Gherman Dan-Gabriel MSD2

Onu Stefan MOC2

**AEA - Tema 2**

Problema de rezolvat a fost TSPD. Deoarece această problemă este cunoscută a fi NP-hard, o abordare deterministă pentru această problemă ar oferi fie o soluție departe de soluția optimă, fie o soluție mult prea consumatoare de timp. Din acest motiv, am implementat 2 soluții euristice pentru a rezolva problema noastră: un algoritm genetic (GA) și un Ant Colony Optimization (ACO sau pur și simplu AC). O altă soluție implementată este utilizarea programării constrângerilor, dar soluțiile pe care ni le-a oferit au fost furnizate prea încet, așa că nu am comparat GA și ACO cu acesta.

Atât algoritmii genetici, cât și optimizările coloniilor de furnici sunt metaheuristici inspirate din natură și ambele roiesc în jurul noțiunii de individ (în ACO se mai numește și furnică). Acest individ este evaluat pe baza structurii sale și, ulterior, sunt utilizate diferite metode de căutare. Pentru a căuta soluția, trebuie să definiți structura individului; această structură este comună atât pentru GA, cât și pentru ACO:

**Structură**

* 𝑖𝑛𝑑𝑖𝑣𝑖𝑑𝑢𝑎𝑙 = (𝑝𝑒𝑟𝑚𝑢𝑡𝑎𝑡𝑖𝑜𝑛, 𝑑𝑟𝑜𝑛𝑒\_𝑣𝑖𝑠𝑖𝑡𝑠) = (𝑝, 𝑣)
* Numărul orașelor este 𝑛
* Permutare = 𝑝𝑒𝑟𝑚𝑢𝑡𝑎𝑡𝑖𝑜𝑛 = 𝑝:
  + o permutare a listei [1,. . . , 𝑛 - 1], adăugând un 0 la începutul și la sfârșitul permutării => 𝑝 = [0, 𝑖1, 𝑖2,. . . , 𝑖𝑛 − 1, 0], unde 𝑖𝑗 ∈ {1,…, 𝑛 - 1}, 𝑗 ≠ 𝑘 =>

𝑖𝑗 ≠ 𝑖𝑘, ∀𝑗, 𝑘 = 1̅̅, ̅𝑛̅̅̅ − ̅̅̅1̅

* Noduri vizitate de dronă = 𝑑𝑟𝑜𝑛𝑒\_𝑣𝑖𝑠𝑖𝑡𝑠 = 𝑣:
  + O listă de booleni. Pentru fiecare 𝑖 = ̅1̅, ̅𝑛̅̅̅ − ̅̅̅1̅, avem 𝑣𝑖 ∈ {0, 1} și avem 𝑣0 = 𝑣𝑛 = 0

=> 𝑣 = [0, 𝑣1, 𝑣2,… 𝑣𝑛 − 1, 0], unde 𝑣𝑖 ∈ {0,1}

* + Valoarea fiecărui element 𝑣𝑖 din această listă creează 4 cazuri, unde 𝑖 = ̅1̅, ̅𝑛̅̅̅ − ̅̅̅1̅:
    - Drona se află în camion și 𝑣𝑖 = 0: Drona nu face nimic în timp ce camionul trece de la 𝑝𝑖 − 1 la 𝑝𝑖.
    - Drona se află în camion și 𝑣𝑖 = 1: Drona trece de la 𝑝𝑖 − 1 la 𝑝𝑖, în timp ce camionul nu face nimic
    - Drona nu se află în camion și 𝑣𝑖 = 0: Ultima acțiune a dronei a fost livrarea unui colet; drona nu face nimic, în timp ce camionul merge de la 𝑝𝑖 − 1 la 𝑝𝑖
    - Drona nu se află în camion și 𝑣𝑖 = 1: Ultima acțiune a dronei a fost livrarea unui colet; drona și camionul se află la case diferite; ambii merg, de la locul lor actual, la 𝑝𝑖 (exact unul dintre ei va fi la 𝑝𝑖 − 1 înainte de a merge la 𝑝𝑖)  această listă începe și începe cu 0
  + practic, dacă există un număr par de 1 în [𝑣1, ..., 𝑣𝑖], atât drona cât și camionul se află în aceeași casă 𝑖; dacă există un număr impar de 1, înseamnă că drona și camionul sunt la case diferite
  + dacă numărul de valori 1 din întreaga listă este impar, camionul și drona se întâlnesc la nodul 0 (practic, dacă numerele de 1 sunt impare, ultima valoare a listei devine 1 în loc, deci numărul de 1 va fi întotdeauna chiar = camionul și drona se vor sincroniza întotdeauna)

