Documentatie Lab3

Cerinta

1. Să se implementeze **un algoritm evolutiv pentru problema rucsacului**.
    a. Codificare binară
    b. Operatori specifici (încrucișare, mutație)
    c. Algoritm şi parametrizare
    d. Experimente pe cele două instanţe primite la Tema 1

a. Codificare binara
Solutiile sunt reprezentate de siruri binare unde fiecare bit rreprezinta un cromozom

b.Operatori specifici
Am implementat doua variante de algoritm evolutiv care difera prin operatorii de incrucisare si mutatie

- Incrucisare

Single Crossover(Incrucisare cu un punct de taietura)

Incrucisarea cu un punct de taietura

- Un punct de taietura este un numar intreg $\mathbf{k} \in \{1,2, ...,r-1\}$
- X – multimea tuturor cromozomilor de lungime fixata **r**
- C – operatorul de incrucisare
- $C : X \times X \rightarrow X \times X$
- $C(x,y) = (x',y')$

Parinti

$$x = x_1 x_2 \dots x_k x_{k+1} \dots x_r$$
$$y = y_1 y_2 \dots y_k y_{k+1} \dots y_r$$

Cromozomii copii vor fi:

$$x' = x_1 x_2 \dots x_k y_{k+1} \dots y_r$$
$$y' = y_1 y_2 \dots y_k x_{k+1} \dots x_r$$

Implementare

```
def single_cross_over(self, parent1, parent2):
    """
    Perform a crossover with one cut between two parents
    :param parent1: first parent
    :param parent2: second parent
    :return: two children
    """
    cut_point = random.randint(3, self.knapsack.objects_number - 3)
    child1 = [0] * self.knapsack.objects_number
    child2 = [0] * self.knapsack.objects_number

    for index, bit in enumerate(parent1):
        if index < cut_point:
            child1[index] = bit
        else:
            child2[index] = bit

    for index, bit in enumerate(parent2):
        if index < cut_point:
            child2[index] = bit
```

```
        else:
            child1[index] = bit

    return child1, child2
```

Incrucisare adaptiva

Incrucisarea uniforma

- Nu foloseste puncte de taietura
- *Parametru*: **p** - probabilitatea ca gena unui descendent sa provina din primul sau al doilea parinte
  - $x = x_1 x_2 \ldots x_k \ldots x_r$
  - $y = y_1 y_2 \ldots y_k \ldots y_r$
  - Pentru fiecare pozitie *i* din x' se alege parintele care va da valoarea pozitiei respective cu prob **p**
  - Pentru y' se ia valoarea pozitiei corespunzatoare din celalalt parinte; *Alternativ*: independent de x'
- Poate combina caracteristici indiferent de pozitia relativa

Implementare

```
def uniform_cross_over(self, parent1, parent2, prob):
    """
    Perform a uniform crossover with a probability between two
parents
    :param parent1: first parent
    :param parent2: second parent
    :param prob: the probability that a gene comes from one parent
or the other
    float number between 0 and 1
    :return: two children
    """
    child1 = []
    child2 = []

    for i in range(self.knapsack.objects_number):
```

```
        choice = np.random.choice([1, 2], p=[prob, 1-prob])
        if choice == 1:
            child1.append(parent1[i])
            child2.append(parent2[i])
        else:
            child1.append(parent2[i])
            child2.append(parent1[i])

    return child1, child2
```

- Mutatie

Mutatia tare

Mutatia tare

- P1. Pentru fiecare cromozom al populatiei curente si pentru fiecare pozitie a cromozomului se executa:
  - P1.1. Se genereaza un numar aleator q in intervalul [0,1].
  - P1.2. Daca $q < p_m$ atunci se executa mutatia pozitiei respective, schimband 0 in 1 si 1 in 0.

    In caz contrar ($q \geq p_m$), pozitia respectiva nu se schimba

Implementare

```
def strong_mutation(self, entity, mutation_probability):
    """
    Performs a strong mutation on a population entity on binary
representation
        :param mutation_probability: the probability that mutation
occurs
        :param entity: a guy from population
    :return: the modified entity
    """
```

```
    for index, bit_gene in enumerate(entity):
        q = random.uniform(0, 1)  # a random number between 0 and 1
        if q < mutation_probability:
            entity[index] = int(not bit_gene)  # flip the bit

    return entity
```

- Mutatia slaba

Mutatia slaba

- P1. Pentru fiecare cromozom al populatiei curente si pentru fiecare pozitie a cromozomului se executa:
  - P1.1. Se genereaza un numar aleator q in intervalul [0,1].
  - P'1.2. Daca $q < p_m$ atunci se alege aleator una din valorile 0 sau 1. Se atribuie pozitiei curente valoarea astfel selectata.

    Daca $q \geq p_m$ atunci pozitia curenta nu se schimba.

Implementare

```
def weak_mutation(self, entity, mutation_probability):
    """
    Performs a weak mutation on a population entity on binary representation
    :param mutation_probability: the probability that the mutation occurs
    :param entity: a guy from population
    :return: the modified entity
    """
    for index, bit_gene in enumerate(entity):
        q = random.uniform(0, 1)  # a random number between 0 and 1
        if q < mutation_probability:
            val = random.randint(0, 1)
            entity[index] = val

    return entity
```

c. Algoritm si parametrizare

Implementarea a fost facuta utilizand pseudocodul din cursul 4

Un algoritm evolutiv standard

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

1. Initializarea populatiei
   Populatia se initializeaza aleator (din indivizi generati aleator)
2. Selectia parintilor
   Parintii i-am ales cu o probabilitate direct proportionala cu fitnessul individual
   p = fitness_individual / fitness total, unde fitnessul total este suma fitness-urilor tuturor indivizilor din populatie
3. Selectia supravietuitorilor
   Supravietuitorii se aleg descrescator dupa fitness din reuniunea parintilor si copiilor astfel incat pentru noua generatie astfel incat sa se pastreze dimensiunea initiala a populatiei

Observatii

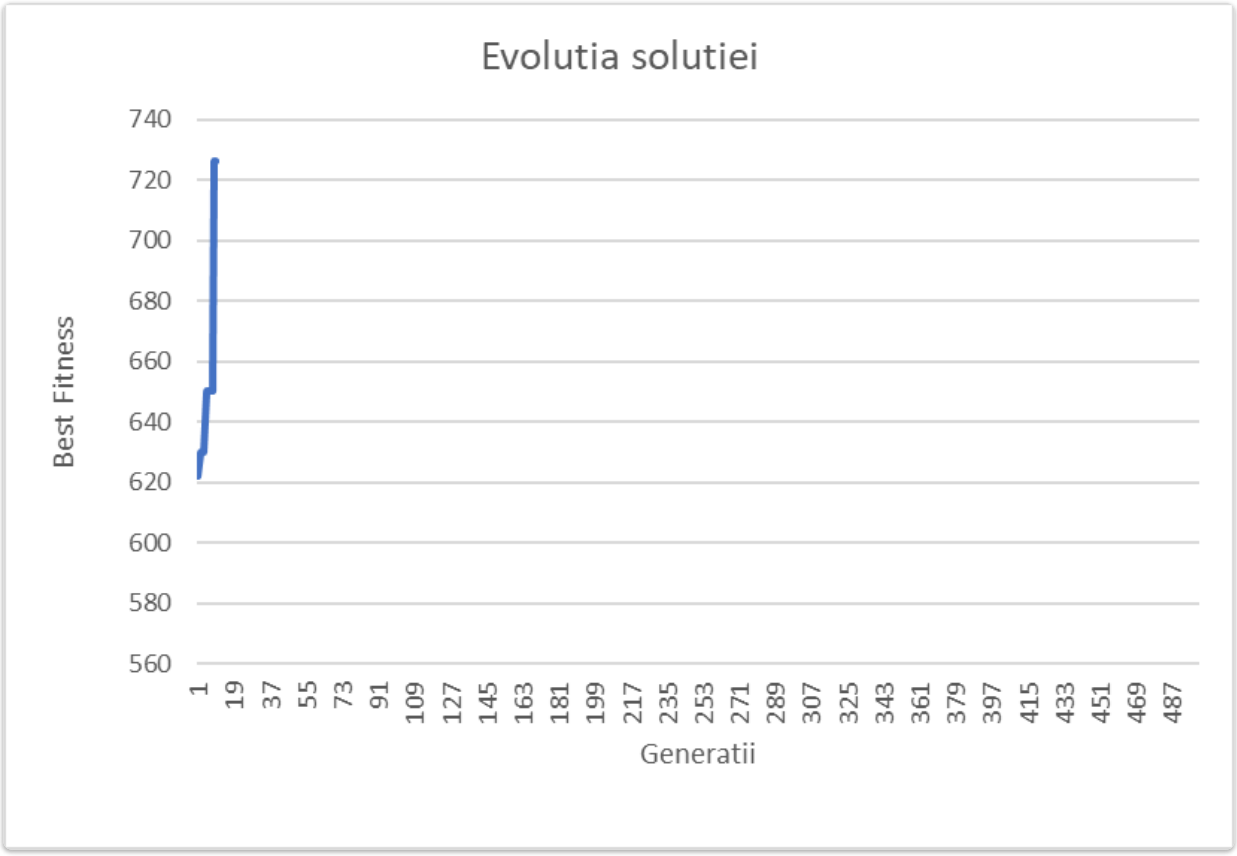
- Daca unul dintre copiii rezultati dupa incrucisare si mutatie nu reprezinta o solutie valida va fi adaugat in populatie, insa cu o penalizare a fitnessului(acesta va fi 0)
- Pentru prima varianta a algoritmului am folosit incrucisarea cu un punct de taietura si mutatie tare

