PROIECT EXAMEN

Algoritmi metaeuristici și aplicații

Bancoș Ionela-Ștefania & Iura Alexandru-Petru (IIS-2)

4 februarie 2022

Nivel dificultate A

Problema Comis-Voiajorului rezolvată utilizând metodele Ant Colony Optimization și Simulated Annealing

1.Introducere

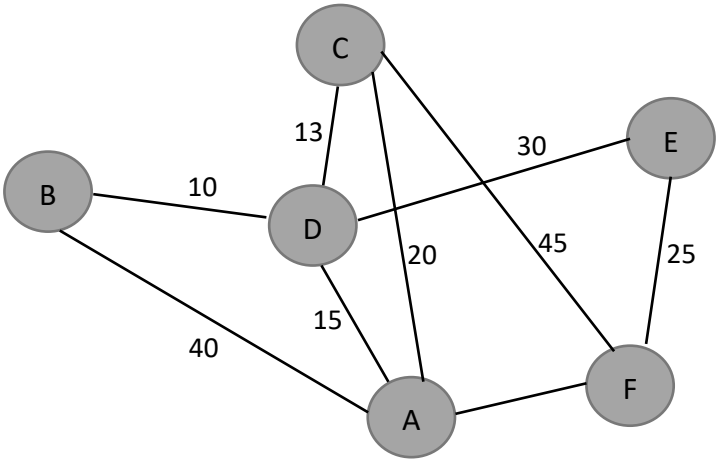
Proiectul de față urmărește rezolvarea Problemei Comis-Voiajorului(Traveling Salesman Problem) prin metoda Ant Colony Optimization și Simulated Annealing, precum si compararea celor două metaeuristici, având ca mediu de lucru Pycharn și limbajul de programare Python. Algoritmii meta-euristici sunt recunoscuți pe scară largă ca fiind una dintre cele mai practice abordări pentru problemele de optimizare combinatorie.

2.Descriere problemă

Fie G = (V, E) este un graf neorientat, cu V= setul de noduri și E=setul de laturi, în care oricare două vârfuri diferite ale grafului sunt unite printr-o latură căreia ii este asociat un cost strict pozitiv. Cerința este de a determina un ciclu care începe de la un nod aleatoriu a grafului, care trece exact o dată prin toate celelalte noduri și care se întoarce la nodul inițial, cu condiția ca ciclul sa aibă un cost minim. Costul unui ciclu este definit ca suma tuturor costurilor atașate laturilor ciclului.[1]

Numele problemei provine din analogia cu un vânzător ambulant care pleacă dintr-un oraș, care trebuie să viziteze un număr de orașe dat şi care apoi trebuie să se întoarcă la punctul de plecare, cu un efort minim (de exemplu timpul minim, caz în care costul fiecărei laturi este egal cu timpul necesar parcurgerii drumului).[2]

Model: se construiește graful complet G=(V, E)=Kn, n=|V|, în care nodurile reprezintă locațiile, iar muchiile au ca valoare distanța dintre locațiile capăt. Problema revine la determinarea unui ciclu hamiltonian de valoare optimă.



Într-un graf Kn există $\frac{(n-1)!}{2}$ cicluri hamiltoniene.

3.Algoritmi metaeuristici

3.1. Ant Colony Optimization

O furnică poate găsi cele mai scurte căi între sursele de hrană și cuib. În timp ce merg de la sursele de hrană la cuib și invers, furnicile depun pe sol o substanță numită feromon, formând o dâră de feromoni. Furnicile pot mirosi feromonul și, atunci când își aleg drumul, au tendința de a alege căile marcate de concentrații mai mari de feromoni. S-a demonstrat că acest comportament de urmărire a traseelor de feromoni utilizat de o colonie de furnici poate duce la apariția celor mai scurte trasee.[3]

