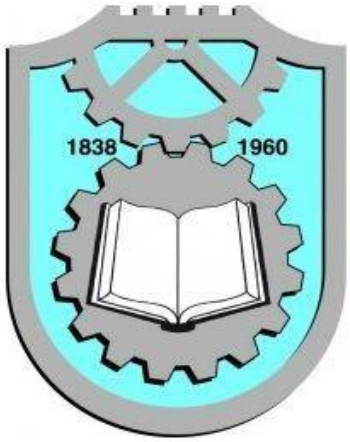
UNIVERZITET U KRAGUJEVCU
FAKULTET INŽENJERSKIH NAUKA



Veštačka inteligencija

Dokumentacija uz domaći zadatak

Problem trgovačkog putnika

Student:
Nikola Džajević

Profesori:
Dr Vesna Ranković
Tijana Šušteršič

Kragujevac, 2021.

Sadržaj

1. Uvod

Veliki broj problema kombinatorne optimizacije su NP-teški, a sa druge strane praktični kombinatorni problemi su obično i slabo struktuirani, nelinearni, višekriterijumski, što ih sve čini komplikovanim i za modeliranje i za rešavanje. Zbog toga je do sada razvijen čitav niz najraznorodnijih heurističkih metoda koje su prilagođene karakteristikama i strukturi specijalnih vrsta problema (koji rešavaju samo takve probleme).

Međutim, kasnije su počeli da se razvijaju i neki opšti heuristički pristupi rešavanju problema kombinatorne optimizacije. Ovi pristupi su se primenjivali još polovinom osamdesetih godina prošlog veka, a kasnije se njihova primena intenzivirala i proširila na čitav niz kako standardnih, tako i veoma složenih realnih kombinatornih problema. Veliki uspeh ovakvih opštih heuristika doveo je do ubrzanog usavršavanja njihovih osnovnih koncepata, kao i do daljih teorijskih istraživanja koje treba da objasne njihovu efikasnost.

Problem trgovačkog putnika spada u veliku klasu problema koji se nazivaju kombinatorni problemi optimizacije. Ovo je jedan od najviše proučavanih problema optimizacije, a njegova rešenja i primene će biti opisane u ovom radu. Korišćene metode rešavanja su:

1. Algoritam planinarenja (Hill climbing algorithm)
2. Genetički algoritam (Genetic algorithm)

2. Implementacija algoritama

2.1 Algoritam planinarenja (Hill climbing algorithm)

[1] Ovaj metod heurističke pretrage zasniva se na metodama isrpljujuće pretrage i pohlepne pretrage. Ime metode - planinarenje, potiče od izbora sledećeg čvora koji će se otvoriti. Temeljen je na logici uspona: algoritam počinje od nekog rešenja i ukoliko postoji sused koji bolje optimizuje funkciju, novo rešenje postaje taj sused. Naime, sledeći čvor koji se otvara ima maksimalnu vrednost heurističke funkcije. Kada se dođe do rešenja onda to odgovara vrhu.

Za neko međustanje koje može biti i početno, generišu se sva moguća stanja primenom svih operatora. Time se jednostavno eliminiše heuristička funkcija za operatore, pošto se primenjuju svi operatori na odabrani čvor Zatim se ispituje da li je neko od tih novih stanja možda rešenje. Ako jeste, to je kraj pretrage, a ako ne, na osnovu heurističke funkcije bira se sledeći čvor iz skupa novih generisanih čvorova

Opisana procedura se dalje ponavlja, sve dok se ne dođe do rešenja.

Sama funkcija može biti nedovoljno adekvatna, tako da vrednost za dati čvor ne mora uvek biti merodavna, jer možda ne uzima u obzir baš sve merodavne faktore za najperspektivniji čvor.

Može se desiti da funkcija daje istu vrednost za više različitih čvorova, u kom slučaju sedeći čvor koji treba otvoriti nije jednoznačno određen.

Algoritmi planinarenja nemaju načina da primete da se nalaze u lokalnom maksimumu.

Čvor algoritma za planinarenja ima dve komponente koje su stanje i vrednost.

