SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2383

METODE ZA RJEŠAVANJE PROBLEMA NARUČIVANJA VOŽNJI

Stjepan Zelić

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2383

METODE ZA RJEŠAVANJE PROBLEMA NARUČIVANJA VOŽNJI

Stjepan Zelić

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2383

Pristupnik:	**Stjepan Zelić (1191228113)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Metode za rješavanje problema naručivanja vožnji**

Opis zadatka:

Proučiti problem naručivanja vožnje za prijevoz putnika automobilima. Istražiti korištene postupke i varijante probleme korištene u literaturi te pronaći odgovarajuće primjerke problema za testiranje. Implementirati postojeći heuristički postupak za rješavanje danog problema. Odabrati i prilagoditi prikladni metaheuristički postupak za rješavanje zadanog problema. Primijeniti algoritam na rješavanje Analizirati rad algoritma te predložiti i implementirati poboljšanja predloženog algoritma s ciljem poboljšanja dobivenih rezultata. Usporediti učinkovitost predloženog algoritma s postojećim postupcima za rješavanje problema prijevoza putnika. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 28. lipnja 2021.

Hvala mojim roditeljima Ivani i Filipu koji su mi pružili sve mogućnosti i bezuvjetnu podršku, baki Mandi koja mi je pružila najveću motivaciju, i dobrim prijateljima koji su bili uz mene na ovom dugom putovanju.

Sadržaj

Uvod

Genetski Algoritmi, kao vrsta optimizacijskih algoritama, su se od svoje koncepcije do danas pokazali kao vrlo vrijedan alat u području računarske znanosti. Širok raspon domena uključuje optimizacijske probleme koji se ne mogu riješiti deterministčkim optimizacijskim algoritmima jer su takvi problemi ili nerješivi u praktičnom vremenu ili, još nepovoljnije, ne možemo ih riješiti trenutno poznatim metodama. No, mnogi scenariji iz stvarnog života nemaju potrebu za pronalaskom optimalnog rješenja, već onog koje je dovoljno dobro iz perspektive aktera kojem je u interesu doći do istog.

Što znači dovoljno dobro? Ako je skup mogućih rješenja velik, a imamo način za odrediti kvalitetu pojedinog rješenja, onda se može odrediti granica s kojom bismo bili zadovoljni. Također, može se dogoditi da je problem rješiv u praktičnom vremenu, ali da se omjer vremena uloženog za nalaženje optimalnog rješenja i vremena uštođenog korištenjem optimalnog rješenja, umjesto nekog dovoljno dobrog, jednostavno ne isplati.

Genetski algoritmi su vrlo poznata evolucijska računarska metoda koja vuče inspiraciju iz prirode. Preživljavanje najjačih (ili prirodna selekcija) je osnovni mehanizam evolucije u prirodi gdje jedinke neke vrste, koje su svojim specifičnim atributima najbolje prilagođeni uvjetima u kojima se nalaze, imaju dulji životni vijek, pa to povlači veću šansu da stvore potomke i prenesu svoje uspješne osobine na svoje potomke prije nego umru.

Ovaj naizgled jednostavan mehanizam se imitira u računarstvu koristeći slične principe kao u prirodi. Jedno moguće rješenje se u memoriji reprezentira kao jedinka - obično kao lista bitova, no u Dial-a-Ride problemu, odnosno Problemu Naručivanja Vozila, kao i nekim drugim problemima, možemo koristiti drugačiji zapis, što je nekad i poželjno.

Nasumičnim ili heurističkim stvaranjem velikog broja jedinci dolazimo do početne populacije koja evoluira kroz ciklus selekcije, reprodukcije i mutacije. Selekcija je proces odabira jedinke koje želimo održati “na životu”, to jest prenijeti ih u slijedeću generaciju. Reprodukcija je koncipirana tako da iz dvije jedinke prethodne generacije stvaramo “djecu”, tj. jedinke nove generacije, tako da po mogućnosti djeca naslijede osobine roditelja. Mutacija nam omogućava da unesemo novu osobinu u neku jedinku nasumičnom promjenom njezinog zapisa, što je simulacija mutacije u prirodi i omogućava nam da “iskočimo” van lokalnog prostora pretraživanja i otvorimo si mogućnosti za još bolja rješenja.

Problem naručivanja vozila (DARP) je problem u kojem skupinu korisnika moramo prevesti od početke točke do odredišta tako da minimiziramo troškove prijevoza, a maksimiziramo zadovoljstvo korisnika uz vremenska ograničenja. Korisnici u DARP-u mogu definirati ili vrijeme polaska s početne lokacije, ili vrijeme dolaska na odredište. Troškovi prijevoza mogu biti ukupan prijeđeni put, broj vozila koji je potreban da se sve korisnike preveze, i ukupno vrijeme vožnje vozila. Zadovoljstvo korisnika možemo mjeriti ukupnom vožnjom korisnika i ukupnim kašnjenjem korisnika, to jest kršenjem vremena polaska ili dolaska.

Primjer ovakvog problema najčešće korišten u literaturi je prijevoz osoba s posebnim potrebama, djece ili umirovljenika. Najčešće je to također unaprijed zadan (statički) problem, gdje sve potrebne informacije imamo prije nego krenemo tražiti rješenje problema, a gdje se nijedan novi zahtjev ne pojavljuje tijekom izvršavanja rješenja. Putnici se ne voze direktno od početne lokacije do odredišta, već mogu dijeliti prijevoz s drugim putnicima, što razlikuje ovu vrstu prijevoza od običnih taksi službi.

U posljednjih nekoliko godina značajno su popularizirane tvrtke za dostavljanje hrane biciklističkim i automobilskim kuririma. Proces dostave hrane se u mnogim elementima može direktno usporediti s DARP-om, jer se više narudžbi mogu voziti u isto vrijeme, imamo vremenske prozore, i hrana se preuzima i dostavlja s mnogih različitih lokacija. Glavna razlika je, naravno, što u stvarnom svijetu teško možemo unaprijed imati sve narudžbe hrane, budući da se u najvećem dijelu radi o dinamičkim zahtjevima.

U praksi se ovaj problem najčešće rješavao heurističkim metodama koje proširuju osnovne algoritme pretraživanja grafova, no u ovom radu pokazuje se primjena genetskih algoritama za optimizaciju. S obzirom da je problem naručivanja vozila podvrsta VRP (Vehicle Routing Problem, ili Problem Raspoređivanja Vozila), to ga čini NP-teškim problemom pa se posezanje za stohastičkim metodama poput GA pokazalo vrlo uspješnim u rješavanju.

1. Genetski algoritmi

1.1. Uvod u genetske algoritme

Genetske algoritme su kao ideju začeli i razvijali John Holland i njegovi suradnici u 1960-ima i 1970-ima kao računalni model evolucije u prirodi baziranoj na Charles Darwinovoj teoriji prirodne selekcije [1]. Holland i suradnici su prvi koji su koristili križanje, mutaciju i selekciju u proučavanju optimizacije teških problema kroz računalne sustave umjetne inteligencije. Od tada do danas, genetski algoritmi su postali široko poznata metoda rješavanja raznih problema, i genetskim algoritmima se pristupilo velikoj većini vrlo poznatih računalnih problema.

Mnogi moderni evolucijski algoritmi direktno potječu od osnovne ideje genetskih algoritama, te imaju velike sličnosti u osnovnim konceptima i izvedbi.

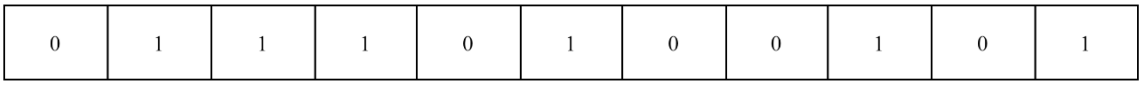
Genetski algoritmi imaju mnoge prednosti u odnosu na tradicionalne optimizacijske algoritme, a dvije najznačajnije su mogućnost rješavanja kompleksnih problema i paralelizam [2]48. S obzirom na to da su jedinke u populaciji neovisne jedna o drugoj, populacija može u isto vrijeme pretraživati prostor u mnogim smjerovima, što GA čini idealnim za paralelizaciju – operacije na jednoj ili dvije jedinke ne utječu na preostale jedinke. Genetski algoritmi se mogu primijeniti na širok spektar problema jer je definicija funkcije dobrote fleksibilna, pa može biti linearna ili nelinearna, prekidna ili neprekidna, i sl.

Genetski algoritmi također imaju i nedostatke. Da bismo formulirali funkciju dobrote, metodu križanja, način mutacije, stopu mutacije, način selekcije, i prikaz jedinke, moramo dobro razumjeti problem i „pametno“ odabrati sve navedene značajke. Ako jedna od njih nije dobro definirana, potencijalno ćemo imati problem s genetskim algoritmom koji neće moći dobro optimizirati problem.

Međutim, pokazalo se da hiperparametrizacijom značajki genetskih algoritama možemo postići značajno bolje rezultate od „ručnog“ oblikovanja, na način da optimiziramo same parametre izvršavanja genetskog algoritma, npr. dinamički gradimo metodu križanja, optimiziramo stopu mutacije, i sl. Po principu No Free Lunch teorema, tako ćemo Genetskim algoritmima napadati pogodne probleme, a one gdje su genetski algoritmi bespomoćni ćemo prepustiti drugim pametnim metodama.

1.2. Značajke genetskih algoritama

Jedinka, populacija, selekcija, križanje, mutacija, i funkcija dobrote su osnovni građevinski blokovi genetskih algoritama. Kao što je ranije spomenuto, jedinka u genetskom algoritmu je jedno rješenje problema kojeg optimiziramo. U ovom slučaju, kod problema sa širokim prostorom rješenja, rješenjem smatramo bilo koje validno rješenje koje ne mora nužno biti kvalitetno prema definiciji problema i cilju optimizacije. Ideja je do kvalitetnih rješenja doći počevši od onih nekvalitetnih. Najčešće se kod genetskih algoritama kao jedinka koristi lista bitova (Sl. 1.1 Prikaz jedinke listom bitova Sl. 1.1).



Sl. 1.1 Prikaz jedinke listom bitova

No nisu svi problemi stvoreni jednakima. Naime, neki problemi (poput Problema Naručivanja Vožnji) ne mogu biti dobro riješeni korištenjem liste bitova kao jedinkom u rješenju jer jedna promjena u listi bitova obično može dovesti do velike razlike u funkciji dobrote, što u našem problemu nije poželjna karakteristika, o čemu će se više reći kasnije.

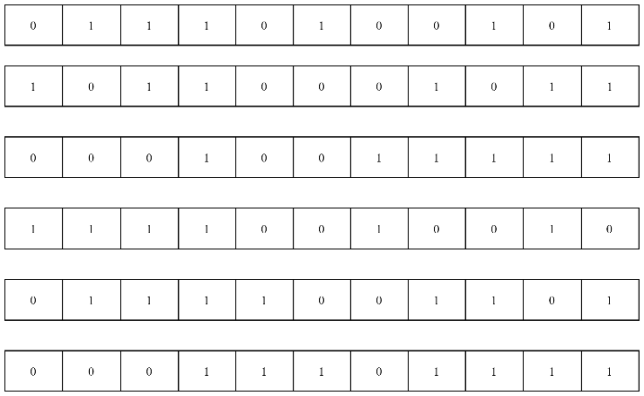
1.2.1. Funkcija dobrote

Funkcija dobrote je funkcija koja preslikava vrijednosti iz prostora jedinki u (obično) prostor realnih brojeva. Njezin cilj je odrediti koliko je dobro jedno rješenje u odnosu na naš zadani problem. Formulira se ovisno o problemu kojeg želimo optimizirati, pa ne postoji generička funkcija dobrote – za svaki problem moramo posebno definirati što želimo uračunati u funkciju dobrote. Ovisno o tome kako je definirana, u implementaciji treba voditi računa hoće li vrijednosti funkcije dobrote moći biti negativni brojevi, hoće li vrijednosti biti velike, pa je sukladno tome normalizirati ili manipulirati njome tako da ju možemo koristiti u kontekstu selekcije i interpretacije rezultata.

1.2.2. Populacija

Populacija u genetskom algoritmu (Sl. 1.2) označava trenutni skup jedinki (rješenja) u jednom trenutku u vremenu, tj. u jednoj generaciji. Populacija se može kreirati potpuno nasumično, što odmah čini genetski algoritam stohastičnim, no možemo je kreirati i

heuristikom ovisnom o problemu kako bismo dobili “bolju” početnu populaciju i moguće brže konvergirali ka rješenju.

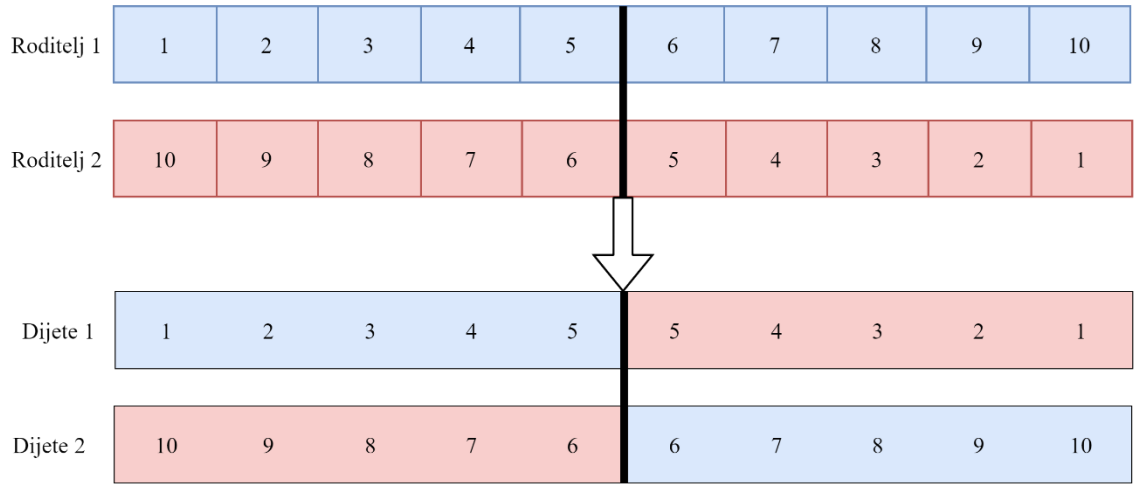


Sl. 1.2 Primjer populacije u jednoj generaciji

Kako genetski algoritam napreduje kroz generacije, tako se mijenja populacija. Generalni cilj nam je da nam bolje jedinke iz populacije „prežive“, to jest prijeđu u slijedeću generaciju, kako bismo zadržali njihove dobre značajke i prenijeli ih dalje, a nove jedinke u generaciji se stvaraju križanjem jedinki roditelja i mutacijom.

1.2.3. Križanje (eng. *crossover*)

Križanje u genetskom algoritmu se može definirati na razne načine, no osnovna je ideja da iz dva roditelja kombiniramo značajke i prenesemo ih u novu generaciju, slično kao kod genetskog nasljeđivanja kod živih bića. Ovisno o problemu i prikazu jedinke, koristit ćemo različite operatore križanja kako bismo što bolje uspjeli prenijeti važne značajke roditelja u sljedeće generacije. Jedan primjer križanja je *One Point Crossover* (Sl. 1.3), ili križanje jednom točkom. U ovom slučaju, nasumično se odabere jedna točka na kromosomu roditelja i oba roditelja se podijele po toj točki. Prvo dijete dobije prvu polovicu prvog roditelja i drugu polovicu drugog, a drugo dijete obrnuto, čime dobijemo dva potomka koji zadržavaju neke karakteristike roditelja, ali tako sastavljenom kombinacijom gena možemo dobiti još bolju jedinku nego što su bili roditelji.