- Pentru a doua varianta am folosit incrucisarea uniforma si mutatia slaba
- Initializarea populatiei, selectia parintilor si selectia supravietuitorilor raman la fel

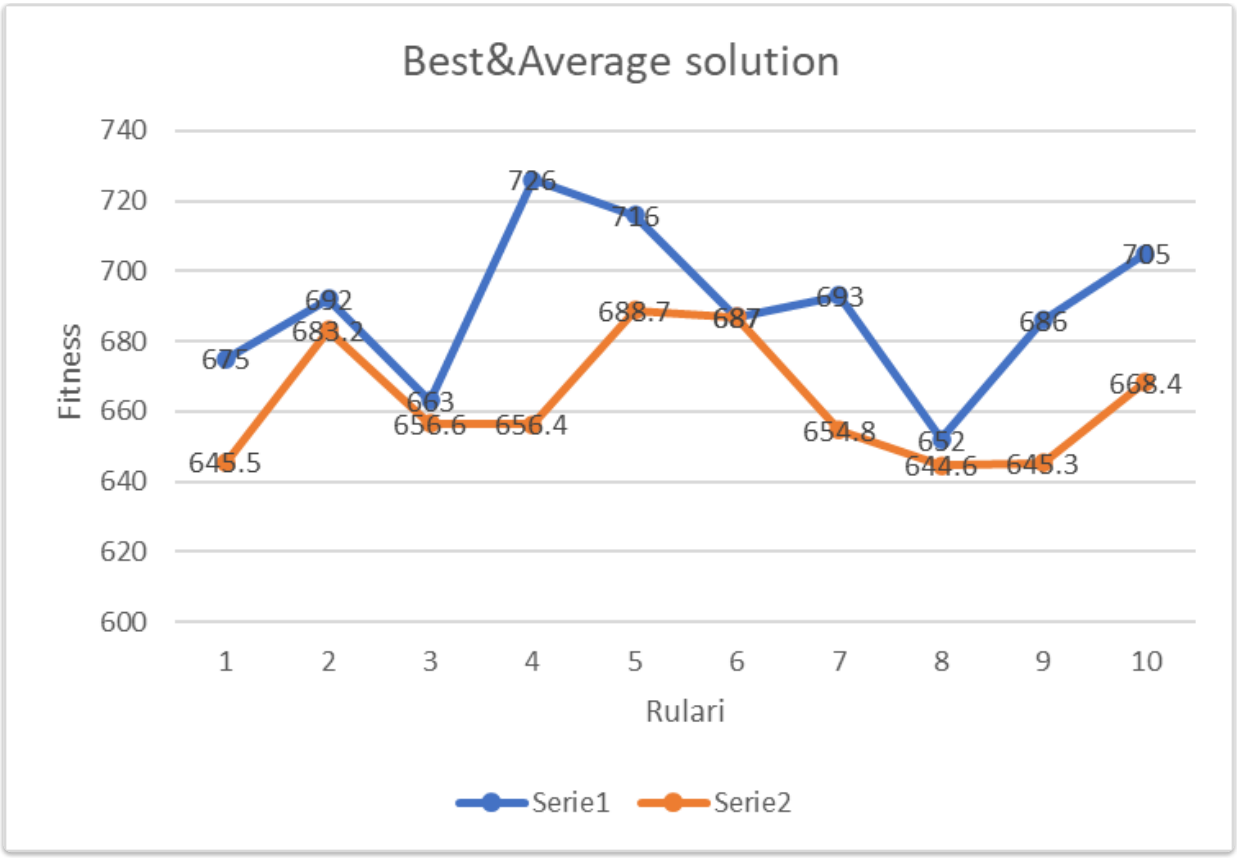
d. Experimente

Grafice

- rucsac20.txt
  populatie=100, generatii=10

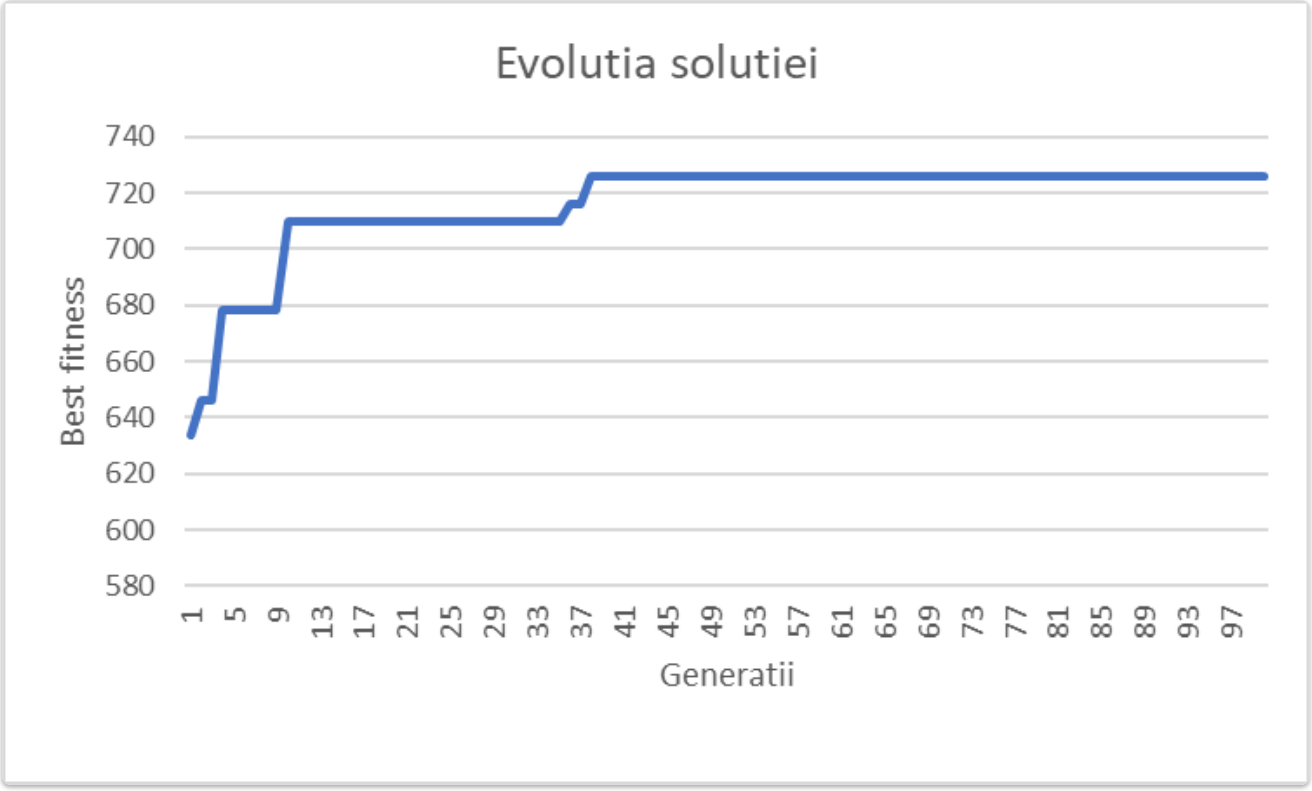


Rezumat 10 rulari

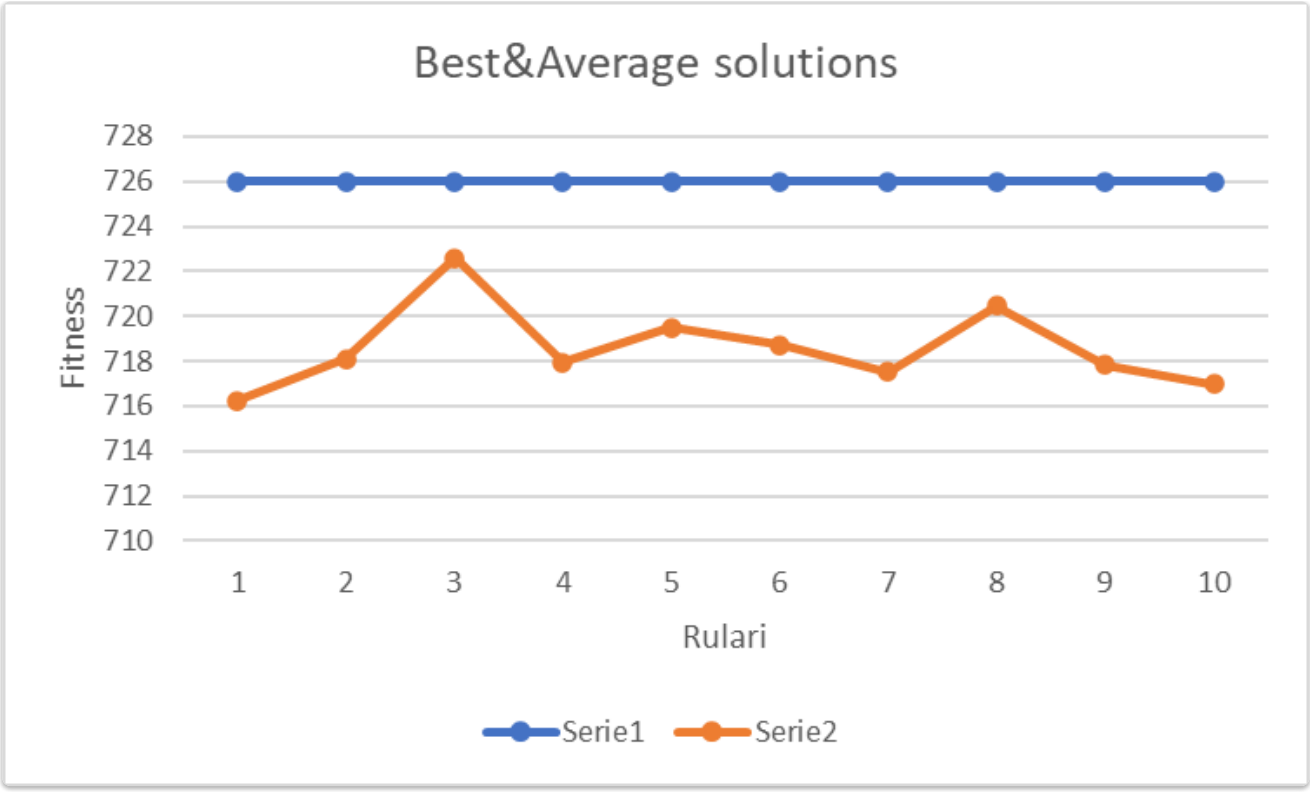


Serie 1 = best sol, Serie 2 = avg sol

populatie=100, generatii=100



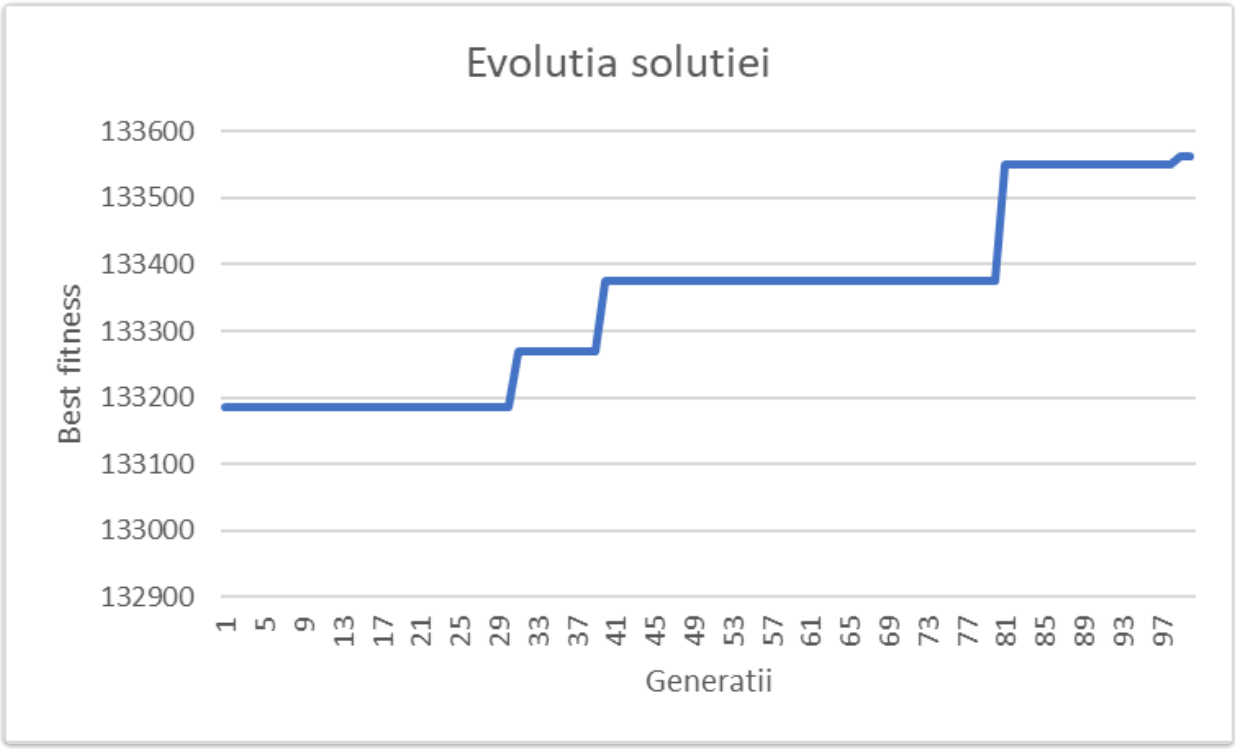
10 rulari



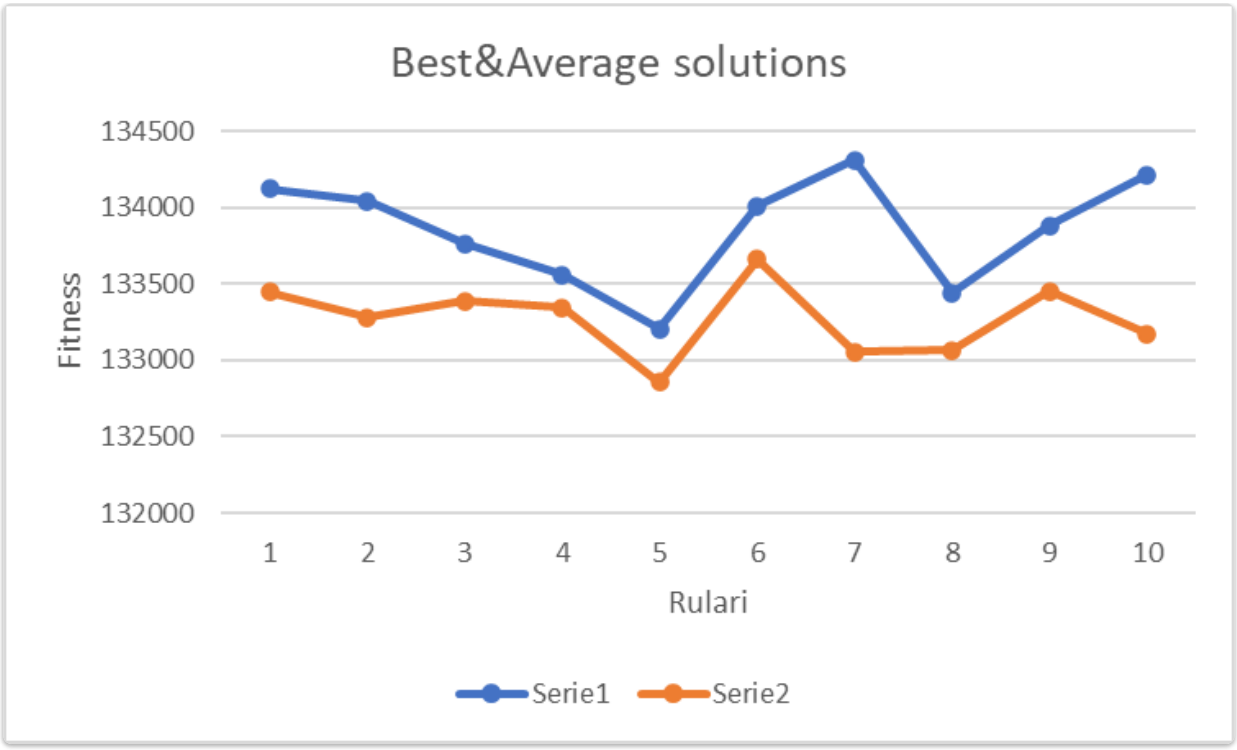
Serie 1 = best sol, Serie 2 = avg sol

- rucsac200.txt

  populatie=100, generatii=100

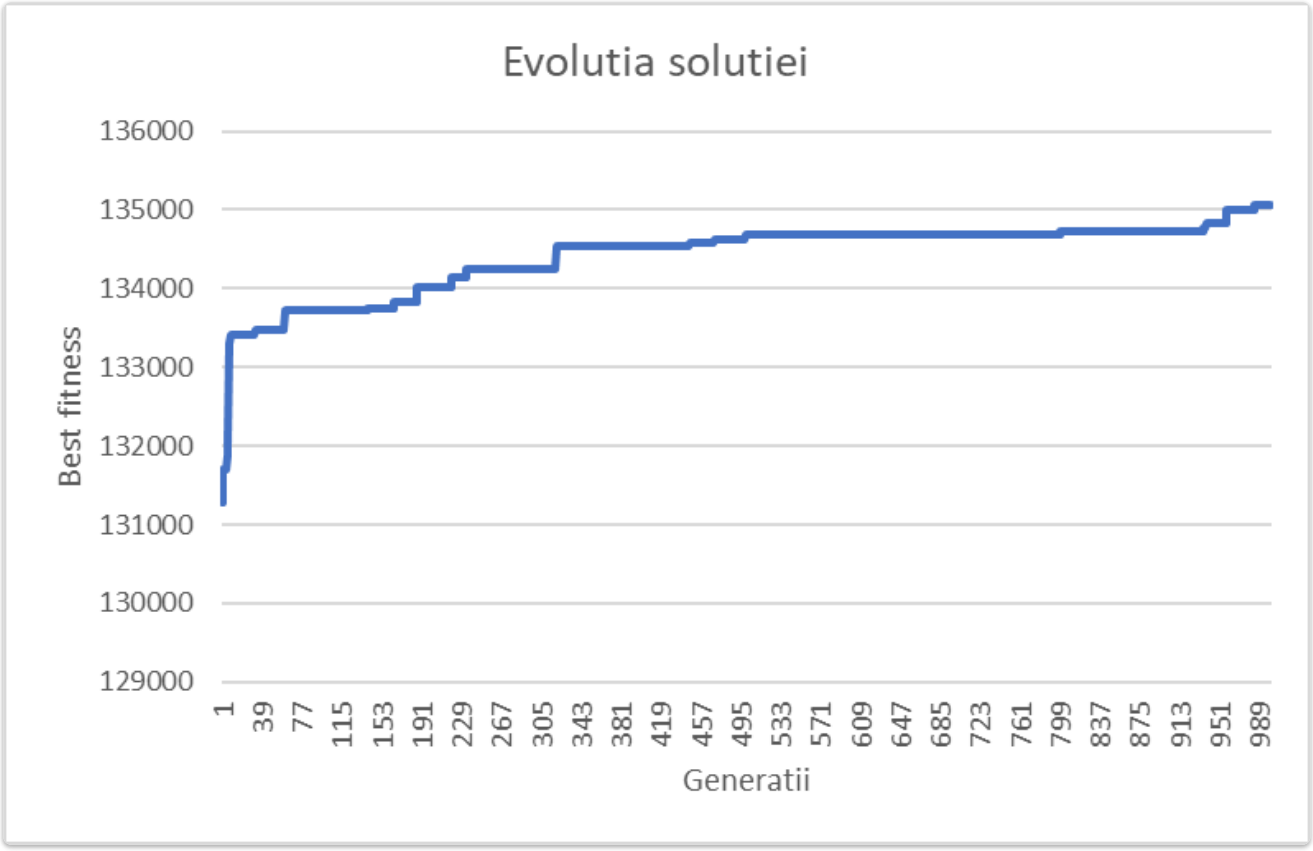


  10 rulari

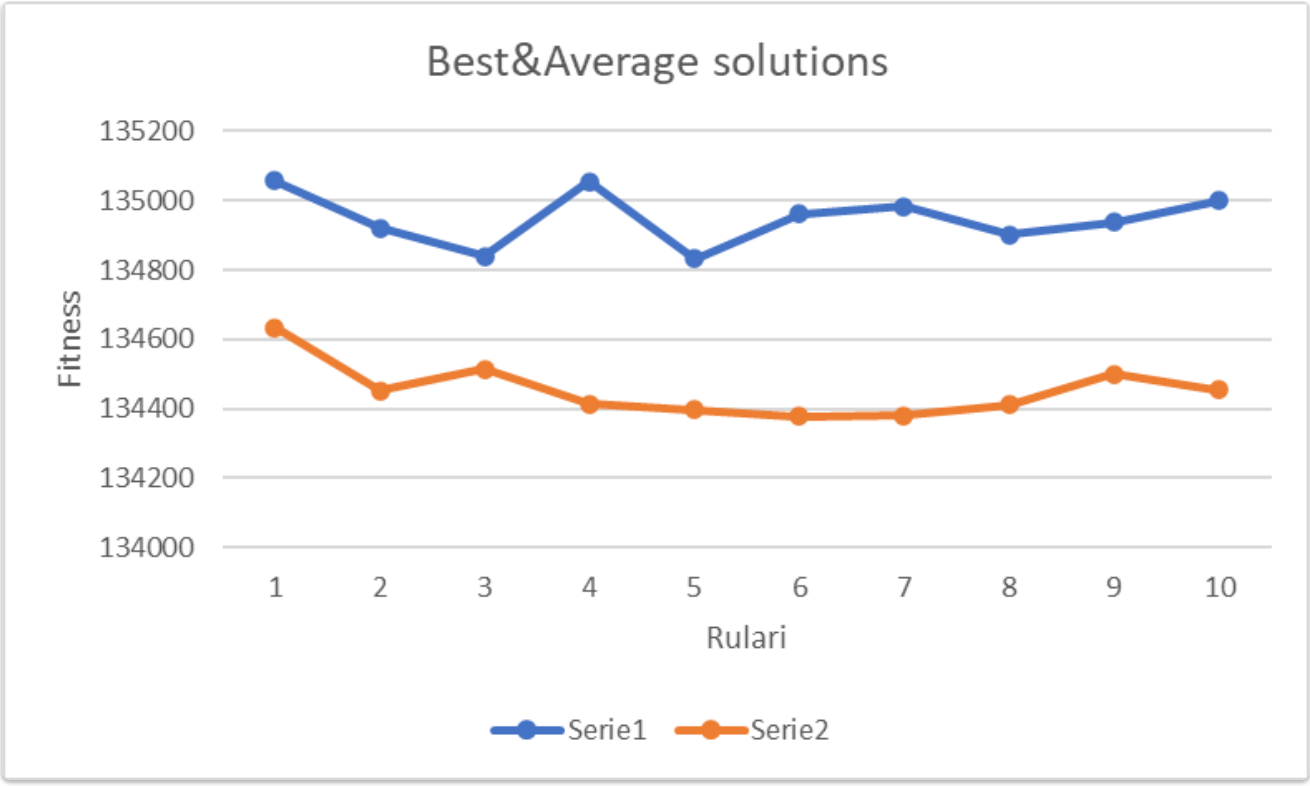


Serie 1 = best sol, Serie 2 = avg sol

populatie=100, generatii=1000



