Inteligență artificială

###### TEMA 4

# Descrierea algoritmilor

### Algoritm Evolutiv pentru Ackley

# 

# 

# 

# 

# evolutiveAlgorithm(option:int) - Această funcție implementează algoritmul evolutiv, inițializând populația, creând descendenți și selectând supraviețuitorii. Aceasta primește un parametru opțional, opțiunea de crossover, pe care îl utilizează în funcția createOffspring. În timpul execuției, calculează valoarea fitness-ului pentru cea mai bună soluție și media valorilor fitness-ului pentru populație.

# initializePopulation() - Această funcție generează și întoarce o matrice bidimensională reprezentând populația inițială. Fiecare rând din matrice este o soluție validă generată aleatoriu.

# createOffspring(population:double[][], option:int) - Această funcție primește o matrice reprezentând populația curentă și opțiunea de crossover și returnează o nouă matrice care conține descendenții obținuți prin încrucișare și mutație.

# uniformCrossover(parent1:double[], parent2:double[]) - Această funcție primește doi părinți și returnează un copil obținut prin încrucișarea uniformă a genelor. Pentru fiecare genă, copilul preia gena de la unul dintre părinți în funcție de un număr generat aleatoriu.

# arithmeticCrossover(parent1:double[], parent2:double[]) - Această funcție primește doi părinți și returnează un copil obținut prin încrucișarea aritmetică a genelor. Pentru fiecare genă, copilul primește o combinație liniară a valorilor genelor părinților.

# mutate(individual:double[]) - Această funcție primește un individ și aplică o mutație aleatoare la gena individului, cu o probabilitate dată de MUTATION\_RATE.

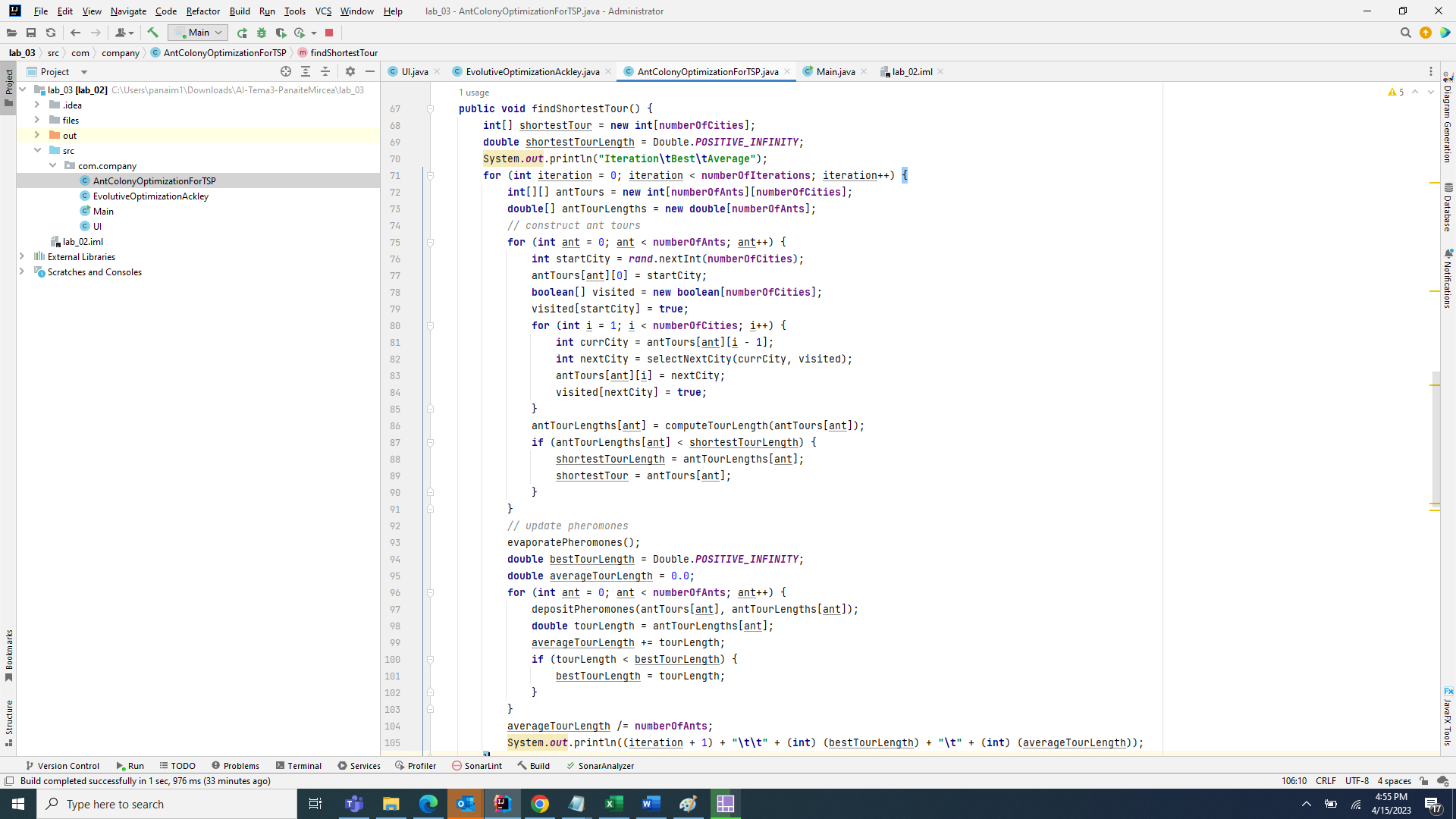
# selectSurvivors(population:double[][], offspring:double[][]) - Această funcție primește două matrici, populația curentă și descendenții, le combină într-o singură matrice și returnează o submatrice care reprezintă cei mai buni indivizi selectați ca supraviețuitori, în funcție de valorile fitness-ului.

