Mattia Colbertaldo - Assignment 2 AI

Here I provide a brief explanation and discussion about my findings and report code fragments, plots and data as evidence of completion of the tasks. This file comes with a Jupyter notebook where each function in commented to be self-explanatory enough.

- Initialization:
  - Here I simply generate a random population of shape `population_size` and `dimension (2D, 3D, 4D, …)` with values between a given range. I return the produced population encoded using gray code.

- Selection
  - Roulette:

    Since I have to find the minimum fitness, the probabilities have to be inversely proportional to the fitness.

    ```
    selection_probabilities = [1 / fit if fit != 0 else 1 for fit in self.population_fitness]
    ```

    After that, I normalize so they sum up to 1.

    ```
    selection_probabilities = selection_probabilities / np.sum(selection_probabilities)
    ```

  - Rank:
    I sticked to the paper, so I calculate the selection probabilities as

    ```
    Pi = 1 / N * ( Pworst + (Pbest - Pworst ) *((i-1) / (N-1)))
    ```

    Where `Pbest` and `Pworst` are the minimum and maximum fitness in the population, respectively, since I am trying to find a minimum.
    I finally normalized them to make them sum up to 1.

- Crossover:
  - Here I perform a single-point crossover on each pair of parents, selected by pairing the first half of the parents with the second half of them.

- Mutation:
  - In this function I implemented a loop that, for each bit of the encoded individual, it decides whether to flip the current bit or not, given a mutation rate.

- Reconstruction

- Elitism: the new population is composed by both the parents (population_size) and children (population size). So, at the next iteration, I firstly select them through the selection method, so that they go from being 2N to N, and after that I make the new individuals using the crossover.
- No elitism: the new generation is made up only by the children.

- Evaluation:
  - Here I simply updated the statistics (mean and standard deviation) as well as managed keeping track of the best solution and its objective value. The only thing I would like to point out is that I update the array of the best fitness value at each iteration, so that I can have a plot that goes up if the current solution is worse, but I obviously keep safe the overall best solution.

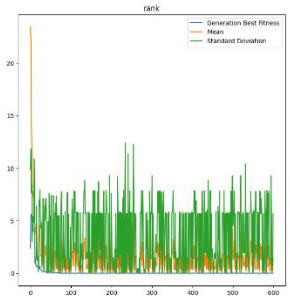
```
# Evaluate population
# ...
self.update_statistics()

self.compute_fitness()
indices = list(range(self.population_size))
indices.sort(key=lambda i: self.population_fitness[i])
self.population_array = [self.population_array[i] for i in indices]

if(iteration==0 or self.best_fitness>self.population_fitness[0]):
  self.best_iteration=iteration
  self.best_solution=self.population_array[0]
  self.best_fitness = self.population_fitness[0]

# Outside the if block so I can plot the current best fitness,
# even if it is worse than the best one
self.best_fitness_values.append(self.population_fitness[0])
```

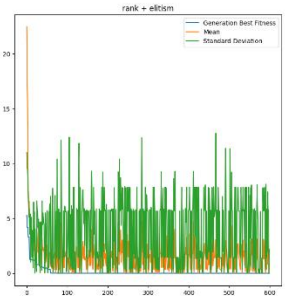
This is the main “solve” method:

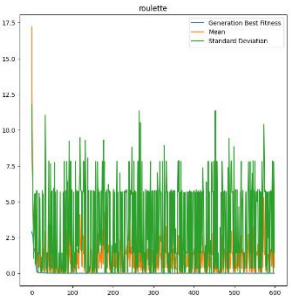
```
def solve(self):
  # print(self.selection_rule, self.elitism)
  """
  This function is the main of the class `GeneticAlgorithm`. It performs the main loop of the algorithm,
  and it calls the other functions of the class.
  """
  #initialize population ...
  self.initialize_population()

  # Evaluate population
  # ...
  self.compute_fitness()
  self.best_solution=self.population_fitness[0]


  for iteration in range(self.iterations):

    # Select parents
    # ...
    parents_twenty=self.selection()

    # Generates new individuals using reproduction rules
    # ...

    #crossover
    children=[]
    while(len(children)<self.population_size*10):
      for i in range(0, len(parents_twenty), 2):
        parents_two = parents_twenty[i], parents_twenty[(i + 1)%len(parents_twenty)]
        child1, child2 = self.crossover(parents_two)
        children.append(child1)
        children.append(child2)

    #one bit mutation
    children= self.mutation(children)

    # Form a new population
    # ...
    self.reconstruction(parents_twenty, children)
```

