SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2485

Primjena prirodom inspiriranih optimizacijskih algoritama na kombinatorne probleme

Filip Kujundžić

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2485

Pristupnik:	**Filip Kujundžić (0036479155)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	izv. prof. dr. sc. Tomislav Burić

Zadatak: **Primjena prirodom inspiriranih optimizacijskih algoritama na kombinatorne probleme**

Opis zadatka:

Kombinatorni su problemi danas prisutni u mnogim znanstvenim područjima, no njihovo rješavanje tradicionalnim algoritmima često nije moguće zbog kombinatorne eksplozije. Stoga se neprestano traže novi pristupi u rješavanju, a jedna skupina takvih algoritama su prirodom inspirirani optimizacijski algoritmi. Glavna ideja njihovog pristupa je oponašanje načina na koji se jedinke kreću u prirodi prilikom pretraživanja prostora za optimalnim ili dovoljno dobrim rješenjima. Cilj rada je implementirati prirodom inspirirane optimizacijske algoritme u rješavanju klasičnih kombinatornih problema. Potrebno je napraviti analizu i usporedbu rezultata dobivenih različitim algoritmima.

Rok za predaju rada: 28. lipnja 2021.

SADRŽAJ

1. Uvod

U svakodnevnom se životu često susrećemo s problemima koje bismo željeli riješiti učinkovitije nego što nam predloženo rješenje nalaže. Na primjer, želimo li od kuće do posla stići gradskim prijevozom, radije ćemo odabrati onu opciju u kojoj nema presjedanja, bez obzira na to što je možda malo skuplja od one druge. Navedeni problem pripada u skupinu optimizacijskih problema. U takvim problemima želimo pronaći najbolje dostupne vrijednosti funkcije nad definiranom domenom. No, vrijeme potrebno za izvršenje takvog algoritma može biti i nekoliko tisuća godina, pa i više. Navedena se pojava naziva kombinatorna eksplozija te nastaje zbog slijednog pretraživanja cjelokupnog prostora u kojem se može nalaziti rješenje. Ovdje je dobro zapitati se da li je pretraživanje cjelokupnog prostora nužno. Postoje li neke informacije koje bi nas mogle navesti u kojem smjeru krenuti prilikom pretraživanja? Odgovor na ovo pitanje rezultirao je pojavom algoritama usmjerenog pretraživanja. Jedan je od takvih algoritama algoritam A^*.

Srećom, optimalno rješenje nam gotovo nikad neće biti potrebno. Zato koristimo heurističke algoritme koji pronalaze zadovoljavajuće dobra rješenja pri tome ne garantirajući da će uspjeti pronaći optimalno rješenje. Heurističke algoritme dijelimo na konstrukcijske algoritme (rješenje problema grade dio po dio sve dok ne izgrade kompletno rješenje) i algoritme koji koriste lokalnu pretragu (kreću od početnog rješenja koje uz skup definiranih izmjena inkrementalno poboljšavaju). Posebno su nam zanimljive metaheuristike - heuristike opće namjene čiji je zadatak usmjeravanje problemski specifičnih heuristika u prostoru rješenja u kojem se nalaze dobra rješenja. (Dorigo i Stützle, 2004) Metaheuristike dijelimo u dvije velike skupine algoritama. Prvi rade nad jednim rješenjem dok drugi rade sa skupovima rješenja.

Algoritmi sa zajedničkom idejom da rade s populacijom rješenja nad kojima se primjenjuju evolucijski operatori (selekcija, križanje, mutacija, zamjena) čime populacija iz generacije u generaciju postaje sve bolja nazivaju se evolucijski algoritmi. Velika skupina algoritama koji se dobro nose sa spomenutim problemima inspirirana je prirodnim procesima. Razlog tome je što prirodni procesi optimiraju život u svakom segmentu već više od četiri milijarde godina. Proučavanjem ovih procesa znanost je došla do niza tehnika kojima je moguće rješavati navedene probleme.

U nastavku će biti opisana dva poznata kombinatorna problema - problem trgovačkog putnika (The Traveling Salesman Problem) i problem naprtnjače (Knapsack Problem). Bit će analizirana primjena prirodom inspiriranih optimizacijskih algoritama na navedene probleme te analizirana učinkovitost. Za implementaciju je odabran Python 3, programski jezik opće namjene. Izvorni kôdovi programske implementacije dostupni su na otvorenom GitHub repozitoriju. Oba problema su stara preko dvjesto godina te je zanimljivo kako im relativno nedavno (prije dvadesetak godina) razvijeni algoritmi pristupaju. Iako izgleda da smo u stalnoj potrazi za idealnim algoritmom, onim koji će biti primjenjiv na sve probleme, zapravo nije tako. Wolpert i Macready u svojim su radovima dokazali da najbolji algoritam ne postoji te je navedena formulacija poznata kao *no-free-lunch* teorem. To znači da će za različite probleme različiti algoritmi biti najbolji. Što više algoritama znamo, i što više razumijemo njihovo ponašanje i način rada, veće su nam šanse da odaberemo pravi algoritam za problem koji pokušavamo riješiti. (Čupić, 2013)

2. Kombinatorni problemi

2.1. Problem trgovačkog putnika

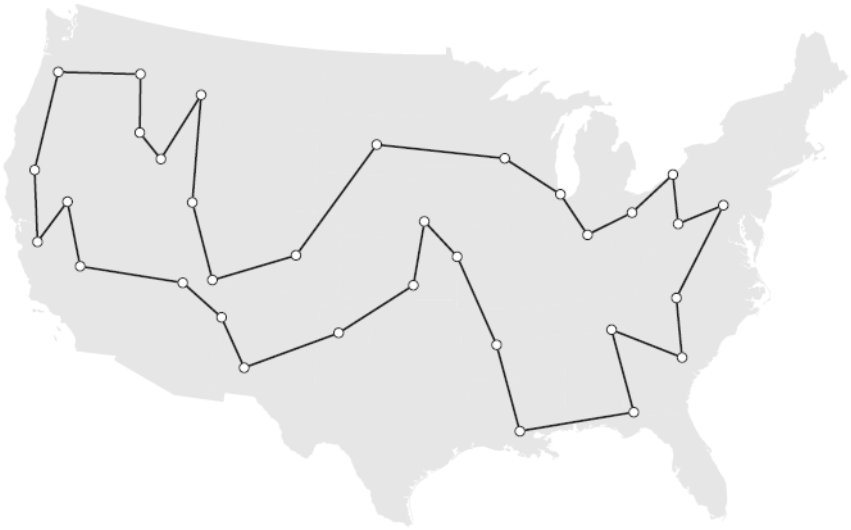
Problem trgovačkog putnika (engl. The Traveling Salesman Problem) vrlo je jednostavno formulirati. Za dani skup gradova s cijenom puta između para svakoga od njih potrebno je pronaći najjeftiniji put kojim se obilaze svi gradovi i vraća u početni grad. Pojam načina obilaska svih gradova predstavlja poredak kojim su gradovi posjećeni. Taj poredak se još naziva i tura. U teoriji grafova bi mogli reći da tražimo hamiltonovski ciklus, ciklus koji prolazi svim vrhovima zadanog grafa. Ovaj je problem jedan od najistraživanijih problema u računskoj matematici. Inspirirao je istraživanja matematičara, računarskih znanstvenika, kemičara, fizičara, psihologa i brojnih neprofesionalnih znanstvenika. Predavači koriste ovaj problem za uvod u diskretnu matematiku, dok su neka od područja primjene logika, genetika, proizvodnja, telekomunikacije i neuroznanost. Jedan od razloga velike rasprostranjenosti je jednostavnost formulacije koja ga čini idealnom platformom za razvoj ideja i tehnika kako pristupiti općenitim računarskim problemima.

Podrijetlo imena problema trgovačkog putnika je nepoznato. Ne postoji nikakva dokumentacija o osobi koja je prva upotrijebila ovo ime. U njemačkom priručniku za trgovačke putnike iz 1832. godine opisana je potreba za dobrim rutama što je zapravo eksplicitna definicija problema trgovačkog putnika. Knjiga opisuje pet ruta kroz regije Njemačke i Švicarske. Četiri od njih uključuju povratak u prijašnji grad koji je služio kao polazišna točka putovanja. Peti je zaista obilazak trgovačkog putnika koja bi mogla biti optimalna ako se uzmu u obzir uvjeti putovanja u to vrijeme. Način putovanja trgovaca se mijenjao tijekom godina, od konja i kočija do vlakova i automobila, no u svakom se od ovih slučajeva planiranje ruta uzimalo u obzir osim udaljenosti među gradovima. Smišljanje dobrih obilazaka je postala redovita praksa trgovaca na putu.

U matematičkoj literaturi, varijantu problema trgovačkog putnika je prvi spomenuo Karl Menger u bilješkama s matematičkog kolokvija koji se održao u Beču, 5.2.1930. Problem je bio naći samo putanju kroz točke, bez povratka u početnu točku. Ova se verzija problema lako pretvori u problem trgovačkog putnika dodavanjem dodatne točke udaljenosti 0 od svih početnih točaka.

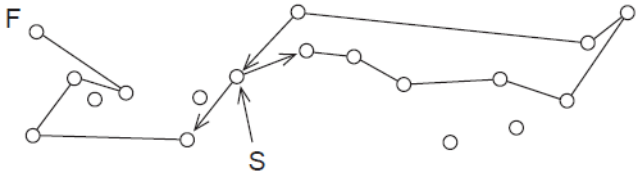
U ranim 1960-im godinama pronalaženje rute kroz Sjedinjene Države bilo je u interesu javnosti zbog reklamne kampanje s 33 grada u problemu trgovačkog putnika. Tvrtka "Protector & Gamble" ponudila je nagradu od 10 000$ za najkraći put kojim bi se obišli svi gradovi. Vozači su bili Toody i Muldoon, policijski heroji iz televizijske serije "Car 54, Where Are You?" Iako dužnost ova dva policajca nije izlazila izvan gradske četvrti Bronx New Yorka, policajci i noćni čuvari često su u svakodnevnom poslu morali raditi obilaske hodajući ili vozeći se. Velika novčana nagrada privukla je pozornost primijenjenih matematičara. Nekoliko je znanstvenih radova o nalaženju obilazaka za trgovačkog putnika citiralo ovaj problem. Pobjednik natječaja je bio Gerald Thompson (Carnegie Mellon University). U znanstvenom radu iz 1964. zajedno s R.L.Kargom opisao je optimalan obilazak koji je prikazan na slici 2.2. Thompson nije sa sigurnošću znao da je pronašao najkraći obilazak, no na kraju se ispostavilo je pronašao isto rješenje kao i nekoliko natjecatelja. Thompson je pobijedio jer je predložio najbolji slogan za jedan od proizvoda tvrtke.

Slika 2.1: Natječaj tvrtke Protector & Gamble



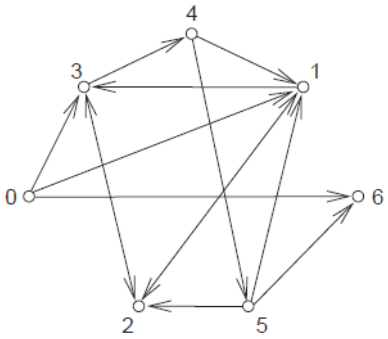
Slika 2.2: Optimalni obilazak kroz 33 grada

Tijekom godina, životinje su promatrane u načinu rješavanja problema trgovačkog putnika. Naravno, poseban izazov ovdje bio je uvjeriti životinje koje su sudjelovale u eksperimentu da traže najkraće obilaske. Središnji znanstveni rad u ovoj kategoriji napravio je Emil Menzel 1973. godine proučavajući tim čimpanzi. Svrhu eksperimenta opisao je na sljedeći način. *Ako je čimpanza u prošlosti vidjela lokacije nekoliko skrivenih objekata u području, kako će se snaći da bi došla do njih ponovno, i kako će organizirati smjer kretanja? Što će nam smjer kretanja čimpanze reći o prirodi kognitivnog mapiranja, strategiji i njezinom kriteriju učinkovitosti?* U svom radu Menzel je smislio lijep način kojim je postigao da se objekti kreću učinkovitim obilaskom. Na početku procesa, šest je čimpanza bilo zatvoreno u kavezu na rubu područja. Trener bi uzeo odabranu čimpanzu iz kaveza i nosio ju po području dok bi asistent sakrio 18 komada voća na nasumičnim lokacijama. Odabrana čimpanza potom bi bila vraćena u kavez i nakon dvije minute čekanja svih šest čimpanzi bi bilo pušteno. Tada bi odabrana čimpanza koristeći pamćenje brzo skupljala hranu prije ostalih koje bi jednostavno tražile hranu. Jedna od čimpanzi, Bido, napravila je kretanje po području prikazano slikom 2.3. Točka S predstavlja početnu točku, točka F završnu, a strelice pokazuju smjer kretanja. Čimpanza je propustila četiri komada voća, ali sveukupno, Bido je napravio značajno dobar obilazak vodeći se pamćenjem lokacija hrane.



Slika 2.3: Čimpanzin obilazak (Bido)

DNA računarstvo je jedno od područja u kojem je problem trgovačkog putnika dao značajan doprinos. Tijekom proteklog desetljeća pojavilo se intrigantno područje istraživanja s ciljem izvršavanja računanja velikih razmjera pomoću uređaja koji rade na molekularnoj razini. Velika količina podataka koja može biti pohranjena u malim količinama DNA potaknula je nagađanja znanstvenika da bi jednog dana bilo moguće koristiti molekularne uređaje za napad na razrede problema koji su izvan dosega standardnim elektroničkim računalima. Primjer koji je napravio Leonard Adleman potaknuo je razvoj ovog područja 1994. godine. Riješio je problem trgovačkog putnika sa sedam gradova koristeći DNA računanje te je njegov rad objavljen na mnogim mjestima, uključujući i *New York Times*. Varijanta problema koju je proučavao Adleman sastoji se od točaka s ograničenim brojem poveznica koje spajaju njihove parove. Cilj je bio pronaći put među poveznicama za putovanje od određene početne do određene završne točke, pritom posjećujući svaku ostalu točku na putu. Takav se put još naziva i *Hamiltonov put*. U problemu koji je proučavao Adleman nije postojala cijena pojedine poveznice između točaka. Cilj je bio samo naći Hamiltonov put, što je još uvijek općenito težak problem. Primjer korišten u DNA eksperimentu prikazan je na slici 2.4.



Slika 2.4: Problem usmjerenog Hamiltonovog puta korišten u DNA eksperimentu

Početna točka je grad s oznakom 0 a završna grad s oznakom 6. Hamiltonov put kroz točke mora ispunjavati sljedeće upute. Putovanje je dozvoljeno u bilo kojem od dva smjera na poveznicama između točaka 1 i 2 te između točaka 2 i 3, dok je putovanje na ostalim poveznicama dozvoljeno samo u jednom smjeru. Adleman je napravio molekularno kodiranje problema tako što je pridružio svaku od sedam točaka nasumičnom nizu od 20 DNA slova. Na primjer, točka 2 bila je pridružena nizu

$$TATCGGATCG|GTATATCCGA$$

a točka 3 nizu

$$GCTATTCGAG|CTTAAAGCTA$$

Razmatranjem niza od 20 znakova za točke koje su kombinacija dva niza od 10 znakova može se vidjeti da je usmjerena poveznica od točke a do točke b prikazana nizom koji sadrži drugu polovicu oznake točke a i prvu polovicu oznake točke b. Na primjer, poveznica između točaka 2 i 3 bi bila:

$$GTATATCCGA|GCTATTCGAG$$

Jedini izuzeci iz ovoga pravila su poveznice koje uključuju početni i završni grad, 0 i 6. Tada se cijeli niz od 20 slova koristi za te točke umjesto ranije opisanog načina kojim se niz dijeli na dva dijela. Za prikaz poveznica kojima je moguć put u oba smjera, napravljen je par poveznica, po jedna u svakom smjeru. U eksperimentu je mnogo kopija DNA za točke i poveznice bilo proizvedeno u laboratoriju. Sljedeći korak u Adlemanovom pristupu bio je osigurati način spajanja poveznica u putanju. U tu svrhu, za svaku točku različitu od 0 i 6 napravljen je komplementarni DNA slijed za oznaku. Takve sekvence djeluju kao "udlage" koje uparuju poveznice u odgovarajućem smjeru. Na primjer, komplementarna sekvenca za točku 3 bi mogla poslužiti za spajanje poveznica $(2, 3)$ i $(3, 4)$ budući da bi se prva polovica sekvence mogla upariti s drugom polovicom kodiranja $(2, 3)$ i druga polovica sekvence bi se mogla upariti s prvom polovicom kodiranja $(3, 4)$. Dodajemo li mnoge kopije ovih "udlaga", njihova je mješavina u mogućnosti generirati dvolančanu DNA koja odgovara putanji u problemu. Nakon pažljivog rada u laboratoriju tijekom sedam dana, dvolančana DNA koja daje Hamiltonov put je bila identificirana.