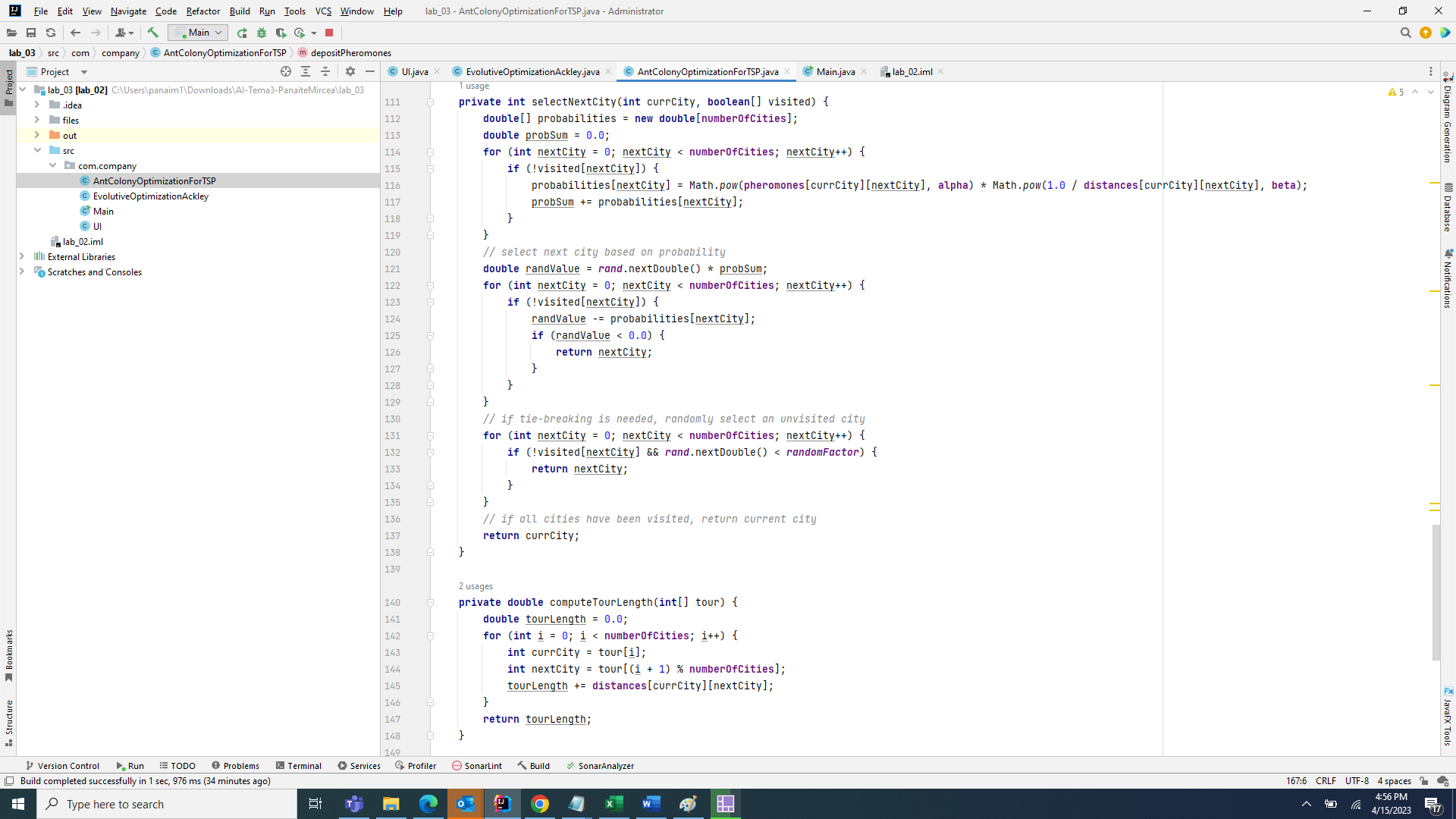
# evaluateFitness(individual:double[]) - Această funcție primește un individ și calculează valoarea fitness-ului pentru el, folosind funcția Ackley.

