Optimizacija kolonijom mrava

Seminarski rad u okviru kursa
Naučno izračunavanje
Matematički fakultet

Vojkan Cvijović, Gorana Vučić
vojkan94@ymail.com , goranavucic94@gmail.com

11. septembar 2019

Sažetak

Sadržaj

1 Uvod

Algoritam optimizacije kolonijom mrava proučava ponašanje mravljih kolonija, u svrhu rešavanja optimizacionih problema, gde se kao najčešći problem javlja pronalaženje najkraćeg puta u zadatom grafu.

Pojam ACO (eng. *ant colony optimization*), kao rezultat istraživanja pristupa kombinatornoj optimizaciji, uveo je Marco Dorigo 1992. godine u svom doktoratu. Algoritam se na samom početku primenjivao na problem trgovačkog putnika, a kasnije se sama ideja proširivala i na neke druge probleme.

Algoritam ACO pripada klasi algoritama inteligencije roja (eng. *Swarm intelligence*). Naime akcije insekata se temelje na interakcijama među pojedincima. Iako te interakcije izgledaju primitivno, globalno gledano često daju dobre rezultate. Takvo kolektivno ponašanje se naziva inteligencija roja. Glavne prednosti jesu fleksibilnost, u smislu brzog prilagođavanja promenljivoj okolini, potom robusnost što predstavlja otpornost na manja odstupanja, kao i sposobnost funkcionisanja bez nadzora. Optimizacija pomoću kolonije mrava je algoritam koji se koristi za nalaženje optimalnih putanja u potpuno povezanom grafu.

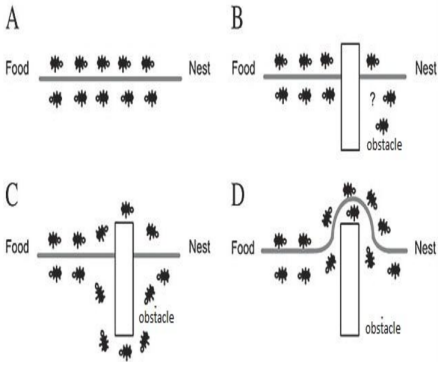
2 Ponašanje mrava

Kako je navedeni algoritam nastao po uzoru na ponašanje iz prirode, neophodno je objasniti sam princip rada na stvarnom ponašanju. Ukoliko postavimo mrave u neku nepoznatu okolinu, kako bi pronašli izvor hrane, oni će se u početku nasumično kretati. Kada neki mrav pronađe hranu, on će pri povratku u svoju koloniju ostavljati trag feromona koji će privlačiti druge mrave u smeru hrane. Na taj način će i ostali mravi koji su se u početku nasumično kretali početi kretati u smeru traga feromona. Svaki mrav koji se priključi pronalasku hrane, pojačavaće trag feromona i na taj način će privlačiti još mrava.

Nakon određenog vremena počinje da se odvija proces evaporacije odnosno dolazi do isparavanja feromona, što je od koristi s obzirom da se pozicija hrane stalno menja. Odnosno nije poželjno da postoji trag feromona koji navodi ostale mrave da idu u smeru gde se hrana nalazila pre određenog vremena, a sada više ne postoji. Ali ovo ukazuje i na to da ukoliko pretpostavimo da postoje 2 izvora hrane, jedan koji se nalazi bliže i drugi koji je udaljeniji od same kolonije, ukoliko se svi mravi na samom početku kreću nasumično oba izvora hrane biće pronađena, i na taj način će mravi ostavljati trag koji usmerava ostale mrave u tom smeru. Međutim s obzirom da kraći put znači i kraće vreme prolaska, to govori da će trag manje da slabi jer će se brže obnavljati u odnosu na duži put. Za duži put je ipak neophodno duže vreme prolaska, što znači da će više feromona oslabiti. Pošto mrave privlači jači trag feromona, više mrava će se odlučiti za kraći put. Neophodno je istaći da isparavanje feromona ima prednost u smislu da se na taj način izbegava konvergencija lokalno optimalnog rešenja, i na taj način se ostavlja prostora i za istraživanje širih prostora, na kojima možda postoji i bolji izvor hrane.