Ova metoda očito nije prikladna za probleme trgovačkog putnika s puno gradova, budući da količina potrebnih DNA komponenata raste vrlo brzo s brojem gradova (u objašnjenom primjeru su tražene sve putanje, ne samo Hamiltonova). No ovaj nevjerojatan način je bio istraživan iz mnogih smjerova za razvoj metoda prikladnih za primjenu na druge razrede problema. Kao i u drugim slučajevima, jednostavna priroda problema trgovačkog putnika omogućila mu je vodeću ulogu u kreiranju nove računske paradigme. (Applegate, 2006)

U opisu kombinatornih problema često se postavlja pitanje koliko je neki problem težak. Ovdje bismo se mogli zapitati da li je problem trgovačkog putnika zapravo teško riješiti. Odgovor na ovo pitanje ne znamo. Menger je opazio da je moguće riješiti problem trgovačkog putnika jednostavno ispitujući svaki obilazak i nakon toga odabrati najjeftiniji. Odmah je pozvao na bolje metode rješavanja, nezadovoljan tehnikom koja je konačna, ali nepraktična. Pojam metode "bolje od konačnog" rješenja diskretan je pojam i ne uzima se u obzir u uobičajenim postavkama klasične matematike. Savjeti oko ovog pitanja pojavljuju se u nekim radovima o problemu trgovačkog putnika, uključujući Floodovu izjavu iz rada 1956. *Vrlo je vjerojatno da je za uspješno rješavanje problema potreban pristup potpuno drugačiji od bilo kojeg korištenog. Zapravo, možda ne postoji općenita metoda pristupa problemu te bi i nemogući rezultati također mogli biti dragocjeni.*

Kako usporediti dvije metode rješavanja ovog problema? Mogli bi jednostavno reći da je metoda A nadmoćnija od metode B ako zahtijeva manje osnovnih koraka za rješavanje svakog slučaja problema. Ovo pravilo je vrlo jasno, no čini izravno rangiranje metoda nemogućim, budući da bi samo usko povezane metode dale tako jednostavnu usporedbu. Vidimo da je potrebno je ublažiti ovaj kriterij. U tu svrhu bi trebalo zanemariti rezultate na malim instancama problema jer ionako za njih imamo tehnike koje nam daju dobre rezultate. Nadalje, za dani broj gradova n, bilo bi se dobro usredotočiti na one gradove koji predstavljaju najviše teškoća u predloženoj metodi. Uz ovaj pristup metodu A bi rangirali ispred metode B ako je za svaku veću vrijednost od n gradova najgori primjer rješiv za manje vremena metodom A nego metodom B. Što onda možemo reći o metodama rješavanja problema trgovačkog putnika? Ovise o broju gradova, n, no vrijeme izvođenja proporcionalno s $(n-1)!$ je nepraktično ako uzmemo u obzir da već za 50 gradova nije moguće pronaći rješenje, čak ni koristeći sva računala na svijetu. Ovo se često navodi kao razlog zašto je *problem trgovačkog putnika težak za rješavanje* no to je zapravo pogrešno. Navedeni argument samo pokazuje da provjera svih putova ne dolazi u obzir, te ne isključuje mogućnost potpuno drugačijeg pristupa rješavanju problema.

Važno je teorijsko pitanje postavio Edmonds oko 1960. godine pitajući da li postoji ili ne postoji dobar algoritam za problem trgovačkog putnika. Do današnjeg dana to pitanje nije odgovoreno. Clay Mathematics Institute ponudio je nagradu od $1,000,000\$$ za pronalazak dobrog algoritma ili dokaz da takav ne postoji. Problemi za koje postoje dobri algoritmi poznati su kao *klasa* P (slovo P dolazi od polinomijalnog vremena). Općenitija klasa problema su NP problemi, koji se rješavaju u nedeterminističko polinomijalnom vremenu. Problem je NP ako je odgovor na problem "da" i da se može potvrditi u polinomijalnom vremenu. Na primjer, ako je odgovor na problem trgovačkog putnika "da" onda se to može potvrditi izlaganjem obilaska koji zaista ima cijenu manju od konstante K. Dakle, problem trgovačkog putnika je NP težak problem.

2.2. Problem naprtnjače

Svaki segment ljudskog života određen je odlukama. U 21. stoljeću profesionalno okruženje zahtjeva proces odlučivanja koji mora biti formaliziran i ocijenjen neovisno pojedincima koji su u njega uključeni. Da bi se postigao ovakav učinak, potrebno je prikazati utjecaj bilo koje odluke brojčanim vrijednostima. U najjednostavnijem slučaju ishod odluke mogao bi biti prikazan jedinstvenom vrijednošću kao što je dobitak, gubitak, trošak ili slično. Usporedba ovih vrijednosti dovodi do ukupnog poretka na skupu svih opcija koji su dostupni za donošenje odluke. Problem bi mogao nastati ako želimo pronaći najveću ili najmanju vrijednost u skupu opcija koji je prevelik i/ili eksplicitno nepoznat. Često su jedino poznati uvjeti koji nam govore koji je skup teoretski moguće odabrati. Najjednostavniji mogući oblik odluke je izbor između dvije alternative. Takva binarna odluka je formulirana u kvantitativnom modelu kao binarna varijabla $x \in \{0, 1\}$ s očitim značenjem da $x = 1$ predstavlja prvu alternativu dok $x = 0$ ukazuje na odbacivanje prve alternative i odabir druge opcije.

Mnogi praktični procesi odlučivanja mogu biti prikazani kombinacijom nekoliko binarnih odluka. To znači da se konačni problem odluke sastoji od odabira jedne ili dvije alternative za veliki broj binarnih odluka koje možda utječu jedna na drugu. U osnovnoj verziji linearnog modela odlučivanja, ishod cijelog procesa ocijenjen je kombinacijom vrijednosti pridruženoj svakoj binarnoj odluci. Za prikaz međuovisnosti odluka koje su u paru mogu se koristiti i kvadratne funkcije. Izvedivost pojedinog odabira može biti vrlo teška u praksi zato što binarne odluke mogu utjecati ili čak predstavljati kontradikciju jedna prema drugoj. Formalno govoreći, linearan model odlučivanja definiran je s n binarnih varijabli, $x_j \in \{0, 1\}$ što odgovara odabiru j-te binarne odluke i varijable dobiti p_j koja ukazuje na razliku vrijednosti postignutu odabirom prve alternative, npr. $x_j = 1$, umjesto druge alternative $x_j = 0$. Bez smanjenja općenitosti, možemo pretpostaviti da nakon prikladnog dodjeljivanja dvije opcije dvama slučajevima $x_j = 1$ i $x_j = 0$, uvijek imamo $p_j \geq 0$. Ukupna dobit pridružena određenom odabiru za svih n binarnih odluka dana je sumom vrijednosti p_j za sve odluke u kojima je prva alternativa odabrana.

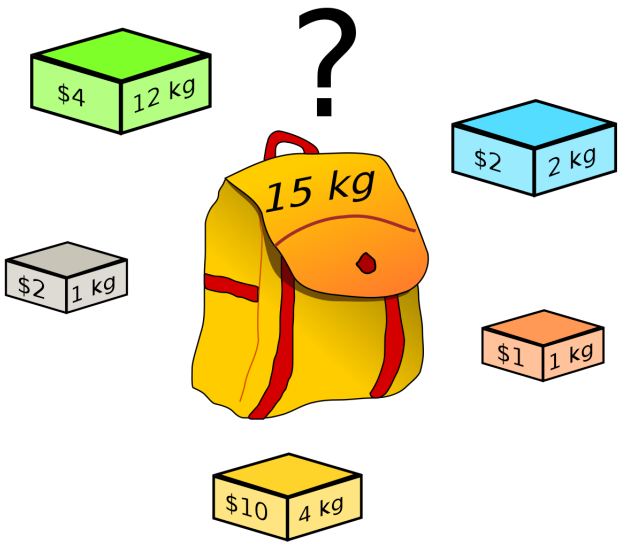
Nas zanimaju problemi odlučivanja u kojima izvedivost određenog odabira može biti ocijenjena linearnom kombinacijom koeficijenata za svaku binarnu odluku. U ovom modelu izvedivost odabira alternativa je određena ograničenjem kapaciteta na sljedeći način. U svakoj binarnoj odluci j odabir prve alternative ($x_j = 1$) zahtjeva težinu w_j dok odabir druge alternative ($x_j = 1$) ne zahtijeva. Odabir alternativa je izvediv ako suma težina svih binarnih odluka ne prelazi dani prag kapaciteta c. Ovaj se uvjet može zapisati kao

$$\sum_{j=1}^{n} w_j x_j \leq c.$$

Uzmemo li u obzir ovaj proces odlučivanja kao optimizacijski problem u kojem bi ukupan

profit trebao biti što veći mogući, došli smo do problema naprtnjače. (Kellerer, 2004)

Problem naprtnjače ima slikovitiju interpretaciju od binarnih odluka po kojoj je vjerojatno dobio i ime, iako za to službeni podatak ne postoji. Uzmimo za primjer planinara koji sprema svoju naprtnjaču za obilazak planine te treba odlučiti koje što ponijeti sa sobom. Na raspolaganju mu je veliki broj stvari koje bi mu mogle biti korisne na putu. Svaka od tih stvari označena brojem od 1 do n omogućit će mu neku korist koja je označena pozitivnim brojem p_j. Svaka stvar koju odabere ima težinu w_j koja otežava teret koji će nositi. Planinar želi ograničiti ukupnu težinu naprtnjače najvećim mogućim opterećenjem, kapacitetom c.



Slika 2.5: Problem naprtnjače

Nakon ove slikovite interpretacije, problem naprtnjače možemo definirati i formalno. Američki matematičar Tobias Dantzing ovo je ime koristio u svojim ranim radovima pa bi podrijetlo moglo biti vrsta usmene predaje. Dana nam je instanca problema naprtnjače sa skupom N koji sadrži n predmeta j s dobiti p_j, težinom w_j i kapacitetom c (obično su sve ove vrijednosti pozitivni brojevi). Cilj je odabrati podskup od N takav da ukupna dobit bude što veća moguća i da ukupna težina ne bude veća od c.

Problem naprtnjače može biti formuliran kao rješenje sljedećeg problema cjelobrojnog programiranja:

$$\begin{aligned} \text{minimiziraj} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{uz} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j \in \{0,1\}, \quad j = 1, \ldots, n \end{aligned}$$

Kao oznaka za optimalni vektor rješenja koristi se $x^* = (x_1^*, \ldots, x_n^*)$, a za optimalnu vrijednost rješenja z^*. X^* je optimalni skup, npr. skup predmeta koji odgovaraju optimalnom vektoru.

Problem naprtnjače je najjednostavniji netrivijalni model cjelobrojnog programiranja s binarnim varijablama te jednim ograničenjem. Koeficijenti su također cjelobrojni. Unatoč tome, dodavanje uvjeta $x_j \in \{0, 1\}, j = 1, \ldots, n$ jednostavnom linearnom programu, problem naprtnjače svrstava u kategoriju teških problema.

Tijekom stoljeća ovaj je problem bio proučavan kao najjednostavnija verzija problema u kojem se tražila najveća vrijednost. U prvim definicijama problema ograničenja su navođena neovisno jedno drugom. Mathews je $1897.$ godine pokazao kako nekoliko ograničenja može biti spojeno u jedno ograničenje. Time je na neki način napravio prototip redukcije općeg problema cjelobrojnog programiranja na problem naprtnjače i tako dokazao da je problem naprtnjače težak barem kao problem cjelobrojnog programiranja.

Iako se formulacija problema uz pomoć planinara koji sprema stvari u naprtnjaču možda čini neprimjenjivom na svakodnevni život (u situacijama drugačijim od pakiranja stvari), zapravo nije tako. Opisani problem maksimizacije mogli bismo protumačiti kao problem ulaganja. Zamislimo da bogati ulagač ima određenu količinu novca c koju želi uložiti u isplative poslovne projekte. Kao temelj svoje odluke, ulagač je napravio dugu listu projekata u koje bi mogao ulagati. Za svaki je projekt navedena potrebna količina novca w_j i očekivana dobit (povratna vrijednost) p_j nakon određenog vremenskog perioda uz pretpostavku da rizika nema. Očito je da ovom problemu možemo pristupiti kao problemu naprtnjače.

Još jedna često korištena interpretacija problema naprtnjače je u zrakoplovnom prijevozu tereta. Otpravljač zračnog prometa mora odlučiti koje zahtjeve prijevoza korisnika mora ispuniti, tj. kako napuniti određeni avion teretom. Odluka se temelji na popisu zahtjeva koji sadrže težinu pojedinog tereta w_j te stopu po težini naplaćenu za svaki teret. Ova stopa ne mora biti fiksna nego ovisi o ugovoru s pojedinim kupcem. Stoga profit p_j avionske kompanije koji ostvari preuzimanjem zahtjeva i stavljanjem odgovarajućeg tereta u zrakoplov nije direktno proporcionalan težini paketa. Svaki avion ima kapacitet c koji ne smije biti premašen ukupnom težinom tereta. Ponovno vidimo da je ovaj logistički problem analogan pakiranju stvari u naprtnjaču.

Dobro je primijetiti kako problem naprtnjače nije samo problem pakiranja, nego i problem smanjivanja. Zamislimo da u pilani treba izrezati trupac na sitnije dijelove. Dijelovi moraju biti izrezani na unaprijed određene duljine w_j, pri čemu je svaka duljina pridružena prodajnoj cijeni p_j. U svrhu maksimizacije profita, pilana može formulirati problem kao problem naprtnjače u kojem duljina trupca predstavlja kapacitet c.

Problem naprtnjače može biti prisutan i u akademskom životu. Postupak ispitivanja na fakultetu u Norfolku (Connecticut) temelji se upravo na ovom problemu. Studentima je dan skup od n pitanja uz navedeni najveći broj bodova koji mogu dobiti na pojedinom pitanju, npr. 125 bodova. Zadatke boduje nastavnik te dodjeljuje konačnu ocjenu skaliranu na npr. 100 bodova takvu da je odabrani podskup pitanja presudan. Idealno bi bilo kad bi se podskup pitanja odabirao automatski tako da studenti mogu osvojiti što je više moguće bodova. Ovdje također vidimo ekvivalenciju problemu naprtnjače sa j-tim predmetom za j-to pitanje gdje w_j predstavlja maksimalne bodove a p_j osvojene bodove. Kapacitet c je ograničenje preko kojeg ne smije prijeći ukupan zbroj bodova odabranih pitanja.

Kao što smo mogli vidjeti kroz prethodne primjere, problem naprtnjače ima puno različnih proširenja i varijacija. Neka su od tih proširenja, koja ubrajamo u generalnije varijante problema naprtnjače, postala problemi za sebe. Osim problema u kojima se eksplicitno naglašava korištenje problema naprtnjače, brojne metode za rješavanje kompleksnijih problema također koriste ovaj postupak (često i iterativno) iako to možda nije očito.

Kada govorimo o kombinatornoj optimizaciji i operativnim istraživanjima, pristupi problemima mogu se razvrstati u dvije skupine. "Top-down" pristup podrazumijeva pronalaženje metoda za najteže probleme kao što je već spomenuti problem trgovačkog putnika ili problem izrade rasporeda. Ako pronađena metoda pokazuje dobre rezultate, možemo pretpostaviti da će biti uspješna i pri rješavanju raznolikih problema. Suprotan pristup je razvoj metoda za najjednostavniji problem, kao što je problem naprtnjače te pokušaj generaliziranja na složenije probleme. Oba pristupa zahtijevaju daljnji razvoj metoda te opravdavaju istraživački napor za rješavanje relativno jednostavnih problema.