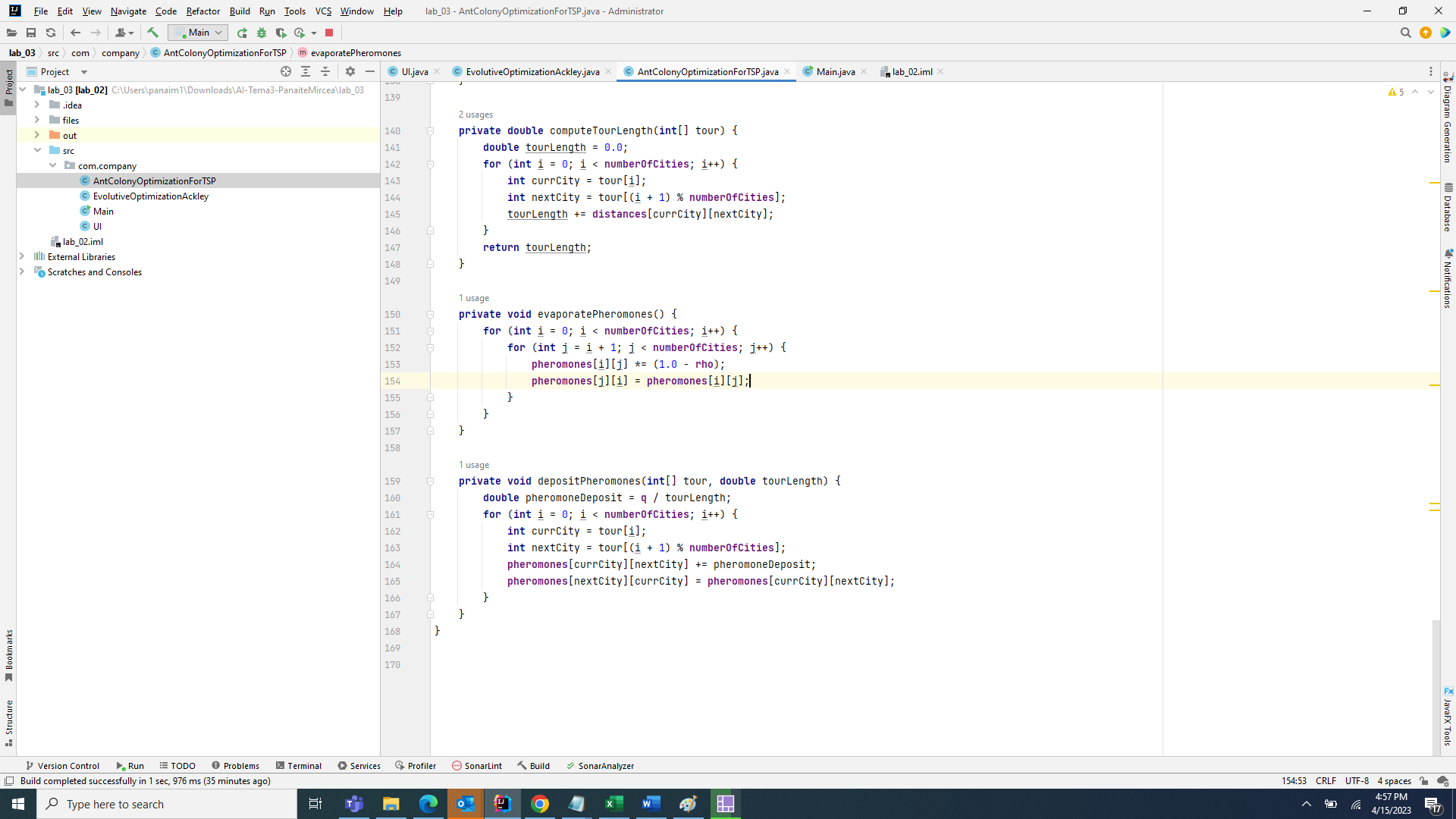
# generateRandomValidSolution() - Această funcție generează și întoarce o soluție validă aleatorie pentru problema Ackley, verificând dacă aceasta are o valoare fitness sub un anumit prag.