Această structură ne oferă posibilitatea de a crea și calcula orice soluție pentru problema noastră TSPD. Acestea ne oferă următoarele:

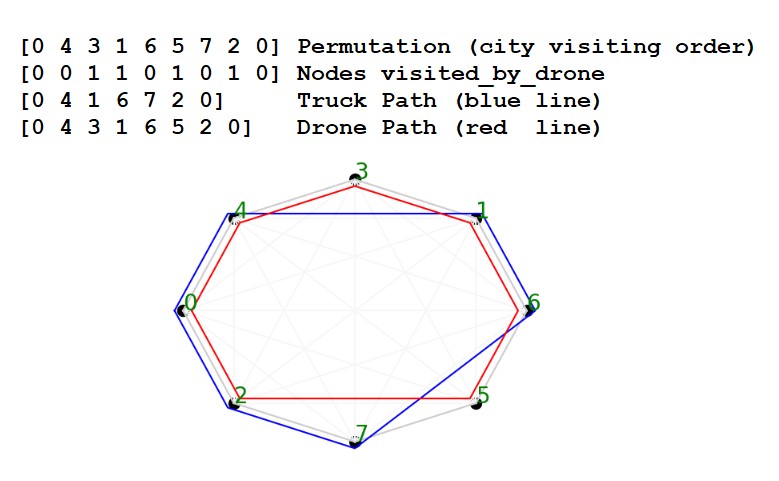
* Traseul camionului:
  + calea pe care o ia camionul path Calea dronei:
  + calea pe care o ia drona

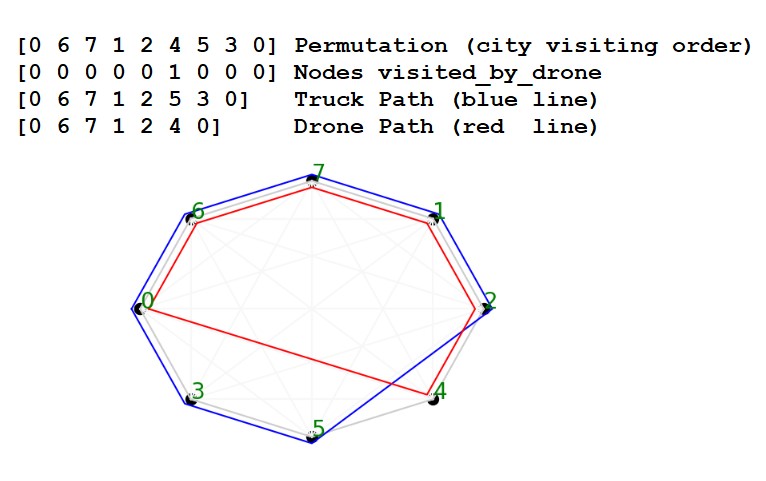
* Timpul total va fi

max (𝑙𝑒𝑛𝑔𝑡ℎ (𝑡𝑟𝑢𝑐𝑘\_𝑝𝑎𝑡ℎ),

𝑙𝑒𝑛𝑔𝑡ℎ (𝑑𝑟𝑜𝑛𝑒\_𝑝𝑎𝑡ℎ))

Pentru a vedea mai bine cum sunt definite toate acestea, observați următoarele 2 exemple din imaginile de mai jos:





# Algoritm genetic

Pentru algoritmul genetic, avem nevoie de două seturi separate de operații: un set pentru partea de permutare, un set pentru partea de vizitare a dronelor. Parametrii utilizați:

**POP\_SIZE = 200**

**N\_GENERAȚII =** 𝒊𝒏𝒕 (3000 / 𝒍𝒏 (n))

**PROBABILITY\_FOR\_INDIVIDUAL\_TO\_BE\_SELECTED\_FOR\_MUTATION = 5%**

**PROBABILITY\_FOR\_ELEMENT\_OF\_INDIVIDUAL\_PERMUTATION\_TO\_BE\_MUTATED = 1%**

**PROBABILITY\_FOR\_ELEMENT\_OF\_INDIVIDUAL\_DRONE\_VISIT\_TO\_BE\_MUTATED = 5%**

**PROBABILITY\_FOR\_CROSSOVER = 20%**

## Mutaţie

**Permutare:** mutația se face prin comutarea a 2 valori aleatorii = numere de oraș din permutație (mutație Shuffle Indexes).