Algoritam planinarenja uglavnom se koristi kada je na raspolaganju dobra heuristika.

Predeo stanja i prostora je grafički prikaz algoritma planinarenja koji prikazuje grafikon različitih stanja algoritma i ciljne funkcije / cene.

Na Y osi smo uzeli funkciju koja može biti objektivna funkcija ili funkcija troškova i prostor stanja na X osi. Ako je funkcija na osi Y trošak, cilj pretraživanja je pronaći globalni minimum i lokalni minimum.

Slika 1(grafik principa rada algoritma Hill Climbing)

Lokalni maksimum: Lokalni maksimum je grad koji je bolji od susednih gradova, ali postoji i drugi grad koji je bolji od njega.

Globalni maksimum: Globalni maksimum je najbolje moguće stanje grafika funkcije grada. Ima najveću vrednost ciljne funkcije.

Trenutno stanje: To je stanje u dijagramu gde je trgovac trenutno prisutan.

Ravni lokalni maksimum:To je ravni prostor u predelu gde sva susedna stanja trenutnih gradova imaju istu vrednost.

Plato: To je deo grafika koja ima ivicu.

```
def Solution(put):

    cities = list(range(len(put)))
    solution = []

    for i in range(len(put)):
        CityA = cities[0]
        solution.append(CityA)
        cities.remove(CityA)

    return solution

def routeLength(put, solution):

    routeLength = 0

    for i in range(len(solution)):
        routeLength += put[solution[i - 1]][solution[i]]
    return routeLength

def getNeighbours(solution):

    neighbours = []

    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(put, neighbours):

    bestRouteLength = routeLength(put, neighbours[0])
    bestNeighbour = neighbours[0]

    for neighbour in neighbours:
        currentRouteLength = routeLength(put, neighbour)
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength
```

Slika 2(realizacija algoritma Hill Climbing)

```
def hillClimbing(put):

    currentSolution = Solution(put)
    currentRouteLength = routeLength(put, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(put, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(put, neighbours)

    print("\nNajbolja ruta: ", currentSolution)
    print("Cena: ", currentRouteLength)

def main():

    put = [
        [0, 7, 6, 10, 13],
        [7, 0, 7, 10, 10],
        [6, 7, 0, 8, 9],
        [10, 10, 8, 0, 6],
        [13, 10, 9, 6, 0]
        ]

    print("Optimizacija puta trgovca: ", put)
    hillClimbing(put)

if __name__ == "__main__":
    main()
```

Slika 3(realizacija algoritma Hill Climbing)

2.2 Genetički algoritam (Genetic algorithm)

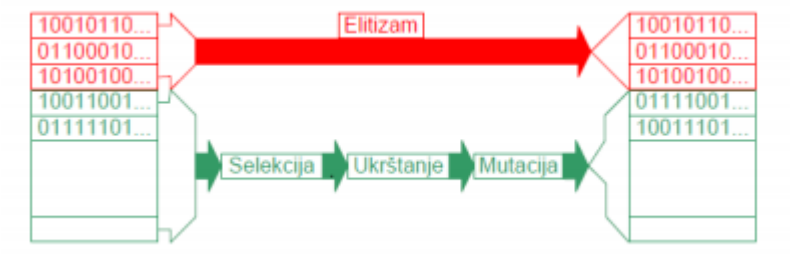
[4]Prve ideje o genetskim algoritmima izložene su u radu Holland-a [6], 1975. i javile su se u okviru tzv. teorije adaptivnih sistema, koja proučava modele efikasnog adaptivnog ponašanja nekih bioloških, specijalno genetskih, sistema. Ovakvi algoritmi su prvobitno kreirani da simuliraju proces genetske evolucije jedne populacije jedinki pod dejstvom okruženja i genetskih operatora. U ovom procesu je svaka jedinka okarakterisana hromozomom koji predstavlja njen genetski kod. One jedinke koje su u vedoj meri prilagođene okruženju, međusobno se dalje reprodukuju (primenom genetskih operatora na njihove hromozome) i tako se stvara nova generacija jedinki, prilagođenija od prethodne. Ovaj proces se ponavlja, pri čemu se iz generacije u generaciju prosečna prilagođenost članova populacije povećava.