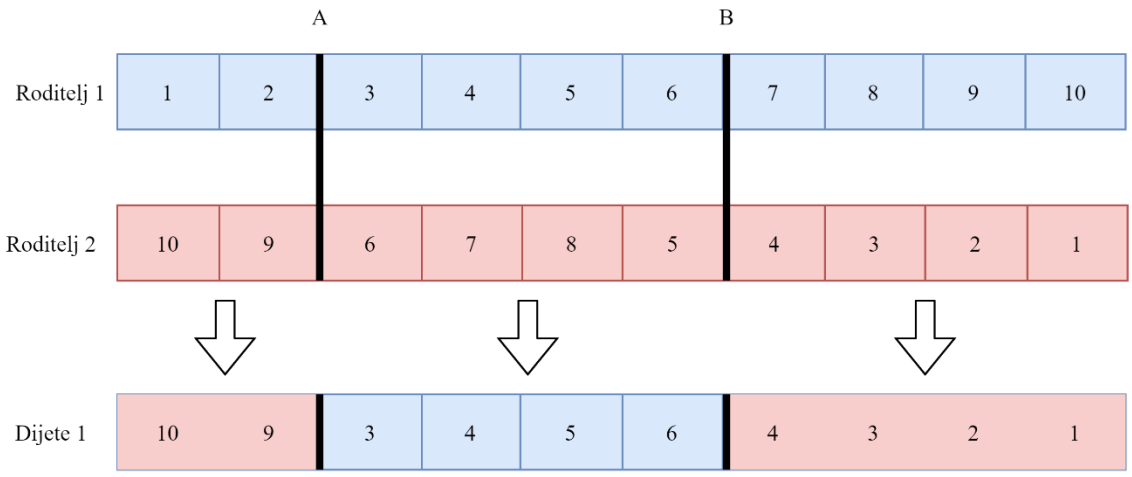


Sl. 1.3 One Point Crossover

Drugi primjer križanja je tzv. *Partially Mapped Crossover*, ili križanje djelomičnim preslikavanjem, čija se varijanta koristila u implementaciji rješavača u ovom diplomskom radu.

Kao što smo mogli vidjeti u križanju jednom točkom, uspješno smo prenijeli dijelove roditelja na djecu, no dobili smo vrijednosti koje se ponavljaju. Konkretno, sada su djeca postala zrcaljena po sredini, pa nam se prvih pet vrijednosti ponovilo u drugoj polovici kromosoma. Kod nekih problema ne želimo imati takvo rješenje jer uopće nije validno, npr. ako je ono što pokušavamo optimizirati poredak, tada bismo imali besmislena rješenja kod djece. Križanje djelomičnim mapiranjem popravlja taj problem, na način opisan ispod:

Po dužini kromosoma roditelja (svejedno je kojeg, kada će roditelji uvijek imati jednaku duljinu) nasumično odaberemo dvije točke A i B, i podijelimo roditelje na 3 dijela: dio prije točke A, dio između A i B, te na dio nakon točke B. U prvo dijete između točke A i B stavimo sve gene (vrijednosti) koje su se nalazile između točke A i B u **prvom** roditelju, a ostatak kromosoma popunimo s vrijednostima iz **drugog** roditelja (Sl. 1.4).

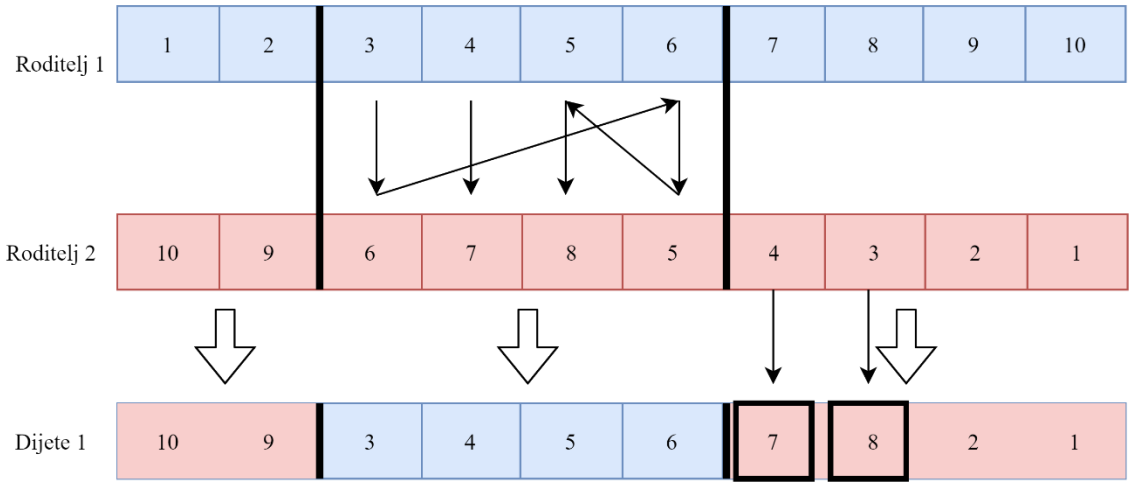


Sl. 1.4 Partially Mapped Crossover, prvi korak

Kao što možemo vidjeti, još uvijek imamo isti problem kao i u One Point Crossoveru, ali sada ćemo napraviti nešto drugo: Kao primjer vidimo da prve dvije ćelije sadrže vrijednosti 10 i 9 koje se ne ponavljaju kasnije u kromosomu, pa ih nemamo potrebe mijenjati. Zapravo vidimo kako su jedine vrijednosti koje se mogu ponoviti one između točaka A i B, jer njih "guramo" u drugog roditelja, čime stvaramo dijete, pa možemo izazvati ponavljanje samo tih točaka. Te vrijednosti su u ovom slučaju 3, 4, 5 i 6. Kako bismo te vrijednosti zadržali u djetetu, morat ćemo malo korigirati vrijednosti izvan srednjeg dijela, tj. lijeve i desne "repove". U desnom repu djeteta vidimo da se ponavljaju vrijednosti 4 i 3, pa ih želimo preslikati u neke druge vrijednosti, ali u koje vrijednosti i na koji način?

Logično je gledati preko kojih su vrijednosti 3, 4, 5 i 6 "pregazile", pa ih samo kasnije zamijeniti s pregaženim vrijednostima. Vrijednost 4 je pregazila vrijednost 7, i vidimo kako se 7 neće ponavljati, pa ćemo umjesto 4 staviti 7.

Međutim, sada vidimo da se 3 preslikava u 6, ali ako zamijenimo 3 sa 6, opet ćemo imati ponavljanje. No što ako zamijenimo tu novodobivenu vrijednost 6 sa onom vrijednošću u koju se 6 preslikava? Dobit ćemo 5, no to opet znači da ćemo imati ponavljanje. Nastavimo li istom metodom, sada ćemo pogledati u koju vrijednost se preslikava 5, i dobit ćemo da se preslikava u vrijednost 8. Budući da se 8 ne nalazi u području između točaka A i B, za 8 smo sigurni kako se neće ponoviti pa ćemo završiti postupak za ovaj gen i ponoviti ga za svaki slijedeći (Sl. 1.5). Vrijednosti 2 i 1 se ne nalaze u srednjoj traci prvog roditelja, pa njih samo prepisujemo bez potrebe za preslikavanjem.



Sl. 1.5 Partially Mapped Crossover, drugi korak

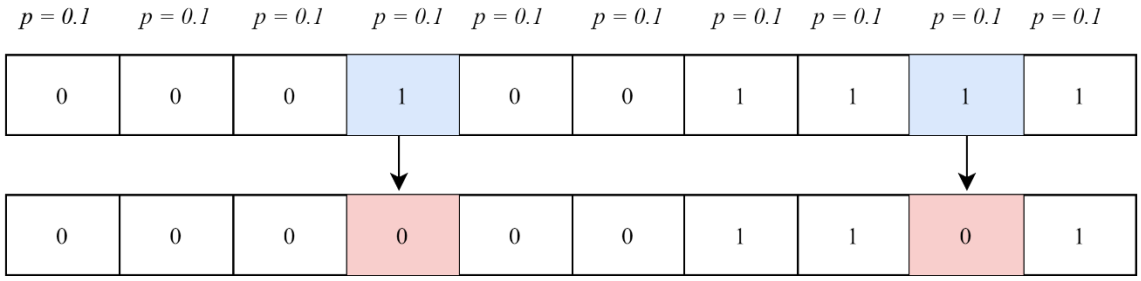
Ovako opisan postupak ćemo na isti način primijeniti i za drugo dijete, ali ćemo obrnuti poredak roditelja: uzimat ćemo srednju traku iz drugog roditelja, a repove iz prvog. Vidimo kako se Partially Mapped Crossoverom ne zadržavaju sasvim točne informacije iz drugog roditelja, ali to je cijena koju moramo platiti kako bismo zadržali validnost rješenja. Potencijalno nam te promjene mogu značiti i skok u funkciji dobrote, samo je bitno da ipak zadržimo neke značajke drugog roditelja, kako ne bismo u potpunosti ignorirali njegovo naslijeđe.

1.2.4. Mutacija (eng. *mutation*)

Međusobnim križanjem jedinki početne populacije, pa onda međusobnim križanjem potomaka, i tako kroz brojne generacije, može se dogoditi da dođemo do stagnacije, to jest da nam svako novo dijete ima već viđene značajke i da ne možemo napustiti lokalni prostor pretraživanja pa tako ni konvergirati prema boljem rješenju.

Tu u priču ulazi mutacija kao operator nad jedinkama u genetskom algoritmu. Slično kao i u prirodi, mutacija će nam donijeti nepredviđenu promjenu koja će možda donijeti veliku prednost toj jedinci nad drugim jedinkama, i u populaciju uvesti novu značajku koja će se prenositi dalje pa tako izbaciti algoritam iz lokalnog prostora pretraživanja prema boljem lokalnom minimumu. Mutacija također ostaje na mašti implementatora genetskog algoritma u granicama problema, a kao primjer se može izdvojiti zamjena jednog ili više bitova (eng. *one-bit flip*), koja je povijesno dokazana i robusna mutacija na bitnom zapisu jedinke. Princip ove mutacije je da svaka ćelija ima jednaku šansu “mutirati”, i nasumično određujemo koja

hoće a koja neće promijeniti bit iz 0 u 1 i obrnuto. Obično je ta šansa $p = 1/n$, gdje je n broj gena u jedinci (Sl. 1.6). U pokazanom primjeru, promijenjeni su četvrti i deveti bit.



Sl. 1.6 One-bit flip mutacija

U prikazu jedinke cijelim brojevima primjer mutacije može biti zamjena mjesta dviju vrijednosti, ili po potrebi problema nešto kompleksnije od toga, što ćemo imati prilike vidjeti u implementaciji rješenja.

1.2.5. Selekcija

Kada stvaramo populaciju u novoj generaciji, želimo zadržati neke dobre značajke iz prethodne populacije. Već smo pokazali kako možemo prenijeti dobre značajke križanjem s roditelja na djecu, no kako odabrati roditelje iz kojih ćemo dobiti djecu za novu generaciju? Selekcija je odgovorna za taj proces, a implementator genetskog algoritma ima slobodu odabira ili smišljanja algoritma koji će to omogućiti.

Jedan takav primjer je Jednostavna Selekcija (eng. *Roullete wheel selection*), gdje se jedinke koje će postati roditelji biraju na temelju njihove ocjene dobrote. Što je ocjena dobrote veća, to je veća šansa da će jedinka biti odabrana. Kada se zbroje ocjene dobrote svih jedinki u populaciji, onda možemo podijeliti svaku pojedinačnu ocjenu dobrote s ukupnom, i dobiti vjerojatnost da će pojedinačna jedinka biti odabrana iz cijele populacije (Sl. 1.7).

Sl. 1.7 Roulette wheel selection

Ova metoda ima nedostatke: moguće je da je jedna jedinka toliko bolja od drugih da će biti odabrana veliki broj puta, i time onemogućiti malo lošijim jedinkama koje bi mogle imati

kvalitetne značajke da ih prenesu u sljedeće populacije. U drugu ruku, ako se funkcija dobrote ponaša tako da ima sve velike vrijednosti, npr. kreće se u rasponu između 1000 i 1100, gdje je 1000 najbolja vrijednost, a 1100 najlošija, tada vjerojatnost odabira neće odraziti stvarnu kvalitetu jedinki, jer će sve jedinke imati približno jednake šanse za odabir i križanje.

Jedna alternativa, koja je korištena u ovom radu, jest turnirska selekcija. Klasična turnirska selekcija funkcionira tako da se nasumično odabere m jedinki od ukupno n u populaciji, pa se od tih m jedinki odabere ona s najboljom funkcijom dobrote, te se ta jedinka onda koristi kao roditelj za iduću generaciju. Korišten je također još jedan mehanizam opstanka najboljih jedinki, o čemu će se detaljnije pričati kasnije.

1.2.6. Kriterij zaustavljanja

Kriterij zaustavljanja genetskog algoritma može biti broj generacija, tempo stagnacije konvergencije, ili ocjena dobrote svih jedinki u generaciji. Kako se približavamo optimalnom rješenju, tako će naše jedinke sve teže postajati bolje, pa će se u jednom trenutku dogoditi stagnacija u više generacija gdje se najbolji rezultat svake iduće generacije ne mijenja ili je promjena minimalna. Moguće je i npr. da nam 95% jedinki iz jedne generacije ima vrlo sličnu ocjenu dobrote, što nam govori da sljedeće generacije neće puno napredovati. Kada prepoznamo da se dogodi stagnacija, možemo prisilno zaustaviti genetski algoritam i iskoristiti te rezultate. Nekad se može dogoditi da, ovisno o nasumičnoj početnoj populaciji, vrlo brzo konvergiramo do zadovoljavajućeg rješenja a onda velik dio vremena stagniramo, a nekada da se konzistentno spuštamo prema dobrom rješenju, pa je potreban velik broj generacija da do njega i dođemo. Prema tome je dobro imati kriterij zaustavljanja, ali treba paziti da algoritam ne zaustavimo prerano ili prekasno, što se određuje testiranjem.

1.3. Algoritam

Sada kada su objašnjene sve osnovne sastavnice genetskih algoritama, pokažimo kako se genetski algoritam izvršava. Kao prvi korak stvaramo početnu populaciju jedinki koje odgovaraju našem problemu. Veličina i način kreiranja početne populacije su hiperparametri genetskog algoritma koji se također daju optimizirati.

Kada imamo početnu populaciju, pridružimo svakoj jedinci unutar populacije njezinu ocjenu koristeći *fitness* funkciju, odnosno funkciju dobrote. Odrađujemo selekciju jedinki jednim

od gore spomenutih načina, i stvaramo nove potomke koristeći odabrane jedinke kao roditelje koje onda križamo. Mutacija se izvršava na djeci, ali se ne odrađuje prilikom svakog križanja, jer bi zadržavanje dobrih značajki bilo otežano, pa se zato odredi vjerojatnost mutacije. Obično je ta vrijednost vrlo mala (< 1%), kako bismo održali općenitu kvalitetu populacije a ipak dozvolili skok u područje boljeg lokalnog minimuma. Moguće je još odrediti postotak najboljih jedinki koje će sigurno prijeći u slijedeću generaciju direktno iz prethodne, a ostatak nadomjestiti križanjem, kako bismo bili sigurni da ćemo zadržati značajke.

Tako novoformiranu generaciju jedinki ponovno provučemo kroz funkciju dobrote, pa ponovimo cijeli postupak dok ne ostvarimo kriterij zaustavljanja. Pseudokod genetskog algoritma nalazi se ispod [3].

```
Genetski_algoritam
      {
            t = 0
            generiraj početnu populaciju potencijalnih rješenja   P(0)
            Sve dok nije zadovoljen kriterij zaustavljanja
                  {
                        t = t + 1;
                        selektiraj P'(t) iz P(t-1);
                        križaj jedinke iz P'(t) i djecu spremi u P(t);
                        mutiraj jedinke iz P(t);
                  }
            ispiši rješenje;
      }
```

2. Problem naručivanja vozila (DARP)

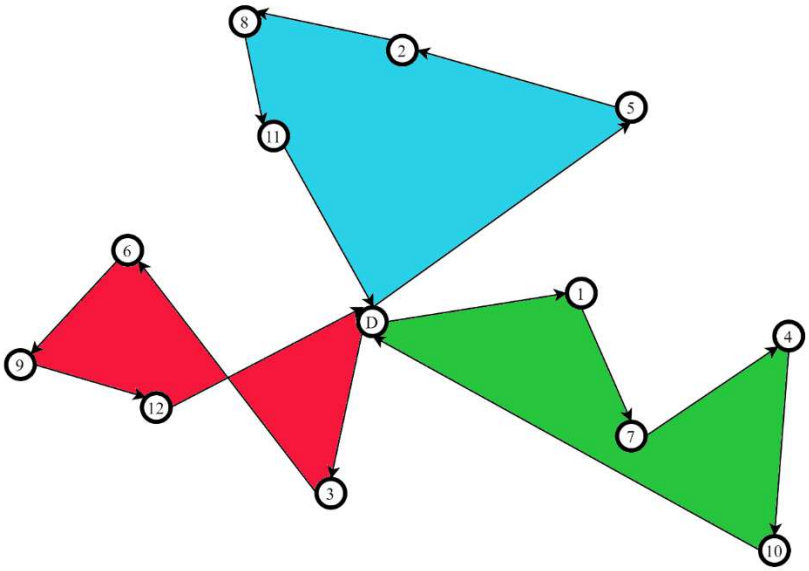
2.1. Pregled problema naručivanja vozila

Problem naručivanja vozila u literaturi je formuliran na različite načine obično ovisno o temeljnom problemu. Formulacija problema u ovom radu temelji se na radu [4], koji svoju definiciju temelji na ranijem radu jednog od autora [5], gdje je inspiracija za problem došla iz stvarnog svijeta – iz Danskog sektora transporta.

