Artificial Intelligence **2023/2024**

Student: LUCA PERNIGO StudentID:19-993-658

Assignment 2 Due date: 22 November 2023, 23:59

Genetics Algorithms for the De Jong functions optimisation

1. GA Implementation

The methods that make up the GA class have been implemented, in order to search for the global minimum of De Jong functions: n.1, n.2, n.3 and n.5.

1.1. initialise_population

In order to initialize the population, a matrix with dimension population_size by dimension of the problem is generated. Each element of this matrix has been sampled uniformly over the domain of the dj_instance passed as input.
The generated population is then gray encoded and returned.

1.2. compute_fitness

First of all, notice that all the De Jong functions considered here are positive functions, i.e.
$f(x) > 0 \quad \forall x$.
Therefore, a valid candidate fitness for minimizing the considered De Jong functions is $\frac{1}{1+f(x)}$. Where $f(x)$ is the objective function that we want to minimize.

1.3. selection

Based on the input parameters, compute selection probabilities either with rank or roulette rule.
N parents are then selected from the population; in our experiments we have N=20 .

1.4. rank_selection

Rank every individual by its fitness with a number from 1 to N. Then divide every rank by the normalizing factor $\frac{N*(N+1)}{2}$, i.e. the sum of all ranks. Doing so we assign every individual a probability proportional to how they rank in terms of fitness inside the whole population.

1.5. roulette_selection

Normalize the vector of fitnesses, in this way every individual is assigned a probability proportional to its fitness.

1.6. crossover

Divide the array of parents in two smaller arrays parent1 and parent2; each array has half the length of the parents array. For every pair i of parents(parent1[i]+parent2[i]) generate two children. Children are created by mixing the elements of the chromosome arrays of their parents; in these experiments single point crossover has been employed.

1.7. mutation

For every bit in the chromosome array of each children, the bit is mutated with a probability of 0.01. That is mutation happens if and only if a value smaller or equal to 0.01 is sampled from the uniform distribution over the interval [0,1]. If the bit to be mutated is a 1, it is converted to a 0 bit, on the other hand if the bit that undergoes mutation is a 0 bit, it is transformed into a 1 bit.

1.8. elitism_selection

It concatenates the current population and the mutated one. Within this pool of individuals select the N best in terms of fitness.

1.9. reconstruction

If the genetic algorithm is run with elitism equal to true then the new population_array is the output of elitism_selection. If elitism equals to false, then the new population is equal to the mutated one.

1.10. update_statistics

Method that keeps track of various summary statistics: mean objective value and std along each generation, best fitness over each generation, best solution of the generation both gray encoded and decoded and best objective value of the generation. Additionally, for each of these variables there exist a list that keep their realization at every iteration of the solve loop.

2. GA Settings and Parameter Combinations

The GA class has been configured with the following settings:

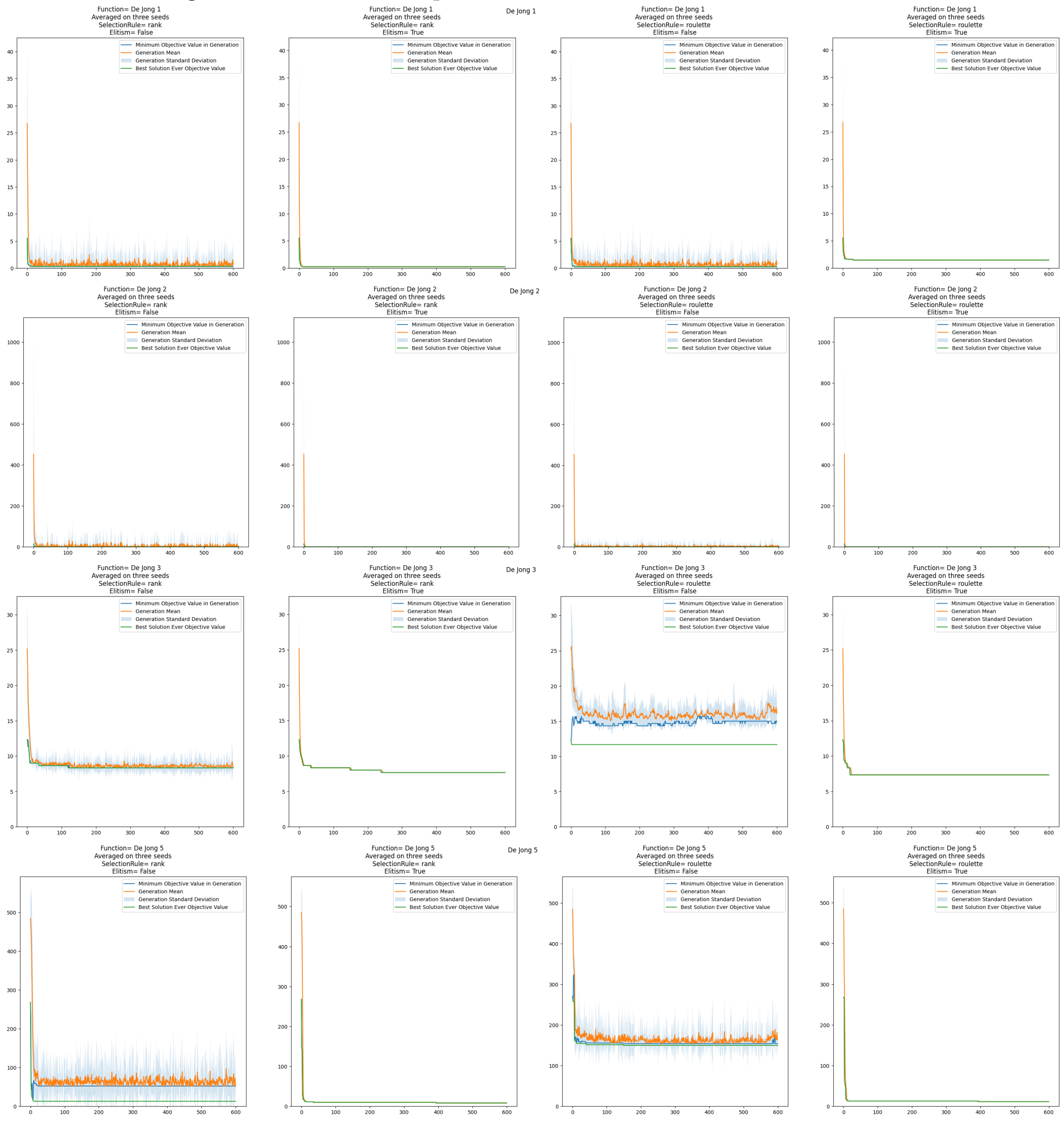
- Population size: 20
- Crossover method: Single-Point
- Mutation type: Bit string with a rate of 1%
- Iterations: 600
- Random: Three different starting seeds