**Vizite cu drone:** mutația se face luând în considerare valorile 1 și 0 ca fiind adevărate și false și răsturnând valoarea acesteia.

## *Crossover*

**Permutare:**

încrucișarea, cunoscută sub numele de Parțial Matched Crossover (PMX), este utilizată pe scară largă la permutari. Acest lucru este bine explicat in articolul: <https://www.researchgate.net/figure/Partially-mapped-crossover-operator-PMX_fig1_226665831>

**Vizite cu drone:** crossover-ul, cunoscut sub numele de crossover Two Point Cut, transfera continutul intre doua puncta de taiere selectate aleatoriu intre 2 indivizi.

## Selecţie

Selecția funcționează pe întregul individ, deci nu este separată pentru permutare și părțile de vizită ale dronelor. A fost utilizată selecția turneului, cu dimensiunea turneului = 3

## Funcția de evaluare

Aceasta este cea mai importantă parte a algoritmului genetic. Este calculat așa cum se specifică mai sus, în capitolul Structură.

# Optimizarea coloniilor de furnici

Pentru soluția ACO (Ant Colony Optimization), am implementat câteva mici variații ale acesteia. Întregul algoritm este descris pe scurt mai jos:

* Parametrii utilizați:

**POP\_SIZE = 10**

**N\_GENERAȚII =** 𝒊𝒏𝒕 (300 / 𝒍𝒏 (n))

**PHEROMONE\_DECAY = 0,1**

**CONSTANT\_HEUR = 2.5**

**CONSTANT\_LOCAL\_PHEROMONE = 0,1**

**CONSTANT\_GREED\_CHANCE = 0,7**

* Furnicile sunt inițializate cu structura descrisă mai sus
* O primă soluție inițială este aleasă dintre primele furnici și, pe baza costului structurii sale, un nivel inițial de feromoni este stabilit pe matricea feromonilor
* După inițializare, pentru fiecare generație, fiecare furnică încearcă euristic să găsească cel mai bun traseu. Pentru aceasta, la fiecare pas, se ia în considerare matricea nivelurilor de feromoni și cea mai bună opțiune care poate fi aleasa. Cu acestea calculate, cu o probabilitate egală cu CONSTANT\_GREED\_CHANCE alege cea mai bună opțiune menționată în prealabil și cu o probabilitate 1 - CONSTANT\_GREED\_CHANCE alege o opțiune aleatorie
* După fiecare soluție candidat găsită, matricea de feromoni este actualizată pe baza costului rutei acelei soluții (cunoscut sub numele de actualizare locală)
* După fiecare iterație, matricea de feromoni este actualizată pe baza celei mai bune soluții găsite până acum (cunoscută sub numele de actualizare globală)

# Implementare

Ambii algoritmi au fost implementați în Python 3.6.3 (Anaconda); deși nu a fost utilizată nici o sintaxă sau librarii specifice Python 3.6.3, ceea ce înseamnă că poate fi rulată pe multe alte versiuni Python. Implementarea folosește, de asemenea, librariile Python numpy == 1.17.0 și deap == 1.2.2 (numpy pentru calcule matrice rapide n-dimensionale și deap pentru operațiile Algoritmului genetic).