U problemu naručivanja vozila imamo flotu vozila za prijevoz s ograničenim kapacitetom vozila, i niz zahtjeva korisnika koji žele doći od početka do odredišta. Svaki od korisnika ima točno dva zahtjeva – jedan za pokupljanje korisnika s početne lokacije, i jedan za dostavljanje korisnika na odredišnu lokaciju.

Kao u primjeru navedenom u uvodu, možemo povući paralelu s kurirskim službama za dostavu hrane koje su se u posljednjih nekoliko godina snažno popularizirale i pojavile širom cijelog svijeta. Korisnici ne moraju nužno biti odvezeni izravno od početka do cilja, već mogu "skrenuti" s puta da bi vozilo u kojem se nalaze pokupilo nekog drugog korisnika, tako da je vožnja dijeljena između više korisnika u isto vrijeme, kao što kurir na biciklu može prvo pokupiti više narudžbi iz više različitih restorana i dostavljati ih jednu po jednu naručiteljima.

Sva vozila kreću s iste početne točke (dalje u radu: depo), i na kraju vožnji se vraćaju u depo time završavajući radni dan. Na Sl. 2.1 možemo vidjeti primjer jednog potencijalnog rješenja. Slovom D označen je depo, a brojevima 1 do 12 označeni su zahtjevi klijenata. Zahtjevi 1-6 su prikupljanja, a zahtjevi 7-12 su dostavljanja korisnika. Zahtjev 1 i 7 simboliziraju istog korisnika – 1 je prikupljanje, a 7 je dostavljanje, tako da moramo osigurati da isti auto pokupi i dostavi jednog korisnika. U ovom slučaju imamo 3 vozila simboliziranih različitim bojama, a putanje im se u ovom primjeru ne sijeku radi bolje preglednosti. Više o preciznoj definiciji instance problema ćemo vidjeti u formalnoj definiciji.



Sl. 2.1 Primjer grafa vožnji

U ovom slučaju razmatramo statički slučaj u kojem se svi zahtjevi znaju unaprijed, i neće se pojaviti dodatni zahtjevi tijekom izvršavanja vožnji. Broj vozila je ograničen, kao i kapacitet svakog pojedinačnog vozila Q, koji je isti za sva vozila. Svaki zahtjev (i prikupljanja i dostavljanja korisnika) imaju definirane vremenske prozore, odnosno najranije vrijeme dolaska vozila i najkasnije vrijeme dolaska vozila, koje definiraju korisnici.

Tu postoji jedno ograničenje: Korisnik smije odrediti vremenski prozor ili za prikupljanje ili za dostavu, ne oboje. Razlog tome je što bi u tom slučaju svaki korisnik potencijalno odredio prozore prikupljanja i dostavljanja tako da je između njih točno ono vrijeme koje treba vozilu da dođe od početka do odredišta, pa bi problem bio nerješiv uz kombinaciju tih uvjeta. Vremenski prozori definirani u problemu su meka ograničenja, to jest dopušteno je da vozilo za nekog korisnika prekrši vremenski prozor, ali takvo rješenje se onda proporcionalno kažnjava u funkciji dobrote. To omogućava genetskom algoritmu da koristi loša rješenja kako bi u konačnici došao do onih koje zadovoljavaju (ili približno zadovoljavaju) sva ograničenja problema.

Definirano je i vrijeme posluživanja (eng. *service time*), koje je vrijeme potrebno da korisnik uđe u vozilo i da vozilo krene s te lokacije. Vrijeme posluživanja je vezano uz prirodu problema: osobe s posebnim potrebama (npr. invalidi) ne mogu tako brzo ući u vozilo; potrebno je u vozilo ubaciti i korisnika i kolica, ili bilo kakvu drugu opremu koja je potrebna jednom korisniku.

Osim vremenskih prozora za prikupljanja/dostavljanja, imamo i ograničenje na trajanje rute jednog vozila. Trajanje rute jednog vozila je vrijeme koje je potrebno vozilu da krene s depoa, obavi sve njemu dodijeljene zahtjeve, i vrati se natrag u depo. U slučaju da premašimo trajanje rute, npr. jedno rješenje je formirano na način da jedan auto odradi sve zahtjeve, a da ostala vozila ne krenu s depoa, takvo rješenje ćemo također prigodno penalizirati u funkciji dobrote, jer nam nije u interesu da jedna osoba radi prekovremeno a da imamo neiskorištene resurse.

Vrlo slično trajanju rute je ukupan prijeđeni put vozila, što možemo gledati i kao potrošnju goriva. Moguće je da jedno vozilo, nakon što pokupi jednog korisnika, dođe do lokacije idućeg korisnika na svojoj listi, i onda mora čekati do najranijeg vremena prikupljanja drugog korisnika, čime mu se povećava trajanje rute, ali se ne povećava ukupan prijeđeni put. Također, na ukupno trajanje rute utječu i vremena posluživanja, a ne utječu na ukupan prijeđeni put.

U ovom primjeru mogli smo vidjeti kako će se onda prvi korisnik voziti dulje vremena nego što je najmanje potrebno od njegove početne točke do lokacije, pa u funkciju dobrote uključujemo i ukupan višak vremena provedenog u vožnji (*excess ride time*), kako bismo pokušali što brže odvesti korisnike i povećati njihovo zadovoljstvo.

Primjeri ograničenja koja bi se možda mogla uključiti u model su ravnomjerno raspoređivanje vozača – ne želimo da jedan vozač radi puno više od drugog, fiksni troškovi – kapital, amortizacija vozila, oportunitetni trošak posjedovanja vozila i nekorištenja istog, trošak korištenja depoa, plaće vozača (pod pretpostavkom da su svi vozači stalno zaposleni, umjesto da bivaju plaćeni po učinku), itd.

2.2. Formalna definicija

Pretpostavimo da imamo skup od 2*n* korisničkih zahtjeva. Svaki zahtjev sastoji se od početne lokacije, v_i, i odredišne lokacije, v_{n+i}. Definiran je i svim korisnicima jednak broj mjesta koji im je potreban kako bi stali u vozilo, kao i preferirani vremenski prozor ukrcaja s početne lokacije, v_i, ili dostavljanja na odredišnu lokaciju, v_{n+i}. Svako vozilo kreće iz depoa i vraća se u isti depo po završetku usluge. Sad možemo definirati problem po uzoru na Cordeaua i Laportea [6], na čijim ćemo instancama problema testirati genetski algoritam :

$V = \{v_0, v_1, \ldots, v_{2n}\}$	*skup vrhova na grafu, gdje je v_0 depo*
$A = \{(v_i, v_j) : v_i, v_j \in V, i \neq j\}$	*skup bridova grafa*
$G = (V, A)$	*potpuni graf*
Q	*ukupan broj mjesta u bilo kojem vozilu*
q_i	*broj mjesta potreban za smještanje korisnika*
	gdje je $q_0 = 0$, jer je riječ o depou
d_i	*vrijeme posluživanja, gdje je $d_0 = 0$*
e_i	*najranije vrijeme dolaska na lokaciju*
l_i	*najkasnije vrijeme dolaska na lokaciju*
$t_{i,j}$	*distanca (vrijeme) od lokacije i do lokacije j*
L	*maksimalno vrijeme vožnje korisnika*
T	*maksimalno vrijeme vožnje vozila*
m	*broj raspoloživih vozila*

U ovakvoj definiciji, $q_1, ..., q_n$ iznose 1, a $q_{n+1}, ..., q_{2n}$ iznose -1: svim korisnicima treba jedno mjesto da uđu u vozilo, a kada izađu iz vozila jedno mjesto se oslobađa. DARP problem se sastoji od dizajniranja m ruta na grafu G tako da:

- svaka ruta počinje i završava u depou
- za svaki zahtjev i, vrhovi v_i *i* v_{i+n} pripadaju istoj ruti (istom vozilu) i vrh v_{i+n} je posjećen nakon vrha v_i
- popunjenost bilo kojeg vozila ne premašuje Q
- ukupno trajanje rute bilo kojeg vozila ne premašuje T
- ako je moguće, usluga u vrhu v_i počinje u intervalu $[e_i, l_i]$, i sva vozila se vraćaju u depo unutar intervala $[e_0, l_0]$
- ukupno vrijeme vožnje bilo kojeg korisnika ne premašuje L
- Ukupna cijena ruta svih vozila je minimizirana

Ukupna cijena ruta svih vozila računa se kao zbroj ukupnog vremena vožnje svih vozila, ukupnog vremena čekanja vozila i ukupnog vremena vožnje svih korisnika.

Vrijeme vožnje svih vozila je vrijeme proteklo od napuštanja depoa do povratka u depo nakon što su odrađene sve vožnje.

Vrijeme čekanja pojedinačnog vozila odnosi se na vrijeme koje vozilo provede na lokaciji v_i ako dođe prije e_i (najranijeg vremena dolaska), i posljedično mora čekati klijenta koji ne može izaći prije tog vremena.

Vrijeme vožnje svih korisnika je ukupno vrijeme koliko su svi korisnici skupa proveli u vožnji, preciznije, vrijeme između završetka vremena posluživanja na početnoj lokaciji i početka vremena posluživanja na završnoj lokaciji. Primijetimo da će taj broj biti veći od ukupnog vremena vožnje svih vozila zato što se, na primjer, u vozilu može voziti Q osoba u isto vrijeme, pa im se svima vrijeme provedeno u istom vozilu zbraja u ukupno vrijeme vožnje svih korisnika.

U implementaciji su dodani dodatni penali kako bi se u rješenju izbjegla kršenja maksimalnog trajanja rute vozila, maksimalnog vremena vožnje pojedinačnog korisnika, te kršenja vremenskih prozora za klijente. Kako bi se u genetskom algoritmu zadržao širok prostor pretraživanja, postavljene su mekane granice na ove zahtjeve, ali se pokušavaju izbjeći većom penalizacijom u funkciji dobrote.

3. Implementacija

3.1. Okruženje i jezik

U sklopu ovog diplomskog rada implementiran je čitav rješavač problema naručivanja vozila. Tijekom razvoja rješavača cilj je bio isprobati različite metode rješavanja te ih usporediti, pa su u konačnici implementirane dvije verzije genetskog algoritma i jedna jednostavna pohlepna heuristika rješavanja.

Rješavač je pisan u programskom jeziku Python 3.9.5 u Windows 10 operacijskom sustavu. Nisu korištene nikakve pomoćne biblioteke osim onih standardnih, pa je kod moguće izvršavati samo s instaliranim Pythonom, bez dodatnih instalacija i postavki. U ovom poglavlju pokazat će se kako je zadatak prikazan u kodu, koje su prednosti i mane različitih verzija rješavača, te detaljnije objasniti korištene algoritme.

Kod je pisan bez paralelizacije, pa se čitav izvršava u jednom procesu.

3.2. Instance problema

Instance na temelju kojih se gradio kod i na kojima se testirao su iste instance koje su koristili Cordeau i Laporte [6]. Te testne instance stvorili su kako bi analizirali ponašanje algoritma simuliranog kaljenja za rješavanje DARP-a. Nasumično su generirali 20 instanci, ali ipak ne posve nasumično: temeljene su na realističnim pretpostavkama. Naime, informacije o vremenskim prozorima, kapacitetima vozila, lokacijama korisnika, maksimalnog vremena vožnje i maksimalnog vremena jedne rute im je ustupila komisija za promet iz Montreala [4]. U testnim instancama možemo naći izmeću 48 i 288 zahtjeva. Prva polovica zahtjeva rezervirana je za prikupljanja, a druga polovica za dostave. U svakoj instanci sve lokacije su generirane kao skupovi vrhova oko izvorišnih točaka, koje se koriste samo za tu svrhu. Nulti zahtjev označen je kao depo, i njegova lokacija je određena kao prosječna lokacija izvorišnih točaka. Sve instance, kao i njihova najbolja rješenja, preuzeta su sa 48[8].

Jedan primjer testne instance nalazi se na Sl. 3.1 Primjer instance problemaSl. 3.1. Pogledamo li detaljnije tu instancu, prvo saznajemo koliko dostupnih vozila imamo, možemo vidjeti kako ukupno imamo 48 zahtjeva – 24 prikupljanja i 24 dostave. Maksimalno trajanje rute, kapacitet vozila, i maksimalno vrijeme vožnje definirani su posebno za svaku

instancu problema. Svaki zahtjev ima korisnički broj, koordinate x i y u Kartezijevom koordinatnom sustavu, vrijeme posluživanja (u svih 20 instanci vrijeme posluživanja je jednako svim korisnicima – 10 minuta). Zauzeće je 1 za prikupljanja, a -1 za dostavljanja. Imamo definirano i najranije i najkasnije vrijeme dolaska, e_i i l_i, respektivno. Svaki zahtjev za prikupljanje i ima odgovarajući zahtjev za dostavljanje $i + n/2$, gdje je n broj zahtjeva.

Može se primjetiti kako su za prvu četvrtinu zahtjeva, ili prvu polovicu prikupljanja (zahtjevi 1-12), e_i i l_i definirani s 0 i 1440. To u prijevodu znači da nemamo definirano najranije ili najkasnije vrijeme dolaska, to jest zahtjev se može odraditi bilo kad (od 0 do 1440 minuta, koliko ih ima u danu). Druga polovica prikupljanja (zahtjevi 13-24) ima definirane nasumične vremenske prozore inspirirane stvarnim podacima.

Treća četvrtina zahtjeva, ili prva polovica dostava (zahtjevi 25-36), koja odgovara prvoj polovici prikupljanja, također ima definirane vremenske prozore. To je konzistentno s već spomenutim uvjetom da korisnici mogu definirati ili vremenski prozor prikupljanja ili vremenski prozor dostavljanja, a ne oboje. Tim principom onda zadnja četvrtina zahtjeva, ili druga polovica dostava, također nema definirane vremenske prozore.

Primjer na Sl. 3.1 je prvi testni primjer (pr01.txt) od njih 20 koji su se koristili za testiranje ove implementacije genetskog algoritma.

