim punilom (stiropor, karton i sl.).

Poglavlje 5

Prikaz rješenja

U prethodnom poglavlju smo opisali i prikazali kako odabir roditelja i kontrola populacije utječu na izvođenje, ali dobiveni rezultati ovise o još mnogo toga. Ovdje ćemo prikazati kako vjerojatnosti rekombinacije i mutacije, ali i kako sam izbor operatora utječe na rezultate.

U tablici vidimo prikaz kako različite vjerojatnosti rekombinacije i mutacije utječu na istu populaciju za problem naprtnjače u 3 dimenzije.

Vjerojatnost rekombinacije p_c	0.3	0.5	0.2	0.7	0	1
Vjerojatnost mutacije p_m	0.8	0.5	0.9	0.7	1	0
Zauzeti volumen	0.97917	0.94167	0.95417	0.95833	0.9375	0.97083

Tablica 5.1: Veličina populacije = 30, broj paketa = 100, dimenzije kontejnera $20 \times 12 \times 1$

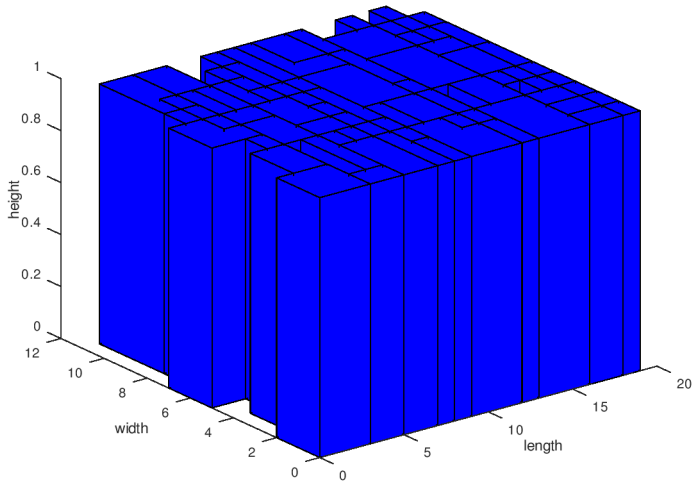
Primijećujemo da su svi rezultati približni, ali i da je najlošiji rezultat kad koristimo samo operator mutacije. Važno je i napomenuti da smo za operator rekombinacije koristili križanje u 1 točki, a za operator mutacije bit-flip na jednom genu. Prikazat ćemo još kako mijenjanje operatora utječe na gore opisani primjer s vjerojatnostima $p_c = 0.3$ i $p_m = 0.8$ jer nam je taj slučaj dao najbolji rezultat.

Križanje u k točaka	1	3	1	5	10	2
Bit-flip na k gena	1	1	4	5	7	10
Zauzeti volumen	0.97917	0.95833	0.95	0.99833	0.97917	0.975

Tablica 5.2: Veličina populacije = 30, broj paketa = 100, dimenzije kontejnera $20 \times 12 \times 1$

Ovdje možemo primjetiti da nam veći broj križanja ili bit-flipova na jedinki ne garantira i bolji krajnji rezultat. Na slici možemo vidjeti kako izgledaju paketi raspoređeni u

kontejner za križanje u 5 točaka te bit-flip na 5 gena s vjerojatnostima $p_c = 0.3$, $p_m = 0.8$.



Slika 5.1: Raspored paketa u kontejneru

Pogledajmo sada kako to izgleda za algoritam s cjelobrojnom reprezentacijom. U ovom primjeru imamo 50 paketa koje treba rasporediti u kontejnere dimenzija $10 \times 5 \times 1$, a veličina populacije je 30. Prema početnim podacima 4 kontejnera će biti potrebna da bismo sve pakete posložili. Možda neki od ovih brojeva ne izgleda baš optimalno, ali omjer

Vjerojatnost rekombinacije p_c	0.85	0.85	0.3	0.7	0	1
Vjerojatnost mutacije p_m	0.2	0.3	0.95	0.7	1	0
Zauzeti volumen - kontejner 1	0.88	0.86	0.84	0.76	0.8	0.64
Zauzeti volumen - kontejner 2	0.96	0.96	0.90	0.90	0.98	0.92
Zauzeti volumen - kontejner 3	0.92	0.64	0.96	0.98	0.88	0.90
Zauzeti volumen - kontejner 4	0.68	0.98	0.74	0.80	0.78	0.98

Tablica 5.3: Veličina populacije = 30, broj paketa = 100, dimenzije kontejnera $20 \times 12 \times 1$

zbroja volumena svih paketa i volumena kontejnera je 3.44, što nam govori da ne možemo popuniti sve kontejnere s više od 90%.