Atunci când un obstacol întrerupe calea, furnicile încearcă să ocolească obstacolul alegând la întâmplare oricare dintre cele două căi. În cazul în care cele două căi care înconjoară obstacolul au lungimi diferite, mai multe furnici trec pe calea cea mai scurtă în timpul mișcării lor continue de pendulare între punctele de cuib, într-un anumit interval de timp. În timp ce fiecare furnică își marchează drumul cu feromoni, traseul mai scurt atrage mai multe concentrații de feromoni și, prin urmare, tot mai multe furnici aleg acest traseu. Această reacție conduce în cele din urmă la un stadiu în care întreaga colonie de furnici folosește calea cea mai scurtă. Există multe variante ale optimizării prin colonii de furnici aplicate la diferite probleme clasice. [3]

Sistemul furnicilor utilizează agenți simpli, numiți furnici, care construiesc iterativ soluții candidate la o problemă de optimizare combinatorie. Construirea soluției de către furnici este ghidată de trasee de feromoni și de informații euristice dependente de problemă. [3]

O furnică individuală construiește soluții candidate pornind de la o soluție goală și adăugând apoi iterativ componente ale soluției până când este generată o soluție candidată completă. Fiecare punct în care o furnică trebuie să decidă ce componentă de soluție să adauge la soluția sa parțială curentă se numește punct de alegere. [3]

După ce construcția soluției este finalizată, furnicile oferă feedback cu privire la soluțiile pe care le-au construit prin depunerea de feromoni pe componentele soluției pe care le-au folosit în soluția lor. Componentele soluției care fac parte din soluții mai bune sau care sunt folosite de mai multe furnici vor primi o cantitate mai mare de feromoni și, prin urmare, este mai probabil ca furnicile să le folosească în viitoarele iterații ale algoritmului. Pentru a evita blocarea căutării, de obicei, înainte ca traseele de feromoni să fie consolidate, toate traseele de feromoni sunt reduse cu un factor. [3]

Un exemplu din viața reală de problemă a unui agent de vânzări ambulant rezolvată cu ajutorul optimizării coloniei de furnici este problema găsirii celui mai scurt traseu pentru un camion de livrări care să livreze pachete la mai mulți clienți.

Datele de intrare ar fi locațiile clienților și distanțele dintre ei. Scopul este de a găsi cel mai scurt traseu posibil pentru ca camionul să livreze toate pachetele, vizitând fiecare client o singură dată și revenind la punctul de plecare.

Algoritmul de optimizare a coloniei de furnici poate fi utilizat pentru a rezolva această problemă prin simularea comportamentului furnicilor care caută hrană. Fiecare furnică reprezintă o soluție potențială, iar urma de feromoni lăsată de furnici ajută la ghidarea celorlalte furnici către cea mai bună soluție.

Algoritmul începe prin inițializarea matricei de feromoni și apoi prin crearea repetată de furnici, fiecare dintre acestea calculând un tur al orașelor pe baza unei combinații de feromoni și a distanțelor dintre orașe. După fiecare iterație, feromonii sunt actualizați pe baza calității tururilor generate de furnici, iar procesul continuă până când se găsește o soluție satisfăcătoare.

În final, cel mai scurt traseu găsit de algoritmul de optimizare a coloniei de furnici poate fi utilizat de camionul de livrări pentru a efectua livrările în cel mai eficient mod posibil.