Broj vozila	Broj zahtjeva		Maksimalno trajanje rute		Kapacitet vozila	Maksimalno vrijeme vožnje
3	48		480		6	90

Korisnički broj	x	y	Servisno vrijeme	Zauzeće	Najranije vrijeme dolaska	Najkasnije vrijeme dolaska
0	-1.044	2	0	0	0	1440
1	-2.973	6.414	10	1	0	1440
2	-3.066	0.546	10	1	0	1440
3	5.164	0.547	10	1	0	1440
4	-1.317	6.934	10	1	0	1440
5	-6.741	6.832	10	1	0	1440
6	4.891	0.627	10	1	0	1440
7	0.524	2.226	10	1	0	1440
8	-6.5	7.723	10	1	0	1440
9	-0.417	-0.157	10	1	0	1440
10	2.303	1.164	10	1	0	1440
11	2.548	0.629	10	1	0	1440
12	-4.261	-2.639	10	1	0	1440
13	-7.667	9.934	10	1	325	358
14	-2.067	5.789	10	1	111	152
15	-5.204	0.657	10	1	395	421
16	-4.138	5.082	10	1	386	401
17	-9.194	2.759	10	1	86	114
18	-6.512	3.021	10	1	409	426
19	1.86	9.672	10	1	454	470
20	-4.094	8.321	10	1	175	202
21	-3.776	-3.333	10	1	416	453
22	2.377	2.908	10	1	147	177
23	-4.303	2.045	10	1	471	499
24	-3.53	-2.49	10	1	321	346
25	-5.476	1.437	10	-1	258	287
26	-4.933	3.337	10	-1	329	361
27	5.74	2.382	10	-1	209	252
28	-2.275	5.541	10	-1	416	460
29	-5.662	7.334	10	-1	305	349
30	-3.856	-0.37	10	-1	432	458
31	-1.678	1.954	10	-1	202	236
32	-1.156	1.161	10	-1	225	252
33	-4.655	9.797	10	-1	102	123
34	1.623	0.932	10	-1	260	276
35	0.129	0.735	10	-1	178	215
36	-2.64	2.953	10	-1	381	397
37	0.435	1.469	10	-1	0	1440
38	-5.066	-2.313	10	-1	0	1440
39	-2.283	-0.981	10	-1	0	1440
40	-7.11	-1.862	10	-1	0	1440
41	-0.785	3.207	10	-1	0	1440
42	1.188	-2.493	10	-1	0	1440
43	-1.893	-2.373	10	-1	0	1440
44	-1.192	1.175	10	-1	0	1440
45	2.984	1.163	10	-1	0	1440
46	1.227	-5.581	10	-1	0	1440
47	-3.793	-2.161	10	-1	0	1440
48	4.288	-0.297	10	-1	0	1440

Sl. 3.1 Primjer instance problema

3.3. Objekti u kodu

3.3.1. Objekt Customer

Objekt Customer reprezentira jedan zahtjev iz instance problema. Ima sljedeće atribute:

- customernumber (broj zahtjeva)
- x (x koordinata)
- y (y koordinata)
- servicetime (Vrijeme posluživanja)
- loadrequired (zauzeće mjesta u vozilu)
- earliestarrivaltime (najranije vrijeme dolaska)
- latestarrivaltime (najkasnije vrijeme dolaska)
- actualarrivaltime (stvarno vrijeme dolaska)

Od gore navedenih atributa, već smo upoznati sa svima osim sa stvarnim vremenom dolaska. To je atribut koji se popunjava tijekom simulacije jednog rješenja kako bismo mogli pratiti rezultate. Na ovom objektu definirana je i metoda `resetCustomer` koja resetira stvarno vrijeme dolaska jer se ovaj objekt u memoriji zadržava cijelo vrijeme rješavanja jedne instance problema, a mijenja se samo stvarno vrijeme dolaska.

3.3.2. Objekt Car

Objekt Car reprezentira jedno vozilo. S obzirom da je u problemu definiran samo broj vozila, na početku izvedbe će se inicijalizirati više identičnih instanci objekta Car s različitim pridruženim brojem (vehiclenumber), što služi tome da se prati koliko je udaljenosti koji auto prešao, koliko mjesta ima slobodnog, itd. Atributi koje objekt Car posjeduje su:

- vehiclenumber (broj ovog vozila)
- x (trenutna x koordinata vozila)
- y (trenutna y koordinata vozila)
- currentcustomer (trenutni zahtjev na kojem se nalazi ovo vozilo)
- totaldistancepassed (ukupan do sada prijeđeni put ovog vozila)
- freespace (trenutni slobodni kapacitet vozila)
- currenttime (trenutno vrijeme vozila)
- startime (vrijeme početka vožnje vozila)

- endtime (vrijeme kraja vožnje vozila)
- idletime (vrijeme čekanja vozila)

Na objektu vozilo definirane su i metode:

goFromDepot (depot, customer)

Ova metoda kao argumente prima nultog customera, tj. zahtjev u koji je upisan depo, i zahtjev koji želimo prvi izvršiti za to vozilo. Ova metoda za trenutno vozilo upisuje početno vrijeme i trenutno vrijeme. Početno vrijeme se računa kao min(0,customer.earliestarrivaltime – distances.getDistance(depot, customer)

Objašnjenje za to je sljedeće: Zamislimo početak novog radnog dana. Sva vozila su u depou u ponoć, tj. u nultome trenutku. Prvi zahtjev toga dana ima najranije vrijeme dolaska u 9 sati ujutro. Vozilo koje će krenuti prema lokaciji tog zahtjeva će krenuti točno onoliko vremena prije najranijeg vremena dolaska koliko mu treba da dođe do te lokacije. Obzirom da pretpostavljamo da se vozilo kreće jednu jedinicu udaljenosti po minuti (da nam je računanje jednostavnije), to je točno onoliko vremena koliko iznosi i udaljenost. U našem primjeru, ako vozilu treba 10 minuta do prve lokacije, krenut će u 9:20 kako bi minimizirao troškove. No, ako vozilu treba 10 sati da dođe do prve lokacije (iako je ovo nerealan primjer), vozilo ipak neće krenuti u 23:00, nego u ponoć, kada počinje radni dan, i doći kasnije od najranijeg vremena dolaska.

Kada se odredi početno vrijeme, tada se to vrijeme upiše u atribut currenttime, i time sada znamo gdje se nalazi vozilo (još u depou), koje mu je trenutno vrijeme i kada je započeo s poslom. Kod iz implementacije nalazi se ispod za bolju referencu:

```
def goFromDepot(self, depot: Customer, customer: Customer):
    self.starttime = max(0, customer.earliestarrivaltime -
distances.getDistance(depot, customer))
    self.currenttime = self.starttime
```

serveCustomer (customer)

Ova metoda služi odrađivanju jednog specifičnog zahtjeva koji se prima u argumentu, te odrađuje sljedeće promjene na vozilu na kojem je pozvana:

- Lokacijski atributi x i y se postavljaju na x i y koordinate iz zahtjeva (vozilo se sada nalazi na lokaciji zahtjeva)
- totaldistancepassed se poveća za udaljenost između trenutne lokacije vozila i lokacije zahtjeva

- slobodan prostor u vozilu se promijeni za -1 ili +1, ovisno o tome je li zahtjev prikupljanje ili dostava
- trenutno vrijeme se poveća za udaljenost između starog i novog zahtjeva
- Ako je trenutno vrijeme manje od najranijeg vremena dolaska iz novog zahtjeva, onda povećavamo idletime (vrijeme čekanja vozila) za onoliko vremena koliko će proći do odrađivanja novog zahtjeva. Trenutno vrijeme se postavlja na najranije vrijeme dolaska iz novog zahtjeva. Ako je trenutno vrijeme između najranijeg i najkasnijeg vremena, u ovom koraku ga ne mijenjamo
- na customer objektu (zahtjevu) primljenom u argumentu postavlja se stvarno vrijeme dolaska na trenutno vrijeme vozila
- trenutno vrijeme vozila se povećava za vrijeme posluživanja zahtjeva
- currentcustomer (trenutnikorisnik) vozila se postavlja na novog korisnika

goToDepot(depot)

Ova metoda „vraća vozilo na depo“, poziva se na kraju simulacije vožnji za sva vozila te postavlja potrebne atribute:

- završno vrijeme vozila se postavi na trenutno vrijeme + udaljenost od zadnjeg korisnika do depoa
- ukupan prijeđeni put se također povećava za udaljenost od zadnjeg korisnika do depoa
- trenutni korisnik se postavi na depo
- trenutne koordinate se postave na koordinate depoa

resetCar(depot, freespace)

Ova metoda u argumentima prima nulti zahtjev (depo) i kapacitet vozila Q iz instance problema. Služi tome da se vozila resetiraju na kraju računanja funkcije dobrote za svaku od jedinki u populaciji. Tako se trenutni korisnik postavi na depo, ukupna prijeđena udaljenost na 0, slobodan prostor u vozilu na Q, trenutno vrijeme, početno vrijeme, završno vrijeme i vrijeme čekanja na 0.

3.3.3. Objekt Customer

Objekt Customer u memoriji predstavlja jedan zahtjev. Budući da tijekom cijelog izvršavanja genetskog algoritma želimo imati približno jednako zauzeće u memoriji, instance ovog objekta također imaju atribute za praćenje i resetiranje korisnika kako bismo mogli izračunati funkciju dobrote. Atributi prisutni u ovom objektu su:

- customernumber (broj trenutnog korisnika) – odnosi se na broj korisnika iz instance problema
- x (x koordinata)
- y (y koordinata)
- servicetime (Vrijeme posluživanja) – postavlja se iz instance problema
- loadrequired (Zauzeće) – kao što je spomenuto ranije, ova vrijednost je 1 za sve zahtjeve prikupljanja, a -1 za sve zahtjeve dostavljanja u svim instancama problema, no algoritam radi i za drugačije instance gdje ove vrijednosti nisu jednake za sve
- earliestarrivaltime – e_i
- latestarrivaltime – l_i
- actualarrivaltime – vrijednost koja se postavlja u metodi ServeCustomer iz vozila. Odnosi se na stvarno vrijeme dolaska kako bi se pratila kašnjenja

3.3.4. Objekt ProblemInstance

ProblemInstance je jednostavan objekt – služi nam da tokom čitavog izvršavanja genetskog algoritma na istom mjestu imamo sve informacije iz definicije problema. Posjeduje sljedeće atribute:

- numberofvehicles – broj dostupnih vozila u ovoj instanci problema
- numberoflocations – broj lokacija/zahtjeva u problemu
- maxrouteduration – iznos najdulje dozvoljene vožnje jednog vozila
- vehiclecapacity – Q (kapacitet vozila)
- maximumridetime – iznos najdulje dozvoljene vožnje jednog korisnika

3.4. Prva verzija genetskog algoritma

Prva verzija genetskog algoritma inspirirana je velikim dijelom radom Cubillosa, Rodrigueza, i Crawforda [7]. U ovoj verziji, genetski algoritam rješava čitav problem, pa svaka jedinka reprezentira jedno cijelo rješenje problema, to jest sadrži informacije o svim rutama vozila i pripadajućim korisnicima, uz pretpostavku nekih temeljnih pravila vožnje. Jedinke kroz generacije evoluiraju pokušavajući kontinuirano minimizirati funkciju dobrote, a genetski algoritam završava kada se dosegne maksimalan broj generacija – nije definiran drugi kriterij zaustavljanja. Jedinke kroz generacije prolaze kroz proces selekcije, križanja i mutacije, te se tako poboljšavaju dok ne konvergiramo blizu optimalnom rješenju.

3.4.1. Jedinka (Unit)

Prikaz kromosoma jedinke u ovoj verziji genetskog algoritma je lista parova vozilo-korisnik. Svaki par odgovara jednom genu u kromosomu te jedinke. Jedinka, kao objekt u memoriji, također ima pomoćne atribute koji nam koriste da bismo zapisali rezultate genetskog algoritma. Na jedinci su također implementirane i različite mutacije koje, kada se pozovu, nasumično mijenjaju jedinku. Mutacije će se detaljnije objasniti kasnije. Popis svih korištenih atributa jest:

- fitnessvalue (rezultat funkcije dobrote)
- customerOrder (redoslijed korisnika, koristi se u drugoj verziji genetskog algoritma)
- totalRouteDuration (ukupno trajanje vožnje svih vozila)
- totalRideTime (ukupno vrijeme vožnje svih korisnika)
- totalWaitingTime (ukupno vrijeme čekanja svih vozila)
- totalDistanceTraveled (ukupno prijeđeni put vozila)
- chromosome (rješenje)

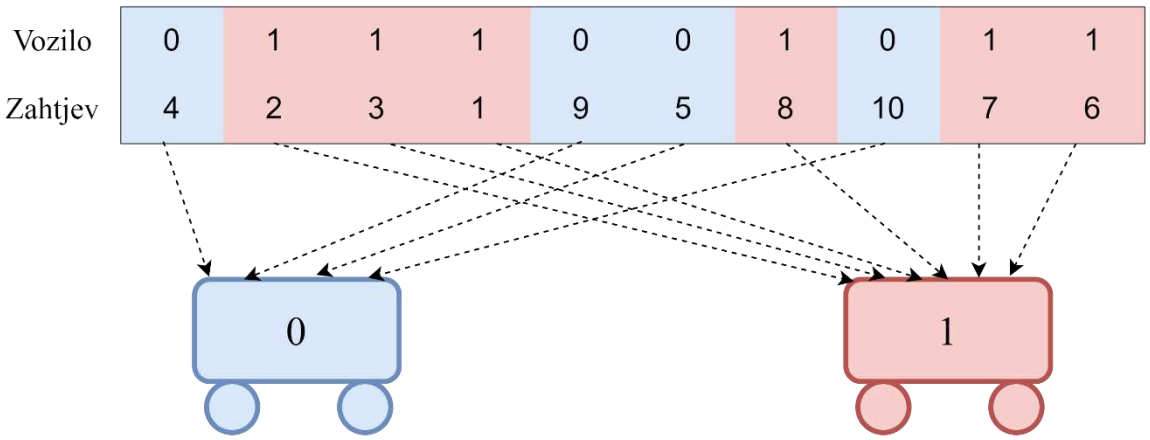
Popis svih metoda na ovom objektu je:

- mutateSwitchCustomers()
- mutateChangeCarForOneCustomer (probleminstance)
- alternateMutate()

Kromosom jedne jedinke određuje koje će vozilo voziti koje korisnike i kojim redoslijedom i nalazi se na Sl. 3.2. U ovom primjeru imamo 10 zahtjeva, gdje su zahtjevi 1-5 prikupljanja, a 5-10 dostave. Vozilo 0 će prvo pokupiti korisnika iz zahtjeva 4, onda će odraditi dostavu iz zahtjeva 9, što je zapravo dostavljanje korisnika 4, jer su 4 i 9 par na skupu od 10 zahtjeva. Tada će pokupiti korisnika iz zahtjeva 5 i odmah potom ga dostaviti. U vozilu 0 će se prvo sam voziti korisnik 4, pa sam voziti korisnik 5.

Vozilo 1, u drugu ruku, će prvo pokupiti korisnika 2, pa korisnika 3, pa korisnika 1. Onda će dostaviti korisnika 3 (zahtjev 8), pa će dostaviti korisnika 2 (zahtjev 7), i konačno dostaviti korisnika 1 (zahtjev 6).

Za ovakav prikaz rješenja ne bi bilo dobro da su za dva zahtjeva istog korisnika (zahtjevi *i* *i+n/2*) definirana različita vozila. Takvo rješenje nije validno iz očitih razloga, pa zato imamo mehanizme koji takvu pojavu tijekom mutacija i križanja sprječavaju, izbjegavaju, ili u najgoru ruku, popravljaju. Takvi mehanizmi će biti podrobnije objašnjeni kasnije.



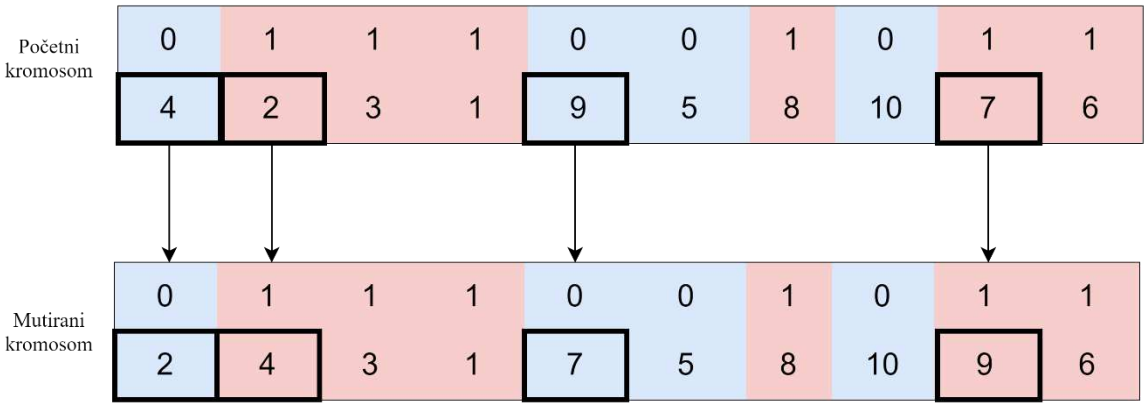
Sl. 3.2 Prikaz rješenja u jednom kromosomu

3.4.2. Mutacije

Kao što je spomenuto ranije, objekt Jedinka ima definirane 3 metode za mutaciju. U ovom odjeljku objasnit ćemo kako funkcioniraju mutateSwitchCustomers() i mutateChangeCarForOneCustomer(probleminstance)

Mutacija zamjene korisnika - mutateSwitchCustomers()

Metoda mutateSwitchCustomers zamjenjuje dva korisnika u njihovim respektivnim rutama. U kromosomu jedinke metoda odabere nasumično gene koji označavaju 2 korisnika i zamijeni im mjesta u kromosomu. Na Sl. 3.3 vidimo primjer mijenjanja gena. Prvo odaberemo dva nasumična zahtjeva različitog broja, nađemo ih u kromosomu, i zamijenimo ih. U ovom primjeru su odabrani zahtjevi 2 i 4, pa će algoritam proći kroz čitav kromosom, i kada naiđe na gen sa zahtjevom 2, promijenit će ga na 4, kada naiđemo na gen sa zahtjevom 4, promijenit će ga na 2, kad naiđe na gen s korisnikom 2 + pola duljine kromosoma (2+5 = 7), zamijenit će 7 sa (4+5=9) 9, i zamijenit će 9 sa 7.



