SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Stanko Smrček
Filip Šoštarić
Karlo Vardijan

Rješavanje kombinatornog problema metodom računalne inteligencije (bus driver scheduling) uz pomoć genetskog algoritma (GA)

Varaždin, 2024.

Sadržaj

Genetski algoritam (GA) i primjena u kombinatornim problemima

Genetski algoritam (GA) je heuristička metoda optimizacije koja oponaša prirodni evolucijski proces. Evolucija je robustan proces istraživanja prostora rješenja, pri čemu se živa bića prilagođavaju uvjetima svoje okoline. Ova analogija između evolucije kao prirodnog procesa i genetskog algoritma kao metode optimizacije očituje se kroz proces selekcije i genetske operatore. U evoluciji, prirodni uvjeti i okolina određuju selekciju živih vrsta. U genetskim algoritmima, funkcija cilja igra ključnu ulogu u selekciji, predstavljajući problem koji se rješava. Kao što su uvjeti u prirodi ključni za preživljavanje i reprodukciju najbolje prilagođenih jedinki, tako je i funkcija cilja ključna za odabir najboljih rješenja u populaciji genetskog algoritma. U ovom algoritmu, jedno rješenje predstavlja jednu jedinku. Selekcija odabire najbolje jedinke koje prelaze u sljedeću generaciju, dok se manipulacijom genetskog materijala stvaraju nove jedinke. Ovaj ciklus selekcije, reprodukcije i genetske manipulacije ponavlja se dok se ne postigne zadovoljavajući uvjet zaustavljanja. Konačni rezultat je populacija jedinki, pri čemu najbolje rješenje iz zadnje iteracije predstavlja konačno optimizirano rješenje. [1][2]

Rješavanje kombinatornog problema metodom računalne inteligencije uz pomoć genetskog algoritma sastoji se od nekoliko ključnih koraka. Prvi korak je stvaranje početne populacije potencijalnih rješenja koja se obično generira nasumično kako bi se osigurala raznolikost rješenja. Zatim se definira funkcija cilja koja procjenjuje kvalitetu svake jedinke u populaciji. U kontekstu kombinatornog problema, funkcija cilja kvantificira koliko je određeno rješenje uspješno. Na temelju vrijednosti funkcije cilja vrši se selekcija jedinki koje će sudjelovati u reprodukciji, pri čemu jedinke s boljim vrijednostima imaju veću vjerojatnost biti odabrane za sljedeću generaciju. Metode selekcije uključuju ruletsku selekciju, turnirsku selekciju i rang selekciju. Križanje tj. kombiniranje genetskog materijala dviju jedinki (roditelja), stvara nove jedinke (potomke), simulirajući biološki proces reprodukcije i omogućujući stvaranje potencijalno boljih rješenja. Postoji nekoliko metoda križanja, poput jednopointnog, dvopointnog i uniformnog križanja. Mutacija, koja nasumično mijenja dijelove jedinke, osigurava genetsku raznolikost unutar populacije i pomaže u izbjegavanju lokalnih optimuma, primjenjujući se s malom vjerojatnošću. Nakon što su novi potomci stvoreni, potrebno je odlučiti koje jedinke će činiti sljedeću generaciju; u nekim strategijama nova generacija potpuno zamjenjuje staru, dok u drugim postoji kombinacija starih i novih jedinki. Evolucijski proces se ponavlja kroz mnoge generacije dok se ne ispuni kriterij zaustavljanja, koji može biti unaprijed definiran broj generacija, zadovoljavajuća vrijednost funkcije cilja ili nedostatak poboljšanja tijekom određenog broja generacija. Kada se proces zaustavi,

najbolja jedinka iz posljednje generacije predstavlja optimalno rješenje kombinatornog problema. Primjena genetskog algoritma na kombinatorne probleme, poput problema trgovačkog putnika, problema rasporeda ili optimizacije mreže, pokazala se vrlo učinkovitom zbog njegove sposobnosti pretraživanja velikih prostora rješenja i pronalaženja približno optimalnih rješenja u razumnom vremenskom okviru. [3][4]

Problem raspoređivanja vozača autobusa (Bus Driver Scheduling Problem)