10 rulari



Serie 1 = best sol, Serie 2 = avg sol

Tabele cu rezultatele experimentelor

Rucsac 20

				Best	Average	Best Average	Running time
Alg 1	p=100	g=10	m=0.1	711	667.24	693.3	0.5 s
		g=100	m=0.1	726	718.59	722.61	1.9 s
	p=200	g=10	m=0.1	726	680.6	697.4	1.79 s
	p=1000	g=500	m=0.1	726	711.39	725.28	117.43 s
	p=1000	g=500	m=0.3	726	712.06	725.47	109.17 s
Alg 2	p=100	g=10	m=0.1	726	663.05	688.7	1.12 s
		g=100	m=0.1	726	717.36	724.33	7.82 s
	p=200	g=10	m=0.1	711	664.20	680.5	2.2 s
	p=1000	g=500	m=0.1	726	725.07	725.58	376 s
	p=1000	g=500	m=0.3	726	724.38	724.96	387 s

p=populatie, g=generatie, m=probabilitatea de mutatie

Rucsac 200

				Best	Average	Best Average	Running time
Alg 1	p=100	g=100	m=0.1	134316	133273.30	133664.82	24.1 s
		g=1000	m=0.1	135058	134453.24	134632.61	128.52 s
	p=1000	g=500	m=0.1	135294	134525.02	134721.78	642.76 s
Alg 2	p=100	g=100	m=0.1	133915	133060.83	133234.95	74.53 s
		g=1000	m=0.1	136031	134724.34	134901.71	702.83 s
	p=1000	g=500	m=0.1	135578	134484.62	134606.65	3570 s

p=populatie, g=generatie, m=probabilitatea de mutatie

Concluzii

Comparand rezultatele din tabele cu cele din tabelele din laboratoarele precendente, se observa ca acesta abordare cu algoritmi evolutivi ofera cele mai bune rezultate de pana acum

Cerinta

2. Să se implementeze **un algoritm evolutiv pentru problema comis-voiajorului**.
    a. Codificare prin permutări
    b. Operatori specifici (încrucișare, mutație)
    c. Algoritm şi parametrizare
    d. Experimente pe instanţa primită la Tema 2

a. Codificare prin permutari

Solutiile sunt reprezentate de permutari unde fiecare lement din permutare reprezinta un oras