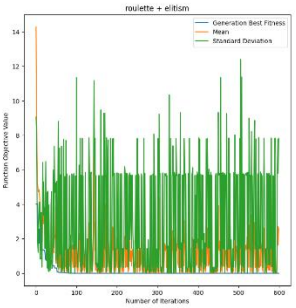
```
    # Evaluate population
    # ...
    self.update_statistics()

    self.compute_fitness()
    indices = list(range(self.population_size))
    indices.sort(key=lambda i: self.population_fitness[i])
    self.population_array = [self.population_array[i] for i in indices]

    if(iteration==0 or self.best_fitness>self.population_fitness[0]):
      self.best_iteration=iteration
      self.best_solution=self.population_array[0]
      self.best_fitness = self.population_fitness[0]

    # Outside the if block so I can plot the current best fitness,
    # even if it is worse than the best one
    self.best_fitness_values.append(self.population_fitness[0])
```

De Jong 1, Random Seed: 0

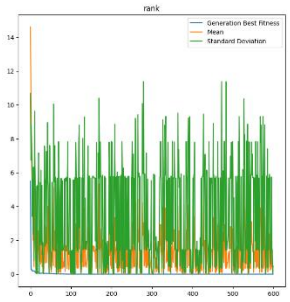


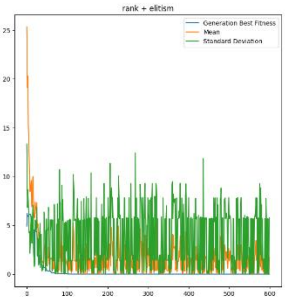


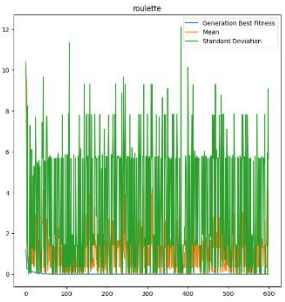


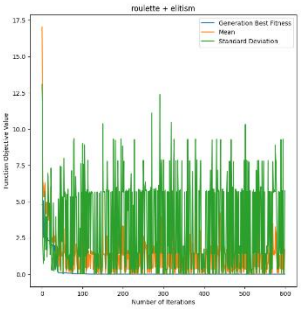


De Jong 1, Random Seed: 42

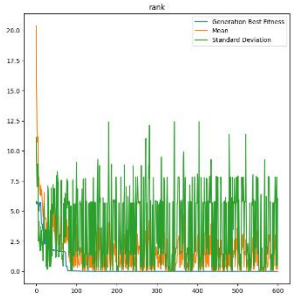


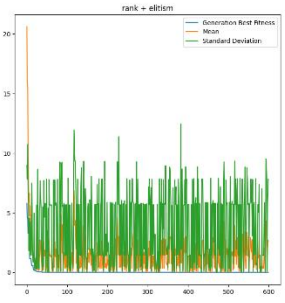