Princip rada

1. Odabrani roditelji dobrih svojstava imaju šansu da daju potomke koji ce imati bolja svojstva od svakog pojedinačnog roditelja.
2. Roditelji dobrih svojstava imaju vedu šansu da daju potomke i prenesu svoja svojstva (gene) u iduću generaciju.
3. Svaka sledeća generacija će imati sve bolja svojstva

Elitizam

Pošto je GA ipak veštački i komandovani sistem "evolucije" jedne populacije u njegovom okviru je moguće izvršiti izbor određenog broja "najboljih" jedinki, i direktno ih preneti u narednu generaciju. Ovakav postupak se naziva elitizam, i grafički je prikazan na slici1



Slika 4(prikaz elitizma)

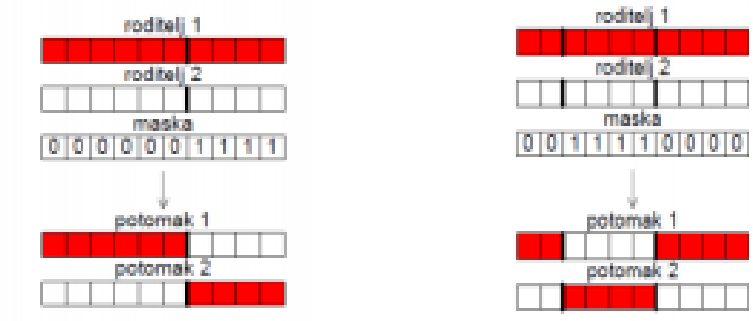
Selekcija

U osnovnoj verziji GA odgovara procesu selekcije u genetskim sisitemima, u kome se biraju one jedinke iz trenutne populacije koje su prilagođenije okruženju, da bi se njihovom reprodukcijom dobile jedinke nove generacije.

Operatori ukrštanja i mutacije

Operator ukrštanja se po ugledu na istoimeni genetski proces, definiše u najopštijem slučaju kao postupak u kome se slučajno uzajamno razmenjuju delovi kodova (hromozoma) dva rešenja (roditelja) i tako dobijaju kodovi 2 nova rešenja (dece).

Operator mutacije u opštem slučaju vrši, kao i odgovarajudi genetski proces, promenu sadržaja koda (hromozoma) nekog rešenja (jedinke) slučajnom zamenom pojedinih simbola ovog koda sa nekim drugim simbolima iz azbuke simbola. Ovakav operator se primenjuje na svako dete sa unapred zadatom verovatnodom mutacije (ova verovatnoda treba da bude veoma mala). Operator mutacije se koristi da bi se povremeno unela raznovrsnost među tačke jedne populacije (posebno u slučaju kada su ove tačke veoma slične), kao i da bi se sprečio neopravdan gubitak pojedinih simbola na nekim od pozicija kodova populacije.



Slika 5 (prikaz mutacije)

```
class Graph:

    def __init__(self, graph_struct = {}):

        self.graph = graph_struct

    def __str__(self):

        grh = ''
        for vrt in self.getVertices():
            for adj in self.getAdjacent(vrt):
                grh += '({0}, {1}, {2})\t'.format(vrt, adj, self.graph[vrt][adj])
        return grh

    def setVertex(self, vertex):

        if vertex not in self.graph.keys():
            self.graph[vertex] = {}
        return self

    def setAdjacent(self, vertex, adj, weight=0):

        if vertex not in self.graph.keys():
            self.graph[vertex] = {}
        if adj not in self.graph.keys():
            self.graph[adj] = {}

        self.graph[vertex][adj] = weight
        self.graph[adj][vertex] = weight
        return self

    def getVertices(self):

        return list(self.graph.keys())

    def getAdjacent(self, vertex):

        if vertex in self.graph.keys():
            return self.graph[vertex]

    def getPathCost(self, path):

        pathCost = 0
        for vrt, adj in zip(path, path[1:]):
            pathCost += self.graph[vrt][adj]
        return pathCost
```