# validateSolution(solution:double[]) - Această funcție primește o soluție și verifică dacă aceasta este validă pentru problema Ackley, adică are o valoare fitness sub un anumit prag.

### Algoritm AS pent ru TSP







* public void findShortestTour():
* Această funcție implementează algoritmul de optimizare a coloniei de furnici pentru rezolvarea problemei de călătorie a unui vânzător. Aceasta inițializează variabilele shortestTour și shortestTourLength, construiește tururile furnicilor pentru fiecare furnică, actualizează nivelurile de feromon, și afișează lungimile tururilor cele mai bune și medii pentru fiecare iterație. Funcția apelează funcția computeTourLength() pentru a calcula lungimea turului pentru fiecare furnică..
* private int selectNextCity(int currCity, boolean[] visited):
* Această funcție selectează următorul oraș de vizitat de către o furnică. Aceasta calculează probabilitățile de a vizita fiecare oraș nevizitat bazat pe nivelurile de feromon și distanțele față de orașul curent. Funcția utilizează un algoritm de selecție cu roată de ruletă pentru a selecta următorul oraș bazat pe probabilitățile calculate.
* private void evaporatePheromones():
* Această funcție evaporă nivelurile de feromon pe fiecare muchie cu un factor de rho. Funcția utilizează o formulă simplă pentru a calcula noile niveluri de feromon: (1 - rho) \* pheromones[i][j].
* private void depositPheromones(int[] tour, double tourLength):
* Această funcție depune feromoni pe fiecare muchie a turului bazat pe lungimea turului și valoarea q. Funcția utilizează o formulă simplă pentru a calcula cantitatea de feromoni care trebuie depusă: q / tourLength.
* private double computeTourLength(int[] tour):
* Această funcție calculează lungimea turului pentru un tur dat prin suma distanțelor dintre toate perechile de orașe consecutive din tur. Funcția returnează lungimea totală a turului.