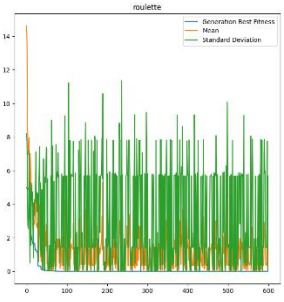


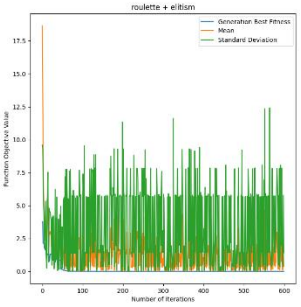


De Jong 1, Random Seed: 666

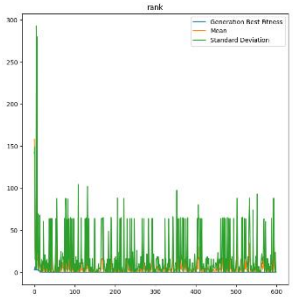


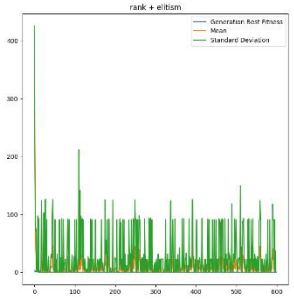


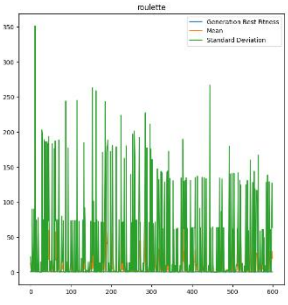


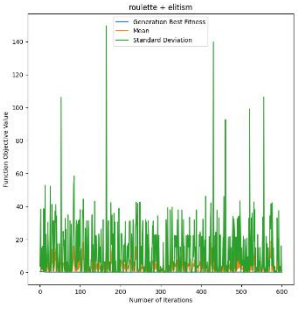


De Jong 2, Random Seed: 0

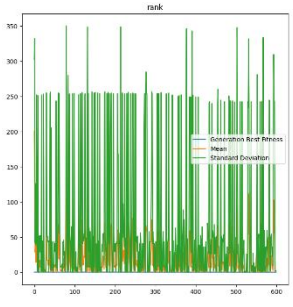


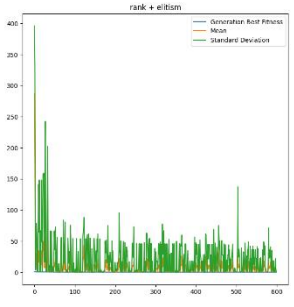


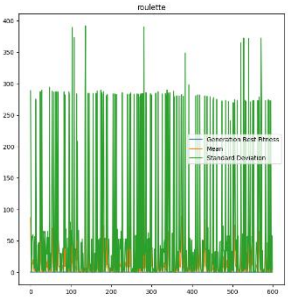


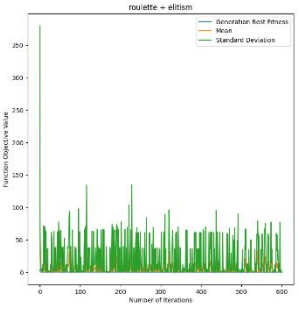


De Jong 2, Random Seed: 42

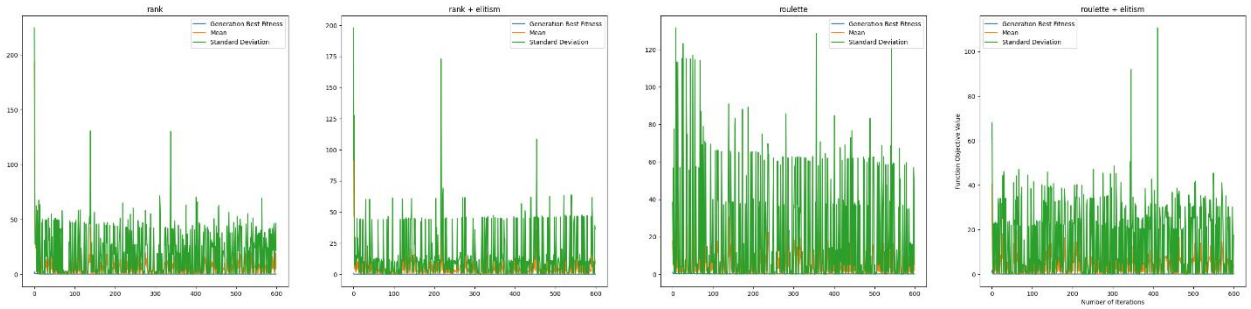




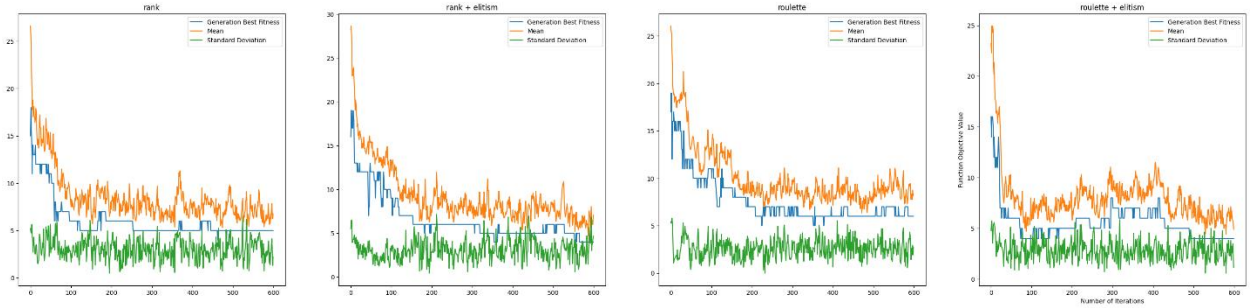




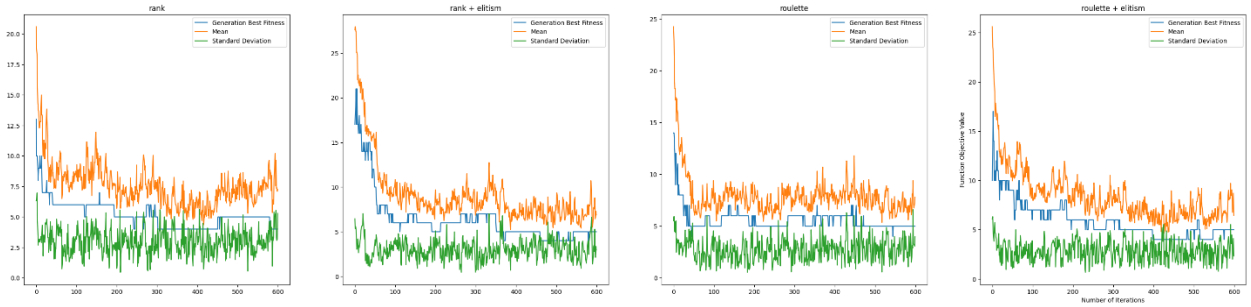
De Jong 2, Random Seed: 666



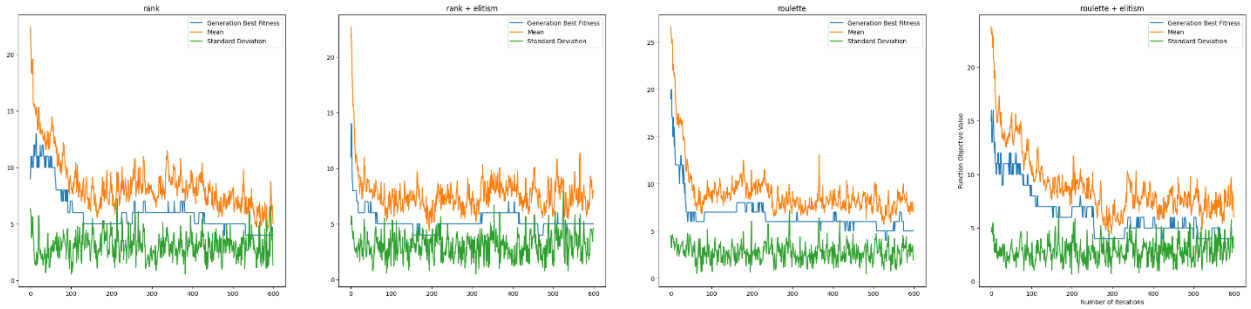
De Jong 3, Random Seed: 0



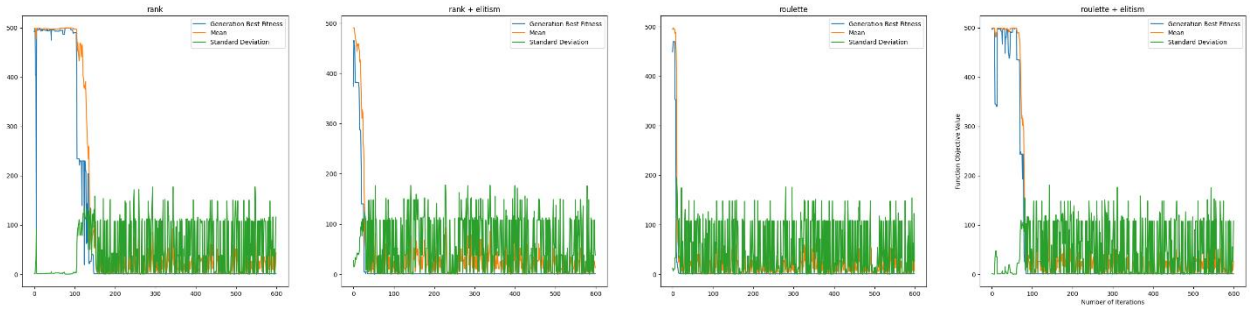
De Jong 3, Random Seed: 42



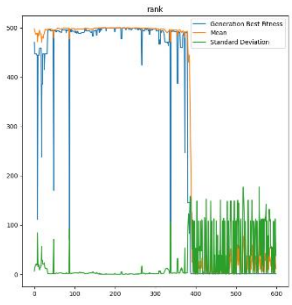
De Jong 3, Random Seed: 666

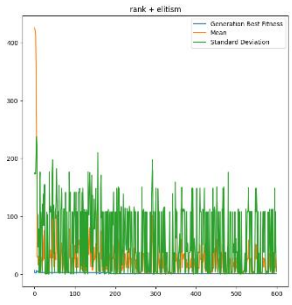


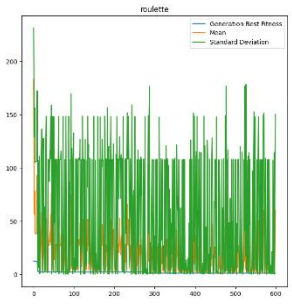
De Jong 5, Random Seed: 0

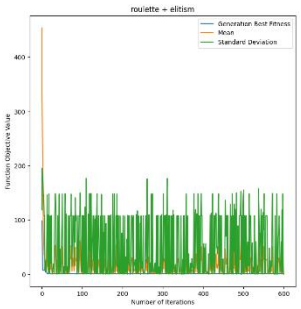