Problem raspoređivanja vozača autobusa predstavlja izazov u operacijskom istraživanju i optimizaciji, čiji je cilj učinkovito planiranje rasporeda rada vozača tako da se zadovolje svi operativni zahtjevi uz minimizaciju troškova i poštivanje ograničenja. Ključni aspekti ovog problema uključuju zadatke i smjene, gdje vozači moraju pokriti sve zadane smjene koje variraju u vremenu i ruti, te pravila rada koja obuhvaćaju maksimalne radne sate, minimalno vrijeme odmora i pauze. Troškovi su također kritični, pri čemu je glavni cilj minimizirati ukupne troškove rada, uključujući plaće, prekovremeni rad i dodatne vozače, dok se poštuju radna pravila. Raspoloživost vozača koja može uključivati njihove preferencije i ograničenja se mora uzeti u obzir kako bi se povećalo zadovoljstvo zaposlenika i smanjila fluktuacija radne snage. Raspored mora biti fleksibilan da bi se prilagodio nepredviđenim situacijama kao što su bolovanja, kvarovi vozila ili promjene u prometnom rasporedu. Ovaj problem koristi napredne metode optimizacije i računalne inteligencije za rješavanje što uključuje genetske algoritme, algoritme simuliranog kaljenja, tabu pretraživanje i linearno programiranje. Ovi algoritmi omogućuju učinkovito pretraživanje velikog prostora mogućih rješenja, uzimajući u obzir sva pravila i ograničenja, te pružaju praktična i primjenjiva rješenja za stvarne probleme raspoređivanja vozača autobusa. [5][6][7]

Set podataka za obradu

Za problem raspoređivanja vozača autobusa koristili smo set podataka preuzet s https://www.gpea.uem.br/benchmark.html. Set sadrži nekoliko tekstualnih datoteka od kojih svaka ima različit broj zapisa voznih redova autobusa. Za ovaj projekt koristili smo datoteku s najviše zapisa *V_TCCC2312.txt*. Kao što naziv govori, datoteka sadrži 2312 zapis. Primjer zapisa koji se nalaze u datoteci:

"008","G","T",5,30,20,1
"008","T","T",5,50,70,1
"008","T","T",7,0,70,1
"008","T","T",8,10,70,1
"008","T","T",9,20,85,1
"008","T","T",10,45,55,1
"008","T","T",11,40,70,1

Redom podaci u tablici su ID autobusne linije, polazište, odredište, sat polaska, minuta polaska, trajanje putovanja u minutama te ID vozila. Svi podaci su poredani po autobusnoj liniji i po vremenu. Gotovo u svim zapisima nema vremenskih razmaka (pauzi) između intervala te se rijetko kad dešava da više vozila vozi na istoj liniji u isto vrijeme što čini naš zadatak optimizacije problematičnim. Nema navedene ni informacije o troškovima prijevoza ni vozačima.

Uzimajući sve to u obzir, odlučili smo napraviti naša vlastita ograničenja za ovaj skup podataka kako bi uopće imalo smisla raditi optimizaciju genetskim algoritmom.

Programsko Rješenje

GitHub repozitorij programskog rješenja: https://github.com/kvardijan/BusDriverSchedulium

Genetski algoritam

```
def genetic_algorithm(tasks, population_size=10, mutation_rate=0.01,
num_generations=1000):
    population = [generate_random_schedule(tasks) for _ in
range(population_size)]

    for generation in range(num_generations):
        fitnesses = [1 / (calculate_cost(individual) + 1) for individual in
population]  # Avoid division by zero
        new_population = []

        for _ in range(population_size // 2):
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            new_population.append(mutate(child1, mutation_rate))
            new_population.append(mutate(child2, mutation_rate))

        population = new_population

    best_schedule = min(population, key=calculate_cost)
    return best_schedule
```