Iată o implementare de bază a algoritmului Anto Colony Optimization pentru problema Comis-voiajorului în python:

```
import random
import numpy as np

def ant_colony_optimization(cities, num_ants, num_iterations, alpha, beta, rho):
    n = len(cities)
    tau = np.full((n, n), 1.0 / n)
    best_distance = float('inf')
    best_tour = []

    # repeta procesul de optimizare pentru un anumit numar de iteratii
    for iteration in range(num_iterations):
        tours = []
        # generare tur pentru fiecare furnica
        for ant in range(num_ants):
            tour = construct_solution(cities, tau, alpha, beta)
            tours.append((tour, get_distance(cities, tour)))

        # sorteaza turul pe baza lungimii lor
        tours.sort(key=lambda x: x[1])
        # updateaza cel mai bun tur gasit departe
        if tours[0][1] < best_distance:
            best_distance = tours[0][1]
            best_tour = tours[0][0]

        # updateaza nivelul de feromoni pe baza lungimii turului
        update_pheromones(tau, tours, n, rho)

    return best_tour, best_distance

def construct_solution(cities, tau, alpha, beta):
    n = len(cities)
    tour = []
    visited = [False] * n
    # alege random un oras de pornire
    current_city = random.randint(0, n - 1)
    tour.append(current_city)
    visited[current_city] = True

    # continua sa construiesti turul pana cand fiecare oras este vizitat
    while len(tour) < n:
        probabilities = [0.0] * n
        total = 0.0
        # calculaeaza probabilitatea vizitarii oricarui oras urmator
        for next_city in range(n):
            if not visited[next_city]:
                probabilities[next_city] = choose_next_city(current_city, next_city, tau, alpha, beta)
                total += probabilities[next_city]

        # normalizarea probabilitatilor daca totalul este mai mare decat zero
        if total > 0:
```

```
            for i in range(n):
                if visited[i]:
                    probabilities[i] = 0
                else:
                    probabilities[i] /= total

        # alege urmatorul oras pe baza probabilitatilor
        next_city = np.random.choice(n, p=probabilities)
        tour.append(next_city)
        visited[next_city] = True
        current_city = next_city

    return tour

def choose_next_city(current_city, next_city, tau, alpha, beta):
    pheromone = tau[current_city][next_city]
    distance = dist(cities[current_city], cities[next_city])
    return (pheromone ** alpha) * ((1.0 / distance) ** beta)

def update_pheromones(tau, tours, n, rho):
    # se scade nivelul de feromoni
    for i in range(n):
        for j in range(i + 1, n):
            tau
```

3.2. Simulated Annealing

Prin analogie cu procesul de recoacere a unui material, cum ar fi metalul sau sticla, prin ridicarea acestuia la o temperatură ridicată și apoi prin reducerea treptată a temperaturii, permițând regiunilor locale de ordine să crească spre exterior, crescând ductilitatea și reducând tensiunile din material, algoritmul perturbă aleatoriu calea originală într-o măsură descrescătoare, în funcție de o "temperatură" logică care scade treptat.[4]

În recoacerea simulată, echivalentul temperaturii este o măsură a caracterului aleatoriu prin care se fac modificări ale traiectoriei, căutând să o minimizeze. Atunci când temperatura este ridicată, se efectuează modificări aleatorii mai mari, evitându-se riscul de a rămâne blocat într-un minim local (care, de obicei, sunt mai multe într-o problemă tipică de tip "comis voiajor"), pentru ca apoi, pe măsură ce temperatura scade, să se orienteze către un minim aproape optim. Temperatura scade într-o serie de pași pe un program de descreștere exponențială în care, la fiecare pas, temperatura este de 0,9 ori mai mare decât cea din pasul anterior.[4]

Procesul de recoacere începe cu un traseu care enumeră pur și simplu toate orașele în ordinea în care pozițiile lor au fost selectate aleatoriu. La fiecare pas de temperatură, se efectuează un număr de transformări aleatorii ale căii. În primul rând, se selectează un segment al traseului, cu orașele de început și de sfârșit alese aleatoriu. Apoi, o monedă software este aruncată pentru a decide ce fel de transformare se va încerca: invers sau transport.[4]

În cazul în care apare invers, se generează o cale alternativă în care orașele din segmentul ales sunt inversate în ordinea vizitei. În cazul transportului, segmentul este decupat din poziția sa curentă în traseu și este introdus într-un punct ales aleatoriu în restul traseului. Se calculează apoi lungimea

traseului modificat și se compară cu traseul înainte de modificare, obținându-se o cantitate numită "diferența de cost". Dacă valoarea este negativă, traseul modificat este mai scurt decât traseul original și îl înlocuiește întotdeauna. În cazul în care există o creștere a costului, însă, exponențialul mărimii sale negative împărțit la temperatura curentă este comparat cu un număr aleatoriu distribuit uniform între 0 și 1 și, dacă este mai mare, calea modificată va fi utilizată chiar dacă a crescut costul. Rețineți că, inițial, când temperatura este ridicată, va exista o probabilitate mai mare de a face astfel de modificări, dar că, pe măsură ce temperatura scade, vor fi acceptate doar creșteri mai mici ale costului. Numărul total de modificări testate la fiecare nivel de temperatură este stabilit în mod arbitrar la de 100 de ori numărul de orașe din traseu, iar după ce se găsește un număr de zece ori mai mare de modificări care reduc lungimea traseului decât numărul de orașe, se reduce temperatura și se continuă căutarea. În cazul în care, după încercarea tuturor modificărilor potențiale la un anumit nivel de temperatură, nu se găsește nicio modificare care să reducă lungimea traseului, soluția este considerată "suficient de bună" și se afișează traseul rezultat.[4]