De Jong 5, Random Seed: 42

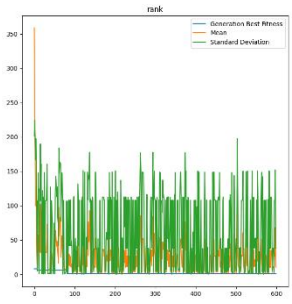


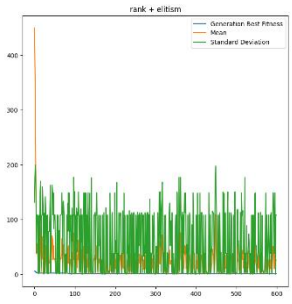


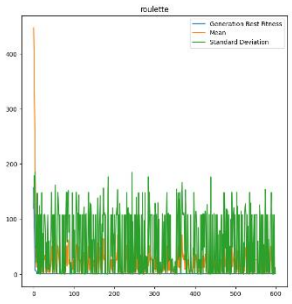


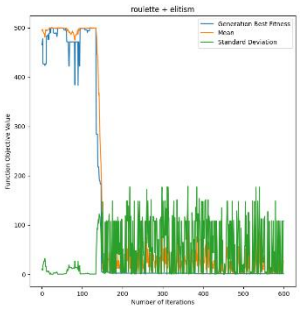


De Jong 5, Random Seed: 666



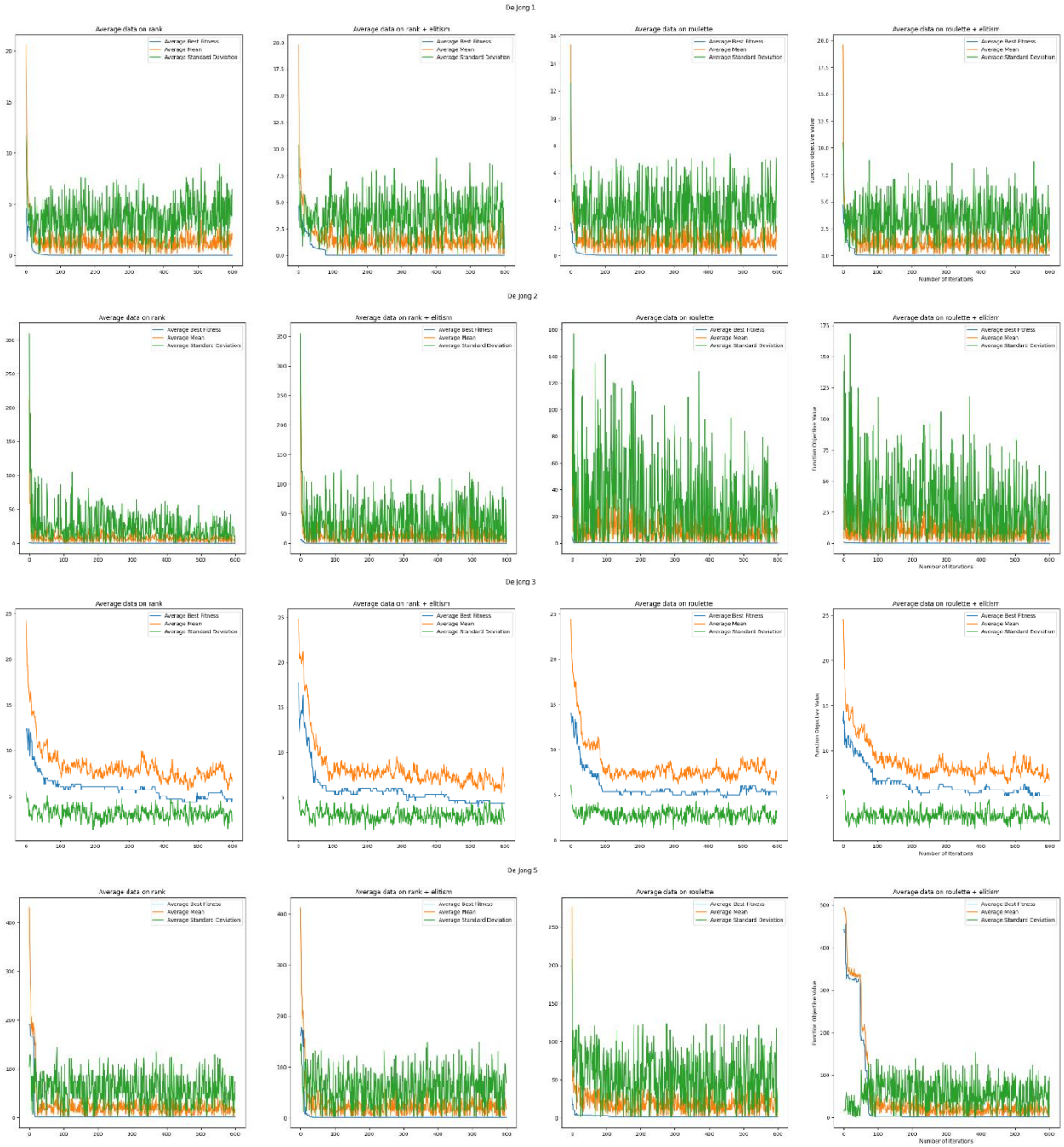






	Function	Method	Seed	Best solution	Best objective value	Best iteration
0	De Jong 1	rank	0	[1100000000, 1100000000, 1100000000]	0.000000	105
1	De Jong 1	rank + elitism	0	[1100000000, 1100000000, 1100000000]	0.000000	118
2	De Jong 1	roulette	0	[1100000000, 1100000000, 1100000000]	0.000100	79
3	De Jong 1	roulette + elitism	0	[1100000000, 1100000000, 1100000000]	0.000100	112
4	De Jong 1	rank	42	[1100000000, 1100000000, 1100000000]	0.000000	155
5	De Jong 1	rank + elitism	42	[1100000000, 1100000000, 1100000000]	0.000000	184
6	De Jong 1	roulette	42	[1100000000, 1100000000, 1100000000]	0.000100	84
7	De Jong 1	roulette + elitism	42	[1100000000, 1100000000, 1100000000]	0.000100	194
8	De Jong 1	rank	666	[1100000000, 1100000000, 1100000000]	0.000000	168
