University of Edinburgh

School of Informatics

Assignment

Natural Computing

Fall 2023

Miquel Muñoz García-Ramos

Contents

1 Problem 1: The effect of the population size.

For the first problem it are asked to investigate the effect of co-operation in a population by means of a population-based methauristic algorithm. It has been decided to implement a PSO algorithm. Particles moving through space with certain positions and velocities are defined. The movements are influenced by their best local position but also by the best positions found globally by other particles. With this mechanism it is expected to arrive at the optimal global solutions.

This algorithm consists of three main parameters:

- Inertia (w): inertia is the tendency to retain a certain movement. A high value of inertia tends to make the particles maintain their current direction, while a low value allows a greater exploration of the search space.
- Attraction to the local best (a1): Also known as the cognitive coefficient, this determines the influence of the best known value for an individual particle on its motion. A high value of c1 favours exploitation by making particles move towards the best known local optimum.
- Attraction to the global best (a2): Also known as the social coefficient, this indicates the influence of the global best position on a particle's motion. A high value of c2 favours exploration by making particles move towards the global optimum found by any particle in the swarm.

As objective function, it has been decided to choose the Rastrigin function. This is a non-convex function used for optimisation algorithms. In the investigation it will be tried to minimise the function to find the global minimum (which is at the point x = 0 where f(x) = 0). Since it has many local minima but only one global minimum it is useful to measure the performance of our investigation.

In order to be able to measure a total number of evaluations of the cost function, a fixed number of iterations has been defined. In this way we can observe without benefiting the larger populations from having more particles.

The populations range is 1 to 100. The number of timesteps was chosen to be 5000 so that for the larger populations the iterations per particle would be

sufficiently large. As for the parameters of inertia, attraction to the local maximum and attraction to the global maximum, it has been chosen the standardised value proposed in the course (w = 0.7, a1 = 2.02 and a2 = 2.02) and other values that have also been proven to give solutions with good performance depending on the problem (w = 0.5, a1 = 1.5 and a2 = 1.5). It is also worth noting that as there is randomness to start with the initial solution, and for the calculation of velocities it is important to have a sufficient sampling (10 trials have been done with each of the experiments and the average has been computed).

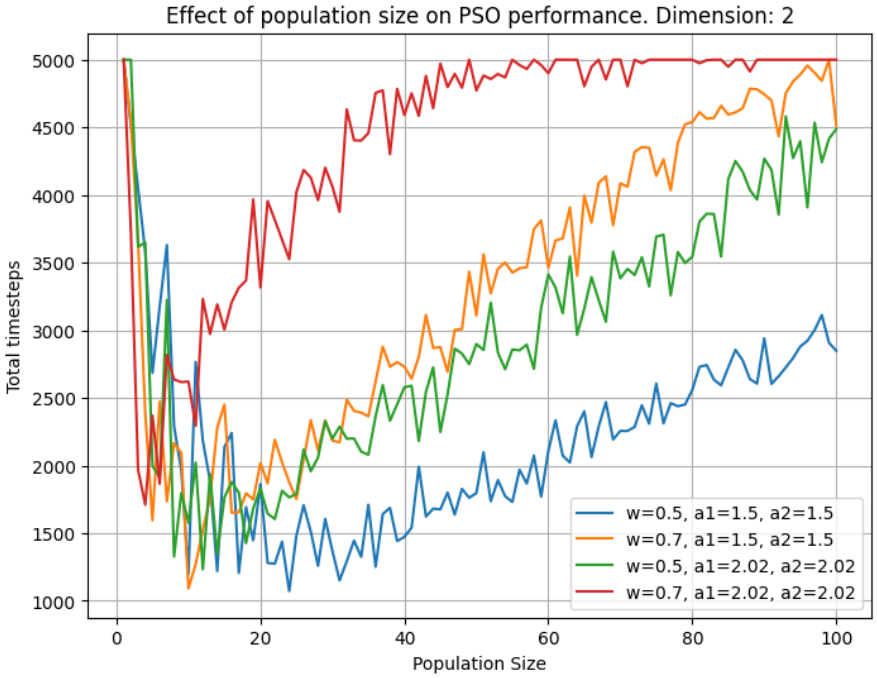

Figure 1: Effect of population size on PSO performance (dimension 2)