Logično je zapitati se kako bi mogli proširiti problem naprtnjače. Vjerojatno bi ubrzo došli do točnog odgovora. Željeli bismo riješiti problem ne samo za jedan dani kapacitet c nego i za sve kapacitete c do zadane gornje granice c_{max}. Takva se formulacija problema naziva *problem naprtnjače svih kapaciteta*. U nekim problemima s planiranjem točan kapacitet nije unaprijed poznat ali se može saznati na temelju predloženih rješenja. Problem naprtnjače svih kapaciteta možemo ilustrirati prethodno spomenutim primjerom prijevoza tereta zrakoplovom. Što više tereta zrakoplov nosi, potrebna je veća količina goriva. Količinu goriva također možemo računati i u ukupnoj težini tereta. Ovisnost goriva o teretu nije linearna funkcija, potrebno je riješiti problem naprtnjače svih kapaciteta za sve kapacitete do danog gornjeg ograničenja težine tereta. Za svaki kapacitet računamo potrebu za gorivom i oduzimamo odgovarajuću cijenu goriva od dobiti. Optimalno je rješenje najveća vrijednost konačnih dobiti. Prirodna metoda za rješavanje problema naprtnjače svih kapaciteta bila bi jednostavno rješavanje problema naprtnjače za svih c_{max} kapaciteta. No, postoje algoritmi koji mogu ubrzati izvođenje odbacujući računanje za kapacitete koji su manje bitni ili nebitni.

Glavni je cilj proučavanja problema naprtnjače razvoj metoda rješavanja (algoritama) koji daju optimalno ili aproksimativno rješenje za svaki primjer ovog problema. Očito je da svi algoritmi koji rezultiraju optimalnim rješenjem nisu jednaki po performansama. Možemo očekivati da će algoritmi koji ne pronalaze optimalno nego samo približno rješenje imati bolje vrijeme izvođenja. U ovom kontekstu performanse nekog algoritma definirane su vremenom izvođenja i količinom računalne memorije (prostora). Još jedan bitan faktor u ocjenjivanju algoritma bi bila razina težine algoritma jer su jednostavne metode preferirane u odnosu na složenije koje je zahtjevnije implementirati.

Slično kao i u problemu trgovačkog putnika, i ovdje bismo željeli imati neku mjeru kojom ćemo procijeniti da li je neki algoritam brži ili sporiji. To možemo napraviti na tri različita načina. Prvo, testirati algoritam na različitim primjerima problema. Pri tome treba paziti u kakvom s okruženju odvija testiranje. Različit hardver i softver bitni su čimbenici koji bi mogli dovesti do krivih zaključaka prilikom usporedbe algoritama. Drugi način bi bio istražiti prosječno vrijeme izvođenja algoritma. Pronalazak odgovarajući vjerojatnosnog modela koji odražava pojavu u stvarnom svijetu prilično je zahtjevan zadatak. Uz to treba voditi brigu o činjenici da se samo jednostavniji vjerojatnosni modeli mogu analizirati jer su napredniji vjerojatnosni modeli nemoćni kad su u pitanju složeni algoritmi. Veliki nedostatak ovog pristupa je potreba za velikim brojem primjera da bi pristup bio učinkovit. Treća i najčešća metoda za mjerenje performansi algoritama je analiza trajanja najgoreg slučaja. To znači da ćemo postaviti gornju granicu za broj osnovnih aritmetičkih operacija koje su potrebne da bi se riješio svaki problem (određene veličine) iz skupa problema. Ta granica bi trebala biti funkcija veličine primjera problema koja ovisi o broju ulaznih vrijednosti te njihovoj dimenziji. Pretpostavimo da trebamo izvesti određeni skup operacija za svaki predmet ili za svaku cjelobrojnu vrijednost od jedan do najvećeg mogućeg kapaciteta c. Za većinu vremenskih granica dovoljno je kao parametre uzeti u obzir broj predmeta n, kapacitet c i najveću dobit ili težinu p_{max} ili w_{max}.

Brojanje točnog broja nužnih aritmetičkih operacija u algoritmu je skoro nemoguće te također ovisi o implementacijskim detaljima. U svrhu usporedbe različitih algoritama, najviše nas zanima asimptotska gornja granica kako bi prikazali porast reda veličine vremena izvođenja. Konstantni su faktori zanemareni što doprinosi boljoj generalizaciji jer implementacijski "trikovi" i specifikacije samog uređaja na kojemu se analiza izvodi ne pridonose rezultatu u ovakvoj reprezentaciji. Analogan postupak odnosi se na potrebnu količinu računalne memorije.

Asimptotska vremena izvođenja objašnjena su takozvanom O-notacijom. Neformalno, možemo reći da je svaki polinom u n s najvećim eksponentom k $O(n^k)$.

To znači da možemo zanemariti sve što je kraj eksponenata manjih od k te također konstantni koeficijent uz n^k. Formalnije, za danu funkciju $f : \mathbb{R} \to \mathbb{R}$ izraz $O(f)$ predstavlja familiju svih funkcija

$$O(f) := \{g(x) \mid \exists c, x_0 > 0 \text{ takav da } 0 \leq g(x) \leq cf(x)\ \forall x \geq x_0\}.$$

Pretpostavimo da je funkcija $f(x) := x^2$. Tada su sve ove funkcije uključene u $O(f)$:

$$10x,\ xlogx,\ 0.1x^2,\ 1000x^2,\ x^{1.5}.$$

Međutim, $x^{2+\varepsilon}$ nije u $O(f)$ za bilo koji $\varepsilon > 0$. Restriktivnija od navedene je Θ notacija koja asimptotski ograničava funkciju odozdo i odozgo. Formalno, s $\Theta(f)$ označavamo familiju svih funkcija

$$\Theta(f) := \{g(x) \mid \exists c_1, c_2, x_0 > 0 \text{ takav da } 0 \leq c_1 f(x) \leq g(x) \leq c_2 f(x)\ \forall x \geq x_0\}.$$

Ovisno o granicama njihovog asimptotskog vremena izvođenja, algoritmi mogu biti podijeljeni u tri skupine. Najefikasniji su algoritmi oni čije je vrijeme izvođenja ograničeno polinomom, npr. $O(n)$, $O(nlogn)$, $O(n^3)$ ili $O(n^k)$ za konstantu k. Takvi se algoritmi još zovu i algoritmi s polinomijalnim vremenom izvođenja. Pseudopolinomijalni algoritmi, u kojima je vrijeme izvođenja ograničeno asimptotski polinomom u n-tog stupnja i jednom ili više vrijednosti, npr. $O(nc)$ ili $O(n^2 p_{max})$, manje su atraktivni jer bi čak i jednostavan problem s malim brojem predmeta mogao imati dosta velike koeficijente a time i dugo vrijeme izvođenja. Treća i najneugodnija skupina su nepolinomijalni algoritmi, algoritmi čije se vrijeme izvođenja ne može ograničiti polinomijalnom nego npr. samo eksponencijalnom funkcijom u n, kao što je $O(2^n)$ ili $O(3^n)$. Ova se neugodnost može ilustrirati usporedbom npr. $O(n^3)$ i $O(2^n)$. Ako je n povećan na $2n$ prvi izraz raste za faktor 8 dok je drugi izraz kvadriran.

Za problem naprtnjače i njegove generalizacije poznati su nam pseudopolinomijalni algoritmi iako bi preferirali polinomijalne algoritme za ove probleme. Međutim, postoji jak dokaz da za problem naprtnjače i njegove generalizacije ne postoji algoritam koji bi pronašao optimalno rješenje u polinomijalnom vremenu. Svi ovi problemi pripadaju u već ranije spomenutu klasu NP-teških optimizacijskih problema. Vjeruje se da ne postoji polinomijalni algoritam koji bi optimalno riješio NP-težak problem jer su svi NP-teški problemi jednaki u smislu da ako je ijedan NP-težak problem rješiv u polinomijalnom vremenu, navedeno se može primijeniti i na sve ostale NP-teške probleme.

2.3. Problem usmjeravanja vozila

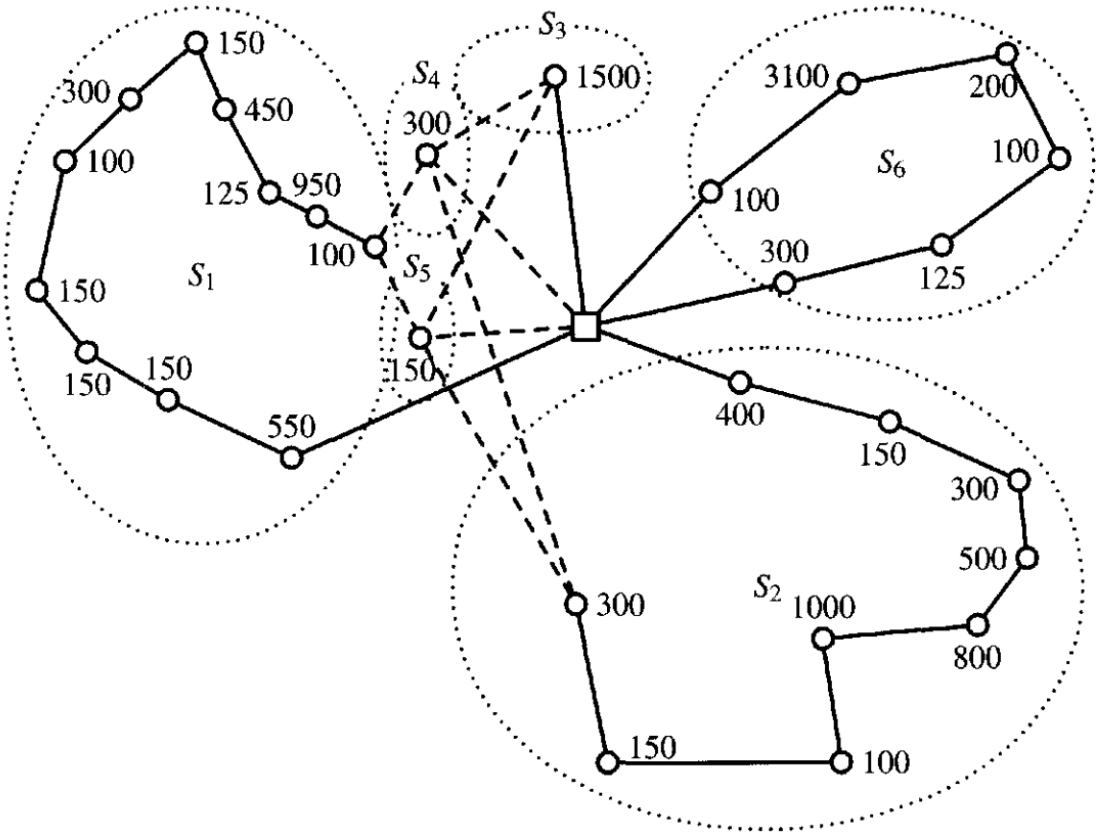
Zadnjih smo desetljeća vidjeli sve veće korištenje optimizacije za učinkovitiju opskrbu robom i uslugama u distribucijskim sustavima. Upotreba računalnih postupaka u distribucijskom procesu dovela je do znatnih ušteda (od 5% do 20%) u globalnim cijenama prijevoza. Jednostavno je vidjeti da je utjecaj ovih ušteda na globalnu ekonomiju ogroman. Proces prijevoza uključuje sve faze proizvodnje i distribucije te predstavlja relevantnu komponentu (od 10% do 20%) konačne cijene proizvoda. Drugačiji faktor uspjeha koji je važan kao i ostali, je razvoj modeliranja i algoritamskih alata implementiranih proteklih godina. Predloženi modeli uzimaju u obzir sve karakteristike distribucije problema koje nastaju u stvarnim primjenama a odgovarajući algoritmi i računalne implementacije pronalaze dobra rješenja u prihvatljivim vremenima računanja.

Problemi oko distribucije dobara između skladišta i korisnika općenito su poznati kao *problemi usmjeravanja vozila*. Modeli i algoritmi predloženi za rješavanje ovih problema mogu biti efektivno iskorišteni ne samo za rješavanje problema usmjeravanja vozila (dostava i skupljanje dobara) nego i za druge probleme koji nastaju u transportu. Neke primjene ovog tipa su prikupljanje otpada, čišćenje ulica, usmjeravanje školskih autobusa, usmjeravanje prodavača i jedinica za održavanje. (Toth, 2002)

Da bi distribucija robe bila uspješna, u određenom vremenskom razdoblju se nizu kupaca treba isporučiti usluga skupinom vozila upravljanih vozačima koji koriste odgovarajuću cestovnu mrežu. Rješenje problema usmjeravanja vozila zahtijeva određivanje skupa ruta, od kojih je svaka izvedena jednim vozilom koje počinje i završava na vlastitom skladištu, tako da su svi zahtjevi kupaca ispunjeni uz zadovoljenje svih operativnih ograničenja te minimalne globalne troškove prijevoza.

Cestovna mreža, korištena za prijevoz dobara, općenito je opisana grafom, čiji lûkovi predstavljaju ceste a vrhovi odgovaraju cestovnim čvorovima do skladišta i lokacija korisnika. Lûkovi (i posljedično odgovarajući grafovi) mogu biti usmjereni ili neusmjereni, ovisno o tome mogu li biti prijeđeni u samo jednom smjeru (npr. jednosmjerne ulice) ili u oba smjera. Svakom je lûku pridružena cijena koja općenito predstavlja duljinu te vrijeme putovanja koje je ovisno o vrsti vozila i dobu godine u kojem je putovanje obavljeno. Tipične karakteristike korisnika:

- vrh na grafu u kojem je korisnik smješten
- količina zahtijevanih dobara, moguće različitih vrsta, koja moraju biti dostavljena ili pokupljena od korisnika
- vremenski periodi u danu tijekom kojih je korisnik dostupan
- vrijeme potrebno za dostavu ili preuzimanje dobara na lokaciji korisnika
- podskup vozila koja se mogu koristiti za pristup korisniku (npr. zbog velikog tereta)



Slika 2.6: Problem usmjeravanja vozila

Ponekad nije moguće u potpunosti ispuniti zahtjeve svakog kupca. U takvim slučajevima, određena količina robe može ostati neisporučena i nepreuzeta ili podskup korisnika može ostati bez usluge. Za takve situacije korisnicima se mogu dodijeliti različiti prioriteti ili kazne za djelomičnost ili nedostatak usluge. Rute izvedene za opskrbu kupaca počinju i završavaju na jednom ili više skladišta u vrhovima grafa puta. Svako skladište karakterizira broj i vrsta vozila povezana s njim te globalnom količinom robe s kojom može raditi. Najčešće su korisnici a priori podijeljeni između skladišta te se vozila trebaju vratiti do njihovih matičnih skladišta na kraju svakog obilaska. U ovim slučajevima, ukupni problem usmjeravanja vozila može se razložiti na nekoliko neovisnih problema od kojih je svaki pridružen određenom skladištu. Prijevoz dobara se izvodi flotom vozila čija se kompozicija i veličina može biti fiksna ili se prilagoditi zahtjevima kupaca. Tipične karakteristike vozila su:

- matično skladište i vjerojatnost da završi uslugu u skladištu koje nije matično
- kapacitet vozila izražen kao maksimalna težina, volumen ili broj paleta koje vozilo može voziti
- moguća podjela vozila na odjeljke u kojem je svako vozilo kategorizirano prema kapacitetu i tipu tereta koji može nositi
- uređaji za utovar i istovar

- podskup lûkova na grafu koji mogu biti prijeđeni vozilom
- troškovi nastali korištenjem vozila

Vozači moraju ispuniti nekoliko ograničenja postavljenih sindikatnim ugovorima i propisima o poduzećima (npr. radna razdoblja tijekom dana, broj i trajanja pauza tijekom službe, maksimalno trajanje vremena vožnje, prekovremeni rad). Ograničenja koja su nametnuta vozačima dio su onih povezanih s odgovarajućim vozilima. Primjeri nekih operativnih ograničenja: na svakoj ruti trenutno opterećenje pripadajućeg vozila ne može premašiti kapacitet vozila; kupci na ruti mogu zahtijevati samo dostavu i/ili slanje robe te mogu biti posluženi samo u dodijeljeno vrijeme. Ako je više kupaca na istoj ruti, oni će imati prednost pred ostalima zbog uštede vremena.