Problema vânzătorului ambulant este de obicei formulată în termeni de minimizare a lungimii traseului pentru a vizita toate orașele, dar procesul de recoacere simulată funcționează la fel de bine cu scopul de a maximiza lungimea itinerariului. Dacă obiectivul este schimbat de la a minimiza la a maximiza, funcția de cost optimizată va fi negativul lungimii traseului, ceea ce va duce la căutarea celui mai lung traseu.[4]

Un avantaj al algoritmului Simulated Annealing este faptul că poate produce soluții rezonabile.[5]

Ca dezavantaje regăsim următoarele: o programare slabă a temperaturii poate împiedica explorarea suficientă a spațiului de stare; și poate necesita unele experimente înainte de a funcționa bine. [5]

Ca precauții, este recomandat să reexecutați cum mai multe soluții inițiale(foarte) diferite. Experimentați întotdeauna cu diferite programe de temperature. Și mai ales, alegeți întotdeauna o temperature inițială care să asigure o probabilitate ridicată de acceptare a unui salt de costuri ridicate. [5]

Iată o implementare de bază a algoritmului Simulated Annealing pentru problema Comis-voiajorului în python:

```
import random
import math


def simulated_annealing(cities, initial_temperature, cooling_rate, stopping_temperature):
    n = len(cities)  # numarul de orase
    current_solution = random.sample(range(n), n)  # selectarea random a unei solutii de pornire
    best_solution = current_solution.copy()  # initializarea celei mai bune solutii ca solutia curenta
    current_distance = get_distance(cities, current_solution)  # calcularea distantei a solutiei curente
    best_distance = current_distance  # initializarea celei mai bune distante ca distanta curenta
    temperature = initial_temperature  # initializarea temperaturii cu temperatura curenta

    # se repeta urmatoarele pana cand temperatura scade sub temperatura de oprire
    while temperature > stopping_temperature:
        new_solution = get_neighbour(current_solution)  # genereaza o noua solutie prin perturbarea solutiei curente
        new_distance = get_distance(cities, new_solution)  # se calculeaza distanta noii solutii
        delta_distance = new_distance - current_distance  # calcularea modificarii distantei
```

```
        # in cazul in care noua solutie este mai buna (adica o distantă mai mica), se accepta
        if delta_distance < 0:
            current_solution = new_solution
            current_distance = new_distance
            if new_distance < best_distance:  # daca noua solutie este cea mai buna solutie de pana acum, se actualizeaza cea mai buna solutie si cea mai buna distanta
                best_solution = new_solution
                best_distance = new_distance
        # daca noua solutie nu este mai bunt, se accepta cu o anumita probabilitate
        else:
            # probabilitatea de a accepta o solutie mai proasta este proportionala cu temperatura si cu modificarea distantei
            # pe masura ce temperatura scade, scade probabilitatea de a accepta o solutie mai proasta
            if random.random() < math.exp(-delta_distance / temperature):
                current_solution = new_solution
                current_distance = new_distance

        temperature *= cooling_rate  # scaderea temperaturii prin rata de racire

    # returneaza cea mai buna solutie si cea mai buna distanta gasita
    return best_solution, best_distance


# genereaza o noua solutie prin inlocuirea a doua orașe selectate aleatoriu in solutia curenta
def get_neighbour(solution):
    n = len(solution)
    a = random.randint(0, n - 1)
    b = random.randint(0, n - 1)
    while a == b:  # ne asiguram ca a și b sunt orase diferite
        b = random.randint(0, n - 1)
    solution[a], solution[b] = solution[b], solution[a]
    return solution


# sa calculeze distanta totala a solutiei, avand in vedere lista de orase si ordinea in care acestea sunt vizitate
def get_distance(cities, solution):
    distance = 0
    n = len(solution)
    for i in range(n - 1):
        distance += dist(cities[solution[i]], cities[solution[i + 1]])
    distance += dist(cities[solution[-1]],
                     cities[solution[0]])  # se adauga distanta de la ultimul oraș la primul oras
    return distance
```