Analyzing Table 1 it can be seen that the parameters that give the best results are a1 = 1.5, 12 = 0.5 and w = 0.5. Knowing that the Rastrigin search function requires further exploration since there are many local minima, it is logical that it may be beneficial to explore the search space more widely. It can be observed that the optimal population of the optimal timesteps is 24 in this case of n = 2. (See Appendix to see the experiments in other dimensions regarding the total timesteps compared to the population size)

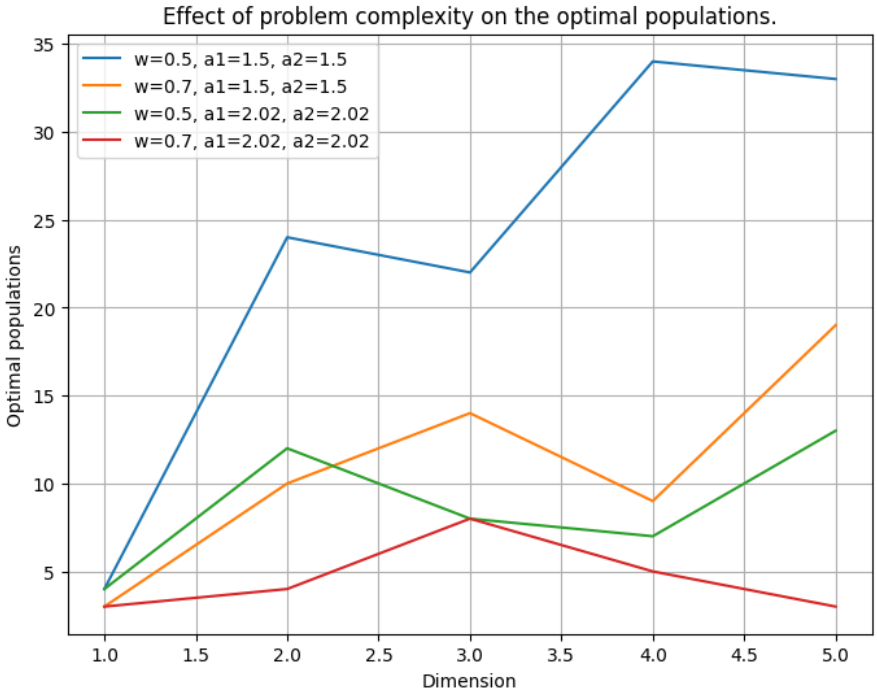

Figure 2: Effect of problem complexity on the optimal populations.

To analyze how the optimal population depends on the complexity of the problem, the optimal populations found have been compared with each of the dimensions. As can be seen in Table 2, the populations with parameters that benefit the most from exploration are those that show an upward trend in the optimal population as a function of dimension. It can be observed that as the value of the parameters decreases and the exploration increases, the more populated populations benefit from the increase in difficulty. In the case of (w1 = 1.5, w2 = 1.5 and w = 0.5) the upward trend is the most noticeable.

2 Problem 2: *Sumplete*, genetic algorithm

Sumplete is a game invented by ChatGPT that tries to find which cells must be removed so that the sum of each column and each row matches the set values.

To solve the problem a genetic algorithm has been developed. It has been chosen to encode the matrix as a vector. The reasons are that for the mutation and crossover phases it makes the management of the elements much more efficient, to be able to cut in a uniform way and to do the merging when it is due in a simple way. The only drawback is that when it has to be calculated the fitness of the function it has to be managed well the indices to iterate through the vector considering that it is a matrix coded as such (although it is not a big disadvantage in clarity and efficiency).

As for the solution, a vector of Booleans of the same size as the initial matrix has been chosen to indicate which positions are active (to be summed) and which are not.