Procjena globalnih troškova ruta i provjera operativnih ograničenja na njima zahtjeva znanje o troškovima i vremenu putovanja između svakog para korisnika i između skladišta i korisnika. U tu je svrhu, graf (koji je često rijedak) transformiran u potpuni graf, čiji vrhovi odgovaraju korisnicima i skladištima. Za svaki par vrhova i i j u potpunom grafu, lûk (i, j) je definiran cijenom, c_{ij}, najkraćeg puta između vrhova i i j. Vrijeme putovanja t_{ij} pridruženo svakom lûku (i, j) izračunava se kao suma vremena putovanja lûkovima koji pripadaju najkraćem putu između i i j. Umjesto originalnog grafa cesta, radimo s pridruženim potpunim grafom, koji može biti usmjeren ili neusmjeren, ovisno o odgovarajućoj cijeni i matricama vremena putovanja. U prvom slučaju će biti asimetričan, a u drugom simetričan.

U problemu usmjeravanja vozila postavlja se nekoliko ciljeva koji trebaju biti uzeti u obzir. Tipični ciljevi su:

- minimizacija cijena globalnog prijevoza, ovisnog o globalnoj udaljenosti (ili vremenu potrošenom na putovanje) i fiksnim cijenama pridruženim odabranim vozilima
- minimizacija broja vozila potrebnih da bi se poslužili svi kupci
- balansiranje ruta kako bi se postiglo što kraće vrijeme putovanja i optimalna popunjenost vozila
- minimizacija kazni povezanih s djelomičnom uslugom kupaca

ili bilo koja kombinacija ovih ciljeva u kojima težina (zapremnina) igra glavnu ulogu.

Neke primjene podrazumijevaju da svako vozilo može odraditi više ruta u određenom vremenskom razmaku, ili da rute mogu biti duljine da zahtijevaju više od jednog dana da se obiđu. Dodatno, ponekad je potrebno uzeti u obzir stohastičke oblike problema, npr. problemi u kojima je prije početka obilaska dostupno samo djelomično znanje o zahtjevima korisnika ili cijenama pridruženo lûkovima cestovne mreže.

Osnovna verzija problema usmjeravanja vozila je *kapacitivni problem usmjeravanja vozila*. U ovoj verziji problema svi kupci odgovaraju na isporuke te su svi zahtjevi deterministički, unaprijed poznati i ne mogu se podijeliti. Sva su vozila jednaka i smještena u jednom centralnom skladištu te su samo nametnuta ograničenja za kapacitet. Cilj je minimizirati ukupni trošak (npr. težinske funkcije broja ruta i njihovih duljina ili vremena putovanja) da bi se poslužili svi korisnici.

Kapacitivni bi problem usmjeravanja vozila mogao biti opisan kao sljedeći problem iz teorije grafova. Neka je $G = (V, A)$ potpun graf, u kojem je $V = \{0, \ldots, n\}$ skup vrhova i A skup lûkova. Vrhovi $i = 1, ..., n$ odgovaraju korisnicima, dok vrh 0 odgovara skladištu. Ponekad je skladištu pridružen vrh $n+1$. Nenegativna cijena, c_{ij}, dodijeljena je svakom lûku $(i, j) \in A$ koji predstavlja cijenu puta potrošenu od vrha i do vrha j. Općenito, korištenje petlji, (i, i), nije dozvoljeno te je taj uvjet nametnut definiranjem $c_{ii} = +\infty$ za sve $i \in V$. Ako je G usmjeren graf, matrica cijene c je asimetrična te se odgovarajući problem naziva *asimetrični kapacitivni problem usmjeravanja vozila*. Inače, imamo $c_{ij} = c_{ji}$ za sve $(i, j) \in A$. Tada problem nazivamo *simetrični kapacitivni problem usmjeravanja vozila*. Skup lûkova A je općenito zamijenjen skupom neusmjerenih bridova, E. Za dani brid $e \in E$, neka $\alpha(e)$ i $\beta(e)$ označavaju njegove krajnje vrhove. Skup vrhova neusmjerenog grafa G označavamo s A kada su bridovi definirani njihovim krajnjim točkama $(i, j), i, j \in V$ a s E kad su definirani duljinom brida e.

Graf G mora biti čvrsto povezan te se općenito pretpostavlja da je potpun. Za dani vrh i, neka $\Delta^+(i)$ označava *prednju zvijezdu* od i, definiranu kao skup vrhova j takav da lûk $(i, j) \in A$, tj. vrhovi su direktno dostupni iz i. Analogno, neka $\Delta^-(i)$ označava *stražnju zvijezdu* vrha i, definiranu kao skup vrhova j takvih da je lûk $(j, i) \in A$ tj. vrhovi iz kojih je direktno dostupan i. Neka je dan skup $S \subseteq V$ te neka $\delta(S)$ i $E(S)$ označavaju skup vrhova $e \in E$ koji imaju samo jednu ili obje krajnje točke u S. Kao i obično, kad je jedan vrh $i \in V$ uzet u obzir, pišemo $\delta(i)$ radije nego $\delta(\{i\})$. Iz više praktičnih razloga, matrica cijene ispunjava nejednakost trokuta.

$$c_{ik} + c_{kj} \geq c_{ij} \text{ za sve } i, j, k \in V$$

Drugim riječima, nije zgodno odstupati od direktne poveznice između dva vrha i i j. Prisutnost nejednakosti trokuta je ponekad potrebna algoritmima za kapacitivni problem usmjeravanja vozila, te se može dobiti dodavanjem prikladno velike pozitivne cijene M svakome lûku. Ovakvom promjena mogu se iskriviti donja i gornja granica u odnosu na one koje odgovaraju izvornim troškovima. Dobro je primijetiti da kada je cijena svakoga lûka na grafu jednaka cijeni najkraćeg puta između njegovih krajnjih točaka, odgovarajuća matrica cijene ispunjava nejednakost trokuta.

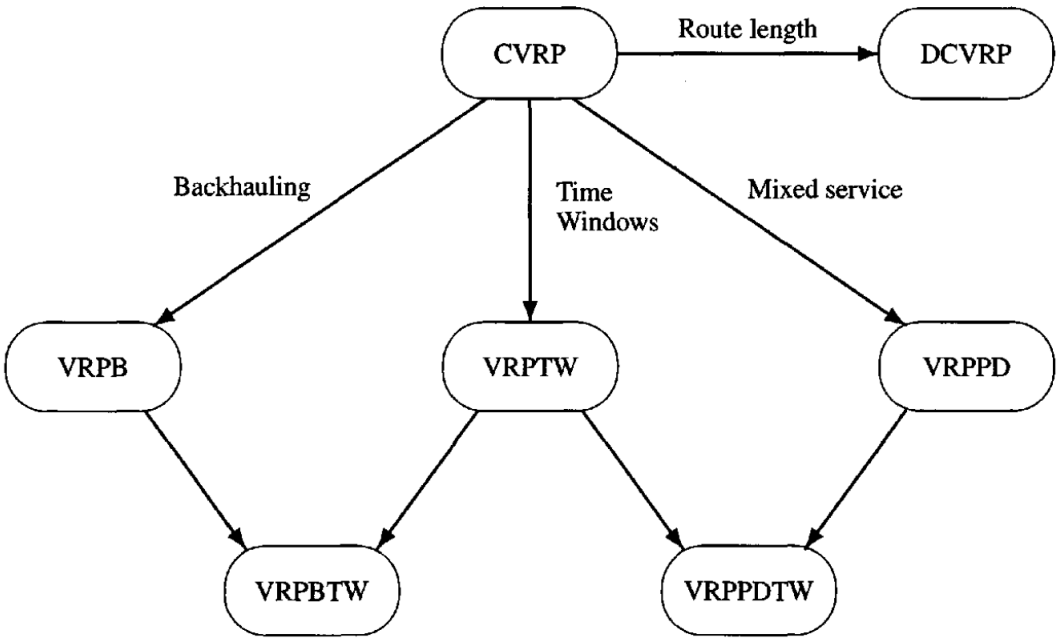
U nekim instancama problema, vrhovi su pridruženi točkama ravnine s koordinatama, te se cijena c_{ij} za svaki lûk $(i, j) \in A$ definira kao euklidska između dvije točke koje odgovaraju vrhovima i i j. U ovom slučaju matrica cijene je simetrična te ispunjava nejednakost trokuta te ovaj problem nazivamo *Euklidski simetrični kapacitivni problem usmjeravanja vozila* (engl. Euclidean Symmetric Capacitated Vehicle Routing Problem, SCVRP).

Nekoliko se verzija kapacitivnog problema usmjeravanja vozila razmatra u literaturi. U slučaju da je broj dostupnih vozila K veći od K_{min}, moguće je ostaviti neka vozila nekorištena te odrediti najviše K ruta. U ovom slučaju, fiksne cijene su često pridružene uporabi vozila. Dodatni cilj je minimizirati broj ruta. Još se jedna varijanta problema pojavljuje kad su dostupna vozila drugačija, npr. imaju različit kapacitet, $C_k, k = 1, ..., K$. Konačno, obilasci koje imaju samo jednog korisnika najčešće nisu dopuštene.

Kapacitivni problem usmjeravanja vozila pripada u skupinu NP-teških problema i predstavlja generalizaciju ranije obrađenog problema trgovačkog putnika, pozivajući se na utvrđivanje minimalne cijene jednostavnog obilaska svih vrhova od G (hamiltonski obilazak) a koje nastaju kad je $C \geq d(V)$ i $k = 1$. Stoga su sva uklanjanja ograničenja za problem trgovačkog putnika vrijede za problem usmjeravanja vozila.

Postoje razne varijante problema usmjeravanja vozila koje su sve na neki način povezane s kapacitivnim problemom usmjeravanja vozila.

- *Distance-Constrained VRP* (DCVRP) varijanta je u kojoj je za svaku rutu ograničenje kapaciteta zamijenjeno ograničenjem maksimalne duljine rute ili maksimalnog vremena potrebnog da se ista prijeđe
- dodijelimo li svakom korisniku i vremenski prozor (interval $[a_i, b_i]$) dobili smo VRPTW (*VRP with Time Windows*); također je dodijeljeno vrijeme polaska vozila te vrijeme putovanja za svaki lûk
- u varijanti VRPB (*VRP with Backhauls*) skup korisnika podijeljen je na dva podskupa; korisnici koji očekuju pošiljke te korisnici koji šalju pošiljke. Dodan je uvjet prema kojem, u slučaju da ruta sadrži obje vrste korisnika, korisnici koji preuzimaju pošiljke moraju biti usluženi prije onih koji šalju pošiljke
- u osnovnoj verziji problema VRPPD (*VRP with Pickup and Delivery*), svakom su korisniku i dodijeljene količine d_i i p_i koje predstavljaju homogenu robu koja se dostavlja i preuzima od korisnika
- VRPBTW (*VRP with Backhauls and Time Windows*) i VRPPDTW (*VRP with Pickup and Deliveries and Time Windows*) kombinacije su već navedenih varijanti problema

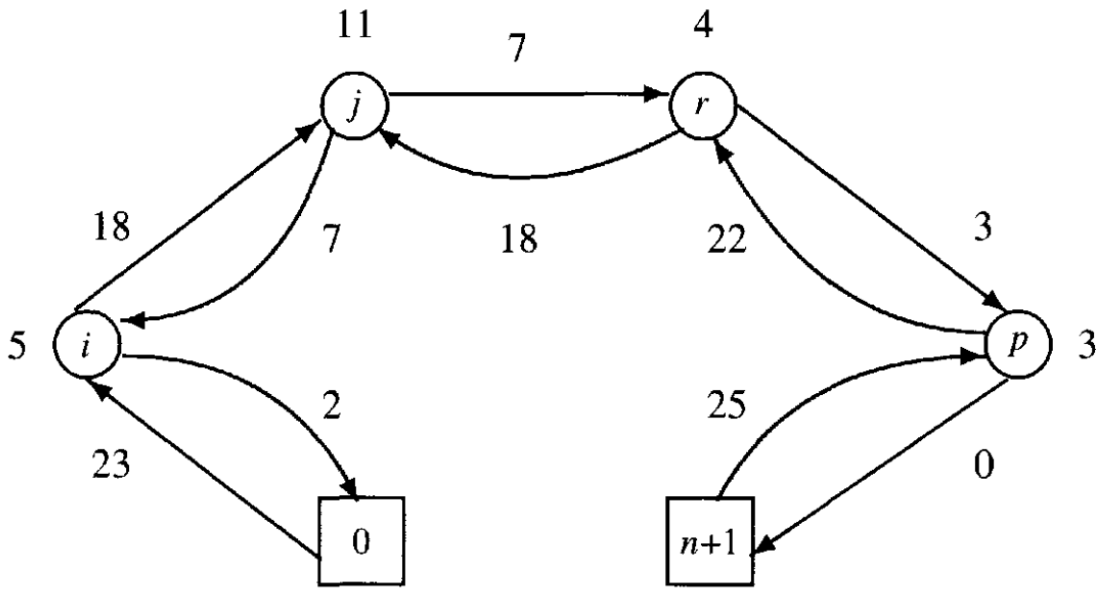


Slika 2.7: Osnovni problemi usmjeravanja vozila i njihova povezanost

Još je jedan zanimljiv model povezan s problemom usmjeravanja vozila. Radi se o modelu *toka robe* koji razvijen za dostavu ulja te kasnije proširen u varijante problema trgovačkog putnika i problema usmjeravanja vozila. U ovim se formulacijama zahtjeva korištenje binarne varijable za svaki lûk koja označava putuje li vozilo optimalnom rutom te skup kontinuiranih varijabli koje predstavljaju količine potražnje duž svakoga od lûkova. Model toka robe zahtjeva orijentaciju lûkova, te je potrebno definirati ekvivalentni usmjereni graf.

Formulacija zahtjeva prošireni graf $G' = (V', A')$ dobiven od G dodavanjem vrha $n+1$, koji je kopija skladišta. Rute su sada staze od vrha 0 do vrha $n+1$. Dvije nenegativne varijable, y_{ij} i y_{ji}, pridružene su svakom lûku $(i,j) \in A'$. Ako vozilo putuje od i do j, tada y_{ij} i y_{ji} označavaju opterećenje vozila i preostali kapacitet vozila, tj. $y_{ji} = C - y_{ij}$. Uloge su zamijenjene ako vozilo putuje od j prema i. Stoga jednadžba $y_{ij} + y_{ji} = C$ vrijedi za svaki lûk $(i,j) \in A'$.

Za svaku rutu u izvedivom rješenju, varijable toka definiraju dva usmjerena puta, jedan od vrha 0 do $n+1$, čije varijable predstavljaju teret vozila, te drugi, od vrha $n+1$ do 0, čije varijable predstavljaju preostali kapacitet vozila. Drugim riječima, na ovako opisan problem mogli bismo gledati na sljedeći način. Neka jedno vozilo ide iz 0 do $n+1$, napuštajući vrh 0 sa tek toliko proizvoda da dostavi svakom kupcu onoliko koliko treba te da dođe prazno u vrh $n+1$. Neka drugo vozilo kreće iz $n+1$ prazno te od svakog korisnika prikuplja količinu robe prema njegovom zahtjevu. Primjer s četiri klijenta i kapacitetom $C = 25$ prikazan je na slici 2.8. Zahtjevi su prikazani kraj svakog vrha.



Slika 2.8: Primjer toka robe na ruti (C = 25)

Zbog ograničenja protoka razlika između sume varijabli protoka robe povezane s lûkovima koji ulaze i izlaze iz svakog vrha i jednaka je dvostrukom zahtjevu od i. Također, postoje i ograničenja koja određuju ispravne vrijednosti za varijable protoka u vrhovima koji predstavljaju skladišta. Konačno, zadnji skup ograničenja koja je potrebno dodati za potpuni problem nameću odnos između toka vozila i varijabli protoka robe te stupnja vrhova (čvorova).