Na slici 1 je predstavljena situaciju u kojoj se vidi kako se mravi ponašaju u slučaju da na putu do hrane naiđu na prepreku. U samom

početku verovatnoća da odaberu levi i desni put je jednaka, stoga polovina mrava će ići levim a druga polovina desnim putem. S obzirom da je levi put kraći to znači da će na levom putu ostati jači trag feromona. Kako vreme odmiče i veći broj mrava prolazi tim putem, biće viši nivo feromona i na kraju će se čitava kolonija mrava kretati tim putem.



Slika 1: Ponašanje mrava u slučaju prepreke na putu do hrane

Sama ideja algoritma oponaša spomeutno ponašanje mrava. Naime, uvodimo pojam "mrava" koji će se kretati po grafu i tražiti najkraći put. Algoritam će biti objašnjen na problemu trgovačkog putnika. Ono što je prednost datog algoritma jeste upravo ta što se može prilagoditi dinamičkoj situaciji.

3 Primena ACO algoritma na TSP

3.1 Problem trgovačkog putnika

Problem trgovačkog putnika je jedan od najpoznatijih problema iz grupe NP - teških problema diskretne i kombinatorne optimizacije. Pojam "trgovački putnik" prvi put je upotrebljen 1932. godine. U osnovi se svodi na to da trgovački putnik tačno zna koje gradove treba da poseti, takođe zna i kolika je međusobna udaljenost među gradovima. Jedini problem je taj što je u obavezi da poseti svaki grad samo jednom i da se vrati u grad iz kog je pošao. Rešenje problema se ogleda u tome da trgovački putnik odredi redosled gradova i da pritom putuje najoptimalnijom mogućom rutom, odnosno najkraćim i najbržim putem. Kretanje od grada do grada se odvija istom brzinom, odnosno kraći put je brži put. Na prvi pogled ovaj problem ne izgleda teško, ali ako se uzme u obzir da ima faktorijalnu složenost, računanjem se dobija da već za 10 gradova posotji 3,628,800 mogućih kombinacija obilaska gradova.

3.2 ACO algoritam za TSP i parametri

U svakom koraku ACO algoritma, mrav se nalazi u nekom čvoru i i treba da odabere naredni čvor j u koji treba da pređe i to između onih čvorova koje na svom putu još uvek nije posetio. Verovatnoća preslaska u naredni čvor je određena rastojanjem između čvorova i i j kao i količinom feromona na grani koja ih spaja. Verovatnoća prelaska je data narednom formulom:

$$p_{i,j}^{k} = \frac{t_{i,j}^{\alpha} * distance_{i,j}^{-\beta}}{\sum_{l \in J_k} t_{i,j}^{\alpha} * distance_{i,j}^{-\beta}} \tag{1}$$

Ova formula predstavlja verovatnoću da će mrav k u narednoj iteraciji algoritma preći u čvor j iz čvora i.

Promenljiva $t_{i,j}^{\alpha}$ predstavlja količinu feromona između čvorova i i j, dok $distance_{i,j}^{-\beta}$ predstavlja takozvanu vidljivost čvora j iz čvora i i ona je definisana kao:

$$distance_{i,j} = \frac{1}{d_{i,j}} \tag{2}$$

gde $d_{i,j}$ predstavlja rastojanje između gradova i i j. Što je proizvod $t_{i,j} * distance_{i,j}$ veći to je veća i verovatnoća prelaska u grad j. Trag feromona i vidljivost podignuti su na stepene α i β respektivno. Ovi parametri kontrolišu uzajamnu bitnost informacija o tragu feromona $t_{i,j}$ nasuprot informacije o udaljenosti $distance_{i,j}$.

Nivo traga feromona mora da se ažurira nakon što svi mravi završe svoje kretanje kroz graf. Naredna formula predstavlja način na koji se trag feromona ažurira:

$$t_{i,j} = (1 - \rho) * t_{i,j} + \Delta t_{i,j} \tag{3}$$

Faktor ρ obezbeđuje da se feromon ne akumulira beskonačno i određuje količinu starog feromona koji će biti prenet u sledeću iteraciju algoritma.

$$\Delta t_{i,j} = \sum_{k=1}^{m} \Delta t_{i,j}^{k} \tag{4}$$

U ovoj formuli m predstavlja broj mrava, dok je $\Delta t_{i,j}^{k}$ količičina feromona koju je deponovao mrav k na poziciji između čvora i i j.