a. Operatori specifici

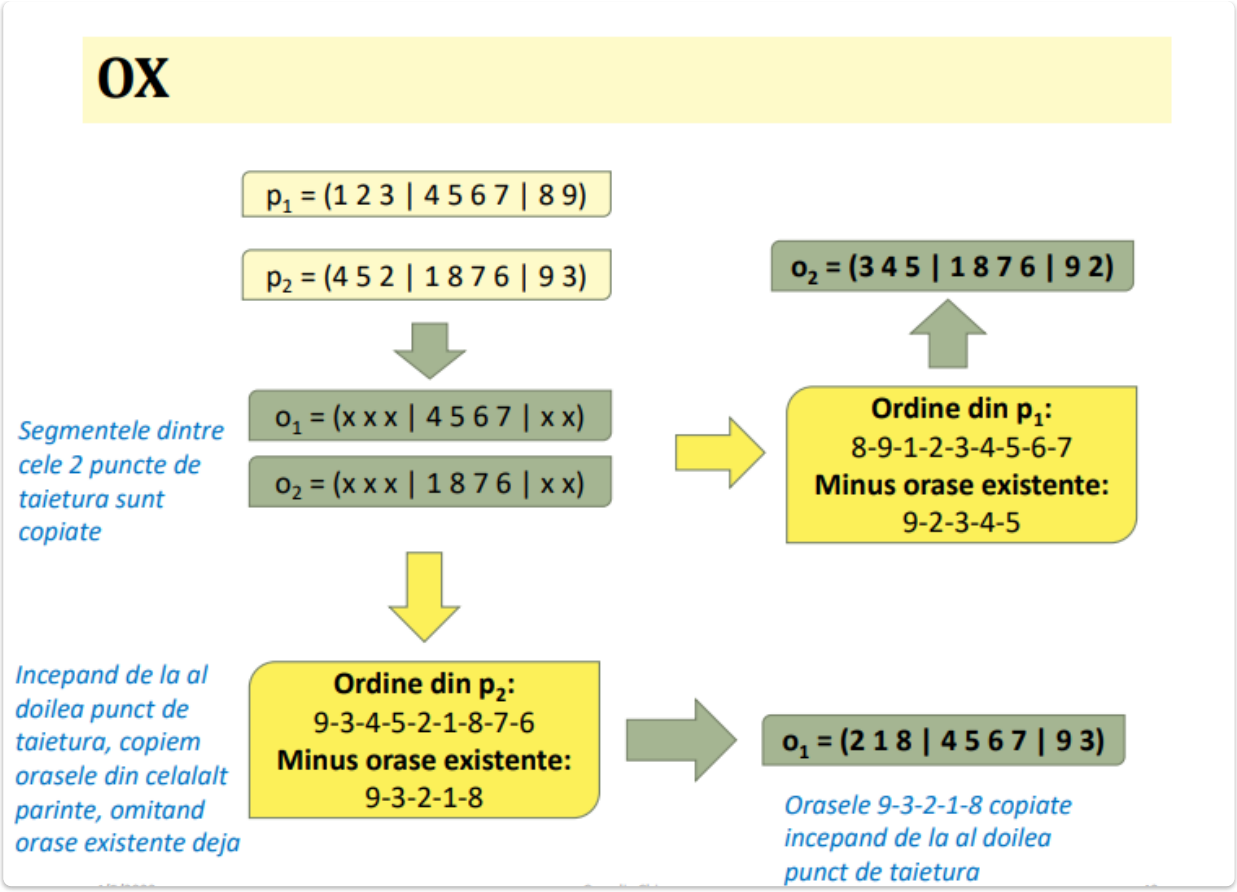
Am implementat doua variante de algoritm evolutiv care difera prin operatorii de mutatie

- **Incrucisarea** - Order Crossover

OX (Order Crossover)

- Alege o subruta dintr-un parinte si pastreaza ordinea relativa a oraselor din celalalt parinte

- ***Algoritm***:
    1. Alege aleator o parte (i...j) din primul parinte
    2. Pentru primul descendent
        2.1 Copiaza partea (i...j)
        2.2 Seteaza celelalte pozitii astfel:
            - Incepand de la pozitia imediat urmatoare lui j
            - Folosind **ordinea** din al doilea parinte
            - Continuand circular pana la pozitia dinaintea lui i
    3. Al doilea descendent se creaza similar cu primul dar cu rolurile parintilor schimbate