9	De Jong 1	rank + elitism	666	[1100000000, 1100000000, 1100000000]	0.000000	110
10	De Jong 1	roulette	666	[1100000000, 1100000000, 1100000000]	0.000100	111
11	De Jong 1	roulette + elitism	666	[1100000000, 1100000000, 1100000000]	0.000100	140
12	De Jong 2	rank	0	[111001110001, 111011110001]	0.005499	483
13	De Jong 2	rank + elitism	0	[111000001011, 111000000010]	0.000101	69
14	De Jong 2	roulette	0	[110101100101, 110011100011]	0.320843	516
15	De Jong 2	roulette + elitism	0	[111010011101, 111101011011]	0.048043	291
16	De Jong 2	rank	42	[101110001110, 100111000010]	0.084691	421
17	De Jong 2	rank + elitism	42	[111110101000, 111100000111]	0.079053	591
18	De Jong 2	roulette	42	[101110111011, 100110110010]	0.108550	500
19	De Jong 2	roulette + elitism	42	[111101000000, 110101010111]	0.131039	598
20	De Jong 2	rank	666	[111110100010, 110100001011]	0.086191	591
21	De Jong 2	rank + elitism	666	[111011111101, 111110110101]	0.021326	548
22	De Jong 2	roulette	666	[111111011010, 110101101101]	0.121105	594
23	De Jong 2	roulette + elitism	666	[111110001011, 111100100110]	0.069238	251