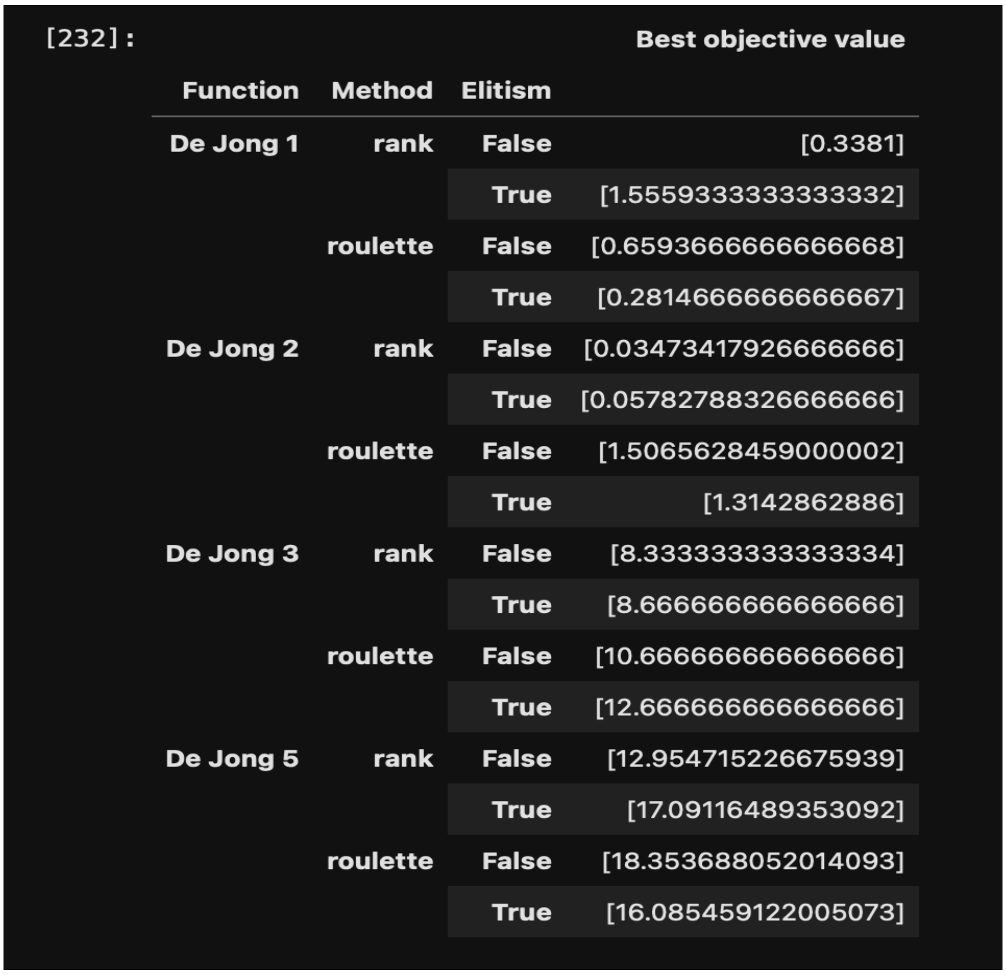
The GA has been tested with the following selection rules:

- Rank
- Roulette
- Rank with Elitism
- Roulette with Elitism

3. Results

The genetic algorithm has been run trying all combinations of the above mentioned parameters. Moreover, each combination of parameters has been run with three different random seeds. The results of the experiments have been visualized by plotting average objective value across iterations (orange line), objective function standard deviation across iterations (light-blue band), minimum objective value across iterations (blue line) and cumulative minimum of the objective value (green line). In addition, a data-frame containing the best solution and the best objective value obtained for each set of parameter combinations and seeds has been created. Hereafter, the plots and the data-frame averaged on the seeds are reported.