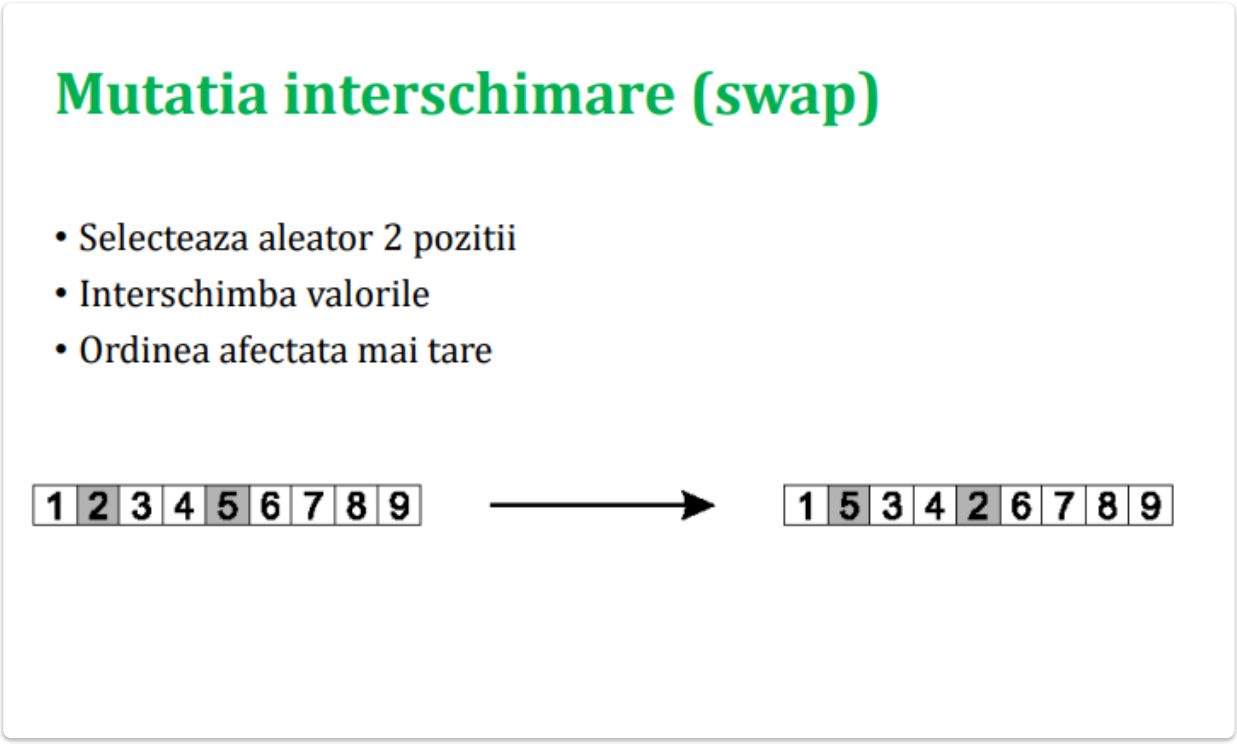


Implementare

```
def order_crossover(self, parent1, parent2):
    """
    Alege o subruta dintr-un parinte si pastreaza ordinea
relativa a oraselor din celalalt parinte
    :param parent1: first parent
    :param parent2: second parent
    :return: two children
    """
    parent1_cities = parent1.cities
    parent2_cities = parent2.cities

    # randomly choose a sub route from one parent to keep in one
child
    i = random.randrange(4, self.dimension - 4)
    j = random.randrange(4, self.dimension - 4)
    if i > j:
        temp = i
        i = j
        j = temp
    sub_route1 = parent1_cities[i:j]
    sub_route2 = parent2_cities[i:j]
```

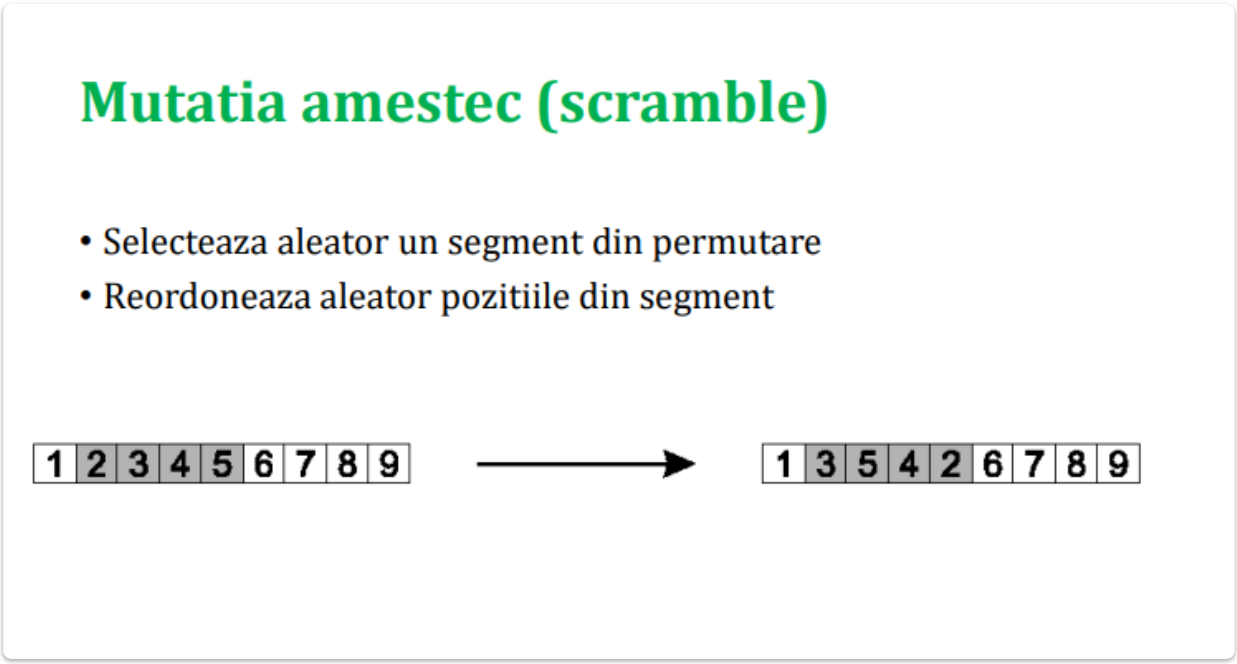
```
    cities1 = sub_route1  # se copiaza partea [i,j] din primul
parinte in primul descendent
    cities2 = sub_route2  # se copiaza partea [i,j] din al doilea
parinte in al doilea descendent
    shifted_ordered_parent2 = parent2_cities[j:] +
parent2_cities[:j]  # se stabileste ordinea in al doilea parinte
incepand cu pozitia j (se roteste lista de orase cu dimensiune-j
pozitii)
    remaining_cities_2 = [city for city in shifted_ordered_parent2
if city not in cities1]  # se scot orasele din al doilea parinte
care apar deja in primul copil
    for elem in range(j, self.dimension):  # se adauga circular in
primul copil orasele ramase in al doilea parinte
        cities1.append(remaining_cities_2.pop(0))
    for elem in range(0, i):
        cities1.insert(0, remaining_cities_2.pop())

    # acelasi lucru ca mai sus pentru al doilea copil
    shifted_ordered_parent1 = parent1_cities[j:] +
parent1_cities[:j]
    remaining_cities_1 = [city for city in shifted_ordered_parent1
if city not in cities2]
    for elem in range(j, self.dimension):
        cities2.append(remaining_cities_1.pop(0))
    for elem in range(0, i):
        cities2.insert(0, remaining_cities_1.pop())

    child1 = Route(self.dimension, cities1)
    child2 = Route(self.dimension, cities2)
    return child1, child2
```

- **Mutatia** - mutatia interschimbare(2-swap)



Implementare
-implementata in laboratorul 2, folosita de acolo

- **Mutatia** - mutatia amestec(scramble)