4 Implementacija

U samoj implementaciji korišćene su tri klase *Graph*, *ACS* i *Ant*.
Klasa *Graph* sadrži neke opšte informacije o gradovima kao što su:

- **distances**: predstavlja matricu rastojanja između gradova
- **rank**: predstavlja broj gradova, važi da između svaka dva grada postoji put
- **pheromone**: predstavlja matricu nivoa feromona između gradova

```
class Graph(object):
    def __init__(self, distances: list, rank: int):
        self.distances = distances
        self.rank = rank
        self.pheromone = [[1 / (rank * rank) for j in range(rank)] for i in range(rank)]
```

Slika 2: Klasa Graph

Klasa *ACS* (eng. *ant colony system*) je klasa koja se koristi za rešavanje problema putujućeg trgovca primenom ACO algoritma. Parametre koje sadrži su:

- **generations**: predstavlja broj iteracija samog algoritma

- **ant_count**: predstavlja broj mrava koji se kreira u svakoj iteraciji
- **alpha**: pozitivan parametar iz formule 7 koji određuje uticaj feromona pri određivanju narednog čvora u koji mrav treba da pređe
- **beta**: pozitivan parametar iz formule 7 koji određuje uticaj udaljenosti između gradova(čvorova) pri određivanju narednog čvora u koji mrav treba da pređe
- **rho**: pozitivan parametar iz formule 3 koji oređuje koja količina starog feromona se prenosi u narednu iteraciju algoritma
- **Q**: pozitivan parametar

```
class ACS(object):
    def __init__(self, ant_count: int, generations: int, alpha: float, beta: float, rho: float, q: int):
        self.generations = generations
        self.ant_count = ant_count
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = q
```

Slika 3: Klasa ACS

Ova klasa sadrži metode *update_pheromone* i *solve*. Metod *update_pheromone* vrši ažuriranje matrice feromona između gradova koristeći formule koje su objašnjene u 3.2.

$$t_{i,j} = (1 - \rho) * t_{i,j} + \Delta t_{i,j} \tag{5}$$

$$\Delta t_{i,j} = \sum_{k=1}^{m} \Delta t_{i,j}^{k} \tag{6}$$

Polje *i,j* matrice feromona se ažurira tako što se na već postojeću vrednost jačine feromona koja se umanjuje množeći se sa parametrom ρ, dodaje vrednost $\Delta t_{i,j}$ koja predstavlja sumu jačine feromona koju ostavlja svaki od mrava prolazeći tim putem.

```
def update_pheromone(self, graph: Graph, ants: list):
        for i, row in enumerate(graph.pheromone):
            for j, col in enumerate(row):
                sum_ants_pheromone_delta = 0
                for ant in ants:
                    sum_ants_pheromone_delta += ant.pheromone_delta[i][j]
                graph.pheromone[i][j] = (1 - self.rho) * graph.pheromone[i][j] + sum_ants_pheromone_delta

ACS.update_pheromone = update_pheromone
```

Slika 4: Metod koji vrši ažuriranje feromona

Metod *solve* je od ključnog značaja za rešavanje problema trgovačkog putnika. Ovaj metod pronalazi najbolje rešenje u smislu najkraćeg puta koje trgovac treba da obiđe. Kao povratna vrednost same funkcije vraća se najbolje rešenje koje predstavlja putanju, odnosno čvorove grafa kao i cenu, odnosno dužinu puta.

U prvoj petlji se prolazi kroz niz iteracija koja se zadaje kao parametar grafa. U svakoj iteraciji formiraju se instance mrava. Potom se za svakog mrava iz skupa mrava u petlji prolazi niz čvorova iz grafa pri čemu mrav svaki put posećuje čvor iz skupa neposećenih čvorova sve dok ne obiđe sve čvorove. Nakon završetka te petlje računa se ukupna cena pređenog puta za trenutno izabranog mrava. Ukoliko je trenutna ukupna cena manja od

najbolje, ažurira se najbolja cena, takođe ažurira se i redosled posećenih čvorova kako bi se znao put kojim je mrav prošao. Na samom kraju svaki mrav ažurira količinu feromona koju je proizveo krećući se datim grafom, taj podatak se koristi pri ažuriranju matrice feromona na nivou iteracije.

```
def solve(self, graph: Graph):
        best_cost = float('inf')
        best_solution = []
        for _ in range(self.generations):
            ants = [Ant(self, graph) for i in range(self.ant_count)]
            for ant in ants:
                for i in range(graph.rank - 1):
                    ant.select_next_node()
                ant.total_cost += graph.distances[ant.visited_nodes[-1]][ant.visited_nodes[0]]
                if ant.total_cost < best_cost:
                    best_cost = ant.total_cost
                    best_solution = [] + ant.visited_nodes
                ant.update_pheromone_delta()
            self.update_pheromone(graph, ants)
        return best_solution, best_cost

ACS.solve = solve
```