The algorithm to solve this problem starts by generating a population of a certain number of individuals and generations. Then, random solutions are generated to generate the initial population (in each of the Boolean vectors). Afterwards, each of these solutions is measured with the fitness function in order to rank the quality of the solutions and generate a probabilistic distribution *fitness(individual)/sum(fitness(individuals))* where *individual* is an element of the population and *sum(fitness(individuals))* is the evaluation of all the individuals of the population.

Once the distribution is generated, it is proceeded to choose two parents using these probabilities iteratively until the intermediate population is generated. If the random value generated is below the crossover ratio, it is proceeded to crossover the parents to generate two children. This procedure is done by randomly generating an index along our matrix coded as a vector (starting from position 1). Then 4 partitions are generated between the two parents. 2 of these are exchanged to generate two new vectors which are called children.

The next phase is mutation, the bits of the matrix coded as a vector will be inverted, with a low probability of mutation rate. The population is updated and one generation has been completed. It only remains to evaluate if among the new

population there is an individual that solves the initial problem.

For this first part of the experimentation the following standard parameters have been chosen populationSize = 100, crossoverRate = 0.7, mutationRate = 0.01 and numGenerations = 5000.

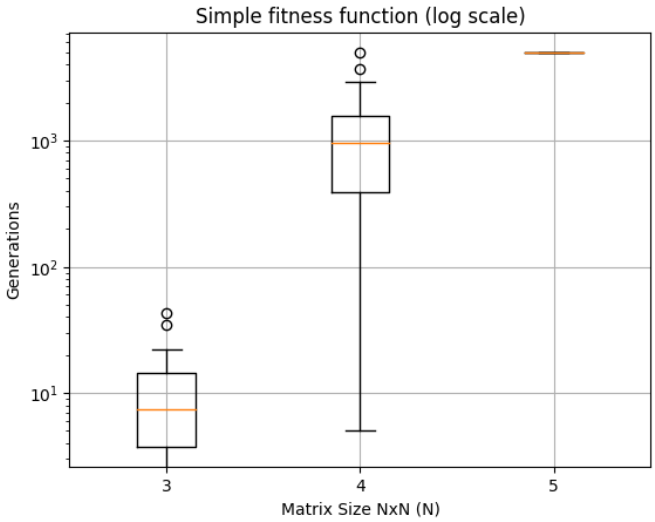

Figure 3: Basic fitness function

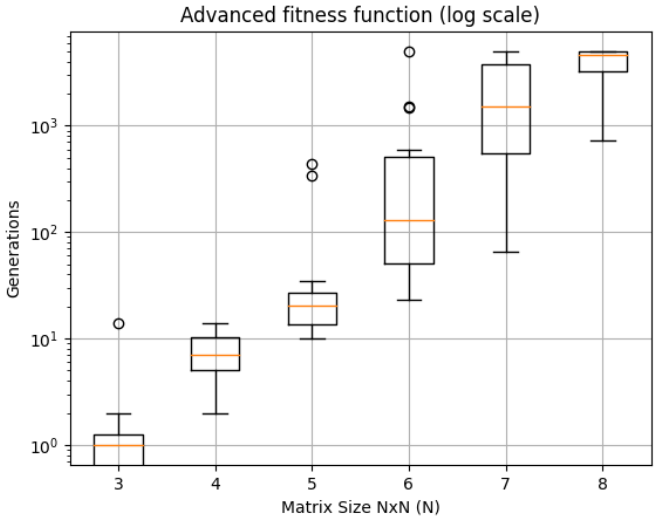

Figure 4: Advanced fitness function

The first basic fitness function that has been asked to develop, it can be seen

from the results that it does not perform the purpose it should. Since it only assigns a 1 to those correct solutions, it is not useful for the whole process up to the solution (basically it evaluates the actual solution, if it is a solution or not). This generates that, the probabilistic distribution becomes random since as long as the solution to the problem is not found, all the individuals of the population will have value 0 as fitness function.