Slika 6 (kreiranje klase Graph za učitavanje pozicija gradova i rastojanja između njih)

```
import numpy as np
import operator
import Graph
import math


class GeneticAlgorithm:

    def __init__(self, generations=100, population_size=10, tournamentSize=4, mutationRate=0.1, elitismRate=0.1):

        self.generations = generations
        self.population_size = population_size
        self.tournamentSize = tournamentSize
        self.mutationRate = mutationRate
        self.elitismRate = elitismRate

    def optimize(self, graph):

        population = self.__makePopulation(graph.getVertices())
        elitismOffset = math.ceil(self.population_size*self.elitismRate)

        print ("Optimizacija puta trgovca:\n{0}".format(graph))

        for generation in range(self.generations):
            print ("\nGeneracija: {0}".format(generation + 1))
            print ("Populacija: {0}".format(population))

            newPopulation = []
            fitness = self.__computeFitness(graph, population)
            print ("Cene puteva:    {0}".format(fitness))
            fittest = np.argmin(fitness)

            print ("Najbolja ruta: {0} ({1})".format(population[fittest], fitness[fittest]))

            if elitismOffset:
                elites = np.array(fitness).argsort()[:elitismOffset]
                [newPopulation.append(population[i]) for i in elites]

            for gen in range(elitismOffset, self.population_size):
                parent1 = self.__tournamentSelection(graph, population)
                parent2 = self.__tournamentSelection(graph, population)
                offspring = self.__crossover(parent1, parent2)
                newPopulation.append(offspring)

            for gen in range(elitismOffset, self.population_size):
                newPopulation[gen] = self.__mutate(newPopulation[gen])

            population = newPopulation

            if self.__converged(population):
                break

        return (population[fittest], fitness[fittest])
```

Slika 7 (kreiranje klase GeneticAlgorithm i njenih metoda)

```
def __makePopulation(self, graph_nodes):

    return [''.join(v for v in np.random.permutation(graph_nodes)) for i in range(self.population_size)]


def __computeFitness(self, graph, population):

    return [graph.getPathCost(path) for path in population]


def __tournamentSelection(self, graph, population):

    tournament_contestants = np.random.choice(population, size=self.tournamentSize)
    tournament_contestants_fitness = self.__computeFitness(graph, tournament_contestants)
    return tournament_contestants[np.argmin(tournament_contestants_fitness)]


def __crossover(self, parent1, parent2):

    offspring = ['' for allele in range(len(parent1))]
    index_low, index_high = self.__computeLowHighIndexes(parent1)

    offspring[index_low:index_high+1] = list(parent1)[index_low:index_high+1]
    offspring_available_index = list(range(0, index_low)) + list(range(index_high+1, len(parent1)))
    for allele in parent2:
        if '' not in offspring:
            break
        if allele not in offspring:
            offspring[offspring_available_index.pop(0)] = allele
    return ''.join(v for v in offspring)


def __mutate(self, genome):

    if np.random.random() < self.mutationRate:
        index_low, index_high = self.__computeLowHighIndexes(genome)
        return self.__swap(index_low, index_high, genome)
    else:
        return genome
```

Slika 8 (kreiranje klase GeneticAlgorithm i njenih metoda)

```
    def __mutate(self, genome):

        if np.random.random() < self.mutationRate:
            index_low, index_high = self.__computeLowHighIndexes(genome)
            return self.__swap(index_low, index_high, genome)
        else:
            return genome


    def __computeLowHighIndexes(self, string):

        index_low = np.random.randint(0, len(string)-1)
        index_high = np.random.randint(index_low+1, len(string))
        while index_high - index_low > math.ceil(len(string)//2):

            try:
                index_low = np.random.randint(0, len(string))
                index_high = np.random.randint(index_low+1, len(string))
            except ValueError:
                pass
        return (index_low, index_high)


    def __swap(self, index_low, index_high, string):

        string = list(string)
        string[index_low], string[index_high] = string[index_high], string[index_low]
        return ''.join(string)


    def __converged(self, population):

        return all(genome == population[0] for genome in population)

if __name__ == '__main__':

    graph = Graph.Graph()
    graph.setAdjacent('a', 'b', 7)
    graph.setAdjacent('a', 'c', 6)
    graph.setAdjacent('a', 'd', 10)
    graph.setAdjacent('a', 'e', 13)
    graph.setAdjacent('b', 'c', 7)
    graph.setAdjacent('b', 'd', 10)
    graph.setAdjacent('b', 'e', 10)
    graph.setAdjacent('c', 'd', 8)
    graph.setAdjacent('c', 'e', 9)
    graph.setAdjacent('d', 'e', 6)


    GA = GeneticAlgorithm(generations=20, population_size=7, tournamentSize=2, mutationRate=0.2, elitismRate=0.1)

    optimal_path, path_cost = GA.optimize(graph)

    print ("\nNajbolja ruta: ", optimal_path)
    print ("Cena: ", path_cost)
```