Sl. 3.3 Mutacija zamjene zahtjeva

Prisjetimo se, svaki zahtjev manji ili jednak polovici ukupnog broja zahtjeva (u ovom slučaju su to 1, 2, 3, 4, i 5) odgovara ukrcaju korisnika, a svaki zahtjev veći od polovice odgovara iskrcaju onog korisnika koji je označen s brojem tog zahtjeva umanjenog za polovicu duljine kromosoma.

Primijetimo da se poredak ukrcaja/iskrcaja u kromosomu nije promijenio. Ovakva mutacija je rađena tako da zadržava validnost kromosoma i ne uzrokuje prepunjenost jednog vozila. Ako je kromosom bio validno rješenje prije mutacije, ostat će validno rješenje i nakon mutacije. Također primijetimo da mijenjamo ukrcaj s ukrcajem i iskrcaj s iskrcajem, pa, pod uvjetom da je početni kromosom bio validan, ne može se dogoditi da se u novom kromosomu iskrcaj nalazi prije ukrcaja, što bi uzrokovalo rješenje koje nije validno.

Ovakva mutacija dozvoljava i zamjenu dva zahtjeva koja imaju isto pridruženo vozilo. Inicijalno je bila zamišljena kao zamjena poretka zahtjeva koji imaju različita pridružena vozila (kao u Sl. 3.3), i nije dozvoljavala zamjenu poretka dva korisnika koji se voze istim vozilom. No, isprobavanjem takve mutacije došlo se do zaključka kako je korisno zamijeniti i položaje korisnika u ruti jednog vozila. Konvergencija prema rješenju je u konačnici postala stabilnija koristeći poboljšanu verziju mutacije.

Mutacija promjene vozila – mutateChangeCarForOneCustomer(probleminstance)

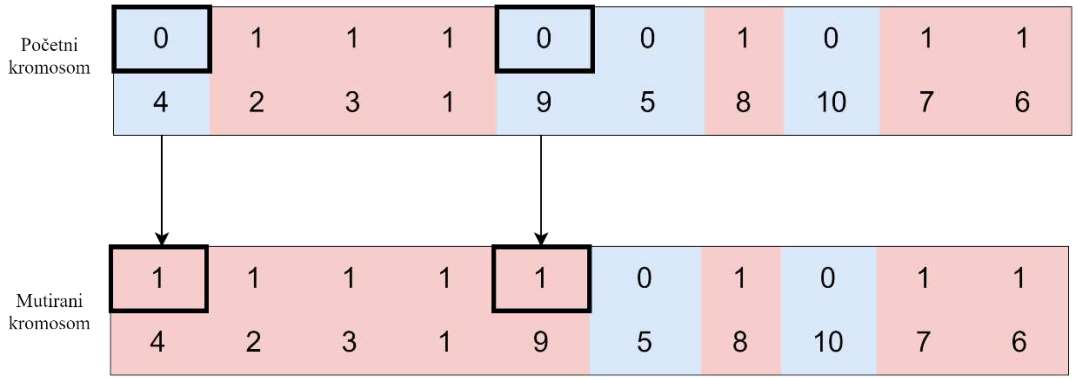
Metoda mutateChangeCarForOneCustomer(probleminstance) je jednostavija od mutateSwitchCustomers mutacije, ali sa sobom nosi dodatne poteškoće. Naime, ova mutacija samo odabire nasumičan gen iz kromosoma i mijenja mu broj vozila u neki drugi. Na prvi pogled se čini kao jednostavna mutacija koja ne mijenja puno stvari – samo broj vozila za 2 mjesta (ukrcaju i iskrcaju), no može se dogoditi slijedeći problem: što ako je nakon mutacije ukrcaj promijenjenog korisnika postavljen na poziciju u kojoj bi prepunio broj mjesta u vozilu?

Neka se uzme isti kromosom kao maloprije, ovaj put uz mutaciju promjene vozila (Sl. 3.4). Funkcija u argument prima instancu problema kako bi imala informaciju o maksimalnom kapacitetu vozila.

Pretpostavimo da je maksimalan kapacitet vozila u ovoj instanci problema, Q, bio 3. Naš početni kromosom je kao rješenje validan – ni u kojem trenutku nismo prekoračili broj mjesta u vozilu: vozilo 0 je prvo pokupilo korisnika 4, pa ga iskrcalo (zahtjev 9), pa je pokupilo korisnika 5, pa ga iskrcalo (zahtjev 10). Vozilo broj 1 je u jednom trenutku bilo

puno: prvo je pokupilo korisnike 2, 3, i 1, pa je iskrcalo korisnike 3, 2 i 1, navedenim redoslijedom. Nasumično smo odabrali kako ćemo mutirati gen koji sadrži korisnika 4. Jedino drugo dostupno vozilo je vozilo broj 1, pa će se genima sa zahtjevima 4 i 9 zamijeniti vozilo s broja 0 na 1 (inače se vozilo mijenja na nasumično vozilo iz skupa ostalih vozila). No, sada se ispostavlja da će vozilo 1 pokupiti 4 korisnika zaredom, a ima samo 3 dostupna mjesta.

Budući da se ovaj problem "pretrpavanja" događa i kod križanja ovakvih jedinki, implementirana je i svojevrsna "popravljač funkcija" kromosoma koja će ga promijeniti na način da ponovno presloži korisnike uz minimalno promjena kako bi se ostvarila validnost rješenja: funkcija simulira vožnje prema zadanom poretku kromosoma i kada primijeti da u vozilu više nema mjesta, prisilno prebaci gen koji označava iskrcaj jednog od trenutnih korisnika u vozilu na mjesto koje prethodi promijenjenom genu. Više o ovoj i o ostalim funkcijama koje popravljaju kromosom će se objasniti u sekciji križanja.



Sl. 3.4 Mutacija zamjene vozila

Još jedna mutacija je implementirana u sklopu ovog rada, ali ona djeluje na drugom prikazu rješenja, pa će se detaljnije objasniti u poglavlju drugog genetskog algoritma.

3.4.3. Simulacija vožnji – simulateRides

Sada kada su definirani svi potrebni elementi za odrađivanje simulacije vožnji, može se objasniti postupak kojim se ona radi. Funkcija simulateRides(unit, customers, cars) u argument prima jedinku za koju simulira vožnju, listu zahtjeva, i listu vozila generiranih iz instance problema. Jedinka unit sadržava redoslijed posluživanja zahtjeva i njima pridruženih vozila.

Prisjetimo se kako je prvi član liste korisnika uvijek depo. Pretpostavka je da su sva vozila resetirana, tj. currentCustomer atribut im je postavljen na depo, a svi atributi koji mjere stanja

su postavljeni na 0. Također pretpostavlja se da je lista zahtjeva sortirana po broju zahtjeva, tako da customers[0] odgovara depou, customers[1] odgovara zahtjevu broj 1, itd. U algoritmu simulacije vožnji, u petlji se prolazi kroz svaki gen u kromosomu jedinke, te se izvršavaju zadane radnje. Ako je trenutni zahtjev vozila (currentCustomer) jednak depou, onda se na vozilu (instanca objekta Car) poziva metoda goFromDepot i time počinju mjerenja.

Za svaki ostali gen poziva se metoda serveCustomer na vozilu za zahtjev iz trenutnog gena, koja simulira kretanje vozila do tog zahtjeva. Kada ova petlja završi, na svim vozilima se pozove metoda goToDepot(customers[0]) i simulira povratak vozila na depo. Tada su sve informacije o prijeđenim rutama zapisane na vozilima i zahtjevima, pa se lako može izračunati ocjena dobrote za tu specifičnu jedinku.

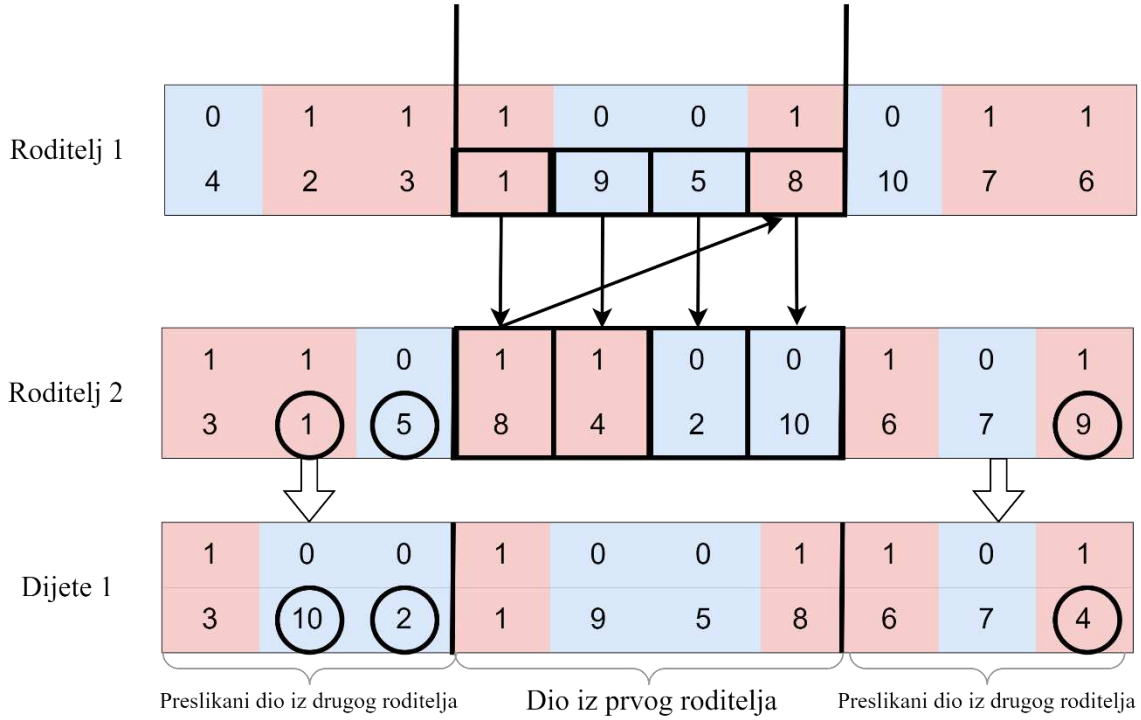
3.4.4. Adjusted Partially Mapped Crossover – Prilagođeno križanje s djelomičnim preslikavanjem

Glavni mehanizam genetskog algoritma oslanja se prvenstveno na križanja, pa onda na mutacije. Zato je važno naći križanje koje će kvalitetno zadržati djelomične značajke jedinki roditelja, bile te značajke loše ili dobre. Ako su roditelji bili dobri, važno je da nakon križanja njihova djeca budu slična dijelom i jednom i drugom roditelju, pa potencijalno i biti bolja od oba roditelja. Prvo križanje implementirano u ovoj verziji algoritma je križanje s djelomičnim preslikavanjem koje je u svojoj osnovnom obliku objašnjeno ranije. S obzirom na specifičnu strukturu kromosoma korištenog ovdje, križanje je prilagođeno kako bi odgovaralo zadanom problemu.

Kao i u osnovnom obliku križanja, u kromosomu se na dva različita mjesta nasumično odaberu točke. Sve između tih točaka preslikava se na djecu bez promjena, a ostatak se preslikava pomoću mapiranja kako ne bismo dobili dvostruke vrijednosti. Detaljan primjer nalazi se na (Sl. 3.5).

No, budući da je izgled kromosoma ovdje ponešto drugačiji nego u primjeru običnog Partially Mapped Crossovera, mora se voditi račun o dodatnim problemima. U primjeru smo nasumično odabrali niz od četvrtog do sedmog gena. Kao i ranije, sada moramo preslikati te vrijednosti u vrijednosti iz drugog roditelja kako ne bismo imali ponavljanje zahtjeva. Tako se zahtjev 1 preslika u zahtjev 8 pa, obzirom da je 8 također u odabranoj traci, 8 se preslikava u 10, pa se 1 tako tranzitivno preslika u 10. Međutim, nećemo pratiti samo koji se zahtjev

preslika u koji zahtjev, već u koji gen. Tako će nam se zahtjev 1 preslikati u gen [0, 10], što nam omogućuje da ga čitavog zamijenimo. Zahtjev 9 će se preslikati u [1,4], a zahtjev 5 u [0,2] (zaokruženi zahtjevi).



Sl. 3.5 Prilagođeni Partially Mapped Crossover

Vidimo kako je prvo dijete dobilo kompletan srednji dio iz prvog roditelja bez promjena, a ostali dijelovi su prepisani iz drugog roditelja, osim ako se (poput 1, 5, i 9) nalaze u srednjoj traci samo prvog roditelja i ne mogu se prepisati u dijete jer će to uzrokovati ponavljanje zahtjeva. Tako se zahtjev 1 preslikao u [0,10], zahtjev 5 u [0,2], a zahtjev 9 u [1,4].

Dijete sada nema zahtjeve koji se ponavljaju, ali sada su se pojavile druge poteškoće. Može se vidjeti kako je sad u djetetu 1 nastao mali kaos s poretkom zahtjeva; iako bi zahtjev 10 trebao doći nakon zahtjeva 5, on se u ovom slučaju nalazi prije, što bi efektivno značilo kako ćemo iskrcati korisnika prije nego smo ga ukrcali u vozilo. Slična je priča sa zahtjevima 9 i 4, obrnuti su im redoslijedi. Tu vidimo još jedan problem: zahtjevi 9 i 4 imaju različito pridruženo vozilo, što nema smisla. Zato će se ovo dijete provući kroz dvije popravljačke funkcije i popraviti će mu se svi nedostaci.

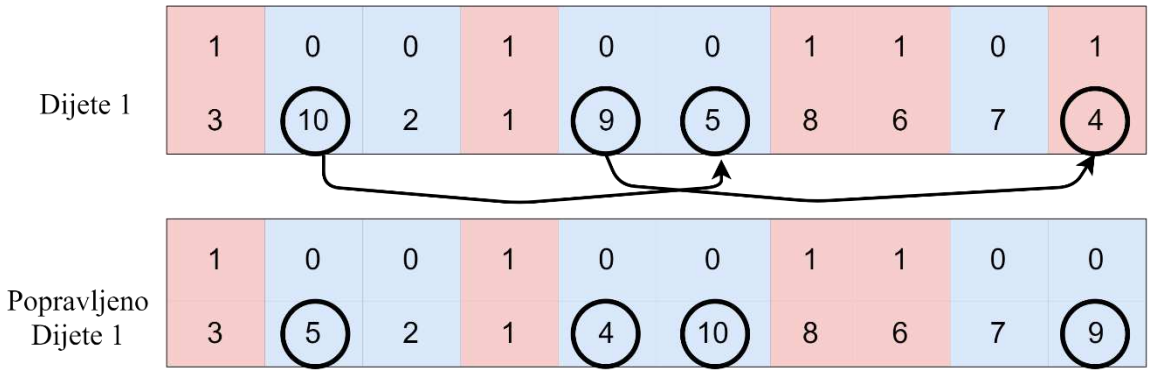
U ovoj implementaciji genetskog algoritma, dvoje roditelja generiraju dvoje djece, pa se drugo dijete generira na isti način kao i prvo, s istim granicama, ali s obrnutim poretkom roditelja – srednji dio će uzeti od drugog roditelja, a ostatak mapirati iz prvog.