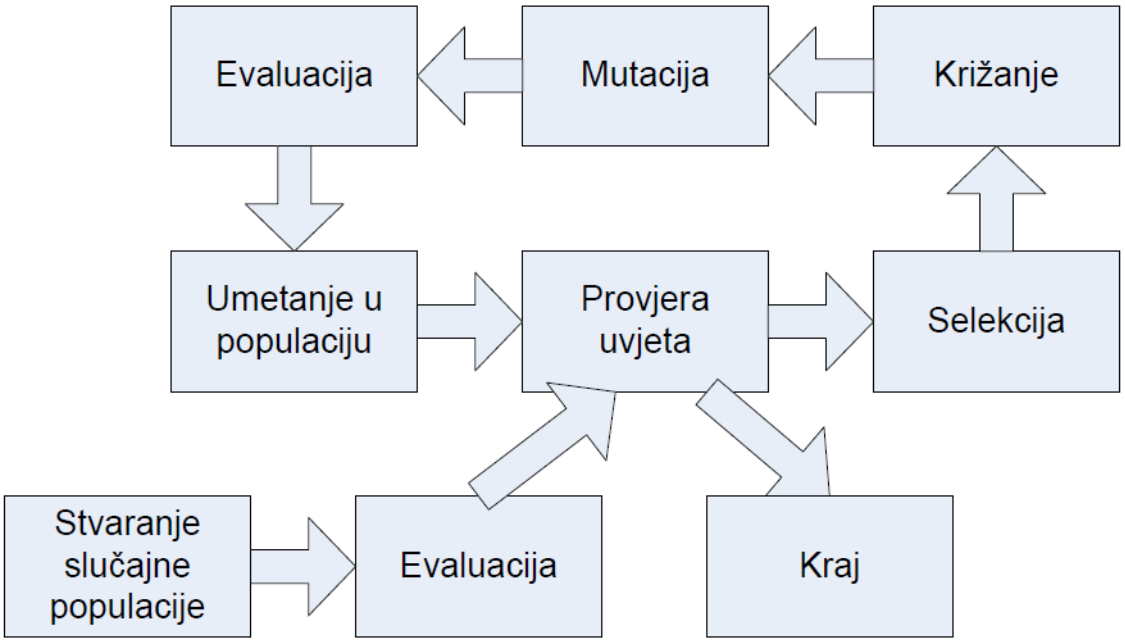
3. Prirodom inspirirani optimizacijski algoritmi

3.1. Genetski algoritmi

Genetski algoritam, inspiriran Darwinovom teorijom o postanku vrsta iz 1859. godine, jedan je od najpoznatijih prirodom inspiriranih optimizacijskih algoritama. Temelji se na činjenici da u svakoj populaciji postoji borba za preživljavanje. Ako potomaka ima više nego što je potrebno a količina hrane je ograničena, neke jedinke neće preživjeti. Jedinke u populaciji koje su bolje imat će veću šansu preživjeti te imati potomke s genetskim materijalom koji je određen njihovim, ali ne u potpunosti isti.

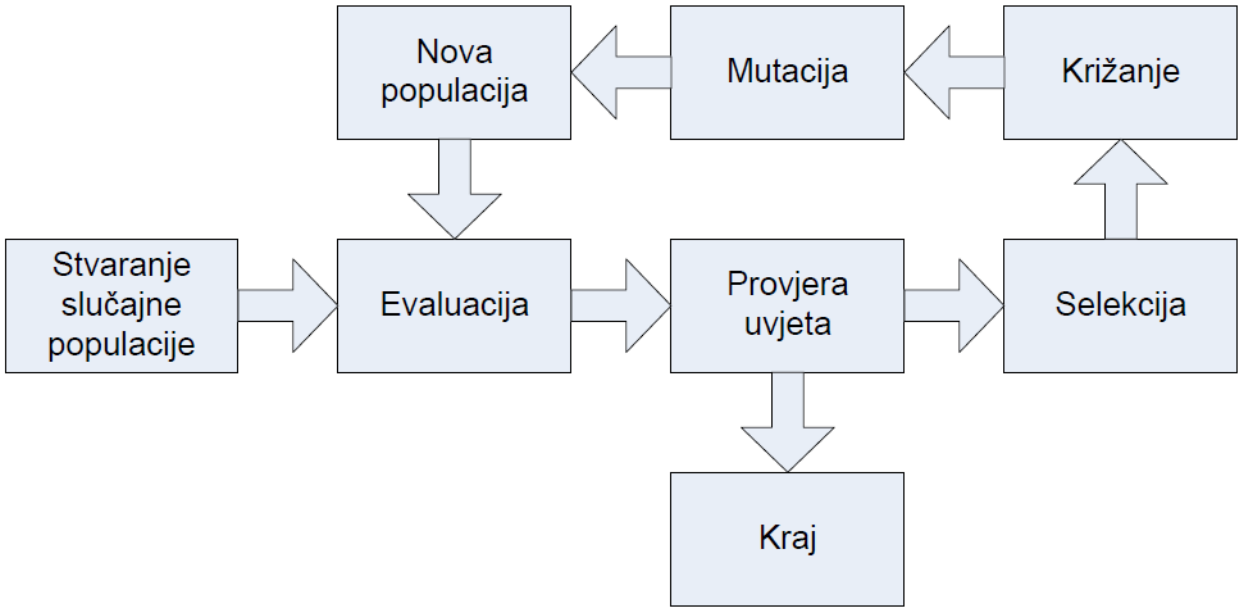
Genetskim algoritmom rješavamo optimizacijske probleme. Za zadanu funkciju $f(\vec{x})$ u kojoj je $x = (x_1, x_2, \ldots, x_n)$ želimo pronaći $\vec{x}^*$ koja maksimizira ili minimizira funkciju f. Svaki genetski algoritam započinje rad s populacijom jedinki, populacijom mogućih rješenja problema. Svaka jedinka (kromosom) ima dobrotu koja predstavlja vrijednost funkcije u točki koju jedinka predstavlja. Rješavamo li maksimizacijski problem, jedinka je lošija što ima manju vrijednost, a ako je problem minimizacijski, veća vrijednost jedinke znači veću udaljenost od rješenja. Nakon toga na red dolazi rad s operatorima. Razlikujemo četiri vrste operatora. Operatorom *selekcije* biraju se jedinke koje će postati roditelji nakon primjene operatora *križanja*. Nad djecom djeluje operator *mutacije* te ona ulaze u populaciju operatorom *zamjene*.

Sekvencijski genetski algoritmi dijele se u dvije skupine. Glavna karakteristika eliminacijskog genetskog algoritma je zamjena određene jedinke u populaciji novom. Jedinka koja će biti zamijenjena može biti odabrana na više načina, npr. možemo zamijeniti najgoru. Također, i dijete se može eliminirati (npr. ako je lošije od svih članova u populaciji). Grafički prikaz pseudokoda ovog algoritma prikazan je slikom 3.1.



Slika 3.1: Eliminacijski genetski algoritam

U generacijskom se populacijskom algoritmu iz koraka u korak stvaraju populacije djece koja postaju roditelji (naravno kao i kod evolucijskog algoritma, nakon mutacije). Stara populacija roditelja time se zamjenjuje novom. Na slici 3.2. prikazan je grafički prikaz pseudokoda generacijskog populacijskog algoritma.



Slika 3.2: Generacijski genetski algoritam

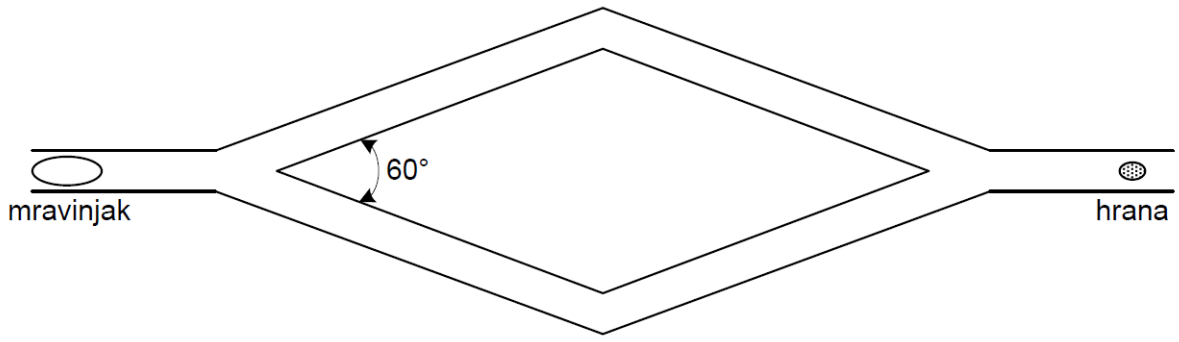
U genetske se algoritme često ugrađuje *elitizam*, svojstvo koje nam garantira da se najbolje pronađeno rješenje ne može izgubiti. Na primjer, moglo bi se dogoditi da je dijete lošije od najboljeg roditelja. Ako bismo koristili generacijski populacijski algoritam, ostali bismo bez najboljeg rješenja. Jedna od ideja kako riješiti ovaj problem je najbolje rješenje odmah kopirati u novu populaciju te normalno dalje nastaviti postupak.

Prikaz rješenja u genetskim algoritmima moguće je napraviti na nekoliko načina, a najjednostavniji je nizom bitova. Nakon odabira vrste prikaza, potrebno je odrediti kakvo će se križanje i mutacija koristiti. Križanje može biti s jednom točkom prekida, s t točaka prekida ili uniformno, a mutacija može biti jednolika ili nejednolika uz zadanu vjerojatnost mutacije bita.

Također je potrebno odrediti i operator selekcije. Najpoznatije vrste selekcija su proporcionalna i turnirska selekcija. Proporcionalna selekcija ili *Roulette-wheel selection* temelji se na sljedećoj ideji. Sve jedinke postave se na kolo, ali tako da ona jedinka koja je bolja zauzima veću površinu. Kolo zavrtimo te pogledamo na kojoj se jedinki zaustavilo. Zbog veće površine koje zauzimaju bolje jedinke, ponavljanjem eksperimenta, češće ćemo birati upravo njih. U turnirskoj selekciji izvučemo slučajni uzorak od n rješenja te odabiremo izvučeno rješenje s najvećom dobrotom. Postoji više vrsta turnirske selekcije kao što su selekcija s ponavljanjem, bez ponavljanja, *k-turnirska* selekcija u kojoj odabiremo k jedinki itd.

3.2. Algoritam kolonije mrava

Priroda nam pokazuje kako možemo učiti i od jednostavnih bića - mrava. Mravi zahvaljujući međusobnim interakcijama uspijevaju izgraditi nevjerojatne organizacijske strukture bez obzira na to što uopće nemaju razvijen vidni sustav. Mnogi su znanstvenici pokazali zanimanje za proučavanje mrava. Ono što je posebno zanimljivo je da su mravi uspješni u rješavanju optimizacijskih problema. Jednostavnim eksperimentom utvrđeno je kako mravi uvijek pronalaze najkraći put između hrane i njihove kolonije. Deneubourg i suradnici su 1990. godine napravili dvokraki most između mravinjaka i hrane (slika 3.3).



Slika 3.3: Eksperiment dvokrakog mosta

U prvom dijelu eksperimenta, oba kraka mosta bila su jednako duga te su se mravi u početku jednako kretali kroz njih. Međutim, nakon nekog vremena, mravi su se počeli kretati samo jednim krakom. Mravi ostavljaju kemijski trag - feromone. Što je koncentracija ovakvog traga jača, mravi će sve više odabirati taj put. U ovom primjeru nekoliko mrava slučajno odabere drugačiji krak mosta od početno odabranog, te na prvotnom kraku koncentracija feromona slabi (isparava). Što je koncentracija feromona slabija, to sve više mravi odabire onaj drugi krak te dolazimo do našeg rezultata. Svi mravi idu samo jednim krakom mosta. Eksperiment je ponovljen s kraćim krakom te je uočeno da mravi nakon nekog vremena upravo biraju taj kraći krak mosta za dolazak do hrane. U zadnjem je dijelu eksperimenta napravljen jednokraki most kojim su mravi prelazili. Nakon nekog vremena, dodan je kraći krak, no mravi su nastavili prolaziti duljim krakom upravo zato što je razina feromona tamo bila jaka.

Zaključke iz prethodno opisanog eksperimenta možemo opisati jednostavnim matematičkim modelom. Polazimo od toga da mravi ostavljaju feromonski trag bez obzira kreću li se od mravinjaka do hrane ili u obrnutom smjeru. Neke vrste mrava čak ostavljaju jači feromonski trag što je izvor hrane bogatiji kako bi pripadnicima svoje vrste pokazali koji je od potencijalno nekoliko izvora hrane bolji. Krakove između mravinjaka i hrane možemo modelirati bridovima grafa. Na sve bridove postavimo jednaku količnu feromona u fazi inicijalizacije. Mrav mora odlučiti u koji će čvor u prvom koraku krenuti. Odluku donosi na temelju vjerojatnosti odabira brida p_{ij}^k:

$$p_{ij}^k = \begin{cases} \frac{{\tau_{ij}}^\alpha}{\sum_{l \in N_i^k} \tau_{il}^\alpha}, & j \in N_i^k, \\ 0, & j \notin N_i^k. \end{cases}$$

U navedenom izrazu τ_{ij} predstavlja vrijednost feromonskog traga na bridu koji se nalazi između čvorova i i j a α je konstanta. Skup N_i^k predstavlja skup svih indeksa svih čvorova u koje je u koraku k moguće prijeći iz čvora i. Ako nije moguće prijeći iz čvora i u čvor j, vjerojatnost je 0. Možemo primijetiti da suma u nazivniku slučaja u kojem vjerojatnost nije 0 ide po svim bridovima koji vode do čvorova u koje se može doći iz čvora i. Mrav navedenu proceduru ponavlja sve dok ne stigne do hrane. Ovako umjetno definirani mravi feromonski trag ostavljaju samo po povratku u mravinjak, s time da je jačina feromonskog traga proporcionalna dobroti rješenja. Potrebno je ažurirati sve bridove kojima je mrav prošao. Ažuriranje radimo na sljedeći način:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau^k,$$

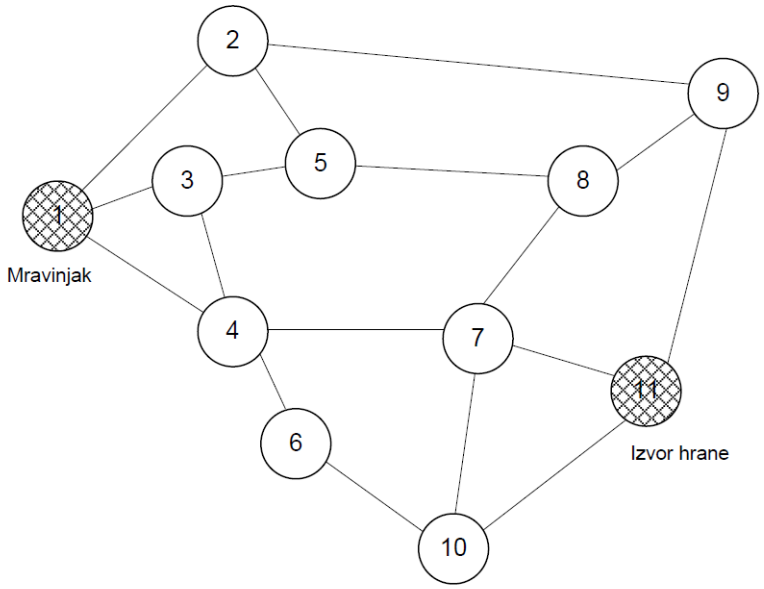
gdje je:

$$\Delta\tau^k = \frac{1}{L}.$$

L predstavlja duljinu prijeđenog puta, za koju vidimo da je obrnuto proporcionalna količini feromona. Isparavanje feromona definirano je s:

$$\tau_{ij} \leftarrow \tau_{ij} \cdot (1 - p)$$

S p smo označili brzinu isparavanja koje se primjenjuje na sve bridove grafa, a može poprimiti vrijednosti iz intervala od 0 do 1. No, i u ovakvom pristupu mogu nastati problemi. Jedan od njih je stvaranje ciklusa ako mravima dopustimo da odabire bilo koji čvor u povezanom putu. Ažuriranje feromonskog traga nakon svakog mrava moglo bi biti doista zahtjevno ako radimo s većim grafovima. Na slici 3.4. prikazan je primjer pronalaska hrane.



Slika 3.4: Primjer pronalaska hrane

Jednostavan mravlji algoritam radi na sljedeći način. Populaciju mrava označili smo s m te pustili da mravi stvore određena rješenja koja vrednujemo prema njihovoj duljini. Svaki mrav pamti pređeni put i kada treba birati u koji će sljedeći čvor krenuti, odbacuje one kroz koje je prošao. Nakon što svih m mrava stvorilo prijedloge rješenja, odabire se podskup od n mrava, $n \leq m$ koji će obaviti ažuriranje feromonskih tragova. Poslije toga primjenjuje se procedura isparavanja feromona te se postupak ciklički ponavlja.

3.3. Algoritam roja čestica

Pokušaj simuliranja kretanja jata ptica na računalu sasvim je slučajno rezultirao pronalaskom algoritma roja čestica (engl. *Particle Swarm Optimization*). C. W. Reynolds je u svom radu promatrao jato ptica kao sustav čestica pri čemu se svaka ptica vodi sljedećim pravilima:

- ne smije doći do kolizije s bliskim pticama
- brzina treba biti usklađena onoj bliskih ptica
- ostati u blizini drugih ptica

R. C. Eberhart i J. Kennedy su primijetili da se ovako definiran sustav može koristiti kao optimizacijski alat. Ideja je kasnije razrađivana u knjigama i radovima te su opisane primjene ovog algoritma na treniranje neuronske mreže te u radu nad diskretnim domenama.