Generira se početna populacija slučajnih rasporeda zadataka, koja je u našem slučaju deset rasporeda. Svaki raspored je potencijalno rješenje problema. Zatim algoritam prolazi kroz 1000 generacija gdje se za svaku generaciju računa fitness svakog rješenja u populaciji. Selekcija roditelja se odvija tako da se bira iz populacije moguće rješenje prema normalnoj distribuciji. Moguće rješenje s boljim fitnessom ima veću vjerojatnost da se izabere. Dva odabrana roditelja se križaju uz pomoć križanja po ciklusima te stvaraju dva nova rješenja. Svako novo rješenje prolazi kroz mutaciju s određenom vjerojatnošću (0.01). Mutacija mijenja dio rješenja kako bi se održala genetska raznolikost. Zatim se djeca dodaju u populaciju i formira se nova generacija koja postaje trenutna populacija. Na kraju najbolje rješenje (ono s najnižim troškom) se vraća kao optimalan raspored zadatka. Ovo rješenje ne mora biti optimalno nego je samo najbolje rješenje koje je pokretanje genetskog algoritma uspjelo stvoriti. U prvim verzijama cjelokupna populacija su nam bile sve moguće rute. Finalno rješenje nam nije imalo smisla pa smo se odlučili da ćemo provoditi poseban genetski algoritam nad svakom rutom kako bi dobili optimalan raspored po ruti pošto rute nisu povezane jedna s drugom. Također smo i zbog tog razloga odabrali malen broj početne populacije (deset) pošto provodimo 1000 iteracija a nema puno redaka po pojedinoj liniji.

Izračun cijena/kazni

Pošto naš set podataka nema unutar sebe definirane nikakve cijene/troškove implementirali smo vlastiti sustav vrednovanja rasporeda koji uzima u obzir vrijeme čekanja, preklapanja rasporeda i dodatne troškove ako vozač vozi više od 4 sata kontinuirano na istoj liniji. Najstrože troškove smo dodijelili preklapanju rasporeda (bus dolazi u 13:15 a sljedeći je krenuo u 13:10) s 10 jedinica troškova po minuti. Vožnje više od 4 sata kontinuirano su kažnjene s 5 jedinica troškova po minuti prijestupa dok vrijeme čekanja s jednom jedinicom troškova po minuti. Ovakav raspored troškova potiče da genetski algoritam evoluira raspored koji nema preklapanje rasporeda i ima smanjeno čekanje uz to da se pridržava vanjskih regulacija za vožnju.

Funkcije križanja

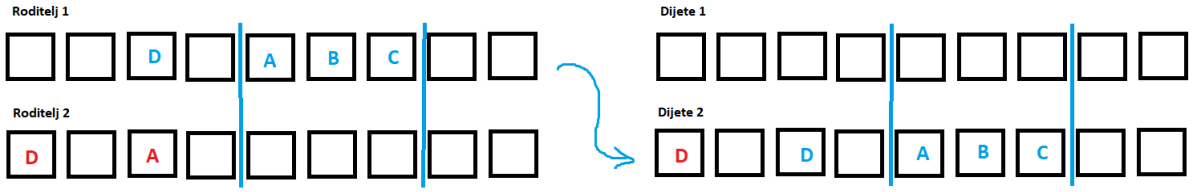
Nakon odabira roditelja, akcija koja se izvršava za samo stvaranje djece tj. potomaka koji će služiti kao evolucijski napredak odvija se uz pomoć funkcija križanja. Za implementirano programsko rješenje pokušale su se koristiti tri različite metode križanja.

Isprva se koristila metoda križanja u jednoj točki.

```
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2
```

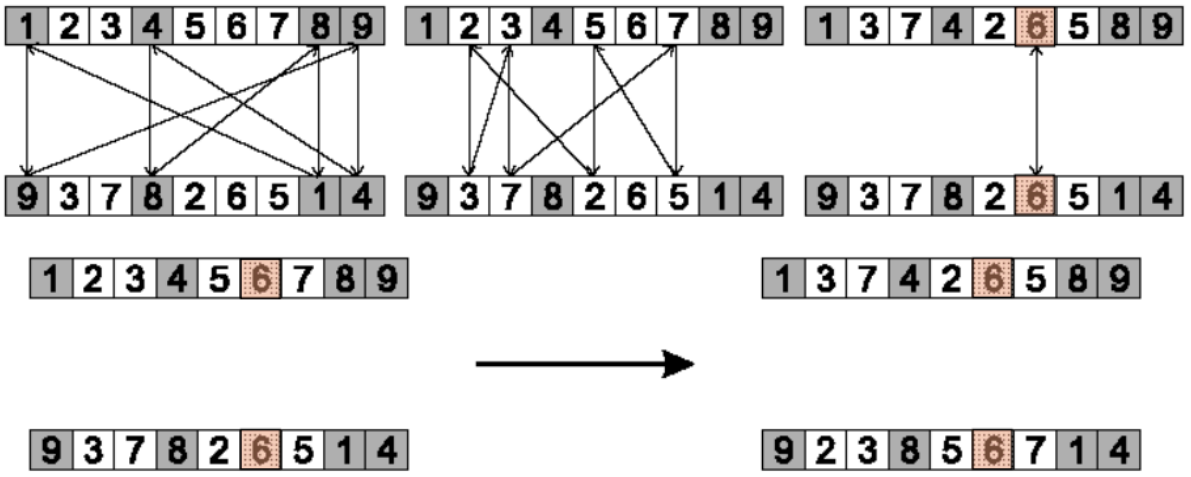
Ovakvim križanjem smo počeli pošto je bilo najjednostavnije za implementirati. Ali odbacili smo ideju brzo je smo naišli na problem da bi se pojavljivali duplikati zapisa jedne vozne linije te predloženo rješenje zatim ne bi imalo smisla. Ovo križanje radi tako da bi se odabrala jedna nasumična točka kojom bi zatim podijelili oba roditelja te križno spojili odsječene dijelove prvog roditelja i drugog kako bi dobili raznolikost rješenja.