3.4.5. fixOrderOfCustomersAndMatchVehicleNumbers (unit) – Popravak redoslijeda i brojeva vozila.

Ova metoda u argument prima jednu jedinku čiji kromosom treba popraviti. Algoritam radi ovako: Prolazi kroz svaki gen i zapisuje zahtjeve u mapu posjećeni zahtjev : vozilo. Za svaki gen provjerava je li njegov suprotan zahtjev već posjećen (suprotan zahtjev iskrcaju je ukrcaj i obrnuto).

Ako je trenutni zahtjev ukrcaj, a nismo posjetili njemu suprotan zahtjev, upisujemo ga u mapu kao ukrcaj : broj vozila, a gen ne mijenjamo. Ako smo pak već posjetili iskrcaj prije ukrcaja, trenutni gen će se promijeniti tako da se umjesto ukrcaja stavi iskrcaj i dodijelit će mu se vozilo iz vrijednosti mape za posjećeni zahtjev. To znači da će se vrijednost vozila uvijek uzimati sa prvog posjećenog gena za jednog korisnika.

U drugu ruku, ako trenutni gen ima zahtjev iskrcaja, a nismo posjetili suprotan zahtjev, u mapu posjećenih zahtjeva upisujemo iskrcaj: vozilo i mijenjamo gen tako da umjesto iskrcaja pišemo ukrcaj. Ako smo trenutno na iskrcaju, a posjetili smo suprotan zahtjev, mijenjamo gen tako da upišemo vozilo iz mape već posjećenog suprotnog zahtjeva. Pokažimo to na primjeru iz prethodnog poglavlja – djetetom koje je rezultiralo s obrnutim redoslijedom korisnika i pogrešno dodijeljenim vozilima (Sl. 3.6).



Sl. 3.6 Popravljanje redoslijeda korisnika i dodijeljenih vozila

Algoritam prolazi kroz čitav kromosom (duljine 10), zapisuje 3:1 u mapu zahtjev: vozilo.

U sljedećem koraku zapisuje 10:0 u mapu, a budući da još nije posjetio zahtjev 5, umjesto 10 u taj gen piše zahtjev 5. Sljedeća dva koraka mogu se preskočiti jer neće biti od važnosti, nakon kojih se dolazi do gena [0,9]. Budući da još nismo posjetili zahtjev 4, umjesto zahtjeva 9 ćemo upisati 4 i u mapu se zapisuje 9:0. Sljedeći gen sadržava zahtjev 5. U mapi već imamo posjećen zahtjev 10, što znači da ćemo umjesto zahtjeva broj 5 upisati zahtjev broj 10, i

postaviti vozilo na ono označeno u mapi s ključem 10: vozilo broj 0. Vozilo broj 0 se već nalazilo u tom genu, pa se zapravo nije promijenio broj vozila, ali kada se dođe do zahtjeva broj 4, vidimo da smo već posjetili njemu suprotan zahtjev, broj 9, pa će se umjesto zahtjeva 4 zapisati zahtjev broj 9 i za vozilo u tom genu će se postaviti vozilo iz mape s ključem 9: vozilo broj 0. U ovom slučaju zamijenjeno je vozilo, te su sada svi geni u dobrom poretku i točno dodijeljenim vozilima.

Iako to nije bio slučaj s roditeljima u ovom primjeru, zamislimo da je u ovoj instanci problema Q bio jednak 2. Tada se vidi kako će vozilo broj 0 prvo ukrcati korisnika broj 5, pa korisnika broj 2, pa korisnika broj 4. To znači da se vozilo u tom trenutku prepunilo, i kako još uvijek imamo rješenje koje nije validno. Tada se poziva funkcija fixOverPopulationOfCarsInChromosome, koja to popravlja.

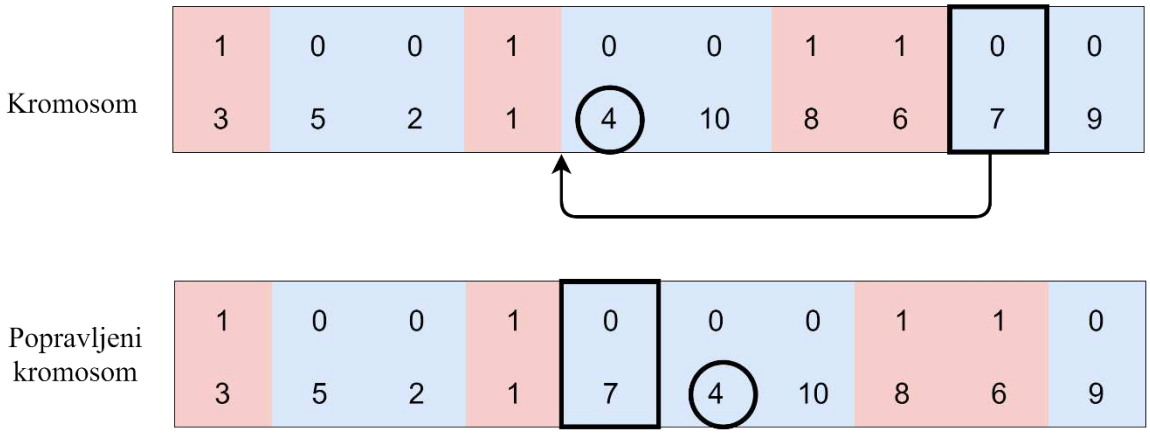
3.4.6. fixOverPopulationOfCarsInChromosome (unit, instance) – popravljanje prepunjenosti vozila u kromosomu

Argumenti koje prima ova funkcija su jedinka i instanca problema – jedinku se popravlja, a instanca problema služi da se zna koji je Q. Prvo se stvori pomoćni kromosom u kojeg će se ugraditi validno rješenje. Funkcija prolazi kroz kromosom, ubacuje validne gene u pomoćni kromosom i prati stanje u vozilima - svaki zahtjev ukrcaja upisuje u mapu (vozilo: skup korisnika u vozilu). Za svaki gen sa zahtjevom iskrcaja funkcija uklanja vezanog korisnika iz skupa korisnika njemu pridruženog vozila i također upisuje taj gen u pomoćni kromosom.

Ako se dogodi ukrcaj radi kojeg će se vozilo prepuniti, nasumično se odabere jedan od korisnika koji je trenutno u vozilu, te se na sljedeće mjesto u pomoćnom kromosomu postavi iskrcaj tog korisnika, njega se ukloni iz skupa korisnika tog vozila, pa se na sljedeće mjesto u pomoćnom kromosomu odmah postavi ukrcaj korisnika koji je uzrokovao problem, ali sad s jednim mjestom praznim u vozilu.

Zahtjev iskrcaja koji je prisilno pomaknut doda se u skup već odrađenih zahtjeva, pa kada u kromosomu naiđemo na gen s tim zahtjevom, ne dodajemo ga u pomoćni kromosom. Kao rezultat onda dobijemo validan kromosom.

Nasumična priroda izbora kojeg ćemo korisnika „prisilno“ iskrcati potencijalno omogućava bolja križanja i mutacije kasnije u algoritmu, iako možda taj izbor nije bio najbolji za kromosom kojeg ispravljamo. Pokažimo kratko na već viđenom primjeru kako radi ova funkcija (Sl. 3.7).



Sl. 3.7 Primjer popravka prenapučenosti vozila

Metoda prolazi kroz kromosom, te kada primijeti da na petom mjestu u kromosomu mora ubaciti korisnika 4 u vozilo 0, a u vozilu 0 nema više mjesta (pod pretpostavkom da je Q = 2), prvo će nasumično odabrati korisnika već u vozilu (2) i odraditi zahtjev iskrcaja korisnika 2 (zahtjev 7) pa će onda pokupiti korisnika 4.

Ovako ne mijenjamo redoslijed ni za koje drugo vozilo, pa možemo biti sigurni da je kromosom nakon ove promjene ostao validan. Ova funkcija se zove nakon svakog križanja i nakon mutacije zamjene vozila jer one mogu uzrokovati prenapučenost.

Prenapučenost vozila bi moglo biti mekano ograničenje, tj. moglo bi se priznavati i takve kromosome, ali ih se prikladno kažnjavati u funkciji dobrote (kao što je napravljeno za kršenje vremenskih ograničenja), ali u stvarnom svijetu je moguće (i nekad bolje) zakasniti minutu jednom korisniku, ali nije moguće pretrpati vozilo, pa je u ovom radu odlučeno kako će se ovakvi kromosomi odmah ispravljati.

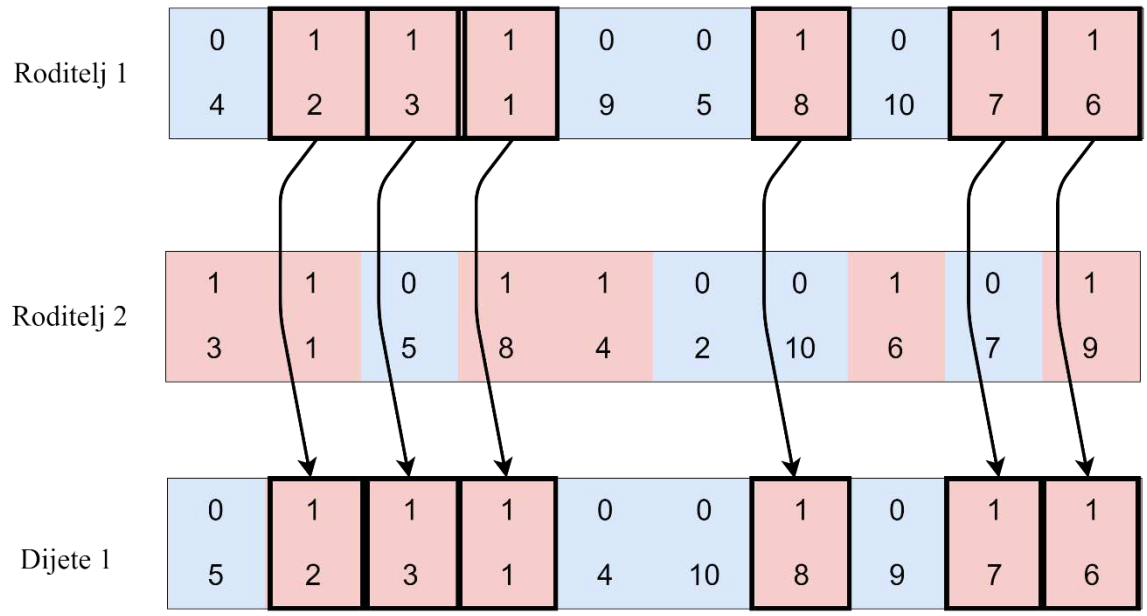
Potencijalno ovakve funkcije ispravljanja smanjuju prostor pretraživanja, jer uvijek „guraju“ iskrcaje bliže ukrcajima, ali u ovom diplomskom radu autor smatra kako se takve „predrasude“ poništavaju kroz mnoge mutacije i križanja. Prostora za poboljšanje ovakvih funkcija uvijek ima.

3.4.7. Partially Car Mapped Crossover – djelomično križanje s preslikavanjem vozila

Drugo križanje implementirano u ovom radu slično je prvom, ali imamo jedan zaokret. Pretpostavka je da će se u jednom trenutku tijekom genetskog algoritma pojaviti jedinka s jako dobro definiranom rutom za nekoliko vozila, a s lošom rutom za ostala. U Partially

Mapped Crossoveru smo nasumično odabrali dvije točke, kromosom podijelili na 3 dijela, i prenosili čitav srednji dio u kromosom djece. Partially Car Mapped Crossover je način križanja specifično kreiran u ovom diplomskom radu za ovaj problem (prema znanju autora, ovakvo specifično križanje nije napravio nitko drugi) – cilj je bio zadržati cijele rute nekih vozila u potomcima, a rute ostalih vozila konstruirati pomoću mapiranja.

Umjesto dvije točke koje dijele kromosom na tri dijela, u ovom križanju odabire se nasumičan broj nasumičnih vozila (može se odabrati samo jedan, mogu svi). Tada se odaberu svi geni koji imaju pridijeljeno jedno od odabranih vozila, te se te gene na tim mjestima prenosi direktno u djecu, a ostatak gena se mapira na sličan način kao i u Partially Mapped Crossoveru (Sl. 3.8).



Sl. 3.8 Primjer Partially Car Matched Crossovera

U ovom primjeru nasumično je odabrano vozilo broj 1, pa su svi geni iz Roditelja 1 koji sadrže vozilo broj 1 direktno prebačeni u Dijete 1, a svi ostali su preslikani iz Roditelja 2, osim u slučaju kada bi se pojavljivali dvaput u kromosomu – tada bi ih se transformiralo na sličan način kao u Partially Matched Crossoveru.

Jedina razlika jest ta što u Djetetu 1 želimo zadržati identičnu informaciju o ruti prvog vozila, pa se npr. gen [1,4] (koji se nalazi na petom mjestu u Roditelju 2) neće preslikati u Dijete 1 kao [1,4], već će mu se broj vozila promijeniti na nasumično odabrano vozilo iz skupa vozila koja nisu odabrani za ovo križanje. Budući da smo u ovom slučaju za križanje odabrali vozilo 1, jedino preostalo vozilo je 0, pa se svi geni iz drugog roditelja koji su imali vozilo 1 sada

prebacuju u vozilo 0. Procedura je jednaka za drugo dijete, zamijenjen je samo poredak roditelja.

U ovom radu korišteno je i treće križanje, ali ono radi s drugom verzijom genetskog algoritma, pa će se objasniti kasnije.

3.5. Jednostavna heuristika za rješavanje

Kao što je spomenuto, u ovom diplomskom radu implementirana su dva genetska algoritma i jedna jednostavna heuristika konstrukcije rješenja. Cilj te heuristike nije bio optimalno riješiti problem, nego usporediti kvalitetu jednostavnog rješenja u odnosu na rješenja genetskog algoritma, kao i nasumično kreiranih jedinki.

Jednostavna heuristika za rješavanje je podijeljena na dva dijela: sortiranje zahtjeva po prioritetu i heuristici za dodjeljivanje vozila za zahtjeve. Ako se prisjetimo instance problema (Sl. 3.1), prva četvrtina zahtjeva su ukrcaji bez vremenskih ograničenja, druga četvrtina su ukrcaji s vremenskim ograničenjima, treća četvrtina su iskrcaji s vremenskim ograničenjima, i zadnja četvrtina su iskrcaji bez vremenskih ograničenja.

Ideja je posložiti korisnike po prioritetu posluživanja tako da pokušamo izbjeći kašnjenje, pa prirodno dolazi zaključak kako zahtjeve trebamo sortirati po l_i, tj. najkasnijem vremenu dolaska. No, budući da je treća četvrtina korisnika (iskrcaji) vremenski ograničena, a ukrcaji tih istih korisnika nisu, ako ih sortiramo po l_i, dogodit će nam se da će nam po prioritetu biti važniji iskrcaji nego ukrcaji istih korisnika. To zaobilazimo tako da privremeno promijenimo l_i za prvu četvrtinu korisnika, i to na način da je postavimo na l_i njihovih iskrcaja umanjenu za vrijeme koje im je potrebno da dođu od lokacije ukrcaja do lokacije iskrcaja direktnom rutom. Od tog iznosa umanjimo također vrijeme posluživanja tog korisnika, pa sada sve korisnike sortiramo po l_i uzlazno.

Nakon što smo sortirali korisnike, moraju im se dodijeliti vozila na „pametan“ način, što će se odraditi u funkciji pridjeljivanja vozila.

appointCarsToCustomersHeuristic(cars, customers, customerOrder) – heuristika dodjele vozila zahtjevima

Ova funkcija u argumente prima listu vozila, listu zahtjeva, te listu cijelih brojeva koja označavaju prioritetni poredak zahtjeva, tj. redoslijed kojima im želimo dodjeljivati vozila. Algoritam ide po redoslijedu korisnika, te za svakog od njih sortira vozila po jednostavnom

prostorno-vremenskom uvjetu: Koje vozilo može biti na lokaciji zahtjeva s najmanje čekanja, a ipak što bliže sredini vremenskog prozora, to vozilo ima viši prioritet.