Implementare

```
def scramble_mutation(self, entity):
    """
    Randomly selects a sub route from the entity and shuffles the
elements
    :param entity: the entity from population
    :return: the mutated entity
```

```
    """
    new_cities = entity.cities
    index1 = random.randrange(5, self.dimension-5)
    index2 = random.randrange(5, self.dimension-5)
    indices = [index1, index2]
    indices.sort()
    selected_sub_route = new_cities[indices[0]:indices[1]]
    random.shuffle(selected_sub_route)
    new_cities[indices[0]:indices[1]] = selected_sub_route
    entity.cities = new_cities
    return entity
```

c. Algoritm si parametrizare

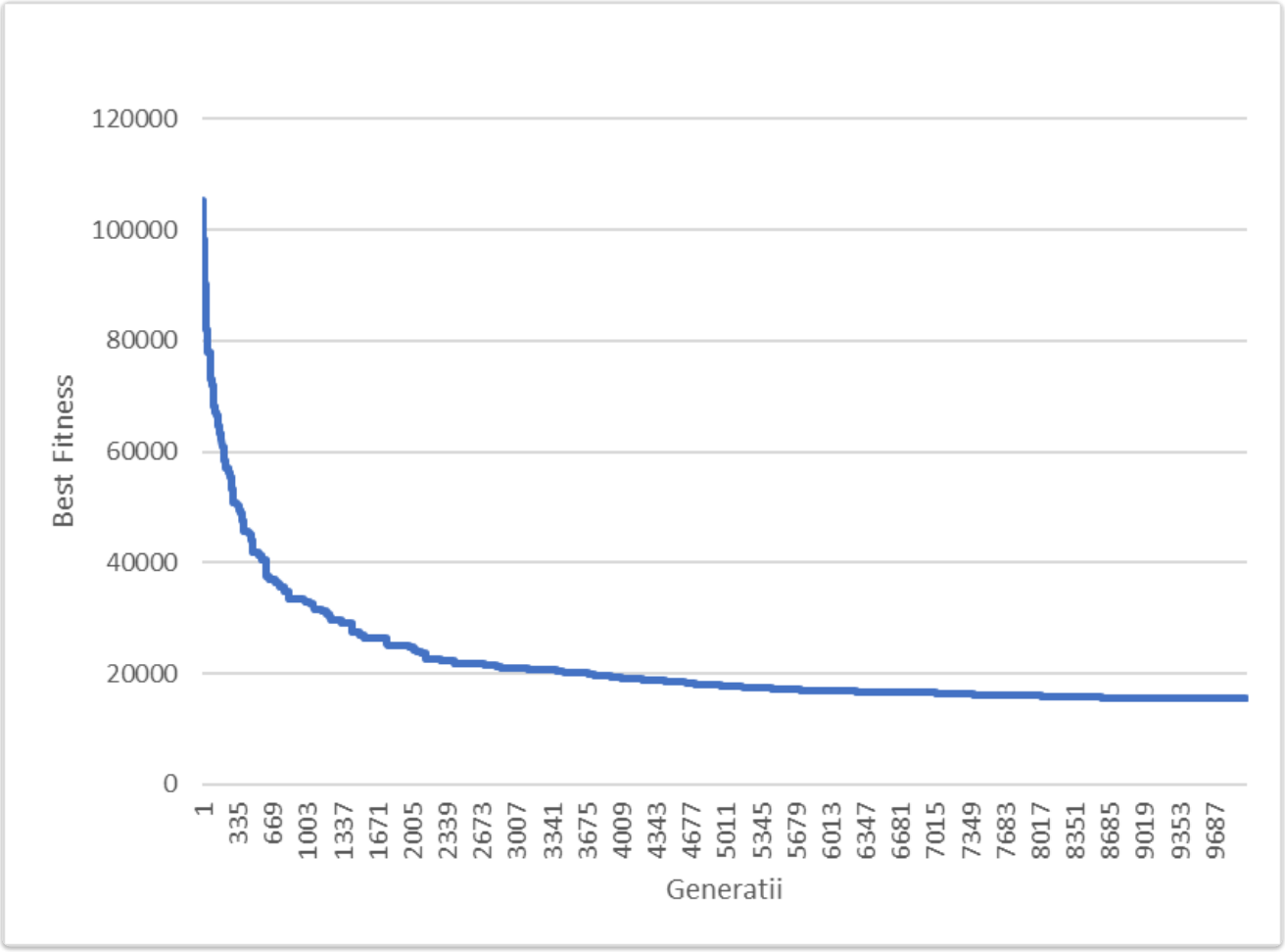
Acelasi pseudocod ca la problema rucsacului, iar initializarea populatiei, selectia parintilor si a supravietuitorilor se face in mod analog

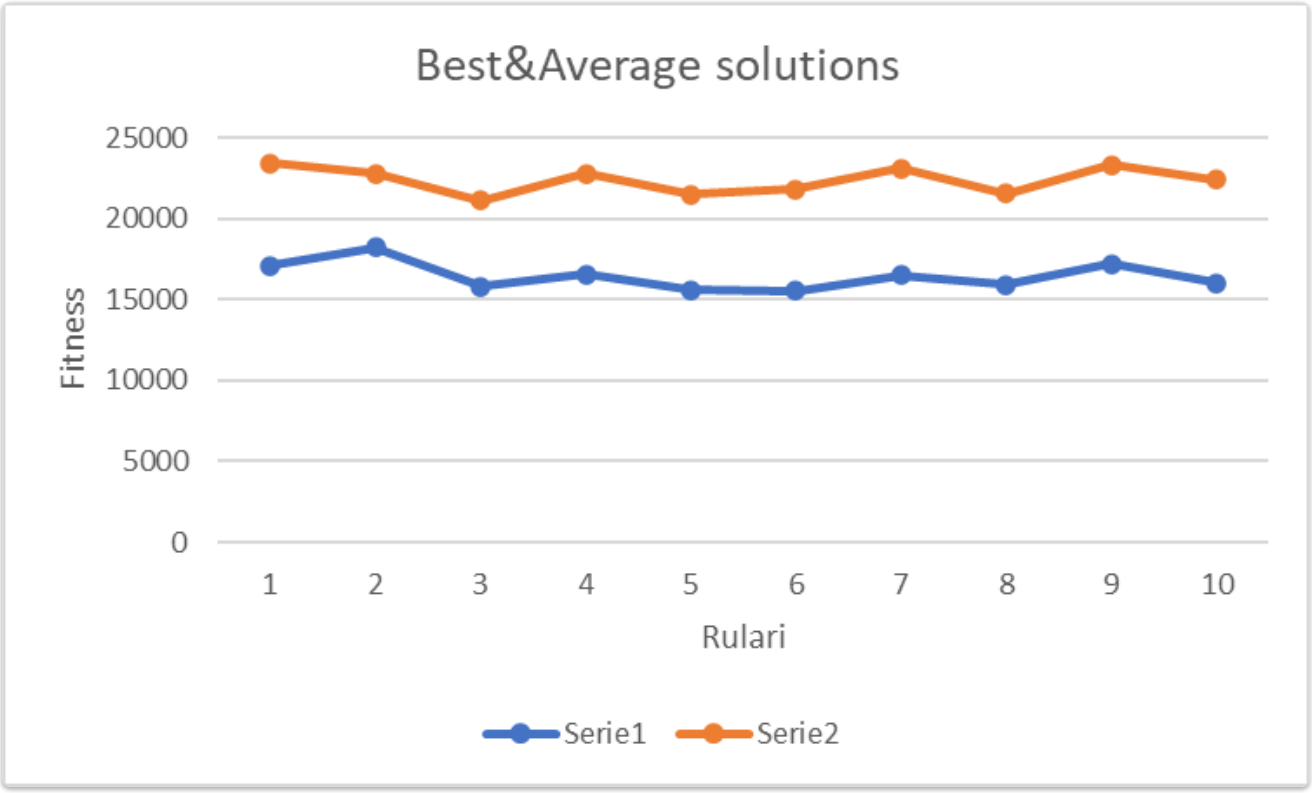
d. Experimente

Experimentele au fost efectuate pe fisierul lin105.tsp, iar din documentatia corespunzatoare, cel mai bun rezultat obtinut este 14379.