24	De Jong 3	rank	0	[0000001110, 0000001010, 0000001010, 0000001101]	5.000000	462
25	De Jong 3	rank + elitism	0	[0000001010, 0000001101, 0000001101, 0000001101]	4.000000	354
26	De Jong 3	roulette	0	[0000001101, 0000001111, 0001111110, 0000000111]	6.000000	224
27	De Jong 3	roulette + elitism	0	[0000000110, 0000001100, 0000001111, 0000000011]	4.000000	73
28	De Jong 3	rank	42	[0000001010, 0000001101, 0000000101, 0000000101]	4.000000	242
29	De Jong 3	rank + elitism	42	[0000000001, 0000000010, 0000000100, 0000000010]	5.000000	370
30	De Jong 3	roulette	42	[0000000110, 0000000001, 0000001100, 0000000001]	5.000000	45
31	De Jong 3	roulette + elitism	42	[0000001111, 0000001100, 0000001010, 0000000111]	5.000000	398
32	De Jong 3	rank	666	[0000001010, 0000000100, 0000000000, 0000000010]	4.000000	478
33	De Jong 3	rank + elitism	666	[0000000100, 0000001010, 0000001110, 0000000011]	5.000000	154
34	De Jong 3	roulette	666	[0000000100, 0000000010, 0000001101, 0000001110]	5.000000	530