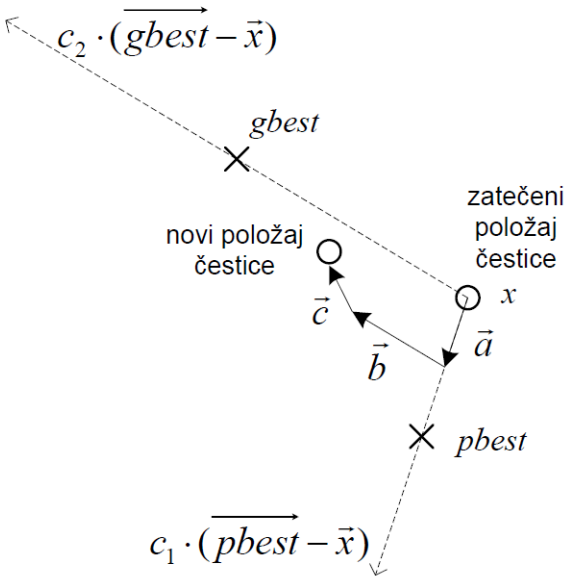
U algoritmu roja čestica jedinke pretražuju višedimenzijski prostor koristeći vlastito iskustvo i iskustvo susjednih jedinki. Pri tome uzima u obzir samo ona iskustva rješenja (svoja i susjedna) koja su najbolja. Zbog takvog načina računanja nove pozicije u prostoru razlikujemo dvije vrste faktora. *Individualni* i *socijalni*. Naravno, navedene komponente ne moraju imati jednak utjecaj. Ako je dominantan individualni faktor, jedinka pretražuje prostor stanja, a dominantan socijalni faktor znači da se uređuje već pronađeno rješenje. Algoritam možemo opisati sljedećim pseudokodom. Varijabla VEL_POP predstavlja veličinu populacije koja pretražuje prostor rješenja dimenzije DIM. U varijabli x pohranjene su trenutne pozicije svih čestica dok su u varijabli v pohranjene njihove brzine. Varijable x i v su dvodimenzijska polja. Broj readaka predstavlja broj čestica, a broj stupaca broj dimenzija. Prostor rješenja je ograničen varijablama $xmin$ i $xmax$. Brzina je također ograničena, vrijednostima $vmin$ i $vmax$.

Algoritam počinje inicijalizacijom populacije (linije 1 do 6) pri čemu se svaka jedinka smješta na slučajno odabranu poziciju te joj se dodjeljuje slučajno odabrana brzina. Algoritam se izvodi sve dok se ne ispuni uvjet zaustavljanja, a to je pronalazak dovoljno dobrog rješenja ili dostizanje maksimalnog broja iteracija. U linijama 10 do 12 radi se evaluacija svake jedinke pomoću funkcije dobrote. U linijama 14 do 19 za svaku česticu provjerimo njeno do tada zapamćeno najbolje rješenje i njeno novo pronađeno rješenje. Ako je novo bolje, pamti se kao novo najbolje rješenje. Zatim u čitavoj populaciji pronađemo najbolje rješenje, i ako je prethodno zapamćeno globalno rješenje lošije, radimo ažuriranje na novo (linije 21 do 26). Da bi algoritam bio potpun, potrebno je još sam ažurirati trenutnu brzinu i položaj. Prvo je potrebno ažurirati brzinu tako da na trenutnu dodamo umnožak faktora individualnosti (c_1) i slučajnog broja iz intervala [0,1]. Zatim dodajemo umnožak faktora socijalnosti (c_2) i slučajni broj iz intervala [0,1]. Ako je brzina izvan dozvoljenog intervala, potrebno ju je korigirati te u skladu s njom ažurirati položaj čestice.

```
za i = 1 do VEL_POP
  za d = 1 do DIM
    x[i][d] = random(xmin[d], xmax[d])
    v[i][d] = random(vmin[d], vmax[d])
  kraj
kraj

ponavljaj dok nije kraj

  za i = 1 do VEL_POP
    f[i] = funkcija(x[i])
  kraj

  za i = 1 do VEL_POP
    ako je f[i] bolji od pbest_f[i] tada
      pbest_f[i] = f[i]
      pbest[i] = x[i]
    kraj
  kraj

  za i = 1 do VEL_POP
    ako je f[i] bolji od gbest_f[i] tada
      gbest_f[i] = f[i]
      gebst[i] = x[i]
    kraj
  kraj

  za i = 1 do VEL_POP
    za d = 1 do DIM
      v[i][d] = v[i][d] + c1*rand()*(pbest[i][d]-x[i][d])
                        + c2*rand()*(gbest[d]-x[i][d])
      v[i][d] = iz_opsega(v[i][d])
      x[i][d] = x[i][d] + v[i][d]
    kraj
  kraj
kraj
```

Faktori c_1 i c_2 su konstante koje se obično postavljaju na vrijednost 2.0. Što je veći faktor c_1, jedinka će biti više individualna i više će istraživati prostor. Veći faktor c_2 znači da se jedinka orijentira na rješenje koje je pronašao kolektiv te da ga dodatno istražuje s ciljem potencijalnog poboljšanja.

Na slici 3.5. grafički je prikazano kako individualni i socijalni faktori djeluju na česticu. Slika je napravljena uz pretpostavku da su c_1 i c_2 jednaki 2.0 te da su generirani brojevi jednaki.



Slika 3.5: Grafički prikaz pomaka čestice (algoritam roja čestica)

Prema formuli kojom ažuriramo brzinu, rezultantni je vektor suma tri komponente, $\vec{a}$, $\vec{b}$ i $\vec{c}$. Vektori $\vec{a}$ i $\vec{b}$ predstavljaju pomake prema najboljem rješenju, vektor $\vec{a}$ prema najboljem rješenju jedinke, a vektor $\vec{b}$ prema najboljem rješenju kolektiva. Obje su vrijednosti odgovarajuće skalirane. Vektor $\vec{c}$ predstavlja staru trenutnu vrijednost brzine te inerciju same čestice. (Čupić, 2013)

4. Implementacija

Ranije opisani kombinatorni problemi implementirani su algoritmom roja čestica u programskom jeziku Python 3. Algoritam roja čestica odabran je kao reprezentativni primjer algoritma kojim ćemo rješavati kombinatorne probleme upravo zbog pristupa u kojem više jedinki, djelujući individualno i u grupi, pokušavaju pronaći rješenje. Za problem usmjeravanja vozila odabran je genetski algoritam kako bi se pokazao i ovaj pristup. U ovom su poglavlju objašnjeni ključni dijelovi programskog kôda, a cjelokupni kôdovi su dostupni na otvorenom GitHub repozitoriju. [1] Također, za svaki je algoritam napravljena analiza učinkovitosti ovisno o promjeni parametara.

4.1. Problem trgovačkog putnika

Implementaciju rješavanja problema trgovačkog putnika algoritmom roja čestica započinjemo definiranjem grafa. Graf definiramo kao razred koji ima skup bridova (*edges*) i vrhova (*vertices*). U varijabli *amount_vertices* pamtimo broj vrhova u grafu.

```
1 class Graph:
2     def __init__(self, amount_vertices):
3         self.edges = {}
4         self.vertices = set()
5         self.amount_vertices = amount_vertices
```

Kako bi razred *Graph* bio kompletan, dodane su i metode koje omogućavaju sljedeće:

- dodavanje brida u graf
- provjera postoji li određeni bridu u grafu
- tekstualni ispis grafa
- izračun cijene puta u grafu
- ispis liste nasumičnih puteva u grafu

te klasa koja generira potpuni graf.

[1] https://github.com/filipkujundzic/diplomskirad

Česticu (tj. jedinku) smo prikazali razredom *Particle*.

```
class Particle:
        def __init__(self, solution, cost):
            self.solution = solution
            self.pbest = solution

            self.cost_current_solution = cost
            self.cost_pbest_solution = cost

            self.velocity = []
```

U varijabli *solution* pamtimo trenutno rješenje, a najbolje rješenje koje je postignuto (koje ujedno odgovara vrijednosti funkcije dobrote za česticu) spremamo u varijabli *pbest*. Varijable *cost_current_solution* i *cost_pbest_solution* predstavljaju cijene trenutnog i najboljeg rješenja. Brzina čestice sastoji se od četiri člana. Npr. $(1, 2, 1, beta)$ Prva dva člana u zagradi predstavljaju argumente operatora zamjene (engl. *swam operator*), oznaka $SO(1, 2)$ koji nam govori da se radi zamjena vrha u grafu označenog s 1 i vrha označenog s 2. Treći argument je vjerojatnost realizacije operatora, a "beta" je argument s kojim uspoređujemo.

Ključni je razred u kôdu *PSO*.

```
class PSO:
    def __init__(self, graph, iterations, size_population,
                 beta=1, alfa=1):
        self.graph = graph
        self.iterations = iterations
        self.size_population = size_population
        self.particles = []
        self.beta = beta
        self.alfa = alfa
```

Razred *PSO* ima atribute graf, maksimalni broj iteracija koji označava najveći dozvoljeni broj iteracija u algoritmu, veličinu populacije, listu jedinki, te vjerojatnosti beta $(gbest - x(t-1))$ i alfa $(pbest - x(t-1))$ za operatore zamjene. Metoda koja pokreće algoritam roja čestica je metoda *run*. Navedenu metodu pokrećemo nakon kreiranja instance *PSO* na sljedeći način.

```
    pso = PSO(graph, iterations=100, size_population=10,
                                     beta=1, alfa=0.9)
    pso.run()
```

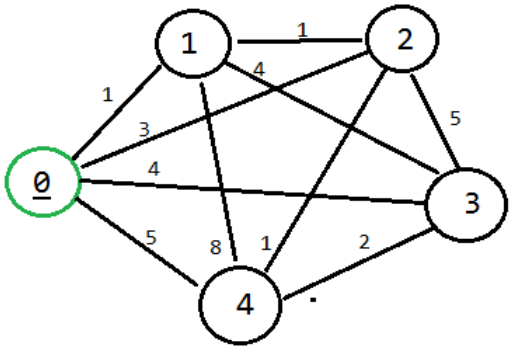
U nastavku je priložen kôd metode *run*.

```
1   def run(self):
2       for t in range(self.iterations):
3           self.gbest = min(self.particles,key=attrgetter('
                cost_pbest_solution'))
4
5           for particle in self.particles:
6               particle.clearVelocity()
7               temp_velocity = []
8               solution_gbest = copy.copy(self.gbest.getPBest())
9               solution_pbest = particle.getPBest()[:]
10              solution_particle = particle.getCurrentSolution()[:]
11
12              for i in range(self.graph.amount_vertices):
13                  if solution_particle[i] != solution_pbest[i]:
14                      swap_operator = (i, solution_pbest.index(
                            solution_particle[i]), self.alfa)
15                      temp_velocity.append(swap_operator)
16
17                      aux = solution_pbest[swap_operator[0]]
18                      solution_pbest[swap_operator[0]] =
                            solution_pbest[swap_operator[1]]
19                      solution_pbest[swap_operator[1]] = aux
20
21              for i in range(self.graph.amount_vertices):
22                  if solution_particle[i] != solution_gbest[i]:
23                      swap_operator = (i, solution_gbest.index(
                            solution_particle[i]), self.beta)
24                      temp_velocity.append(swap_operator)
25
26                      aux = solution_gbest[swap_operator[0]]
27                      solution_gbest[swap_operator[0]] = solution_gbest
                            [swap_operator[1]]
28                      solution_gbest[swap_operator[1]] = aux
29
30              particle.setVelocity(temp_velocity)
31              for swap_operator in temp_velocity:
32                  if random.random() <= swap_operator[2]:
33                      aux = solution_particle[swap_operator[0]]
34                      solution_particle[swap_operator[0]] =
                            solution_particle[swap_operator[1]]
35                      solution_particle[swap_operator[1]] = aux
```

```
36
37                  particle.setCurrentSolution(solution_particle)
38                  cost_current_solution = self.graph.getCostPath(
                        solution_particle)
39                  particle.setCostCurrentSolution(cost_current_solution)
40
41                  if cost_current_solution < particle.getCostPBest():
42                      particle.setPBest(solution_particle)
43                      particle.setCostPBest(cost_current_solution)
```

U drugoj liniji nalazi se petlja koja se izvršava *self.iteration* iteracija. Prvo što radimo u petlji je ažuriranje informacije koja je čestica najbolja u populaciji (linija 3). Prva unutarnja *for* petlja prolazi po svim česticama u populaciji (linija 5). U linijama 6 do 10 prvo se obriše trenutna brzina čestice, dohvati rješenje *gbest*, kopira rješenje *pbest* te dohvati rješenje trenutnog rješenja čestice. U linijama 12 do 19 generiramo sve operatore zamjene kako bi izračunali $(pbest - x(t-1))$. Gotovo isto radimo u linijama 21 do 28, samo ovaj puta računamo $(gbest - x(t-1))$. 30. linija predstavlja ažuriranje brzine čestice koju ćemo koristiti u *for* petlji ispod (linija 31) kako bi generirali novo rješenje za česticu (linije 32 do 3). Ostaje još ažurirati atribut trenutnog rješenja čestice (linija 37), izračunati cijenu trenutnog rješenja (linija 38) te ažurirati cijenu trenutnog rješenja (linija 39). Zadnje što se radi u algoritmu prije iduće iteracije petlje (ili završetka algoritma) je provjera da li je trenutno rješenje *pbest*, tj najbolje rješenje koje je postignuto za česticu.

No, koliko je zapravo važan socijalni faktor u algoritmu roja čestica? Zašto je uopće potrebna suradnja među jedinkama? Pogledajmo jedan primjer. Neka je graf kojim smo opisali problem prikazan na slici 4.1. [2]



Slika 4.1: Algoritam roja čestica - primjer

[2] https://github.com/marcoscastro/tsp_pso

Graf ima pet vrhova, označenih s 0 do 4, gdje je vrh 0 početni vrh. Brojevi na bridovima grafa označavaju cijenu puta, pa je tako na primjer cijena između početnog vrha (0) i vrha 4 jednaka 5. Program smo pokrenuli uz zadani maksimalni broj iteracija 100, veličinu populacije 10 te alfa (individualni faktor) i beta (socijalni faktor) 0.85. Dobiven je sljedeći rezultat.

```
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
pbest: [4, 2, 1, 0, 3]    cost pbest: 9    curr solution: [4, 2, 1, 0, 3]    cost current sol: 9
```

Najbolje rješenje (*gbest*) je $[4, 2, 1, 0, 3]$ s cijenom 9. Pokušajmo isti problem riješiti uz jednake parametre, samo uz socijalni faktor 0.

```
pbest: [4, 1, 3, 2, 0]    cost pbest: 25    curr solution: [4, 1, 3, 2, 0]    cost current sol: 25
pbest: [4, 1, 3, 0, 2]    cost pbest: 20    curr solution: [4, 1, 3, 0, 2]    cost current sol: 20
pbest: [4, 2, 0, 3, 1]    cost pbest: 20    curr solution: [4, 2, 0, 3, 1]    cost current sol: 20
pbest: [4, 2, 3, 0, 1]    cost pbest: 19    curr solution: [4, 2, 3, 0, 1]    cost current sol: 19
pbest: [4, 1, 2, 3, 0]    cost pbest: 23    curr solution: [4, 1, 2, 3, 0]    cost current sol: 23
pbest: [4, 0, 1, 2, 3]    cost pbest: 14    curr solution: [4, 0, 1, 2, 3]    cost current sol: 14
pbest: [4, 0, 3, 2, 1]    cost pbest: 23    curr solution: [4, 0, 3, 2, 1]    cost current sol: 23
pbest: [4, 0, 2, 3, 1]    cost pbest: 25    curr solution: [4, 0, 2, 3, 1]    cost current sol: 25
```

Za isti smo problem, bez međusobne suradnje čestica dobili najbolje rješenje s cijenom 14. Možemo također vidjeti kako vrijednosti trenutnog rješenja variraju u odnosu na prvo pokretanje programa (vrijednosti najgoreg rješenja idu do 25, dok uz uključenu interakciju ostaju vrlo blizu najboljem globalnom rješenju). Pokrenimo sad program uz iznimno mali faktor interakcije, 0.05, dok su svi ostali parametri jednaki kao prije.

```
pbest: [1, 0, 3, 4, 2]    cost pbest: 9     curr solution: [1, 0, 3, 4, 2]    cost current sol: 9
pbest: [1, 0, 3, 4, 2]    cost pbest: 9     curr solution: [1, 0, 3, 4, 2]    cost current sol: 9
pbest: [1, 0, 3, 4, 2]    cost pbest: 9     curr solution: [1, 0, 3, 4, 2]    cost current sol: 9
pbest: [1, 0, 3, 4, 2]    cost pbest: 9     curr solution: [1, 0, 3, 4, 2]    cost current sol: 9
pbest: [1, 2, 4, 3, 0]    cost pbest: 9     curr solution: [1, 2, 4, 3, 0]    cost current sol: 9
pbest: [1, 3, 4, 2, 0]    cost pbest: 11    curr solution: [1, 3, 4, 2, 0]    cost current sol: 11
pbest: [1, 0, 3, 4, 2]    cost pbest: 9     curr solution: [1, 0, 3, 4, 2]    cost current sol: 9
pbest: [1, 0, 3, 4, 2]    cost pbest: 9     curr solution: [1, 0, 3, 4, 2]    cost current sol: 9
```

Vidimo da čak i mala socijalna interakcija uvelike doprinosi kvaliteti rješenja pronađenog algoritmom roja čestica. Zbog toga je ovaj faktor važan te se postavljajući ga nulu, udaljavamo od traženog rješenja.