Kod cjelobrojne reprezentacije za operator mutacije umjesto bit-flipa, koristili smo operator slučajnog razmještaja na k gena.

U tablici 5.4 vidimo kako se ponašaju vrijednosti volumena za različite varijacijske operatore. Ponovno možemo primjetiti da ne utječu previše na optimalan rezultat.

Križanje u k točaka	1	1	5	10	6
Slučajni razmještaj na k gena	1	4	3	2	15
Zauzeti volumen - kontejner 1	0.88	0.76	0.9	0.94	0.94
Zauzeti volumen - kontejner 2	0.96	0.94	0.9	0.6	0.8
Zauzeti volumen - kontejner 3	0.92	0.86	0.74	0.92	0.74
Zauzeti volumen - kontejner 4	0.68	0.88	0.9	0.89	0.96

Tablica 5.4: Veličina populacije = 30, broj paketa = 100, dimenzije kontejnera $20 \times 12 \times 1$

Za svaku jedinku iz populacije moramo provjeriti daje li njen fenotip prihvatljiv genotip, tj. može li se generirani razmještaj posložiti u kontejner bez preklapanja. Kod problema naprtnjače u 3 dimenzije, prolazimo kroz fenotip i ako se na i-tom mjestu nalazi 1, paket s indeksom i stavljamo na prvo dostupno mjestu u kontejneru. Ako dobijemo jedinku čiji fenotip ne prikazuje prihvatljiv razmještaj, mičemo ju. Ovu provjeru radimo nakon izvršenih varijacijskih operatora, tj. na populaciji koja je jednaka $(\mu + \lambda)$ pa nakon pokušaja smještanja paketa u kontejner provjeravamo imamo li μ prihvatljivih jedinki za sljedeću generaciju. Ukoliko to nije slučaj, nasumično generiramo onoliko novih jedinki koliko je potrebno. Za probleme kod kojih je potrebno posložiti pakete u više kontejnera, gore navedenu provjeru radimo na svakom kontejneru posebno. Primjer jednog priihvatljivog rješenja vidimo na slici 5.1.

Svi načini ovdje prezentirani su vrste genetskog algoritma. Zbog reprezentacije populacije i varijacijskih operatora potrebnih za rješavanja ovog problema nije bilo moguće prikazati druge varijante evolucijskih algoritama. S ovim primjerom smo pokazali da dosta različitih parametara utječe na rezultat, ali ne zamjetno. Treba napomenuti da je teško reproducirati rješenja, čak i za iste ulazne podatke, zbog nasumičnosti koja se ponavlja na mnogo mjesta u algoritmima.

Dodatak

U nastavku se nalaze osnovni MATLAB kodovi koje smo koristili prilikom implementacije. Funkcije za varijacijske operatore su jednake za sve načine rješavanja, dok se generiranje nove generacije razlikuje u odabiru roditelja i varijacijskih operatora.

Funkcija za bit-flip na *k* gena

```
function mutant = bit_flip(parent, k)
  n = size(parent, 1);
  mutant = parent;
  for i=1:k
    ind_flip = ceil(rand*n);
    if(mutant(ind_flip)==0)
      mutant(ind_flip)=1;
    elseif(mutant(ind_flip)==1)
      mutant(ind_flip)=0;
    end
  end
endfunction
```

Funkcija za križanje u *k* točaka

```
function [child1, child2] = crossover(parent1, parent2, k)
  n = size(parent1, 2);
  child1 = parent1;
  child2 = parent2;
  ind_1 = ceil(rand*(n/2));
  for i=1:k
    if(k >=n)
      disp('Nije moguce!')
      break;
    end
    ind_2 = ceil((n-ind_1).*rand + ind_1);
```

```
    if (mod(i,2)==0)
      child1(ind_1+1:ind_2) = parent1(ind_1+1:ind_2);
      child2(ind_1+1:ind_2) = parent2(ind_1+1:ind_2);
    else
      child1(ind_1+1:ind_2) = parent2(ind_1+1:ind_2);
      child2(ind_1+1:ind_2) = parent1(ind_1+1:ind_2);
    end
    ind_1 = ind_2;
  end
endfunction
```

Funkcija za slučajni razmještaj na *k* gena

```
function mutant = rand_placement(parent, num_cont, k)
  n = size(parent, 2);
  mutant = parent;
  for i=1:k
    ind = ceil(rand*n);
    mutant(ind) = randi([1 num_cont]);
  end
endfunction
```