Tu vrijednost računamo pomoću izraza (1):

$$\text{abs}((e_i + l_i) / 2 - (c_j + t_{i,k})) \quad (1)$$

gdje je e_i najranije vrijeme dolaska zahtjeva i, l_i najkasnije vrijeme dolaska zahtjeva i, c_j je trenutno vrijeme vozila j, a $t_{i,j}$ udaljenost od trenutne lokacije vozila j do lokacije zahtjeva i.

Naprimjer, ako imamo 2 vozila, od kojih je u trenutnom stanju prvo vozilo udaljeno 10 minuta, a vrijeme na satu mu je 100 minuta prije ($e_i + l_i$) / 2, drugo vozilo udaljeno 20 minuta, ali vrijeme na satu mu je 20 minuta prije ($e_i + l_i$) / 2, ipak ćemo htjeti pozvati drugo vozilo kako bismo imali manje praznog hoda.

No, tu postoji iznimka. Ako je jedno vozilo već odradilo zahtjev, npr. u 9 sati ujutro, tada će mu vrijeme na satu uvijek biti daleko bliže idućem zahtjevu ukrcaja nego ostalim vozilima koji su još u depou i imaju vrijeme na satu 0 pa će se uvijek prioritetno uzimati vozilo koje je već na terenu sve dok se ne popuni. Tada će se uzimati iduće najbolje vozilo. Zato je ovaj algoritam prilagođen tako da, ako je ijedno vozilo trenutno u depou, preferira to vozilo nego bilo koje na terenu. Testiranje i jednog i drugog modela pokazalo je da je nekad bolje koristiti jedno po jedno vozilo kako ih popunjavamo, a nekad bolje odmah pokrenuti sva vozila, no da je češće ipak bolji model u kojem šaljemo sva vozila čim možemo.

Postoji mogućnost da u poretku naiđemo na zahtjev ukrcaja, a nemamo slobodnog mjesta u nijednom vozilu. Tada ćemo taj zahtjev staviti u pomoćni stog zahtjeva na čekanju, i nastavit ćemo stavljati sve zahtjeve u stog sve dok ne naiđemo na zahtjev iskrcaja korisnika koji je trenutno u nekom od vozila – može se dogoditi da naiđemo na zahtjev iskrcaja korisnika kojeg nismo do sada uspjeli ukrcati i stavili smo ga na pomoćni stog, pa ćemo i taj zahtjev iskrcaja također staviti na stog, sve dok ne nađemo iskrcaj koji možemo odraditi. Tada ćemo simulirati iskrcavanje i vratiti sve zahtjeve iz reda čekanja natrag na glavni stog, te nastaviti dalje.

Ovako dodjeljivanje ne daje posebno dobre rezultate jer ne pretražuje niti jedan korak u dubinu, i da se još poboljšati pametnijim izborima vozila, no fokus ovog rada je ipak pokazati moć genetskih algoritama na ovakvim problemima.

3.6. Druga verzija genetskog algoritma

U prvoj verziji genetskog algoritma koristila se jedinka koja je reprezentirala čitavo rješenje problema, uz temeljna pravila ponašanja modela. U drugoj verziji pokazat će se jedinka koja ne sadrži čitavo rješenje, ali koja uz pomoć prethodno objašnjene heuristike dodjeljivanja vozila tvori čitavo rješenje. Već smo imali prilike vidjeti nešto slično u stvaranju poretka zahtjeva za dodjelu, kada smo poredali zahtjeva po najkasnijem dozvoljenom vremenu. U ovoj verziji genetskog algoritma optimizirat ćemo poredak zahtjeva, a te ćemo poretke onda upotpunjavati koristeći heuristiku dodjeljivanja vozila. Tako će naša jedinka sada izgledati kao lista brojeva zahtjeva (Sl. 3.9).

4	1	6	2	7	3	5	9	10	8

Sl. 3.9 Primjer jedinke u drugom genetskom algoritmu

Tehnički, jedinka je isti objekt korišten u prvom genetskom algoritmu, ali će mu ovaj put atribut customerOrder biti popunjen ovim redoslijedom, pa će se iz redoslijeda pomoću heuristike dodjeljivanja vozila graditi puno rješenje kako bi se moglo evaluirati.

Za ovakav prikaz na objektu Unit (jedinka) implementirana je jedna metoda križanja i jedna mutacija, koje ne vrše promjene na čitavom kromosomu, već na customerOrder atributu pa se iz njega gradi puni kromosom.

3.6.1. alternateMutate – mutacija

Mutacija definirana na ovakvom prikazu jedinke je jednostavna – zamijene se dva susjedna korisnika u poretku. Jedini uvjet je da ta dva susjeda nisu zahtjevi ukrcaja i iskrcaja za istog korisnika – ako jesu, onda mutacija ne napravi ništa.

Ako dva susjeda nisu zahtjevi istog korisnika, ne možemo pokvariti redoslijed ukrcaja/iskrcaja ni za jednog drugog korisnika jer su svi ostali zahtjevi ostali na istim mjestima, a o prenapučavanju vozila se ne moramo brinuti jer to odradi heuristika dodjeljivanja vozila.

3.6.2. alternatePartiallyMappedCrossover – alternativno križanje s djelomičnim preslikavanjem

Ovo križanje zapravo se odnosi na regularan Partially Mapped Crossover jer djeluje na običnoj listi cijelih brojeva. U kodu i radu ga se oslovljava „alternativnim“ jer je alternativa prvom Partially Mapped Crossoveru. Objašnjenje kako Partially Mapped Crossover funkcionira je već viđeno u poglavlju 1.2.3, pa se neće ponovno objašnjavati. Ono što je važno nadodati da u ovom slučaju takvo križanje može pokvariti redoslijed korisnika, pa imamo i jednostavnu funkciju alternateFixOrderOfCustomers, koja zamijeni mjesta pogrešno poredanim zahtjevima ukrcaja/iskrcaja na vrlo sličan način kao i metoda koja djeluje na čitavom kromosomu, ali ne mora paziti na brojeve vozila, pa je još jednostavnija.

3.7. Prilagođeni genetski algoritam

U slučaju obje verzije genetskog algoritma, kako problem bio učinkovitije riješen, isprobane su razne kombinacije i varijacije osnovnog genetskog algoritma opisanog u 1.3. U ovom poglavlju objasnit će se razlike u odnosu na osnovni genetski algoritam, i specifičnosti implementacije.

3.7.1. Kreiranje početne populacije

U ovom algoritmu imamo tri načina za kreiranje početne populacije, dva za prvu verziju genetskog algoritma i jedan za drugu. Kreiranje početne populacije, ovisno o problemu kojeg rješavamo, može biti nasumično ili pseudo nasumično, tj. građeno stohastički, ali ipak s nekim kvalitetnim osobinama. Pokazalo se kako dobra početna populacija često može značiti razliku između kvalitetnih i nekvalitetnih rezultata genetskog algoritma.

createRandomStartingPopulation (populationsize, cars, customers) – kreiraj nasumičnu početnu populaciju

Prvi način kreiranja početne populacije je nasumičan, ali moramo paziti na poredak i prepunjenost vozila. Ova funkcija samo zove funkciju za kreiranje nasumičnog kromosoma onoliko puta koliko veliku početnu populaciju želimo (populationsize).

createChromosomeGreedyHeuristic(cars, customers) – pohlepna heuristika kreiranja kromosoma

Ova funkcija prvo napravi jednu permutaciju zahtjeva koja poštuje pravilo poretka ukrcaja/iskrcaja, i onda za svaki zahtjev napravi posebnu permutaciju vozila i pokušava

dodijeliti trenutnom korisniku vozilo po redu definiranom u permutaciji vozila. Ako je već dodijelila vozilo ukrcaju, onda dodijeli isto vozilo zahtjevu iskrcaja istog korisnika. Ako ne može dodijeliti vozilo ukrcajima jer su sva vozila puna, onda ih preskače sve dok ne naiđe na zahtjev iskrcaja, pa ponovno prolazi kroz sve neodrađene zahtjeve iz poretka. Tako nastala jedinka je validna po pitanju preopterećenja vozila i poretka ukrcaja/iskrcaja.

alternateCreateRandomStartingPopulation (populationsize, cars, customers) – alternativna funkcija stvaranja nasumične populacije

Ova funkcija služi za stvaranje nasumične populacije i za prvu i za drugu verziju genetskog algoritma. Funkcija će za veličinu željene populacije stvoriti nasumične permutacije niza zahtjeva koje poštuju pravilo poretka ukrcaja/iskrcaja, i za svaku od tih permutacija pozvati funkciju koja dodjeljuje vozila prema zadanom poretku. Funkcija koja dodjeljuje vozila će u isto vrijeme i generirati kromosom za prvu jedinku, i zapisati poredak korisnika u atribut CustomerOrder, pa ćemo moći raditi operacije križanja/mutiranja i računati funkciju dobrote i u prvoj i drugoj verziji algoritma.

Prednost kreiranja ovakve populacije jest to što vozila ipak nisu nasumično raspoređena, već imaju značajno bolju prosječnu funkciju dobrote od nasumične jedinke, a testiranjima se pokazalo kako takve populacije brže i češće konvergiraju ka dobrom rješenju.

3.7.2. Algoritam

Varijacija osnovnog genetskog algoritma je stvorena testiranjima različitih kombinacija i metodom pokušaja/pogreške. Za selekciju koristimo dvije tehnike.

Jedna od njih je turnirski odabir: za križanje uvijek odabiremo dva roditelja iz populacije na način da odaberemo neki broj *x* nasumičnih jedinki iz populacije, ocijenimo koja je najbolja, i nju uzmemo za prvog roditelja, te ponovimo postupak za drugoga. Što je veći broj odabranih jedinki, to je manja šansa lošijim jedinkama da sudjeluju u križanju. Primjer toga bi bio: ako je veličina turnira 1, odabrat ćemo jednu nasumičnu jedinku koju nemamo s čim usporediti, pa će ona automatski postati roditelj. Ako odaberemo veličinu turnira jednaku veličini populacije, uvijek će biti odabrana ona najbolja jedinka.

Drugi mehanizam je preživljavanje najboljih. Odabere se postotak populacije koji će se prenijeti iz prethodne generacije u novu bez križanja. Ovim mehanizmom osiguravamo kako će dobri geni sigurno preživjeti, no ako se uzme prevelik postotak populacije, raznovrsnost jedinki neće biti dovoljno dobra za stabilnu konvergenciju. Onda na ostatak populacije primijenimo turnirsku selekciju i popunimo novu populaciju do kraja.

Bitan faktor genetskog algoritma je također vjerojatnost mutacije. Prilikom svakog križanja nastaju nova djeca. Ako smo definirali vjerojatnost mutacije kao 0.01, onda će u prosjeku svako stoto dijete mutirati i time donijeti dodatnu raznovrsnost populaciji. Međutim, prevelika vjerojatnost mutacije može donijeti nestabilnost u genetskom algoritmu i otežati konvergenciju.

Ono što je u ovom radu dodano kao mehanizam sprječavanja stagnacije, a što je izvanserijski za genetske algoritme, jest prisilna mutacija. Pri testovima je primijećeno kako u jednom trenutku više od pola populacije postane ista jedinka. Kako konvergiramo ka rješenju, dogodi se da se pojave dvije iste jedinke, a ako su dobre, onda njihovim daljnjim križanjem dolazimo do još identičnih jedinki, itd. Tako populacija može stagnirati, tj. zadržati se na trenutnoj najboljoj ocjeni dobrote i teško napredovati.

Kada se dogodi ta situacija, tj. kada je medijan populacije jednak najboljoj jedinci, onda u ovom algoritmu prisilno primijenimo mutaciju na svaku od jedinki koja je jednaka najboljoj, osim na njoj samoj. Tako se dobije kompletno različita populacija, te takvo ponašanje često uzrokuje skok u kvaliteti populacije. Međutim, kako algoritam konvergira ka optimumu, tako se ova situacija događa sve češće pa prisilna mutacija prestane djelovati, ali to se obično događa kada je algoritam vrlo blizu optimuma i s takvim rezultatima budemo zadovoljni.

Na kraju genetskog algoritma, kada dosegnemo maksimalan broj generacija, ispisujemo najbolju jedinku i njezine rezultate.

Pseudokod korištenog genetskog algoritma nalazi se ispod:

```
Genetski_algoritam {
            Za i od 0 do broja generacija{
                  ako je medijan populacije jednak najboljoj populaciji {
                        prisilnoMutirajPopulaciju();
                  }
                  Za postotak populacije{
                        odaberi prvog roditelja turnirskom selekcijom();
                        odaberi drugog roditelja turnirskom selekcijom();
                        stvori djecu križanjem();
                        ako je nasumičan broj manji od vj. mutacije{
                              mutiraj prvo dijete();
                        }
                        ako je nasumičan broj manji od vj. mutacije{
                              mutiraj drugo dijete();
                        }
```

```
                        izračunaj funkciju dobrote za djecu();
                        resetiraj vozila i zahtjeve();
                    }
                    popuni ostatak populacije najboljim jedinkama iz stare;
                    sortiraj novu populaciju po dobroti();
            Ispiši najbolju jedinku posljednje generacije;
}
```

U sljedećem, ujedno i posljednjem poglavlju detaljnije će se opisati korišteni parametri i postignuti rezultati za sve od navedenih metoda, te će se komentirati prednosti i mane genetskog algoritma, prepreke, te moguća područja poboljšanja ovakvog genetskog algoritma.

4. Rezultati, parametri i primjedbe

Genetski algoritam je u prošlim poglavljima temeljito opisan. U ovom poglavlju pokazat će se koji su se parametri koristili za dobivanje rezultata, rezultati će se usporediti sa rezultatima iz relevantnih radova, te će se interpretirati dobiveni rezultati, područja mogućih poboljšanja, te prednosti i nedostaci genetskog algoritma naspram metoda koje su koristili drugi istraživači.

4.1. Korištena metoda rješavanja

U ovom radu istražile su se tri metode rješavanja problema naručivanja vozila: prvi genetski algoritam, jednostavna heuristika, te drugi genetski algoritam. Kao što je bilo i očekivano, neovisno o odabranim parametrima, prvi genetski algoritam je uvijek davao najbolje rezultate. To se može lako objasniti – heuristika dodjeljivanja vozila je jednostavna i gleda samo jedan korak u dubinu, pa čak i s optimalnim poretkom korisnika nećemo dobiti optimalan raspored po vozilima. No, cilj implementacije jednostavne heuristike nije bio razviti ju tako da je usporediva s povijesno najboljim heuristikama, već pokazati koliko će se, unatoč naizgled pametnom odabiru vozila i poretku korisnika, rješenje ovako teškog problema daleko bolje optimizirati koristeći GA.

Obzirom na performanse druge verzije i heuristike, autor je odlučio fokusirati se na prvu verziju genetskog algoritma za usporedbu s već postojećim rješenjima drugih autora. Druga verzija GA i heuristika su donosile 5-7 puta lošije rezultate u funckiji dobrote nego prvi GA, a detalji se mogu vidjeti na Sl. 4.1.

Problem	Prvi GA	Drugi GA	Heuristika
R1a	1849.6396	10194.904	13002
R2a	5280.8663	22699.24	27995
R3a	5788.616	31864.714	34988
R4a	9605.4583	43216.563	51087
R5a	11060.036	48469.692	64291

Sl. 4.1 Usporedba tri algoritma

4.2. Parametri

Ručnim testiranjem parametara na nekoliko instanci problema došlo se do favorita za prvu verziju GA. Parametri koje se trebalo odabrati su:

- težine za argumente u funkciji dobrote
- broj jedinki u turnirskoj selekciji
- postotak jedinki koji se prenosi iz prethodne u sljedeću generaciju
- veličina populacije
- broj generacija
- vjerojatnost mutiranja jedinke
- mutacija
- križanje

Funkcija dobrote se u ovom radu računala koristeći ukupno vrijeme vožnje vozila, ukupno vrijeme čekanja vozila, ukupno vrijeme vožnje korisnika, ukupno kašnjenje, ukupni prekršaj maksimalnih vremena vožnji vozila, i ukupni prekršaj maksimalnog vremena vožnje korisnika.