Nakon toga se pokušala implementirati metoda redoslijednog jednostrukog križanja. Ideja je bila da se na nasumičan način odaberu indeksi u listi elemenata rasporeda za liniju prijevoza. Elementi između te dvije točke roditelja 1 bi se kopirali i stavili na ista mjesta u djetetu 2. Ostala polja bi se popunila s elementima drugog roditelja. Ako bi se utvrdilo u ovom koraku da se element već nalazi u djetetu iz prvog roditelja, odabrao bi se isti indeks u roditelju 1 te bi se tako razriješili duplikati. Postupak bi se ponovio za drugo dijete s drugim roditeljem. Problem je nastajao u situaciji opisanoj sljedećom skicom:



Ovdje se odabrani dio (A, B, C) iz roditelja 1 prebacio u dijete 2. Nakon toga se preostala polja pune s podacima iz roditelja 2. Prva vrijednost u roditelju 2 je vrijednost D. Ona se ne pojavljuje u već postojećem dijelu te se stoga normalno kopira. Nakon još jedne vrijednosti dolazi se do vrijednosti A koja se u roditelju 2 nalazi izvan granicama odabranog dijela. Algoritam u tom slučaju odabire vrijednost na istoj poziciji u roditelju 1 i kopira ju u dijete 2, no u ovoj situaciji to je vrijednost D koja je već prije bila kopirana iz roditelja 2. Na ovakav način bilo je moguće dobiti duplikatne zapise u djeci što nema smisla za redoslijed autobusnih linija. Također nakon *n* broja iteracija u kojima bi se to desilo bi nova djeca imala sve više i više duplikata u sebi. Mogla se implementirati provjera, no nije bilo definirano što bi se upisalo umjesto duplikatne vrijednosti (u ovom slučaju plavog slova D) u dijete. Stoga se ova metoda križanja također odbacila.

Finalna metoda odabrana je metoda križanja po ciklusima kroz koju je nemoguće dobiti duplikatne vrijednosti u potomcima. Metoda funkcionira na sljedeći način:



Slika x: Metoda križanja po ciklusima (preuzeta s predavanja [8])

Metoda počine tako da se definiraju ciklusi između roditelja te se onda vrijednosti svakog ciklusa naizmjenice kopiraju u djecu. Počinje se od prvog indeksa u roditelju 1 te se "spušta" u roditelja 2, označi se vrijednost te se ta vrijednost traži natrag u roditelju 1. Postupak se ponavlja dok se ne dođe opet do početnog indeksa. To tvori jedan ciklus. Kada se odaberu svi ciklusi i naizmjenice po ciklusima se prebace vrijednosti u djecu može se