Za odabir roditelja koristili smo FPS, ali i nasumičan odabir.

```
function [ind1, ind2] = fps(pop, value, num_par)
  num_pop = size(pop,1);
  for i=1:num_pop
    prob_indiv(i) = value(i)/sum(value);
  end
  [M, ind1] = max(prob_indiv);
  if(num_par>1)
    prob_indiv(ind1) = 0.0;
    [Mi, ind2] = max(prob_indiv);
  end
endfunction
```

Za sve načine rješavanja sličan je način generiranja nove generacije:

```
pc = 0.3;          %vjerojatnost rekombinacije/crossovera
pm = 0.8;          %vjerojatnost mutacije
num_c = round(pc*num_pop); %broj rekombinacija u jednoj iteraciji
num_m = round(pm*num_pop); %broj mutacija u jednoj iteraciji
```

```
%rekombinacija
for i=1:round(num_c/2)
  %i1 = randi([1 num_pop]);
  i2 = randi([1 num_pop]);
  i1 = fps(pop, value, 1);
  [child1, child2] = crossover(pop(i1,:),pop(i2,:),1);
  pop(i+num_pop,:)= child1;
  value(i+num_pop) = 0.0;
  for j=1:num_pack
    if(pop(i+num_pop,j)==1)
      value(i+num_pop) += vol_pack(j);
    end
  end
  value(i+num_pop) = value(i+num_pop)/cont.volume;
  pop(i+round(num_c/2)+num_pop,:) = child2;
  value(i+num_pop+round(num_c/2)) = 0.0;
  for j=1:num_pack
    if(pop(i+num_pop+round(num_c/2),j)==1)
      value(i+num_pop+round(num_c/2)) += vol_pack(j);
    end
  end
  value(i+num_pop+round(num_c/2))=
   =value(i+num_pop+round(num_c/2))/cont.volume;
end

%mutacija
for i=1:num_m
  in = randi([1 num_pop]);
  %in = fps(pop, value, 1);
  mutant = bit_flip(pop(in, :),1);
  pop(i+num_pop+num_c,:)= mutant;
  value(i+num_pop+num_c) = 0.0;
  for j=1:num_pack
    if(pop(i+num_pop+num_c,j)==1)
      value(i+num_pop+num_c) += vol_pack(j);
    end
  end
  value(i+num_pop+num_c)= value(i+num_pop+num_c)/cont.volume;
```

Smještanje paketa u kontejner (pretpostavili smo da su svi paketi visoki koliko i kon-

tejner)

```
function [pop, value] = placement(pop, value, inputc, cont, n_p)
  num_pop = size(pop, 1);
  num_pack = size(pop, 2);
  length = 0; width = 0; height = 0;

   for i=1:num_pop
     flag(i)=0;
     for j=1:num_pack
       if(pop(i,j)==1)
         %stane li paket po sirini
         if((width+inputc(j,2)) < cont.width)
           width += inputc(j,2);
           if(length < inputc(j,1));
             length = inputc(j,1);
           end
           if(height < inputc(j,3))
             height = inputc(j,3);
           end
         %stane li paket do duzini
         elseif(length+inputc(j,1) < cont.length)
           width = 0;            %krecemo s novim redom po duzini
           length += inputc(j,1);
           if(width < inputc(j,2))
             width = inputc(j,2);
           end
           if(height < inputc(j,3))
             height = inputc(j,3);
           end
         %stane li paket po visini
         elseif(height+inputc(j,3) < cont.height)
           width = 0;
           length = 0;
           height += inputc(j,3);
         else
           flag(i)=1;
         end
       end
     endfor
```

```
    endfor

    [flag, ind] = sort(flag, 'descend');
    pop = pop(ind,:);
    value = value(ind);
    idx = min(find(flag<1));

    %izbacujemo one ciji razmjestaj ne stane u kontejner
    pop = pop(idx:num_pop,:);
    value = value(idx:num_pop);


    %ponovno sortitramo po volumenu
    [value, Ind] = sort(value,'descend');
    pop = pop(Ind,:);

    m = size(pop,1);
    if(m < n_p)
      for i = (m+1):n_p
        pop(i,:) = randi([0 1], 1, num_pack);
        value(i) = 0.0;
        for j=1:num_pack
          if(pop(i,j)==1)
            value(i) += inputc(j,1)*inputc(j,2)*inputc(j,3);
          end
        end
        value(i)= value(i)/cont.volume;
      end
    end

    pop = pop(1:n_p,:);
    value = value(1:n_p);

endfunction
```