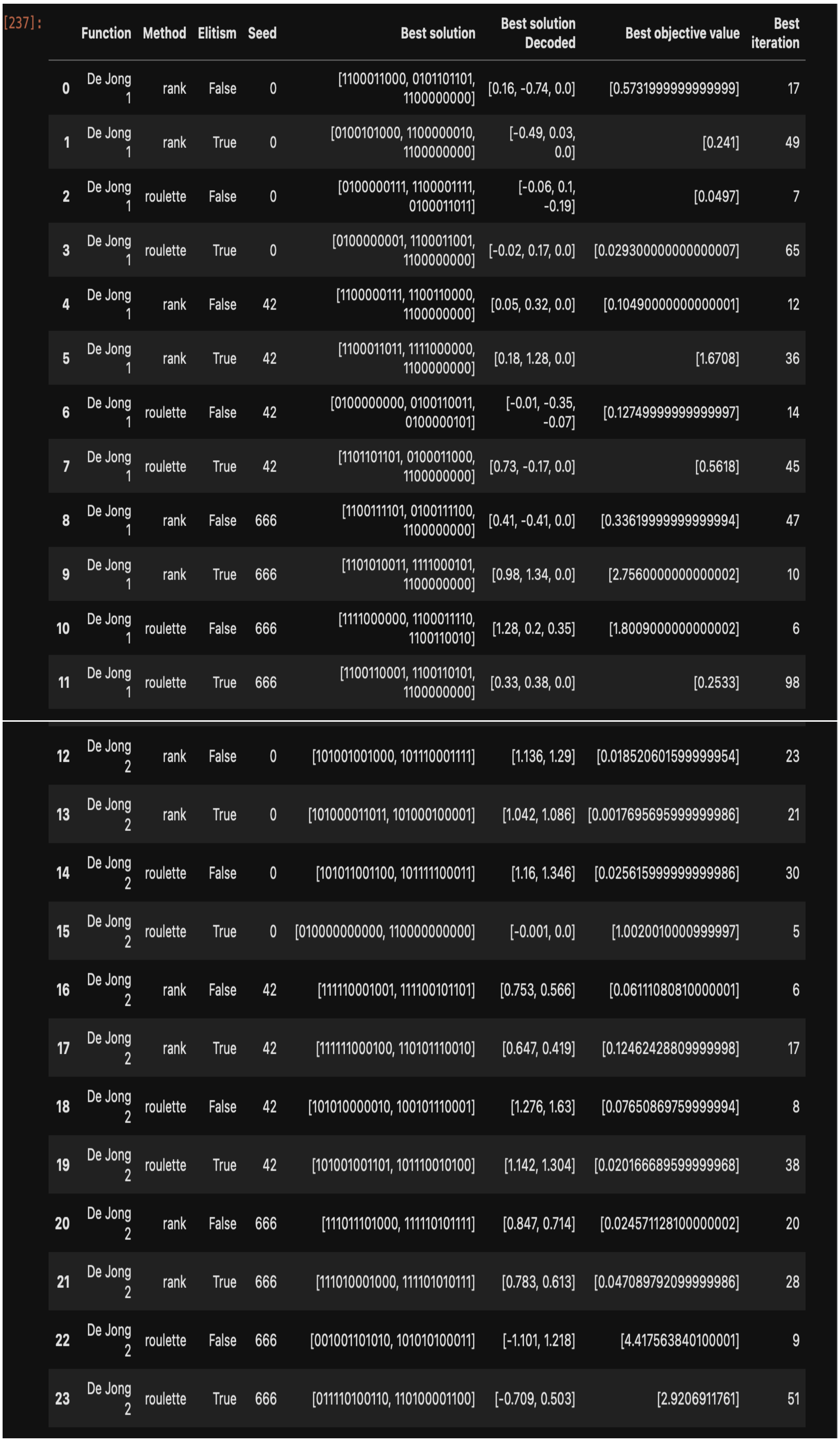




[232]:

Function	Method	Elitism	Best objective value
De Jong 1	rank	False	[0.3381]
		True	[1.5559333333333332]
	roulette	False	[0.6593666666666668]
		True	[0.2814666666666667]
De Jong 2	rank	False	[0.0347341792666666]
		True	[0.0578278832666666]
	roulette	False	[1.5065628459000002]
		True	[1.3142862886]
De Jong 3	rank	False	[8.333333333333334]
		True	[8.666666666666666]
	roulette	False	[10.666666666666666]
		True	[12.666666666666666]
De Jong 5	rank	False	[12.954715226675939]
		True	[17.09116489353092]
	roulette	False	[18.353688052014093]
		True	[16.085459122005073]

We can see that Genetic algorithm permofed well of the instnaces De Jong 1 and De Jong 2, achieving an objective value close to zero. Conversely, for De Jong 3 and De Jong 5 there is room for improvement.
Furthermore, notice that the choice of the selection method (either rank or roulette) does not result in a significant difference in terms of performance.
Finally, it is interesting to consider the effect of enabling elitism in the genetic algorithm. To do so, consider the following pictures that report the results obtained for De Jong 1 and De Jong 2.
By looking at those data-frames, we can see that on average, enabling elitism results in better performance, that is in smaller objective values.