4.2. Problem naprtnjače

Problem naprtnjače također je implementiran algoritmom roja čestica.

```
1 items = ['TV', 'Camera', 'Projector', 'Walkman',
2              'Radio', 'Mobile phone', 'Laptop']
3 prices = [35, 85, 135, 10, 25, 2, 94]
4 kg = [2, 3, 9, 0.5, 2, 0.1, 4]
5 maxKg = 25
```

Definirali smo dostupne predmete, njihove cijene (vrijednosti) te težinu u kilogramima. Tako bi na primjer, projektor imao cijenu 135 a težinu 9 kg. Slično možemo pročitati i za ostale predmete, indeks predmeta u listi *items* odgovara indeksu u listama *prices* i *kg*. Maksimalna težina koja može stati u naprtnjaču je 25 kg.

```
1 def function_max(x):
2     t = f_total_value(x)
3     return t + f_total_kg(x, t)
```

Funkcija koju pokušavamo maksimizirati je funkcija ukupne vrijednosti svih predmeta koji se stavljaju u naprtnjaču. Funkcija koja se poziva u drugoj liniji, *f_total_value*, izgleda ovako:

```
1 def f_total_value(x):
2     total = 0
3     for i in range(len(x)):
4         total += x[i] * prices[i]
5     return total
```

Vidimo da funkcija jednostavno radi umnožak broja predmeta s njihovom vrijednošću. Analogno ovoj funkciji, potrebna nam je funkcija koja će isto napraviti, ali samo s masom predmeta. No, u toj funkciji moramo napraviti provjeru da li je ukupna masa predmeta koje smo uzeli premašila ukupan (maseni) kapacitet naprtnjače. Ako se premaši kapacitet naprtnjače, vraća se negativna vrijednost koja ujedno i resetira povratnu vrijednost tako da se taj konkretni odabir predmeta ne uzima u obzir kao potencijalno rješenje.

```
    if total <= maxKg:
        if total <= reset_elem:
            return reset_elem - total
        else:
            return 0
    else:
        return - reset_elem
```

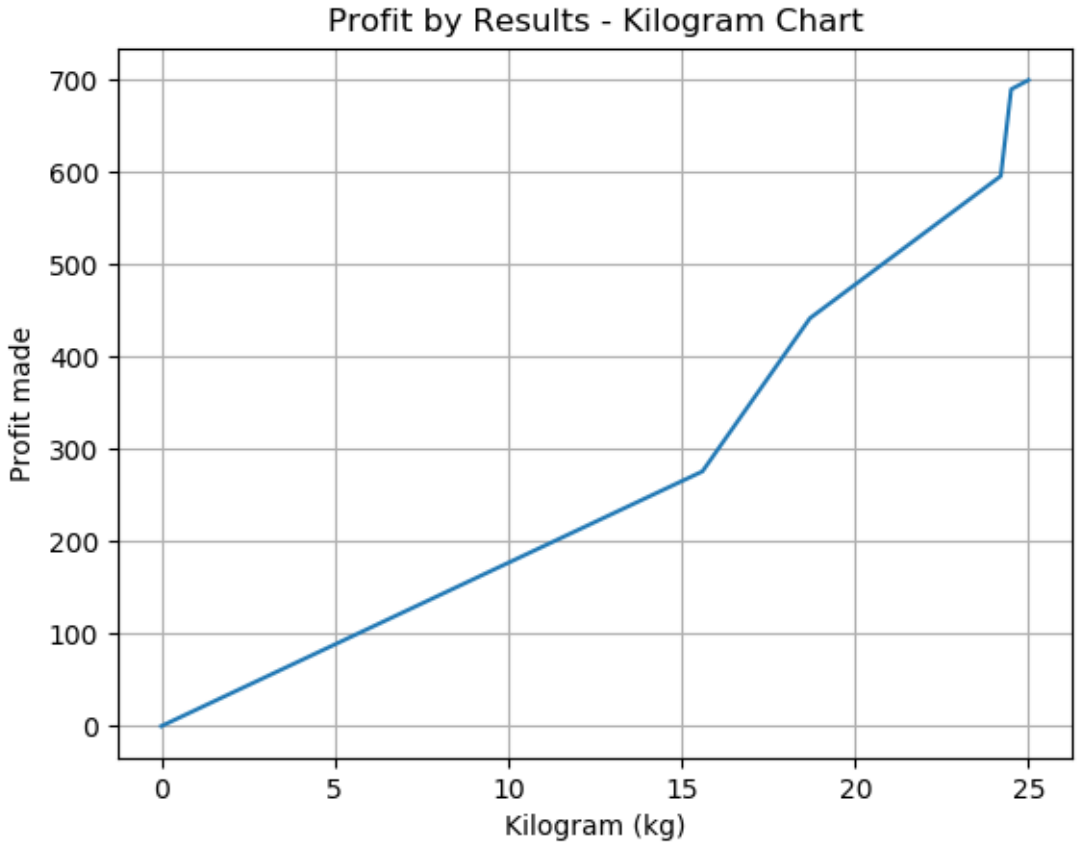
Definicija razreda čestice (engl. *Particle*) slična je kao što smo imali i kod problema trgovačkog putnika.

```
class Particle:
    def __init__(self, startingValues):
        self.position = []
        self.speed = []
        self.pBest = []
        self.pBestapproach = -1
        self.approach = -1

        for i in range(particles_number):
            self.speed.append(random.uniform(-1, 1))
            self.position.append(startingValues[i])
```

Svaka čestica ima svoju poziciju, brzinu, najbolju poziciju te individualni i individualni najbolji pristup. Nakon inicijalizacije razreda, svakoj smo čestici dodijelili nasumičnu brzinu i položaj. Prilikom ažuriranja brzine, uveden je faktor w koji ima cilj zadržavanja prethodne brzine čestice. Vidimo i prisutnost već prije objašnjenih konstanti c_1 i c_2 koje naglašavaju individualni, odnosno grupni rezultat te ukupnu brzinu koja je suma dvije komponente.

```
    def speed_update(self, group_max_position):
        w = 0.99
        c1 = 1.99
        c2 = 1.99

        for i in range(particles_number):
            r1 = random.random()
            r2 = random.random()

            individ_speed = c1 * r1 * (self.pBest[i] - self.position[i])
            social_speed = c2 * r2 * (group_max_position[i]
                                      - self.position[i])
            self.speed[i] = w * self.speed[i] + individ_speed +
                social_speed
```

Nakon što se algoritam izvrši, možemo i grafički prikazati ovisnost dobivenog profita u odnosu na masu predmeta spremljenih u naprtnjaču. Vidimo kako ovisnost nije linearna te da je potreban složeniji algoritam (kao npr. algoritam roja čestica) kako bi se problem riješio.



Slika 4.2: Problem naprtnjače

Kako se odnose veličina populacije i broj dopuštenih iteracija s konačnim rezultatom algoritma pokazano sljedećom tablicom. Program je pokrenut 30 puta, pri čemu su mijenjani parametri veličine populacije i broja iteracija, te su praćeni profit i popunjenost naprtnjače koja ima maksimalno 25 kg.

swarm_size	max_iter	profit	kg	n
100	50	584	25	2
		680	24	1
		689	25	6
		700	25	1
1000	100	689	25	5
		700	25	5
10000	1000	700	25	10

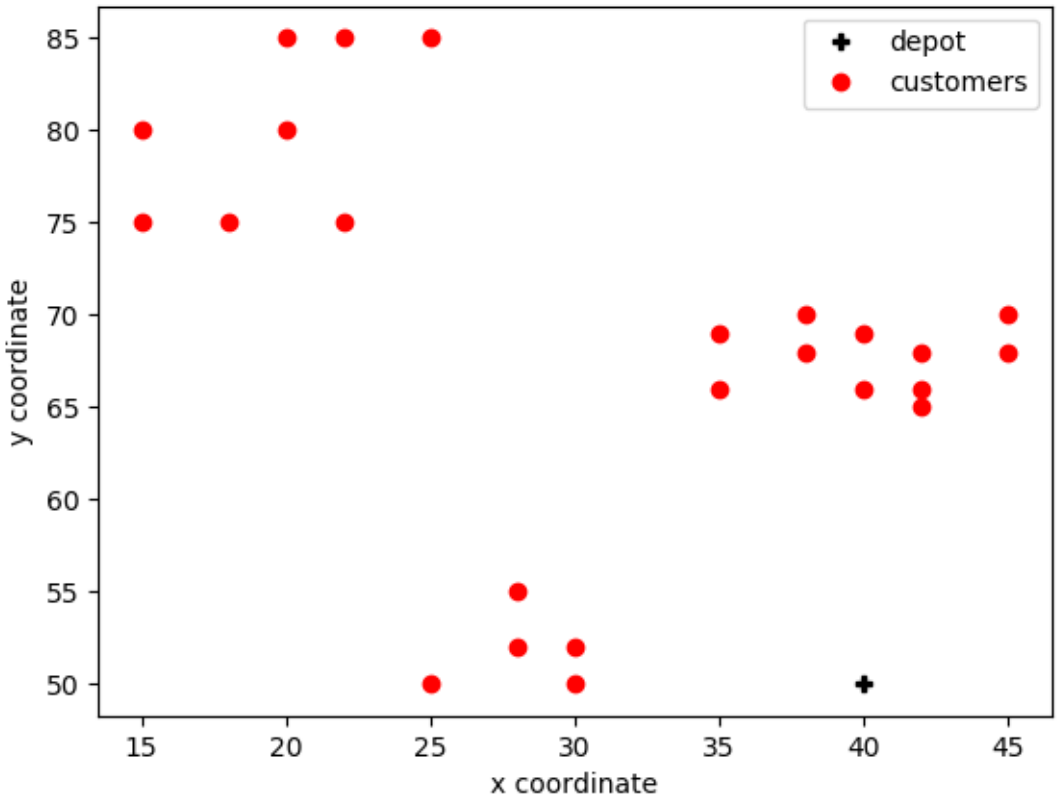
Za prvih 10 pokretanja odabrana je veličina populacije 100 i najveći dozvoljeni broj iteracija 50. Postignuta su četiri različita profita: 584, 680, 689 i 700. Najmanji od navedenih postignut je 2 puta, dok je najveći postignut samo jednom. Znatno poboljšanje već se uočava povećavanjem jedinki u populaciji na 1000 članova te povećavanjem najvećeg dozvoljenog broja iteracija na 100. Dvije najveće vrijednosti iz prethodnih 10 ponavljanja algoritama pojavljuju se i ovdje, uz jednaku zastupljenost (svaki od profita 689 i 700 pojavljuje se 5 puta. Povećanjem broja jedinki u populaciji na 10000 i dopuštanjem do 1000 iteracija u programu, dobili smo svih 10 puta najbolji mogući rezultat - profit 700 uz popunjenih svih 25 kilograma koliko stane u naprtnjaču. No, za jedno pokretanje s 10000 jedinki bilo je potrebno oko 2 minute i 50 sekundi na računalu s 4 fizičke jezgre. Ovdje možemo vidjeti kako je zapravo povećanje broja jedinki na pokazani način zapravo "skupo". Vjerojatno bismo mogli postići ovakav (ili približno ovakav) rezultat i s manje jedinki i manjim najvećim dozvoljenim brojem iteracija. To bismo mogli napraviti binarnim pretraživanjem. Prepolovimo broj jedinki i iteracija (5000 i 500) te pokrenemo 10 puta. Ovisno o rezultatu, polako povećavamo navedene parametre da bismo pronašli vrijednosti za koje algoritam nalazi profit 700, a manje su od 10000 i 1000.

4.3. Problem usmjeravanja vozila

Za rješavanje problema usmjeravanja vozila kao ulaz smo učitali tekstualnu datoteku sljedećeg izgleda:

```
1  C101
2
3  VEHICLE
4  NUMBER        CAPACITY
5     25             200
6
7  CUSTOMER
8  CUST NO.   XCOORD.    YCOORD.    DEMAND    READY TIME   DUE DATE    SERVICE
       TIME
9
10      0        40         50          0          0          1236          0
11      1        45         68         10        912           967         90
12      2        45         70         30        825           870         90
13      3        42         66         10         65           146         90
14      4        42         68         10        727           782         90
15      5        42         65         10         15            67         90
16      6        40         69         20        621           702         90
17      7        40         66         20        170           225         90
18      8        38         68         20        255           324         90
19      9        38         70         10        534           605         90
20     10        35         66         10        357           410         90
21     11        35         69         10        448           505         90
22     12        25         85         20        652           721         90
23    ...       ...        ...        ...        ...           ...         ...
```

Datoteka u prvoj liniji započinje svojim imenom, u ovom slučaju *C101*. Postavljen je broj vozila koji je dostupan za određeni problem, 25 te kapacitet svakog od njih, 200. Za svakog korisnika potrebna nam je informacija o njegovom rednom broju, X i Y koordinatama, njegovom zahtjevu, vremenu od kad je dostupan, vremenu do kad je dostupan te vremenu koje je potrebno da se ispuni njegov zahtjev. Na slici je prikazano nekoliko korisnika od 100 koliko ih ima. Koordinate su potrebne zbog grafičkog prikaza pozicije korisnika. Na grafičkom prikazu crvenim kružićima označen je položaj korisnika, dok je skladište označeno crnim znakom $+$.