# Experimente computaționale

Implementările au fost executate local pe un desktop (procesor Intel i5-9400F - 6CPU 2,60GHz, 16 GB RAM). Ambele au fost rulate de 10 ori pe 5 seturi de date diferite generate aleatoriu, distanțele dintre orașe luând valori între 1 și 99. Seturile de date sunt:

* Foarte mic: 9 noduri
* Mic: 17 noduri
* Mediu: 42 noduri
* Mare: 60 de noduri
* Foarte mare: 563 noduri

Rezultatele statistice pot fi găsite în tabelul de mai jos (timpul este specificat în secunde):

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Algoritm | Set de date | In medie | In medie |  | Minim | Minim |  | Maxim | Maxim |  | STDDEV | STDDEV |
|  |  | cost | timp |  | cost | timp |  | cost | timp |  | cost | timp |
| ACO | Foarte mic | 170,7 | 1.12 |  | 164 | 1,06 |  | 177 | 1.2 |  | 5.22 | 0,04 |
| ACO | Mic | 197.2 | 3.53 |  | 178 | 3.17 |  | 210 | 4.5 |  | 10.07 | 0,42 |
| ACO | Mediu | 298 | 16.29 |  | 277 | 15.55 |  | 318 | 19.26 |  | 13.52 | 1,02 |
| ACO | Mare | 407.6 | 28,87 |  | 376 | 27,65 |  | 438 | 29,88 |  | 17.02 | 0,67 |
| ACO | Foarte larg | 2098.7 | 2178,8 |  | 1865 | 2158,49 |  | 2569 | 2258.03 |  | 213.07 | 27,63 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| GA | Foarte mic | 196 | 17.04 |  | 164 | 12.37 |  | 221 | 24.31 |  | 14,78 | 3,57 |
| GA | Mic | 246,7 | 13,64 |  | 222 | 12.23 |  | 267 | 17.56 |  | 13.48 | 1.6 |
| GA | Mediu | 733,8 | 17,78 |  | 667 | 16,61 |  | 831 | 19.46 |  | 50,21 | 0,81 |
| GA | Mare | 1284.3 | 21.21 |  | 1122 | 19,65 |  | 1430 | 22.57 |  | 101,52 | 0,74 |
| GA | Foarte larg | 22035.7 | 101,98 |  | 21144 | 98.08 |  | 22566 | 106,5 |  | 461.14 | 2.51 |

După cum se poate vedea, timpul necesar pentru a rula variază foarte mult, în funcție de dimensiunea matricei orașului, pentru implementarea ACO, în timp ce pentru implementarea GA nu. Acest lucru se datorează faptului că ACO caută, la fiecare pas, cel mai bun pas următor (partea lacomă) => complexitate pătratică, în timp ce GA își execută pașii la nesfârșit, fără a căuta nimic => complexitate liniară.

Privind funcțiile de evaluare (cost) pentru soluțiile găsite, se poate vedea că implementarea ACO găsește soluții mult mai bune decât GA, mai ales că dimensiunea matricei orașului crește. Acest lucru se datorează faptului că GA generează soluții aleatorii și încearcă să le îmbunătățească încetul cu încetul, nu neapărat solutii adecvate pentru funcțiile de cost pe bază de permutare; iar ACO caută lacom o soluție bună, deoarece ia în considerare soluțiile bune din trecut (bazate pe matricea feromonilor) și, de asemenea, posibilele noi soluții mai bune.

Căutarea aleatorie efectuată de GA poate fi văzută și în coloana Cost deviație standard, unde costurile soluțiilor găsite au o abatere standard mai mare decât costurile soluțiilor găsite de ACO. Pe de altă parte, observând coloana de timp a deviației standard, se poate observa că orice două execuții ale GA durează aproape exact în același timp, în comparație cu ACO, care depinde în mare măsură de soluțiile candidate anterioare găsite (cea mai bună soluție este la fel pentru multe iterații, iar costul acestei soluții este calculat o singură dată). (Execuțiile GA pentru seturile de date foarte mici și mici au fost primele care au fost rulate, deci timpul a fost influențat de alte operații pe mașină la acel moment => pentru aceste 2 timpul de deviere standard este foarte mare).

# Concluzie

Ambii algoritmi implementați pentru rezolvarea TSPD sunt viabile, robuste și ușor de utilizat. Au nevoie doar de numărul de orașe și de matricea distanței ca intrare.

Dacă lucrați cu matrice de distanță mică (să zicem, 𝑛 <50) ACO este mai bun decât GA. Pentru 𝑛> 50 ar trebui să ținem cont de timpul liber pe care îl avem: ACO oferă soluții mai bune decât GA (rute de 10 ori mai mici decât GA pentru setul de date foarte mare), dar trebuie sa ruleze mult mai mult timp (de 20 de ori mai mult) să parcurga setul de date foarte mare.