Slika 5: Metod solve

Klasa *Ant* koja predstavlja jednog mrava u sistemu sadrži sledeće podatke o mravu:

- **colony**: instanca klase (eng. *Ant Colony System*) u kome je mrav kreiran
- **graph**: instanca grafa koji mrav obilazi
- **total_cost**: predstavlja cenu puta koji je mrav prešao
- **visited_nodes**: niz čvorova grafa koje je mrav obišao
- **pheromone_delta**: veličina feromona koju je mrav proizveo
- **unvisited_nodes**: predstavlja neposećene čvorove u grafu
- **start_node**: početni čvor iz kog mrav polazi u obilazak grafa
- **visitied_nodes**: predstavlja niz čvorova grafa koje je mrav obišao
- **current_node**: indeks čvora koji mrav obilazi

```
class Ant(object):
    def __init__(self, acs: ACS, graph: Graph):
        self.colony = acs
        self.graph = graph
        self.total_cost = 0.0
        self.visited_nodes = []
        self.pheromone_delta = []
        self.unvisited_nodes = [i for i in range(graph.rank)]
        start_node = random.randint(0, graph.rank - 1)
        self.visited_nodes.append(start_node)
        self.current_node = start_node
        self.unvisited_nodes.remove(start_node)
```

Slika 6: Metod ant

Ova klasa sadrži i dve metode *select_next_node* i *update_pheromone_delta*. Metoda *select_next_node* određuje naredni čvor u koji mrav treba da pređe prema formuli koja je objašnjena u 3.2.

$$p_{i,j}^{k} = \frac{t_{i,j}^{\alpha} * distance_{i,j}^{-\beta}}{\sum_{l \in J_k} t_{i,j}^{\alpha} * distance_{i,j}^{-\beta}} \tag{7}$$

```
def select_next_node(self):
    denominator = 0
    for i in self.unvisited_nodes:
        denominator += self.graph.pheromone[self.current_node][i] ** self.colony.alpha * \
            self.graph.distances[self.current_node][i] ** (-1 * self.colony.beta )

    probabilities = [0 for i in range(self.graph.rank)]
    for i in range(self.graph.rank):
        try:
            self.unvisited_nodes.index(i)  # test if allowed list contains i
            probabilities[i] = self.graph.pheromone[self.current_node][i] ** self.colony.alpha * \
                self.graph.distances[self.current_node][i] ** (-1 * self.colony.beta) / denominator
        except ValueError:
            pass

    # select next node by probability roulette
    selected_node = 0
    rand = random.random()
    for i, probability in enumerate(probabilities):
        rand -= probability
        if rand <= 0:
            selected_node = i
            break

    self.unvisited_nodes.remove(selected_node)
    self.visited_nodes.append(selected_node)
    self.total_cost += self.graph.distances[self.current_node][selected_node]
    self.current_node = selected_node
```

Slika 7: Metod select_next_node

Metoda *update_pheromone_delta* vrši ažuriranje vrednosti feromona za datog mrava u matrici pheromone_delta.

```
def update_pheromone_delta(self):
    self.pheromone_delta = [[0 for j in range(self.graph.rank)] for i in range(self.graph.rank)]
    for _ in range(1, len(self.visited_nodes)):
        i = self.visited_nodes[_ - 1]
        j = self.visited_nodes[_]
        self.pheromone_delta[i][j] = self.colony.Q / self.total_cost
```

Slika 8: Metod update_pheromone_delta

4.1 Pokretanje

Funkcija kojom se pokreće izvršavanje programa je *find_optmal_path*, kojoj se prosleđuje ime fajla. U samoj funkciji isčitavaju se koordinate gradova i pravi se instanca funkcije *Graph* u kojoj se čuva matrica udaljenosti među gradovima. Takođe pravi se instanca klase *ACS* koja čuva informacije o broju iteracija algoritma, broju mrava, kao i ostalim neophodnim parametrima. Potom se poziva metod *solve* klase *ACS* koji vraća rezultat najbolje cene puta kao i optimalnu putanju. Ova funkcija vraća listu čvorova grafa odnosno gradove, potom optimalnu putanju, cenu kao i vreme izvršavanja *solve* funkcije.