Slika 9 (kreiranje klase GeneticAlgorithm i njenih metoda)

2.3 Rezultati algoritama

Opisani algoritmi su dali zadovoljavajuće rezultate, međutim vidna je razlika u tačnosti, naime genetički algoritam daje bolji rezultat, dok je algoritam planinarenja brži.
Takođe tačnost zavisi i od problema kojem pristupamo.

Rezultati oba su prikazani na priloženim slikama.

```
Optimizacija puta trgovca:  [[0, 7, 6, 10, 13], [7, 0, 7, 10, 10], [6, 7, 0, 8, 9], [10, 10, 8, 0, 6], [13, 10, 9, 6, 0]]

Najbolja ruta:  [1, 0, 2, 3, 4]
Cena:  37
>>>
```

Slika 9 (rezultat pretrage Hill Climbing algoritma)

```
Optimizacija puta trgovca:
(a, b, 7)       (a, c, 6)       (a, d, 10)      (a, e, 13)      (b, a, 7)
(b, c, 7)       (b, d, 10)      (b, e, 10)      (c, a, 6)       (c, b, 7)
(c, d, 8)       (c, e, 9)       (d, a, 10)      (d, b, 10)      (d, c, 8)
(d, e, 6)       (e, a, 13)      (e, b, 10)      (e, c, 9)       (e, d, 6)


Generacija: 1
Populacija: ['decba', 'aedcb', 'cedab', 'ecabd', 'edacb', 'cedba', 'cedab']
Cene puteva:    [29, 34, 32, 32, 29, 32, 32]
Najbolja ruta: decba (29)

Generacija: 2
Populacija: ['decba', 'decba', 'cedab', 'decba', 'deacb', 'edacb', 'cedba']
Cene puteva:    [29, 29, 32, 29, 32, 29, 32]
Najbolja ruta: decba (29)

Generacija: 3
Populacija: ['decba', 'edabc', 'decba', 'decba', 'edcba', 'eadbc', 'debca']
Cene puteva:    [29, 30, 29, 29, 28, 40, 29]
Najbolja ruta: edcba (28)

Generacija: 4
Populacija: ['edcba', 'edcba', 'decba', 'edabc', 'edcba', 'edcba', 'decba']
Cene puteva:    [28, 28, 29, 30, 28, 28, 29]
Najbolja ruta: edcba (28)

Generacija: 5
Populacija: ['edcba', 'ebcda', 'edcba', 'edcba', 'ebcda', 'edcba', 'edcba']
Cene puteva:    [28, 35, 28, 28, 35, 28, 28]
Najbolja ruta: edcba (28)

Generacija: 6
Populacija: ['edcba', 'edcba', 'ecbda', 'edcba', 'edcba', 'edcba', 'edcba']
Cene puteva:    [28, 28, 36, 28, 28, 28, 28]
Najbolja ruta: edcba (28)

Najbolja ruta:  edcba
Cena:  28
>>>
```

Slika 10 (rezultat pretrage Genetskog algoritma)

3. Zaključak

Algoritam planinarenja dao je solidne rezultate, za kratko vreme, međutim moramo uzeti u obzir da je testiran na relativno malom problemu u smislu količine podataka koje obrađuje, jer manji broj ulaznih podataka olakšava proračune algoritma. Da je testiran na većem uzorku efikasnost bi bila daleko manja, više vremena bi mu bilo potrebno jer pretražuje najbolju rutu od tačke u kojoj se trenutno nalazi.