În această implementare, funcția simulated_annealing primește ca date de intrare o listă de orașe, temperatura inițială și rata de răcire. Funcția returnează cel mai bun tur și distanța acestuia. Funcția get_distance calculează distanța totală a unui tur, iar funcția get_neighbor generează o nouă soluție prin înlocuirea a două orașe selectate aleatoriu în turul curent. Funcția dist calculează distanța euclidiană dintre două orașe.

4.Comparare algoritmi

În cele ce urmează se va face o comparare, din mai multe puncta de vedere, între algoritmii Ant Colony Optimization și Simulated Annealing pentru rezolvarea problemei comis voiajorului.

Abordare:

- Ant Colony Optimization este un algoritm meta-euristic inspirat de comportamentul furnicilor. Acesta funcționează prin simularea comportamentului furnicilor care caută cea mai scurtă cale între colonia lor și sursele de hrană.
- Simulated Annealing este un algoritm meta-heuristic inspirat de procesul de recoacere în știința materialelor. Acesta funcționează prin imitarea procesului de răcire și încălzire a unui material pentru a găsi starea optimă a acestuia.

Compromisul explorare-exploatare:

- Ant Colony Optimization utilizează un echilibru între exploatare (urmând căi bine stabilite) și explorare (descoperirea de noi căi) pentru a găsi o soluție bună.
- Simulated Annealing utilizează un echilibru între explorare (încercarea de soluții noi, potențial mai proaste) și exploatare (concentrarea pe cea mai bună soluție găsită până în prezent) pentru a găsi soluția optimă.

Determinism:

- Ant Colony Optimization este un algoritm determinist, ceea ce înseamnă că va produce același rezultat de fiecare dată când este rulat cu aceleași intrări.
- Simulated Annealing este un algoritm stocastic, ceea ce înseamnă că poate produce rezultate diferite de fiecare dată când este rulat cu aceleași intrări datorită caracterului său aleatoriu.

Viteza de convergență:

- Ant Colony Optimization converge de obicei mai repede decât SA, dar convergența sa nu este garantată.
- Simulated Annealing are nevoie de mai mult timp pentru a converge decât ACO, dar este garantat să converge la optimul global dacă este rulat pentru o perioadă de timp suficient de lungă și cu o rată de răcire suficient de lentă.

Reglarea parametrilor:

- Ant Colony Optimization necesită o reglare atentă a parametrilor, cum ar fi rata de depunere a feromonilor, rata de evaporare a feromonilor și influența feromonilor asupra procesului decizional.
- Simulated Annealing necesită o reglare atentă a parametrilor, cum ar fi temperatura inițială, rata de răcire și temperatura de oprire.

Performanță:

Performanțele Ant Colony Optimization și Simulated Annealing depind de instanța problemei, de reglarea parametrilor algoritmului și de implementarea specifică. Ambii algoritmi s-au dovedit a fi eficienți în rezolvarea Traveling Ssalesman Problem, dar Ant Colony Optimization este adesea mai rapid și mai eficient. Cu toate acestea, Simulated Annealing este mai robust și poate găsi adesea soluții mai bune în cazurile în care Ant Colony Optimization eșuează.

5.Concluzii

În concluzie, atât Ant Colony Optimization, cât și Simulated Annealing sunt algoritmi puternici pentru rezolvarea Traveling Salesman Problem și au propriile puncte forte și puncte slabe. Alegerea între ei depinde de cerințele specifice ale problemei și de compromisul dorit între explorare și exploatare, viteza de convergență și robustețea.

Bibliografie

[1] Documentation/Reference/Traveling Salesman Problem – HeuristicLab
[2] Problema comis-voiajorului - Wikipedia
[3] Camelia-M. Pintea, Petrică C. Pop, Camelia Chira. The generalized traveling salesman problem solved with ant algorithms. Springer, 07 August 2017
[4] Simulated Annealing.The Travelling Salesman Problem
[5] The Traveling Salesman Problem (TSP)