Slika 4.3: Pozicije korisnika i skladišta u problemu usmjeravanja vozila

Linija koja pokreće cijeli genetski algoritam je sljedeća.

```
1    def run_ga(instance_name, individual_size, pop_size, cx_pb, mut_pb,
         n_gen, plot=False, save=False, logs=False):
```

Kao argumente funkcija *run_ga* prima veličinu jedinke, veličinu populacije (*pop_size*), vjerojatnost križanja (*cx_pb*), vjerojatnost mutacije (*mut_pb*) te broj generacija (*n_gen)*. Najprije je potrebno cijelu populaciju evaluirati.

```
1        for ind in pop:
2            ind.fitness.values = toolbox.evaluate(ind)
```

Zatim odaberemo 10% najboljeg potomstva i pomoću kotača ruleta odaberemo 90% ostalih.

```
1  offspring = tools.selBest(pop, int(numpy.ceil(pop_size * 0.1)))
2  offspring = list(map(toolbox.clone, offspring))
3  offspring_roulette = toolbox.select(pop,int(numpy.floor(pop_size*0.9))-1)
4  offspring.extend(offspring_roulette)
```

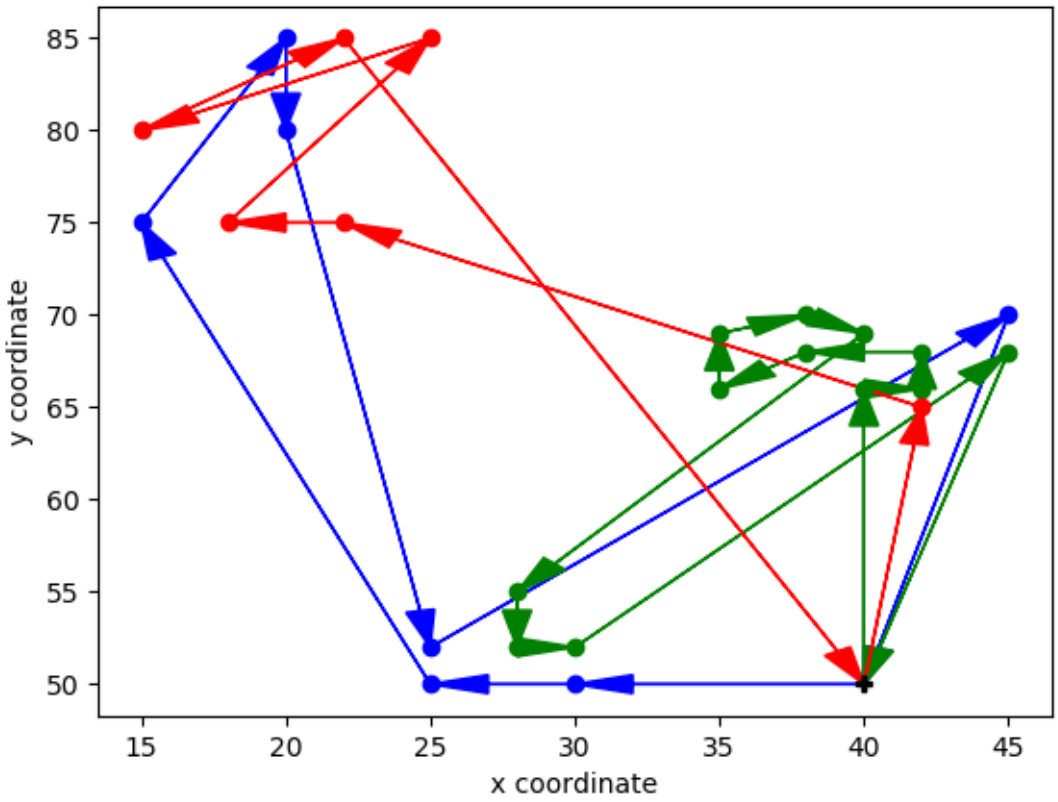
Križanje se obavlja s dvije točke prekida.

```
1  def crossover_pmx(ind1, ind2):
2
3      ind_len = len(ind1)
4      pos_ind1 = [0]*ind_len
5      pos_ind2 = [0]*ind_len
6
7      for i in range(ind_len):
8          pos_ind1[ind1[i]-1] = i
9          pos_ind2[ind2[i]-1] = i
10
11     locus1 = random.randint(0, int(ind_len/2))
12     locus2 = random.randint(int(ind_len/2)+1, ind_len-1)
13
14     for i in range(locus1, locus2):
15         temp1 = ind1[i]
16         temp2 = ind2[i]
17
18         ind1[i], ind1[pos_ind1[temp2-1]] = temp2, temp1
19         ind2[i], ind2[pos_ind2[temp1-1]] = temp1, temp2
20
21         pos_ind1[temp1-1], pos_ind1[temp2-1] = pos_ind1[temp2-1],
               pos_ind1[temp1-1]
22         pos_ind2[temp1-1], pos_ind2[temp2-1] = pos_ind2[temp2-1],
               pos_ind2[temp1-1]
23
24     return ind1, ind2
```

Od 7. do 9. linije radimo popis indeksa položaja, a točke križanja definiramo u 11. i 12. liniji. Proces zamjene je u 18 i 19 liniji. U 21. i 22. liniji spremamo ažurirane vrijednosti. Funkcija *crossover_pmx* vraća dvije vrijednosti, jedinke nakon križanja. Mutacija je također vrlo jednostavna, a opisana je ovim kôdom.

```
1  def mutate_swap(individual):
2      ind_len = len(individual)
3      locus1 = random.randint(0, int(ind_len / 2))
4      locus2 = random.randint(int(ind_len / 2) + 1, ind_len - 1)
5
6      temp = individual[locus1]
7      individual[locus1] = individual[locus2]
8      individual[locus2] = temp
9
10     return individual,
```

Postupak funkcije *mutate_swap* bi se mogao opisati jednom rečenicom: "Odaberi dva indeksa i zamijeni vrijednosti među njima". Nakon što je obavljeno križanje i mutacija, preostaje još evaluiranje novih vrijednosti i umetanje u populaciju. Rješenje možemo prikazati grafički. Strelice koje označavaju kretanje vozila označene su različitom bojom kako bi se lakše razaznalo koji je redoslijed korisnika u obilasku.

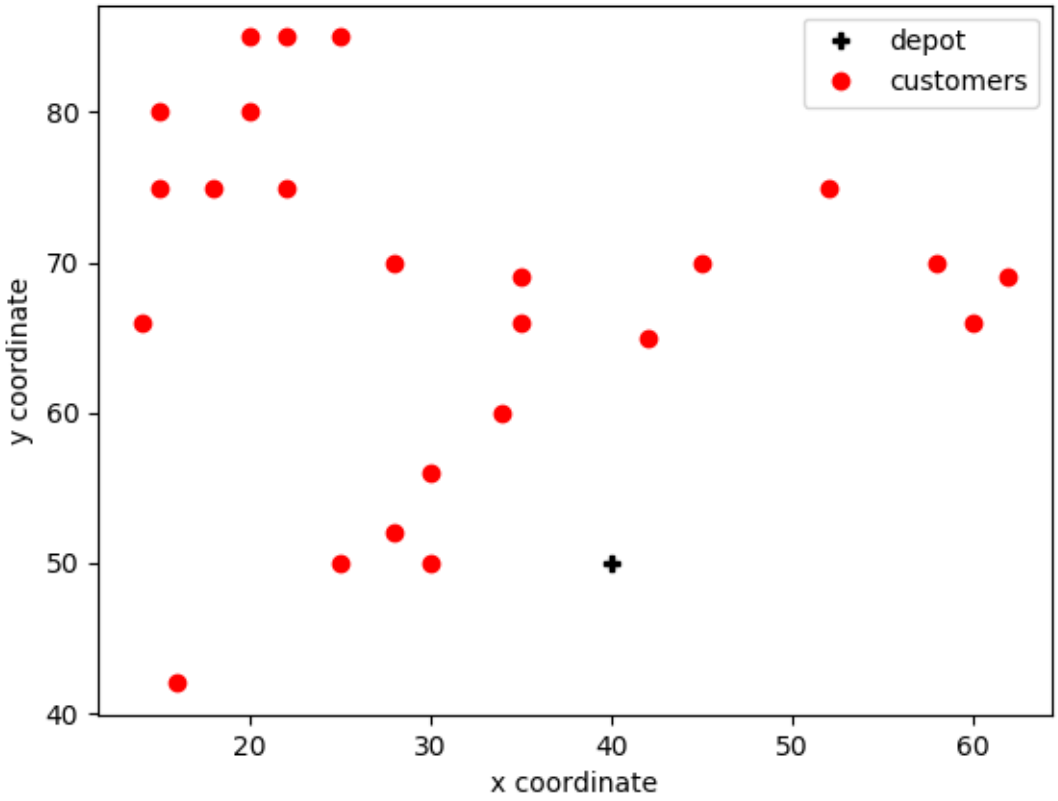


Slika 4.4: Rješenje problema usmjeravanja vozila genetskim algoritmom

Konačni rezultat algoritma dan je ispisom:

```
1 Best individual: [20, 24, 18, 16, 15, 25, 2, 7, 3, 4, 8, 10, 11, 9, 6,
    23, 22, 21, 1, 5, 13, 17, 12, 19, 14]
2  # Route 1 # 0 -> 20 -> 24 -> 18 -> 16 -> 15 -> 25 -> 2 -> 0
3  # Route 2 # 0 -> 7 -> 3 -> 4 -> 8 -> 10 -> 11 -> 9 -> 6 -> 23 -> 22 ->
    21 -> 1 -> 0
4  # Route 3 # 0 -> 5 -> 13 -> 17 -> 12 -> 19 -> 14 -> 0
5 Fitness: 18.41
6 Total cost: 5431.25
7 Found in (iteration): 1355
8 Execution time (s): 40.01
```

Dobili smo informaciju koja je najbolja jedinka, koliko je ruta pronađeno (i koje su), kolika je vrijednost funkcije dobrote, ukupna cijena, u kojoj je iteraciji rješenje pronađeno te koliko je vremena za to bilo potrebno.

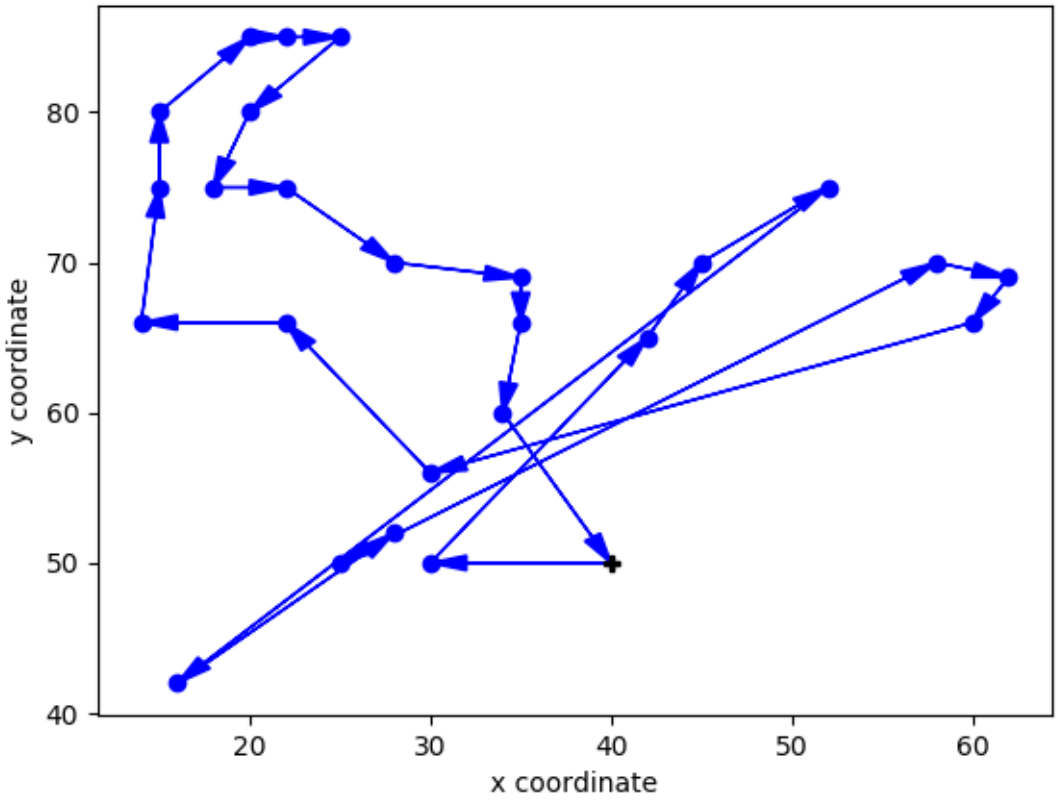


Slika 4.5: Problem usmjeravanja vozila s negrupiranim korisnicima

Složenst problema možemo vidjeti iz konačnog ispisa rješenja.

```
1 Best individual: [20, 5, 2, 1, 6, 22, 24, 7, 3, 4, 21, 25, 23, 18, 19,
    16, 14, 12, 15, 17, 13, 9, 11, 10, 8]
2  # Route 1 # 0 -> 20 -> 5 -> 2 -> 1 -> 6 -> 22 -> 24 -> 7 -> 3 -> 4 -> 21
     -> 25 -> 23 -> 18 -> 19 -> 16 -> 14 -> 12 -> 15 -> 17 -> 13 -> 9 ->
    11 -> 10 -> 8 -> 0
3 Fitness: 21.37
4 Total cost: 4679.07
5 Found in (iteration): 806
6 Execution time (s): 38.76
```

Pronađena je samo jedna ruta, po duljini dosta duža od one u prethodnom primjeru, a za otprilike jednako vremena. No, možemo primijetiti da su u pokazanom primjeru korisnici podijeljeni u tri skupine te su relativno jedan blizu drugome. Pogledajmo primjer kada to nije tako. Broj iteracija nešto je manji, no to obrazlažemo širim pretraživanjem područja obzirom na raspored korisnika. Konačno, pogledajmo i grafički prikaz rješenja ovog problema na kojem ćemo vidjeti jednu rutu koja dominira cijelim područjem u kojem su raspoređeni korisnici.



Slika 4.6: Rješenje problema usmjeravanja vozila s negrupiranim korisnicima

5. Zaključak

Iako su prošla stoljeća od prvih definicija kombinatornih problema, oni su i danas prisutni, i to u sve više područja. Klasični pristup rješavanju, grubom silom, postao je neučinkovit zbog eksplozije broja stanja koje treba pretražiti u ovim problemima da bismo bili sigurni da je pretraga temeljito provedena. No, prisiljeni smo potražiti drugi pristup jer su računala još uvijek previše spora da bi pronašla rješenje na ove probleme u realnom vremenu.

Srećom, pretraživanje cjelokupnog prostora rješenja nije potrebno. Postoje heuristički algoritmi koji nas usmjeravaju k rješenju, koristeći pri tome dostupne informacije o problemu. Štoviše, usmjeravaju nas prema rješenju koje ne mora nužno biti optimalno, ali je zadovoljavajuće dobro, što još više olakšava rješavanje kombinatornih problema.

U radu su opisana tri sveprisutna kombinatorna problema: problem trgovačkog putnika, problem naprtnjače i problem usmjeravanja vozila. Nakon opisa tri prirodom inspirirana optimizacijska algoritma (genetski algoritam, mravlji algoritam te algoritma roja čestica) pokazane su primjene algoritma roja čestica i genetskog algoritma na sva tri problema. Za svaki je algoritam opisano kako radi i koji su principi koji ih razlikuju od "klasičnih algoritama".

Razvoj prirodom inspiriranih optimizacijskih algoritma prije svega predstavlja naznaku kako je zapravo malo načina rješavanja problema do sada otkriveno. Procesi u prirodi optimiraju život u svakome segmentu već više od četiri milijarde godina, te su upravo oni izvor potencijalnih ideja koje će nam pomoći u rješavanju kako kombinatornih, tako i problema drugih skupina. Pri tome je važno naglasiti da ne tražimo najbolji algoritam, za koji je dokazano da ne postoji, nego razvojem što više različitih algoritama dobivamo alate za probleme različitih kategorija.

LITERATURA

Robert E. Bixby; Vašek Chvátal William J. Cook; David L. Applegate. *The Traveling Salesman Problem*. Princeton University Press, 2006.

Dorigo i Stützle. *Ant Colony Optimization Theory*. MIT Press, 2004.

David Pisinger; Ulrich Pferschy; Hans Kellerer. *Knapsack Problems*. Springer, 2004.

Daniele Vigo; Paolo Toth. *The Vehicle Routing Problem*. SIAM, 2002.

Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi. Metaheuristike.* 2013.

Primjena prirodom inspiriranih optimizacijskih algoritama na kombinatorne probleme

Sažetak

Kombinatorni su problemi danas prisutni u mnogim znanstvenim područjima. Njihovo rješavanje tradicionalnim algoritmima nije moguće zbog kombinatorne eksplozije, te se neprestano traže novi pristupi u rješavanju istih. Jedna skupina takvih algoritama su prirodom inspirirani optimizacijski algoritmi. Glavna ideja njihovog pristupa je oponašanje načina na koji se jedinke kreću u prirodi te preslikavanje istog u pretraživanju prostora za optimalnim ili dovoljno dobrim rješenjima. Cilj rada je analizirati prirodom inspirirane optimizacijske algoritme i njihovu primjenu na klasične kombinatorne probleme.

Ključne riječi: Algoritam, kombinatorni problem, optimalnost, prirodom inspirirani optimizacijski algoritmi

Application of nature-inspired optimization algorithms to combinatorial problems

Abstract

Combinatorial problems are present in many scientific fields today. Their solution by traditional algorithms is not possible due to the combinatorial explosion, and new approaches in solving them are constantly being sought. One group of such algorithms are nature-inspired optimization algorithms. The main idea of their approach is to imitate the way individuals move in nature and to map the same in searching for space for optimal or good enough solutions. The aim of this paper is to analyze nature - inspired optimization algorithms and their application to classical combinatorial problems.

Keywords: Algorithm, combinatorial problem, optimality, Nature-Inspired Optimization Algorithms