**Tabele de analiză a soluțiilor**

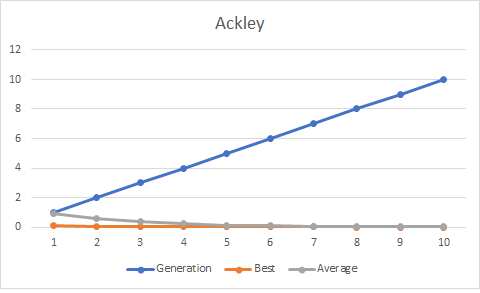
#### Ackley function optimization

#### Uniform crossover

**Parametri: populationSize = 100, maxGenerations = 10**

|  |  |
| --- | --- |
| **Nr rulare** | **Calitate (fitness value)** |
| **1** | 0.0173113918 |
| **2** | 0.0162111724 |
| **3** | 0.0063624542 |
| **4** | 0.0106873022 |
| **5** | 0.0029372952 |
| **6** | 0.0037780084 |
| **7** | 0.0157621154 |
| **8** | 0.0104930302 |
| **9** | 0.0145932014 |
| **10** | 0.0083327463 |

**Time: 0.2341s**

****

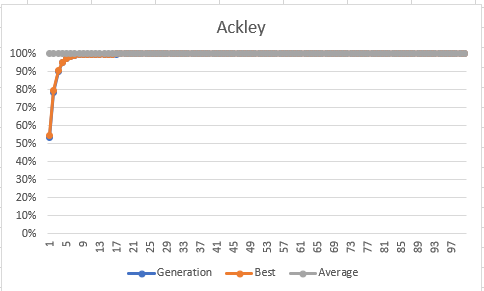
|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 0.0029372952 |
| **Average value** | 0.0111329775 |
| **Worst value** | 0.0173113918 |

**Parametri: populationSize = 1000, maxGenerations = 100**

|  |  |
| --- | --- |
| **Nr rulare** | **Calitate (fitness value)** |
| **1** | 0.0041904467 |
| **2** | 0.0011225073 |
| **3** | 0.0042851709 |
| **4** | 0.0024737814 |
| **5** | 0.0008481309 |
| **6** | 0.0008542318 |
| **7** | 0.0020004612 |
| **8** | 0.0048546848 |
| **9** | 0.0010982524 |
| **10** | 0.0027624876 |

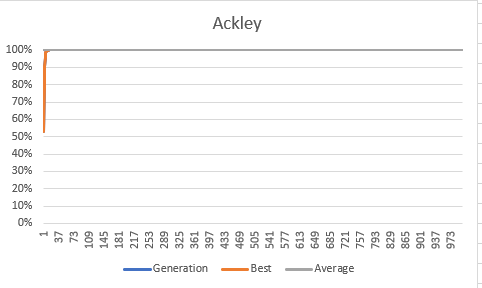
**Time: 5.22110s**

|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 0.0008481309 |
| **Average value** | 0.0022956511 |
| **Worst value** | 0.0048546848 |

****

**Parametri: populationSize = 10000, maxGenerations = 1000**

|  |  |
| --- | --- |
| **Nr rulare** | **Calitate (fitness value)** |
| **1** | 0.0000332634 |
| **2** | 0.0000328139 |
| **3** | 0.0000579142 |
| **4** | 0.0001625612 |
| **5** | 0.0001871092 |
| **6** | 0.0002013138 |
| **7** | 0.0002759686 |
| **8** | 0.0000925317 |
| **9** | 0.0000360019 |
| **10** | 0.0000929895 |

****

**Time: 52.05423**

|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 0.0000328139 |
| **Average value** | 0.0001185258 |
| **Worst value** | 0.0002759686 |