Funkcija *draw_optimal_path* služi za iscrtavanje optimalne putanje. I rezultati funkcije su prikazani uz pomoć plot dijagrama na kome se nalaze gradovi i putanje između njih. Primer se može videti na slici 10

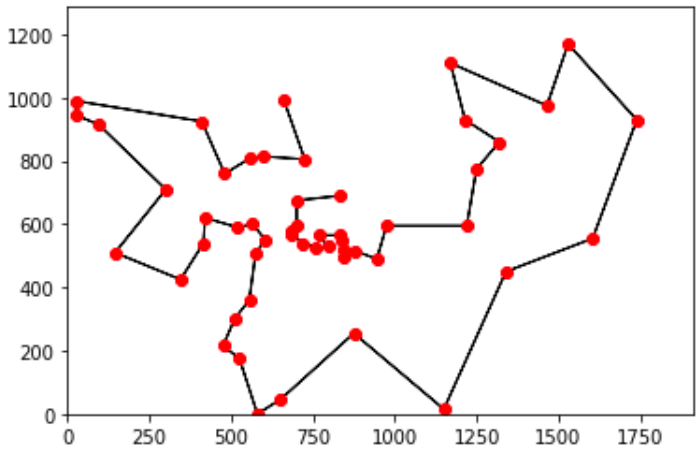
```python
def find_optimal_path(file_name):
    cities = []
    points = []
    with open("./data/{}".format(file_name)) as f:
        for line in f.readlines():
            city = line.split(' ')
            cities.append(dict(index=int(city[0]), x=int(city[1]), y=int(city[2])))
            points.append((int(city[1]), int(city[2])))
    distances = []
    number_of_cities = len(cities)
    for i in range(number_of_cities):
        row = []
        for j in range(number_of_cities):
            row.append(distance(cities[i], cities[j]))
        distances.append(row)

    acs = ACS(ant_count, generations, alpha, beta, rho, q)
    graph = Graph(distances, number_of_cities)
    start = time.time()
    path, cost = acs.solve(graph)
    end = time.time()

    time_execution = end - start

    print('COST: {},\nPATH: {}'.format(cost, path))
    print('TIME EXECUTION: {}'.format(time_execution))
    return points, path, cost, time_execution
```

Slika 9: Funkcija $find\_optmal\_path$



Slika 10: Primer izgleda optimalne putanje za skup podataka berlin52

5 Rezultati merenja

U ovom poglavlju biće izloženi rezultati merenja za različite promene parametara koji figurišu u ACO algoritmu.

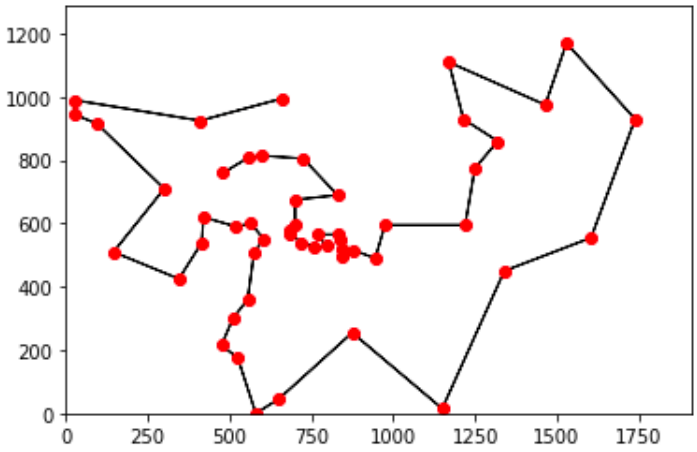
Kao skup podataka za koji se vrši merenje koristi se datoteka u kojoj se čuva informacija o 52 lokacije u Berlinu. Skup podataka se može pronaći na ovoj adresi. Optimalno rešenje za ovaj skup lokacija iznosi 7542.

Kao početni parametri za ACO algoritam uzete su sledeće vrednosti:

- $ant\_count = 10$

- $generations = 100$
- $alpha = 1.0$
- $beta = 10.0$
- $rho = 0.5$
- $q = 10$

Prosečno vreme izvršavanja za zadate parametre je 5.32s. Najbolje optimalno rešenje iznosi 7663.58 sa vremenom izvršavanja 5.56s. Optimalna putanja se može videti na slici 11



Slika 11: Optimalna putanja