Na primeru problema trgovačkog putnika algoritam planinarenja pokazao je manju tačnost, a veću brzinu.

Poznato je da je za svaku vrstu problema potrebno dizajnirati poseban genetski algoritam ili je potrebno prilagoditi problem nekom već postojećem genetskom algoritmu. Evoluciju rešenja korišćenjem genetskog algoritma moguće je usmeravati podešavanjem parametara koje neki genetski algoritam može imati. Od parametrima zavisi koliko će se vremena potrošiti na evoluciju rešenja, kojom će se brzinom algoritam usmeravati prema potencijalnom optimalnom rešenju, hoce li pretraživati veći ili manji deo prostora rešenja, itd… Skup parametara koji za jedan genetski algoritam jedan problem daje kvalitetne rezultate, za neki drugi problem ne mora dati jednako kvalitetne rezultate. Glavni parametri su:

Veličina populacije – parametar koji direktno utiče na kvalitet dobijenih rešenja, ali isto tako i u zavisnosti od selekcije može imati veliki uticaj na dužinu izvođenja genetskog algoritma.

Broj generacija (iteracija) – parametar koji direktno utiče i na kvalitet dobijenih rešenja i na dužinu izvođenja genetskog algoritma. Svakako da veći broj generacija, daje veće vreme izvršavanja genetskog algoritmu da pronađe optimalnije rešenje zadatog problema. Međutim, kod dizajniranja genetskog algoritma za teže probleme, vreme se uzima kao jedan od važnih faktora. Naravno da se uvek pokušava da se pronađe optimalno rešenje za problem, ali je nekada cena za vreme koje bi trebalo genetskom algoritmu jednostavno prevelika, pa se zato pokušava pronaći zadovoljavajude rešenje u što kraćem vremenu izvršavanja. Zadovoljavajuće rešenje je ono rešenje koje je dovoljno blizu optimalnom, a za koje je potrebno mnogo manje vremena kako bi se do njega došlo.

Verovatnoća mutacije – jedan od najvažnijih parametara kod genetskih algoritama, jer direktno utiče na to da li ce doći kod određene jedinke ili kod odredenog bita unutar jedinke do mutacije, a sama mutacija, radi slučajne skokove algoritma po prostoru rešenja, što omogućuje izbegavanje zaglavljivanja algoritma u lokalnim optimumima i proširivanje područja pretrage na još neistražene delove prostora rešenja. Moguće je da se dobijena rešenja biti lošija od originalnih, ali zbog toga ta lošija rešenja imaju manju verovatnoću ulaska u dalje reprodukcije, čime se ostavlja prostor za napredovanje u pravom smeru. Genetski algoritam je dosta osetljiv na promene ovog parametra, pa je dosta bitno pažljivo odabrati njegove vrednosti.

Na primeru problema trgovačkog putnika genetički algoritam pokazao je veću tačnost, a manju brzinu.

Zaključak je da je genetički algoritam daleko precizniji i kompleksniji od algoritma planinarenja i da rezultati koje daje su tačniji na manjem, a i na većem uzorku.

Literatura

Web:

1 Algoritam planinarenja Preuzeto: Maj 15, 2021, - https://razno.sveznadar.info/4_AI/P-IH/30-PenjanjeHil.htm

2 Hill Climbing Algorithm in Artificial Intelligence Preuzeto: Maj 16, 2021, https://www.javatpoint.com/hill-climbing-algorithm-in-ai

3 KnowledgeE Preuzeto: Maj 16, 2021, - https://knepublishing.com/index.php/KnE-Social/article/view/1394/3208

4 Genetic Algorithm Preuzeto: Maj 23, 2021, https://www.sciencedirect.com/topics/engineering/genetic-algorithm

5 An effective method for solving multiple travelling salesman problem based on NSGA-II Preuzeto: Maj 20, 2021, https://www.tandfonline.com/doi/full/10.1080/21642583.2019.1674220

6 Holland J.H., Adaption in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, USA, 1975.s II Preuzeto: Maj 23, 2021