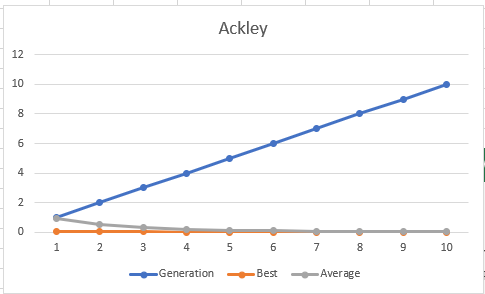
#### Ackley function optimization

#### Arithmetic crossover

**Parametri: populationSize = 100, maxGenerations = 10**

|  |  |
| --- | --- |
| **Nr rulare** | **Calitate (fitness value)** |
| **1** | 0.0001082329 |
| **2** | 0.0006365995 |
| **3** | 0.0005161896 |
| **4** | 0.0002907201 |
| **5** | 0.0001065274 |
| **6** | 0.0005073355 |
| **7** | 0.0000640469 |
| **8** | 0.0000560607 |
| **9** | 0.0004836797 |
| **10** | 0.0005734956 |

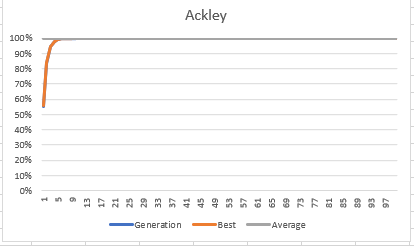
**Time: 0.12231s**

****

|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 0.0000560607 |
| **Average value** | 0.0003446869 |
| **Worst value** | 0.0006365995 |

**Parametri: populationSize = 1000, maxGenerations = 1000**

|  |  |
| --- | --- |
| **Nr rulare** | **Calitate (fitness value)** |
| **1** | 0.0000016188 |
| **2** | 0.0000014565 |
| **3** | 0.0000025846 |
| **4** | 0.0000019390 |
| **5** | 0.0000018486 |
| **6** | 0.0000005775 |
| **7** | 0.0000018865 |
| **8** | 0.0000024898 |
| **9** | 0.0000027413 |
| **10** | 0.0000007829 |

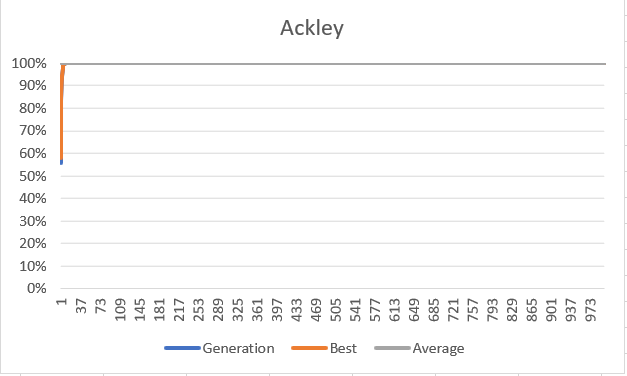
**Time: 0.023253**

|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 0.0000019295 |
| **Average value** | 0.0000027413 |
| **Worst value** | 0.0000007829 |

**Parametri: populationSize = 10000, maxGenerations = 1000**

|  |  |
| --- | --- |
| **Nr rulare** | **Calitate (fitness value)** |
| **1** | 0.0000000234 |
| **2** | 0.0000000150 |
| **3** | 0.0000000018 |
| **4** | 0.0000000211 |
| **5** | 0.0000000091 |
| **6** | 0.0000000269 |
| **7** | 0.0000000058 |
| **8** | 0.0000000722 |
| **9** | 0.0000000430 |
| **10** | 0.0000000000 |

**Time 60.0651**

****

|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 0.0000000000 |
| **Average value** | 0.0000000218 |
| **Worst value** | 0.0000000722 |