35	De Jong 3	roulette + elitism	666	[0000001110, 0000000011, 0000000001, 0000001101]	4.000000	255
36	De Jong 5	rank	0	[01100001010011101, 01100001010011101]	0.998004	289
37	De Jong 5	rank + elitism	0	[01100001010011101, 01100001010011101]	0.998004	149
38	De Jong 5	roulette	0	[01100001110011001, 01100001010011011]	0.998183	77
39	De Jong 5	roulette + elitism	0	[01100001010010100, 01100001010001111]	0.998058	341
40	De Jong 5	rank	42	[01100001010011101, 01100001010011101]	0.998004	538
41	De Jong 5	rank + elitism	42	[01100001010011101, 01100001010011101]	0.998004	381
42	De Jong 5	roulette	42	[01100001110000111, 01100001010111101]	0.998304	305
43	De Jong 5	roulette + elitism	42	[01100001010011100, 01100001010010010]	1.003792	322
44	De Jong 5	rank	666	[01100001010011101, 01100001010011101]	0.998004	326
45	De Jong 5	rank + elitism	666	[01100001010011101, 01100001010011101]	0.998004	155
46	De Jong 5	roulette	666	[01100001010011110, 01100001010001000]	0.998133	346
47	De Jong 5	roulette + elitism	666	[01100001110011000, 01100001110111111]	0.998248	540