Budući da su u radu autora Cordeaua i Laportea [6] dobivena samo rješenja koja ne krše vremenske okvire niti maksimalna ograničenja, cilj je bio u GA više paziti na te vrijednosti i prvo pokušati minimizirati njih, pa onda ostale vrijednosti koje će se uspoređivati s njihovim radom. Tako su odabrane težine 1, 1, 1, 5 5, i 5, gdje smo s 5 množili kašnjenja i kršenja ograničenja, a s 1 (tj. nismo množili) vremena vožnje vozila, vremena čekanja, i vremena vožnje korisnika.

Broj jedinki u turnirskoj selekciji je postavljen na 5, iako algoritmi nisu pokazivali očite razlike korištenjem različitih vrijednosti, osim ako je broj odabiranih jedinki bio premalen (1) ili prevelik (>15).

Postotak jedinki koji se prenosi iz prethodne u sljedeću generaciju je postavljen na 4%. U testiranjima gdje se koristilo 15% ili više, vrlo brzo bi se došlo do „zasićenja“ populacije identičnim jedinkama i konvergencija bi se drastično usporila.

Veličina populacije je postavljena na 200 – u slučaju premalene populacije također bi brzo došlo do zasićenja populacije jednom jedinkom, a u slučaju prevelike populacije patila bi vremena izvršavanja algoritma pa bi GA gubio praktični smisao.

Broj generacija je postavljen na 1500, jer se primijetilo kako za sve probleme GA vrlo uspori konvergenciju nakon 1500 generacija (negdje i nakon puno manje, ovisno o problemu), pa se GA izvodi pretjeranu količinu vremena bez velike koristi.

Vjerojatnost mutacije je postavljena na 10%. Ako bi se koristila veća vjerojatnost, populacija bi bila nestabilna, a ako bi se koristila manja vjerojatnost, algoritam bi sporije radio „skokove“ u kvaliteti populacije.

Odabrana mutacija je bila zamjena vozila dvaju korisnika, jer se pokazalo kako je korisnije zamijeniti poredak i vozila dvojici korisnika, a pogotovo zamijeniti poredak dvojici korisnika istog vozila, nego samo prebaciti jednog korisnika u drugo vozilo. Također, odabrana mutacija čuva validnost rješenja pa se prečestim popravljanjima toliko ne riskira predrasuda (eng. *bias*) prema nekim oblicima rješenja.

Križanje koje se koristilo u najboljoj verziji GA je bilo djelomično križanje s preslikavanjem vozila, ali u ovom slučaju se pokazalo kako oba križanja daju slične rezultate. Odabrano je ovo križanje jer je intuitivnije za ovaj specifičan problem.

4.3. Rezultati

Iako su u GA implementiranog u ovom radu korištena rješenja s mekanim ograničenjima (za razliku od Cordeaua i Laportea), rezultati su ispali iznenađujući.

Za svaku instancu problema GA se izvršio 10 puta, a u rezultatima ćemo koristiti prosjek i najbolje rješenje od 10 dobivenih, gdje za usporedbu koristimo već definiranu funkciju dobrote. Najbolje odabrano rješenje neće nužno biti ono koje ima najmanja ukupna vremena vožnje vozila, čekanja, i vožnje korisnika, već ona koja favoriziraju manja vremena kašnjenja i prekršaja, kako bi rezultat bio što bolje usporediv s Laporteom i Cordeauom [6], kao i s autorima koji su inspirirali oblik korištenog genetskog algoritma, Cubillosa, Rodrigueza, i Crawforda [7]. Vjerojatno postoje rješenja koja su drastično bolja od Cordeauovih i Laporteovih po njihovim mjerilima, ali uz ogromna vremena kašnjenja i kršenja ograničenja vožnje, ali takva rješenja ne želimo razmatrati.

Na Sl. 4.2 se nalaze rezultati objavljeni u [7] i [6], a na Sl. 4.3 rezultati dobiveni u ovom radu.

Rezultati koje su dobili Cubillos, Rodriguez, i Crawford koristeći GA

	Route duration		*Vehicle waiting time*				*Ride time*			
			Avg.		*Best*		*Avg.*		*Best*	
	Avg.	*Best*	*Total*	*Avg.*	*Total*	*Avg.*	*Total*	*Avg.*	*Total*	*Avg.*
R1a	1041	1039	252	5.25	260	5.42	477	19.86	310	12.90
R2a	1969	1994	470	4.90	514	5.36	1367	28.47	1330	27.72
R3a	2779	2781	292	2.03	301	2.09	3081	42.79	2894	40.20
R5a	4250	4274	500	2.08	527	2.20	5099	42.49	4837	40.30
R9a	3597	3526	94	0.44	32	0.15	6251	57.88	6719	62.21
R10a	5006	5025	315	1.09	246	0.86	8413	58.42	8341	57.92
R1b	907	928	143	2.98	164	3.42	630	26.24	549	22.89
R2b	1719	1710	198	2.06	162	1.69	1214	25.30	1300	27.07
R5b	4296	4336	552	2.30	568	2.37	4615	38.46	4720	39.33
R6b	5309	5227	630	2.19	513	1.78	6134	42.59	6397	44.42
R7b	1299	1316	102	1.41	128	1.78	990	27.50	784	21.76
R9b	3679	3676	147	0.68	177	0.82	5362	49.65	5358	49.61
R10b	4733	4678	113	0.39	85	0.29	7969	55.34	8119	56.38
Total	40 584	40 508	3808	27.81	3678	28.21	51 600	514.99	51 657	502.72

Rezultati koje su dobili Cordeau i Laporte tabu pretragom

	Route duration	*Vehicle wait. time*		*Ride time*	
		Total	*Avg.*	*Total*	*Avg.*
R1a	881	211	4.4	1095	45.62
R2a	1985	724	7.54	1977	41.18
R3a	2579	607	4.22	3587	49.82
R5a	3870	833	3.47	6154	51.3
R9a	3155	323	1.5	5622	52.05
R10a	4480	721	2.5	7164	49.75
R1b	965	321	6.68	1042	43.4
R2b	1565	309	3.22	2393	49.86
R5b	3596	606	2.52	6105	50.87
R6b	4072	449	1.56	7347	51.02
R7b	1097	129	1.79	1762	48.94
R9b	3249	487	2.26	5581	51.68
R10b	4041	362	1.26	7072	49.11
Total	35 537	6082	42.92	56 900	634.6

Sl. 4.2 Rezultati drugih autora

	Route Duration		Vehicle Waiting Time		Ride Time			
Problem	Avg.	Best	Avg.	Best	Avg.	Best	Best Avg. Late Time	Best Avg. Ride Time Violation
R1a	890	972	114	201	1138	694	0.48595683	0.07394674
R2a	1601	1975	164	491	2190	1969	0.72425273	1.59624029
R3a	2353.5	2386.7	117.25	92.546	3486.8	2957.6	0.344888407	0.918569627
R4a	3251.9	3598.5	269.71	547.53	4635.1	4494.5	0.529672435	1.204654246
R5a	3812.8	3958.1	193.02	316.91	5885.1	4789.7	0.97634376	1.323063408
R6a	4690.7	4773	279.24	322.58	7228.2	7133.2	0.924522088	2.578248611
R7a	1273	1353.7	133.13	161.56	1571	1295.4	1.49871474	0.194605793
R8a	2270.6	2254.3	45.818	14.286	3348.6	2802.5	0.888685602	2.861205659
R9a	3225.4	3304.5	64.377	117.84	5834.6	5947.4	4.718814796	2.516191584
R10a	4422.3	4517.9	96.778	102.31	8099.3	7795.5	5.185603279	4.264299874
R1b	788	765.54	31.309	3.5452	984.11	666.82	0.482367647	0.055866058
R2b	1498.9	1422	54.698	5.5902	2107.7	1733.1	0.420772778	1.322663886
R3b	2306.4	2282	68.717	34.193	3369.6	2554.8	0.338342375	0.41843259
R4b	3001	2940.9	75.223	37.503	4352.6	3635.5	0.197928391	0.817551003
R5b	3749	3981.2	135.48	242.11	5618.1	5129.8	0.584174884	1.3365416
R6b	4492	4456	148.8	138.85	6653.4	6170.7	1.125918783	1.013345061
R7b	1150.3	1119.8	19.2	9.7416	1570.7	1357.6	0.538261459	1.122871983
R8b	2328.6	2355	99.637	88.487	3504.5	2658.1	0.960419878	2.17390812
R9b	3287.1	3336.7	45.28	89.962	5961.7	5414.9	1.87709699	2.982317569
R10b	4388.3	4442	65.622	70.394	7734.1	7084.3	3.618267801	1.998619899
Total *	35659	36637	1249	1882	57264	51711	21.08272015	20.52453758

Sl. 4.3 Rezultati implementiranog genetskog algoritma

Na ovim rezultatima u stupcima „best“ označeni su rezultati iz jedinke koja ima najbolju funkciju dobrote, a ne nužno npr. najbolje ukupno trajanje ruta. Zato na R9a možemo vidjeti kako je prosječan Route Duration manji od „najboljeg“, dok je zapravo najbolji Route Duration zapravo samo Route Duration odabran iz najbolje ocijenjene jedinke.

Uspoređivali su se samo problemi prisutni kod drugih autora, ali ostali rezultati su također prikazani (neobojani). Plavom bojom označeni su rezultati bolji od Cubillosa, Rodrigueza i Crawforda, žutom bojom rezultati koji su bolji od Cordeaua i Laportea, zelenom bojom rezultati koji su bolji i od jednog i od drugog rada, a crvenom bojom rezultati koji su lošiji od oba rada. U zadnjem retku imamo zbrojene rezultate samo iz onih ćelija koje smo mogli usporediti s drugim radovima, dakle bez neobojanih ćelija. Vidimo kako je ukupno ovaj GA mjerljiv ili bolji od jednog ili oba rada korištena za usporedbu, no već je objašnjeno zašto: naš algoritam dopušta kašnjenja, Laporteov i Cordeauov ne dopušta.

Cubillos, Rodriguez, i Crawford [7] su, prema njihovom radu, također postavili mekana ograničenja na kašnjenja i kršenja ograničenja. No, u svojim rezultatima nisu pokazali vremena kašnjenja, pa se ne može znati jesu uspjeli doći do rješenja koja imaju kršenja ili kašnjenja.

Gledajući iz perspektive stvarnog svijeta i uzimajući u obzir kašnjenja i kršenja ograničenja, vidimo da možemo dobiti značajno poboljšana vremena vožnje vozila, vremena čekanja i vremena vožnje korisnika a da pri tome dozvolimo minimalna kašnjenja i ograničenja.

Za primjer se može uzeti uzeti R3a, gdje je naš rezultat značajno bolji od Laporteovog i Cordeauovog u sve 3 kategorije, a prosječno kašnjenje po korisniku nam je samo 0.8 minuta i prosječno kršenje najdulje vožnje korisnika je 2.23 minute.

U slučaju stvarnog svijeta, često nisu mogući savršeni uvjeti i kašnjenja se uvijek događaju, a bilo bi dobro još istražiti koliko bi zadovoljstvo korisnika bilo kada bi se kašnjenje malo povećalo, a cijena prilično smanjila. Takva istraživanja ćemo ostaviti drugim strukama.

Zaključak

U ovom diplomskom radu uspješno je implementiran jedan genetski algoritam za rješavanje problema raspoređivanja vozila. Druga verzija genetskog algoritma i jednostavna heuristika nisu postigli zavidne rezultate, ali su služili kao kvalitetan materijal za usporedbu s prvim genetskim algoritmom, za uvid u mehanizme genetskog algoritma, i u prednosti i nedostatke GA naspram drugih metoda. Mjesta za napredak svakako ima. Heuristika za rješavanje DARP-a se može značajno unaprijediti, što bi posljedično unaprijedilo i drugu verziju GA. Kod se može jednostavno paralelizirati jer GA ima visoku nezavisnost operacija u kodu: svaka operacija djeluje na jednoj ili dvije jedinke. Parametri korišteni u ovom GA su odabrani ručno na temelju kratkog i lokalnog testiranja, no ti parametri se mogu hiperparametrizirati, to jest optimizirati kroz proces unakrsnog testiranja.

Postoji prostor za implementaciju i testiranje dodatnih mutacija i križanja, koje bi možda poboljšali rezultate ili brzinu i stabilnost konvergencije.

Pružena je malo drugačija perspektiva na DARP u odnosu na originalnu definiciju. Dobiveni rezultati su usporedivi s već postojećim rezultatima, čak u nekim instancama i bolji, te je demonstrirana moć GA na ovakvim problemima.

Literatura

[1] Holland J. *Adaptation in natural and artificial systems.* Ann Arbor, MI, USA: University of Michigan Press; 1975.

[2] Xin-She Y. *Nature-inspired opzimization algorithms.* 1. izdanje. Elsevier, 2014;

[3] Golub M. *Genetski Algoritmi, prvi dio*. Dohvaćeno 10.6.2021 sa http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf

[4] Jorgensen RM, Larsen J, Bergvinsdottir. *Solving the Dial-a-Ride problem using genetic algorithms.* The Journal of the Operational Research Society, 58, 10 (2007), str. 1321-1331.

[5] Jorgensen RM (2002). *Dial-a-ride*. PhD thesis, Center for Traffic and Transportation, Technical University of Denmark

[6] Cordeau J-F, Laporte G (2002). *A tabu search heuristic for the static multi-vehicle dial-a-ride problem.* Trasportation Research Part B 37 (2003), str. 579-594.

[7] Cubillos C., Rodriguez N., Crawford B. (2007) *A Study on Genetic Algorithms for the DARP Problem*. In: Mira J., Álvarez J.R. (eds) Bio-inspired Modeling of Cognitive Tasks. IWINAC 2007. Lecture Notes in Computer Science, vol 4527. Springer, Berlin, Heidelberg.

[8] http://neumann.hec.ca/chairedistributique/data/ , posjećeno 12.11.2021.

Sažetak

Problem raspoređivanja vozila s vremenskim prozorima (DARP) je važan logistički NP-težak problem koji je začet kao verzija Problema usmjeravanja vozila (VRP), koji je pak podvrsta vrlo poznatog računalnog problema, Problema trgovačkog putnika (TSP). U ovom radu istražuju se stohastične metode optimizacije DARP-a, specifično genetski algoritmi. Genetski algoritmi iz obitelji Evolucijskih algoritama su se kroz nedavnu povijest pokazali kao važan alat moderne računarne znanosti u optimizaciji teških problema. Inspirirani prirodnom selekcijom, na razne načine imitiraju mehanizme preživljavanja najbolje prilagođenih jedinki jedne vrste, i odumiranja onih manje prilagođenih. U ovom radu primijenile su se neke generalne metode genetskih algoritama, ali su se razvile i nove metode specifične ovom problemu kako bi se dodatno proučio i unaprijedio proces optimizacije. Iskušane su različite verzije genetskih algoritama, te su se usporedili i komentirali dobiveni rezultati. Istražene su prednosti i mane ovakve optimizacije, te su postignuti rezultati uspoređeni s postojećim rezultatima drugih metoda. Implementirane su i jednostavne heuristike za optimizaciju, te su uspoređene s radom genetskih algoritama. Komentirana su i područja za poboljšanje metoda korištenih u ovom radu, kao i smjernice za budućnost.

Summary

Dial-a-ride problem (DARP) is an important logistic NP-hard problem that was concieved as a version of the Vehicle Routing Problem (VRP), which is a subtype of a famous computer science problem, the Traveling Salesman Problem (TSP). In this thesis we explore stohastic methods for optimizing DARP, specifically the Genetic Algorithms. Genetic Algorithms, from the family of Evolutionary Algorithms, have in recent history proved themselves an important tool in modern computer science in optimizing hard problems. Inspired by natural selection, they imitate survival mechanisms of best adjusted individuals of a species in various ways, and also the mechanisms that deny less adjusted individuals reproduction and survival. Some general methods were used in genetic algorithms were applied in this thesis, but also some new methods specific to this problem were developed in order to additionaly explore and improve upon the optimization process. Various versions of genetic algorithms were tried, and the results were compared to the existing results of other methods. Also, simple heuristic methods were implemented, tested, and compared to the performance of genetic algorithms. Areas to improve this implementation were discussed, as well as points for the future.