**Instanța – problema comis voiajor pentru fișierul ``lin105.tsp``**

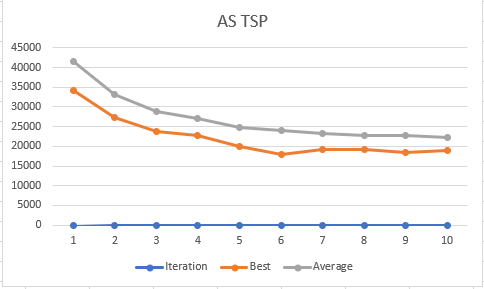
Traveling Salesman Problem – Ant Colony

**Parametri: alpha=0.9, beta=1.9, rho=0.4, q=90.0, numberOfAnts=40, numberOfIterations=10**

|  |  |
| --- | --- |
| **Nr rulare** | **Cost (distanță) – fitness value** |
| **1** | 18125 |
| **2** | 17926 |
| **3** | 18167 |
| **4** | 18360 |
| **5** | 18688 |
| **6** | 18615 |
| **7** | 17931 |
| **8** | 18427 |
| **9** | 18235 |
| **10** | 17807 |

**Time: 1.02331 s**

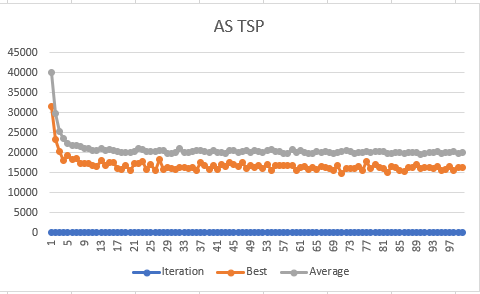
|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 17807 |
| **Average value** | 18205 |
| **Worst value** | 18688 |



**Parametri: alpha=1.0, beta=2.0, rho=0.5, q=100.0, numberOfAnts=50, numberOfIterations=100**

|  |  |
| --- | --- |
| **Nr rulare** | **Cost (distanță) – fitness value** |
| **1** | 15295 |
| **2** | 14920 |
| **3** | 15040 |
| **4** | 15439 |
| **5** | 15120 |
| **6** | 14975 |
| **7** | 15343 |
| **8** | 15324 |
| **9** | 15906 |
| **10** | 15132 |

Time : 23.456I2 s



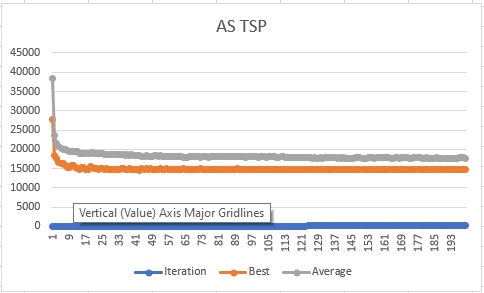
|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 14920 |
| **Average value** | 15223 |
| **Worst value** | 15906 |

**Parametri: alpha=1.1, beta=2.1, rho=0.6, q=110.0, numberOfAnts=500, numberOfIterations=200**

|  |  |
| --- | --- |
| **Nr rulare** | **Cost (distanță) – fitness value** |
| **1** | 14666 |
| **2** | 14726 |
| **3** | 14683 |
| **4** | 14570 |
| **5** | 14613 |
| **6** | 14975 |
| **7** | 14707 |
| **8** | 14706 |
| **9** | 14726 |
| **10** | 14769 |

|  |  |
| --- | --- |
| **Analiza calității** | |
| **Best value** | 14570 |
| **Average value** | 14683 |
| **Worst value** | 14975 |

Time : 91.99133 s



Concluzii

* 1. **Ackley**

Pe măsura ce creștem populationSize, maxGenerations obținem rezultate din ce în ce mai bune apropiate de optim. Am folosit doua tipuri de incrucisari – unfirorma si aritmetica. Cu incrucisarea artimetica optinem chiar optimul global. Cu toate acestea nu merita rulat pentru parametri mai mari de 1000 pentru generatii intrucat se atinge destul de repede convergenta prematura.

* 1. **TSP**

Cu cât creștem numberOfAnts și numberOfIterations obținem rezultate din ce în ce mai bune apropiate de optim și păstrăm **alpha, beta, rho, q apropiate de valorile primului experiment** iar timpul nu crește considerabil. Însă cu cât creștem numărul de furnici si iteratii creștem și timpul de execuție substantial Rezultatele sunt mult mai bune decât pentru simulated annealing si decat pentru algoritmul evolutiv.