krenuti u postupak mutacije. Križanje po ciklusima je u programskom rješenju implementirano na sljedeći način:

```
def cycle_crossover(parent1, parent2):
    child1_locations = defaultdict(BusTask)
    child2_locations = defaultdict(BusTask)

    iskoristeni_indexi = []
    parni_ciklus = False
    while len(iskoristeni_indexi) < len(parent1):  # cjelo krizanje
        index = nadi_prvi_slobodni_index(iskoristeni_indexi, parent1)
        ciklus_indexi = []
        while True:
            ciklus_indexi.append(index)
            iskoristeni_indexi.append(index)
            if parni_ciklus:
                child1_locations[index] = parent2[index]
                child2_locations[index] = parent1[index]
            else:
                child1_locations[index] = parent1[index]
                child2_locations[index] = parent2[index]
            index = nadi_sljedeci_index(parent1, parent2[index])
            if provjeri_index_ciklusa(ciklus_indexi, index):
                break
        parni_ciklus = not parni_ciklus

    child1 = [None] * len(parent1)
    child2 = [None] * len(parent1)
    for route, taskindictionary in child1_locations.items():
        child1[route] = taskindictionary
    for route, taskindictionary in child2_locations.items():
        child2[route] = taskindictionary

    return child1, child2
```

U ovom algoritmu napravi se polje u koje se spremaju svi indeksi koji su posjećeni ciklusima te se postupak ponavlja dok se ne posjete svi indeksi. Funkcija *nadi_prvi_slobodni_index* traži prvi indeks u roditelju 1 koji još nije posjećen. Tu se također definira polje indeksa koji su posjećeni u trenutnom ciklusu kako bi znali kada prekinuti ciklus. Varijabla *parni_ciklus* programu govori u koje dijete se upisuju elementi. Beskonačna petlja je glavni dio koji vrši samo pomicanje kroz elemente. Indeksi se spremaju u spomenuta polja te se u djecu kopiraju vrijednosti iz roditelja za trenutni index. Nakon toga se odabire novi indeks i provjerava se je li program stigao do kraja ciklusa. U slučaju da je izlazi se iz beskonačne petlje ciklusa. Koristi se struktura podataka *dictionary* za varijable *child_locationa* jer se mogu u njega upisati indeks kao *key* i sam element kao *value* koji se na kraju funkcije pune u polja i vraćaju.

Mutacija

Mutacijom novostvoreni potomci populacije imaju vjerojatnost napraviti nasumične promjene u svom genotipu (podacima u programskom slučaju) te na taj način povećati raznolikost populacije za veću vjerojatnost dobivanja optimalnog rješenja. Loša strana toga je što treba duže da se zapravo dođe do optimalnog rješenja ako je vjerojatnost mutacije veća. U ovom programskom rješenju mutacija je implementirana na sljedeći način:

```
def mutate(individual, mutation_rate):
    if random.random() < mutation_rate:
        mutation_point1 = random.randint(0, len(individual) - 1)
        mutation_point2 = random.randint(0, len(individual) - 1)
        individual[mutation_point1], individual[mutation_point2] =
      individual[mutation_point2], individual[mutation_point1]
    return individual
```

U funkciju mutacije prosljeđuje se individualni element i vjerojatnost da taj element mutira. Program određuje nasumični broj te ako je taj broj manji od vjerojatnosti mutacije onda se odabiru dvije točke mutacije (slično kao i kod redoslijednog jednostrukog križanja) te se mijenja mjesta tih dvaju elemenata. U ovom konkretnom slučaju elementi su redovi iz seta podataka. To se naziva mutacijski operator zamjene (eng. *swap*). [8]

Odabir najboljeg rasporeda

Najbolji raspored biramo unutar funkcije “genetic_algorithm”. Najbolji raspored je onaj koji ima minimalnu sumu troškova nad cijelim rasporedom. Funkcija “genetic_algorithm” na kraju svih iteracija vraća najbolji raspored tako da provjeri cijelu populaciju i odredi onaj s najmanjim troškovima. Na prvu izgleda kao da je rezultat samo nasumično izmiješani raspored, no on se zapravo optimalno prilagodio zadanim cijenama i ograničenjima. Sada kad imamo optimalno rješenje možemo odabrati neki dio koji sadrži podatke za određeni dan, na primjer:

BusTask(23, G, T, 6, 20, 40, 252)
BusTask(23, T, G, 7, 55, 20, 84)
BusTask(23, T, T, 12, 15, 60, 79)
BusTask(23, T, T, 18, 45, 60, 80)
BusTask(23, T, T, 22, 5, 65, 79)

Ovo je zapravo jedan od optimalnih rasporeda autobusa kroz dan te možemo po potrebi dodavati u njega autobuse u neka različita vremena što će malo utjecati na optimalnosti, no još uvijek je bolje nego početni raspored.