Funkcija koja provjerava jesu li isti paketi u više kontejnera

```
function[pop,value]=population_check(contn,l,num_pop,value,input,cont
  num_pack = size(contn(l).pop,2);
  k=1;
  while(k>1)
```

```
    for i=1:num_pop
      for j=1:num_pack
        if(contn(k-1).pop(i,j)==contn(l).pop(i,j))
          if(contn(l).pop(i,j)==1)
            contn(l).pop(i,j)=0;
            value(i,1)=
            value(i,1)-(input(j,1)*input(j,2)*input(j,3)/cont.volume)
          end
        end
      end
    end
    k = k-1;
  end
  pop = contn(l).pop;
endfunction
```

Funkcija koja izračunava vrijednost zauzetog volumena i provjerava postoje li jedinke sa sličnom vrijednošću

```
function [pop, value] = population_check_bin(pop, input, cont)
  cont.volume = cont.height*cont.length*cont.width;
  num_pack = size(input, 1);
  num_pop = size(pop, 1);

  for i=1:num_pop
    value(i) = 0.0;
    for j=1:num_pack
      if(pop(i,j)==1)
        value(i) += input(j,1)*input(j,2)*input(j,3);
      end
    end
    value(i)= value(i)/cont.volume;
  end

  while(flag>round(num_pop/3))
    flag = 0;
    for i=1:num_pop
      for j=1:num_pop
        diff = abs(value(i)-value(j));
        if(diff<1e-3)
          pop(j,:) = randi([0 1], 1, num_pack);
```

```
            flag += 1;
            value(j) = 0.0;
            for k=1:num_pack
              if(pop(j,k)==1)
                value(j) += input(k,1)*input(k,2)*input(k,3);
              end
            end
            value(j)= value(j)/cont.volume;
          end
        endfor
      endfor
    end

endfunction
```

Za prikaz optimalnog razmještaja paketa koristili smo funkciju plotcube preuzetu s [1].

Bibliografija

[1] *Olivier (2021) PLOTCUBE*, 2021, `https://www.mathworks.com/matlabcentral/fileexchange/15161-plotcube`, MATLAB Central File Exchange, preuzeto 1. rujna, 2021.

[2] W. Banzhaf, P. Nordin, R.E Keleer i F. D. Francone, *Genetic Programming: An Introduction*, Morgan Kaufmann, 1998.

[3] A.E. Eiben i J.E. Smith, *Intorduction to Evolutionary Computing*, Springer, 2015.

[4] J.R. Koza, *Genetic Programming*, The MIT Press, 1992.

[5] R. Storn i K Price, *Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces*, Teh. izv., Berkeley, 1995.

[6] K. Weicker, *Evolutionäre Algorithmen*, Springer, 2015.

Sažetak

U ovom radu upoznali smo se s osnovama evolucijskih algoritama i njihovom primjenom na konkretnom problemu. Na početku rada iznijeli smo glavnu ideju evolucijskih algoritama, kroz proces evolucije možemo pratiti povećanje sposobnosti cijele populacije. Uz glavnu ideju iznijeli smo i komponente evolucijskih algoritama te smo ih pobliže objasnili. Nakon toga smo se koncentrirali na reprezentaciju i varijacijske operatore i objasnili koji operatori se mogu primijeniti na koju vrstu reprezentacije. Uz ovo je bilo lakše prikazati različite varijante evolucijskih algoritama i navesti sve bitnije značajke. Nakon što smo se upoznali s osnovama evolucijskih algoritama, predstavili smo problem punjenja kontejnera i nekoliko različitih načina rješavanja tog problema koji su ovisili o postavljenim uvjetima i pretpostavkama koje smo zadali. Na kraju smo prikazali rezultate dobivene iz implementiranih načina rješavanja.

Summary

This thesis gives an overview of evolutionary algorithms and their application to a certain problem. In the beginning, we gave the main idea for the evolutionary algorithms, through the process of evolution we can monitor the increase in fitness for the whole population. We also presented and explained components of evolutionary algorithms. We concentrated on representation and variation operators and the link between the two of them. With that knowledge, it was easy to present different variants of evolutionary algorithms. In this thesis, we also presented the optimal container loading problem and gave several different ways to solve it. Lastly, we presented how each solution behaves under different constrictions and assumptions.

Životopis

Rođena sam 07.01.1994. godine u Zagrebu. Završila sam Osnovnu školu Brezovica, a 2008. godine upisujem IV (jezičnu) gimnaziju u Zagrebu. 2012. godine upisujem studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu, tri godine nakon upisa prebacujem se na nastavnički smjer studija matematike koji sam završlia 2017. godine. Potom upisujem diplomski studij Primijenjene matematike i paralelno počinjem raditi u razvojnom sektoru jedne telekomunikacijske tvrtke, a dvije godine nakon prelazim u IT sektor iste tvrtke.