Grafice

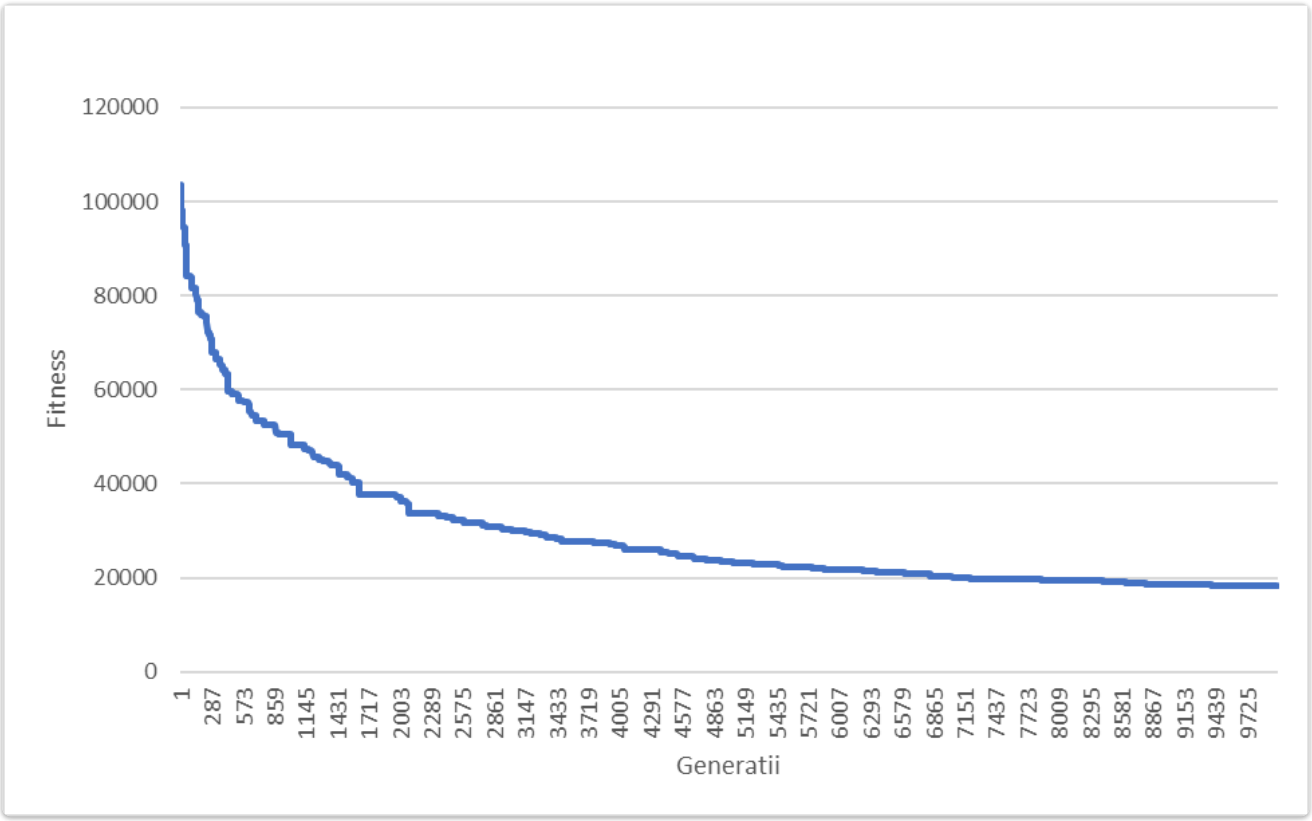
p=500, g=10000, alg 1

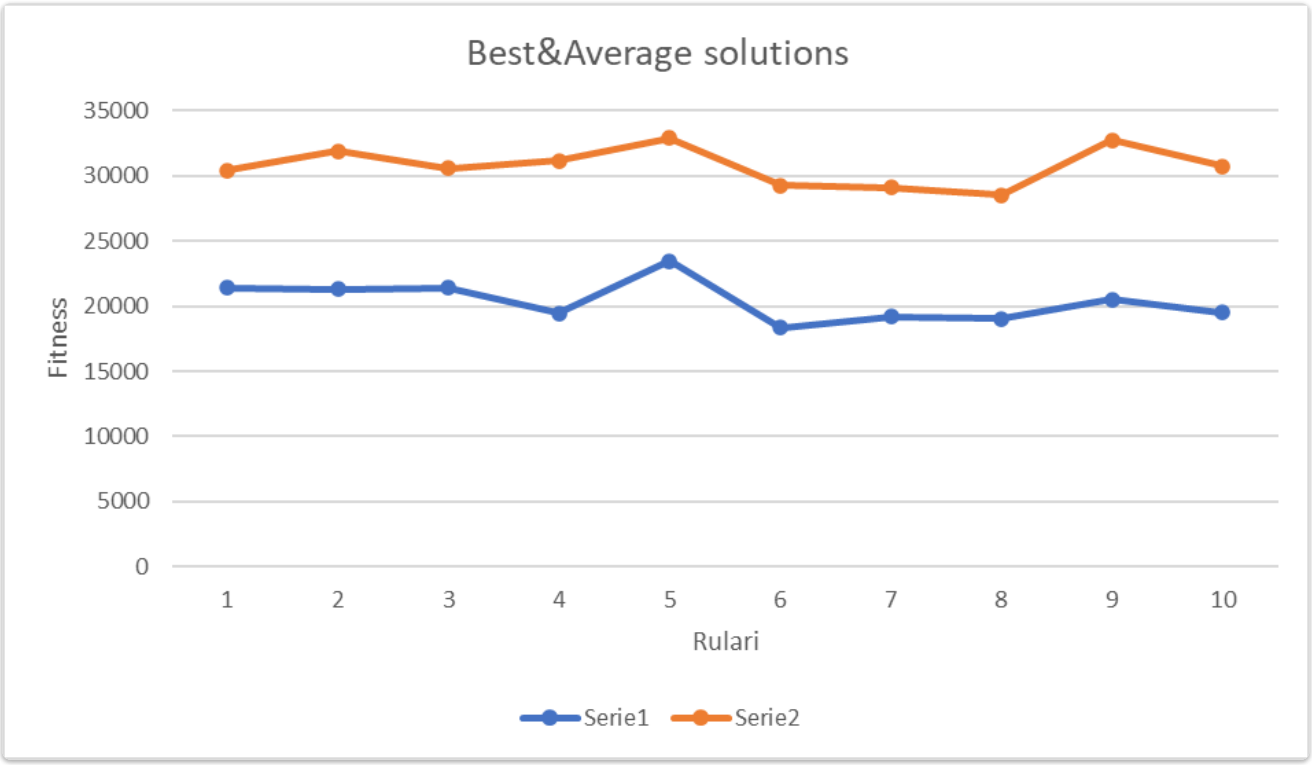




Serie 1 = best sol, Serie 2 = avg sol

p=500, g=10000 alg 2





Serie 1 = best sol, Serie 2 = avg sol

Tabele

TSP

			Best	Average	Best Average	Running time
Alg 1	p=100	g=100	69032.73	84918.02	82360.39	17.56 s
		g=500	43279.55	63043.91	58380.34	35.42 s
		g=5000	19542.54	33829.33	30015.66	366 s ~ 6 min
	p=500	g=10000	15573.91	22399.63	21129.60	3170.01 s ~ 52 min
		g=50000	15585.04	17445.75	16740.67	15981.31 s ~ 4.43 ore
	p=1000	g=500	38378.30	57928.67	54946.211	320.31 s ~ 5 min
		g=1000	26437.18	45620.62	43987.54	644 s ~ 10 min
		g=2500	18953.12	32935.63	31314.91	1591 s ~ 26.5 min
		g=25000	15035.57	18112.50	17216.84	16684 s ~ 4.6 ore
	p=10000	g=1000	25278.40	42238.84	41741.94	6543 s ~ 1.8 ore
Alg 2	p=100	g=100	81095.46	93584.63	89245.35	4.41 s
		g=500	60399.89	77596.23	73757.10	22.5 s
		g=5000	31619.92	47188.08	44889.96	246 s ~ 4 min
	p=500	g=10000	18363.84	30744.57	28556.07	2216 s ~ 37 min
	p=1000	g=500	54880.48	72464.43	69061.33	219.2 s ~ 3.6 min
		g=1000	42974.95	62678.06	59864.64	438 s ~ 7.3 min
		g=2500	27335.19	47419.88	45160.70	1809 s ~ 18.15 min
		g=25000	15995.89	21393.39	20711.79	11224 s ~ 3.11 ore
	p=10000	g=1000	38114.10	57223.27	56275.58	4481 s ~ 1.2 ore

p=populatie, g=generatie

Concluzii

- un numar mai mare de generatii are mai multa pondere in oferirea unei solutii mai bune decat numarul de indivizi din populatie
- rezultatele sunt mult mai bune decat cu metoda simulating annealing din laboratorul 2 (se poate consulta tabelul din documentatia pentru lab2)
- pe aceleasi configuratii de parametri, varianta 2 a algoritmului(cea care are operator de mutatie scramble mutation) da rezultate mai slabe