Zaključak

Genetski algoritam (GA) pokazuje se kao moćan alat za rješavanje kombinatornih problema, uključujući i problem raspoređivanja vozača autobusa (Bus Driver Scheduling Problem). Ovaj algoritam koristi principe evolucije, kao što su selekcija, križanje i mutacija, kako bi pronašao optimalna ili približno optimalna rješenja unutar velikih prostora mogućih rješenja. Primjena GA na problem raspoređivanja vozača autobusa uključivala je nekoliko ključnih koraka: stvaranje početne populacije, definiranje funkcije cilja, selekciju, križanje, mutaciju i generiranje novih generacija. Kroz iterativni proces, algoritam je bio u stanju pretraživati prostor rješenja i poboljšavati raspored kroz svaku generaciju. Tijekom implementacije, naišli smo na izazove s različitim metodama križanja, posebno kada je riječ o duplikatima i nepotpunim rasporedima. Na kraju smo se odlučili za metodu križanja po ciklusima, koja je osigurala jedinstvene vrijednosti u potomcima i omogućila stabilniji evolucijski proces. Mutacija je dodatno povećala genetsku raznolikost unutar populacije, što je pomoglo u izbjegavanju lokalnih optimuma. Izračun troškova i kazni bio je ključan za usmjeravanje evolucijskog procesa prema boljim rješenjima. Naš sustav vrednovanja uzimao je u obzir vrijeme čekanja, preklapanje rasporeda i kontinuiranu vožnju, s ciljem minimiziranja ukupnih troškova i poštivanja radnih pravila. Ovaj pristup omogućio je da algoritam evoluira prema rješenjima koja su praktična i primjenjiva u stvarnom svijetu.

Genetski algoritam, iako heuristički i stohastički, pokazao se kao efikasan alat za optimizaciju problema raspoređivanja vozača autobusa. Iako rješenje možda nije uvijek globalno optimalno, rezultati su dovoljno dobri za praktičnu primjenu, pružajući balans između kvalitete rješenja i vremena potrebnog za njegovu izradu. Ovaj rad pokazuje da GA, uz odgovarajuće prilagodbe i podešavanja, može biti vrlo učinkovit u rješavanju složenih kombinatornih problema u raznim domenama.

Literatura

[1] Marin Golub. 12.7.2022. “Genetski algoritam”, pristupano: 4.6.2024. https://www.zemris.fer.hr/~golub/ga/ga.html#:~:text=Genetski%20ili%20genetički%20algoritam%20(GA,životnoj%20okolini.

[2] Wikipedia. Bez dat. “Kombinatorika”, pristupano: 4.6.2024. https://hr.wikipedia.org/wiki/Kombinatorika

[3] Wikipedia. Bez dat. “Optimal job scheduling”, pristupano: 4.6.2024. https://en.wikipedia.org/wiki/Optimal_job_scheduling

[4] Uwe Schwiegelshohn. 2004. “Scheduling Problems and Solutions”, pristupano: 4.6.2024. https://web-static.stern.nyu.edu/om/faculty/pinedo/scheduling/sched.pdf

[5] Constantino, A. A., Mendonça, C. F. X., Araujo, S. A., Landa-Silva, D., Calvi, R. and Santos, A. F. 28.5.2017. “Solving a Large Real-world Bus Driver Scheduling Problem with a Multi-assignment based Heuristic Algorithm”, pristupano: 4.6.2024. https://www.gpea.uem.br/PaperBSP.pdf

[6] Wikipedia. Bez dat. “Driver scheduling problem”, pristupano: 4.6.2024. https://en.wikipedia.org/wiki/Driver_scheduling_problem

[7] Teresa Galvão Dias, Jorge Pinho de Sousa, João Falcão e Cunha. Ožujak, 2002. “Genetic algorithms for the Bus Driver Scheduling Problem: a case study”, pristupano: 4.6.2024. https://www.researchgate.net/publication/37649793_Genetic_algorithms_for_the_Bus_Driver_Scheduling_Problem_a_case_study

[8] Fakultet Organizacije i Informatike. Bez dat. “Inteligencija rojeva i evolucijsko računanje”, pristupano: 4.6.2024. https://elf.foi.hr/pluginfile.php/229049/mod_resource/content/9/02%20Inteligencija%20rojeva%20i%20evolucijsko%20računanje.pdf