Now I have averaged the results over the three seeds, and plotted the results. As you can see, the plots are very similar to the non-averaged ones, that's because I had similar results among the three seeds.



I have averaged the best objective value, the best iteration, the mean and the standard deviation.

For the best solution (encoded), I kept the best (or the first of the best) among the three seeds, since I though it wouldn't have sense to average the input when there could be multiple minima: the three seeds could reach different minima, and averaging the input could give us an input that, substituted in the function, would correspond to a much higher objective value.

	Function	Method	Best solution	Best objective value	Best iteration
0	De Jong 1	rank	[1100000000, 1100000000, 1100000000]	0.000000	119.000000
1	De Jong 1	rank + elitism	[1100000000, 1100000000, 1100000000]	0.000000	122.666667
2	De Jong 1	roulette	[1100000000, 1100000000, 1100000000]	0.000067	102.000000
3	De Jong 1	roulette + elitism	[1100000000, 1100000000, 1100000000]	0.000033	95.333333
4	De Jong 2	rank	[111011000000, 111010110001]	0.071744	573.666667
5	De Jong 2	rank + elitism	[111001010011, 111011110100]	0.058087	401.000000
6	De Jong 2	roulette	[111000101000, 111001101101]	0.132475	556.000000
7	De Jong 2	roulette + elitism	[111000001001, 111000000111]	0.125535	377.000000
8	De Jong 3	rank	[0000000110, 0000000100, 0000000010, 0000000110]	4.666667	457.666667
9	De Jong 3	rank + elitism	[0000000111, 0000000001, 0000001100, 0000001101]	4.333333	282.333333
10	De Jong 3	roulette	[0000000010, 0000001100, 0000000010, 0000001101]	5.000000	377.333333
11	De Jong 3	roulette + elitism	[0000001010, 0000000011, 0000000110, 0000000110]	5.000000	295.000000
12	De Jong 5	rank	[01100001010011101, 01100001010011101]	0.998004	179.666667
13	De Jong 5	rank + elitism	[01100001010011101, 01100001010011101]	0.998004	151.666667
14	De Jong 5	roulette	[01100001110001101, 01100001010011110]	0.998574	402.000000
15	De Jong 5	roulette + elitism	[01100001010011111, 01100001010001001]	1.000321	425.666667