[237]:

	Function	Method	Elitism	Seed	Best solution	Best solution Decoded	Best objective value	Best iteration
0	De Jong 1	rank	False	0	[1100011000, 0101101101, 1100000000]	[0.16, -0.74, 0.0]	[0.5731999999999999]	17
1	De Jong 1	rank	True	0	[0100101000, 1100000010, 1100000000]	[-0.49, 0.03, 0.0]	[0.241]	49
2	De Jong 1	roulette	False	0	[0100000111, 1100001111, 0100011011]	[-0.06, 0.1, -0.19]	[0.0497]	7
3	De Jong 1	roulette	True	0	[0100000001, 1100011001, 1100000000]	[-0.02, 0.17, 0.0]	[0.029300000000000007]	65
4	De Jong 1	rank	False	42	[1100000111, 1100110000, 1100000000]	[0.05, 0.32, 0.0]	[0.10490000000000001]	12
5	De Jong 1	rank	True	42	[1100011011, 1111000000, 1100000000]	[0.18, 1.28, 0.0]	[1.6708]	36
6	De Jong 1	roulette	False	42	[0100000000, 0100110011, 0100000101]	[-0.01, -0.35, -0.07]	[0.12749999999999997]	14
7	De Jong 1	roulette	True	42	[1101101101, 0100011000, 1100000000]	[0.73, -0.17, 0.0]	[0.5618]	45
8	De Jong 1	rank	False	666	[1100111101, 0100111100, 1100000000]	[0.41, -0.41, 0.0]	[0.33619999999999994]	47
9	De Jong 1	rank	True	666	[1101010011, 1111000101, 1100000000]	[0.98, 1.34, 0.0]	[2.7560000000000002]	10
10	De Jong 1	roulette	False	666	[1111000000, 1100011110, 1100110010]	[1.28, 0.2, 0.35]	[1.8009000000000002]	6
11	De Jong 1	roulette	True	666	[1100110001, 1100110101, 1100000000]	[0.33, 0.38, 0.0]	[0.2533]	98
12	De Jong 2	rank	False	0	[101001001000, 101110001111]	[1.136, 1.29]	[0.018520601599999954]	23
13	De Jong 2	rank	True	0	[101000011011, 101000100001]	[1.042, 1.086]	[0.0017695695999999986]	21
14	De Jong 2	roulette	False	0	[101011001100, 101111100011]	[1.16, 1.346]	[0.025615999999999986]	30
15	De Jong 2	roulette	True	0	[010000000000, 110000000000]	[-0.001, 0.0]	[1.0020010000999997]	5
16	De Jong 2	rank	False	42	[111110001001, 111100101101]	[0.753, 0.566]	[0.06111080810000001]	6
17	De Jong 2	rank	True	42	[111111000100, 110101110010]	[0.647, 0.419]	[0.12462428809999998]	17
18	De Jong 2	roulette	False	42	[101010000010, 100101110001]	[1.276, 1.63]	[0.07650869759999994]	8
19	De Jong 2	roulette	True	42	[101001001101, 101110010100]	[1.142, 1.304]	[0.020166689599999968]	38
20	De Jong 2	rank	False	666	[111011101000, 111110101111]	[0.847, 0.714]	[0.024571128100000002]	20
21	De Jong 2	rank	True	666	[111010001000, 111101010111]	[0.783, 0.613]	[0.047089792099999986]	28
22	De Jong 2	roulette	False	666	[001001101010, 101010100011]	[-1.101, 1.218]	[4.417563840100001]	9
23	De Jong 2	roulette	True	666	[011110100110, 110100001100]	[-0.709, 0.503]	[2.9206911761]	51