To improve the fitness function, a new function has been implemented that calculates the number of rows plus columns that match the expected value. In this manner it can be obtained a partial evaluation that allows us to find a method to order the solutions in an efficient way.

The impact of the improved fitness function can be clearly seen in Figures 3 and 4. For those matrices with more than 5 columns and rows, the first fitness function no longer finds solutions. The optimized fitness function, on the other hand, not only improves the results in dimensions lower than 5 but also finds solutions for higher dimensions. It has been chosen to represent the results in a boxplot since the variance of the random events used in the algorithm is an important factor when analyzing the results. In this case, 20 samples have been performed for each of the experiments.

The following experiments of the parameters (Figures 5,6,7) are then carried out at n = 5 to see the variability of the performance according to other parameter values. The analysis is performed by measuring the generations in which the solution is found with the improved fitness function (with a sampling of 20 samples).

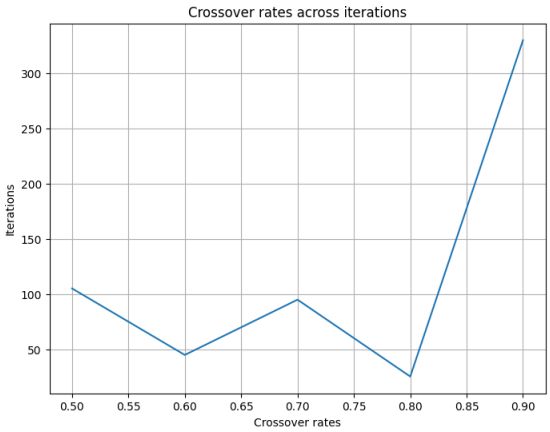

Figure 5: Mutation rates across iterations

Figure 5 shows how the crossover rate optimum is 0.8. It can be seen that the values are similar except for the value of 0.9, which clearly worsens the performance.

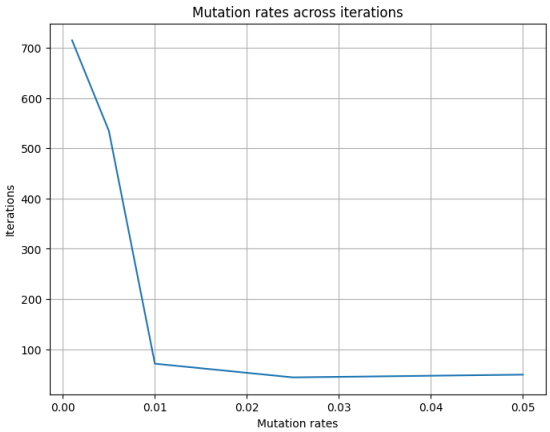

Figure 6: Mutation rates across iterations

Figure 6 shows how mutation rates lower than 0.01 worsen the results. However, the results are similar as the rate increases.

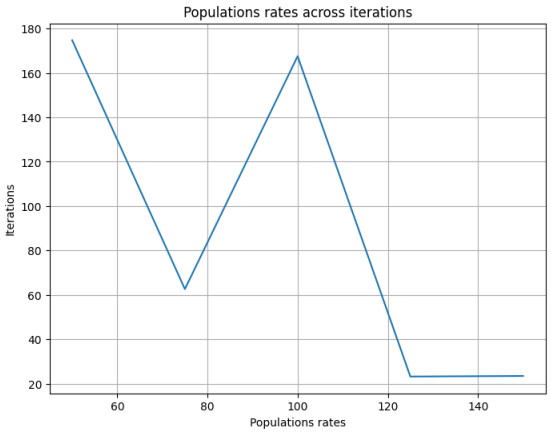

Figure 7: Enter Caption

Figure 7 shows how the higher values (125 and 150) of population generate much better results compared to the others. As for the lower values, it can be seen that there is no clear trend.

3 Problem 3: Genetic Programming

In this exercise it has been requested to develop an algorithm using genetic programming. The sequence chosen is the Jacobsthal numbers. This sequence of integers named after the mathematician Ernst Jacobsthal has been a good option to choose a sequence with different recurrences but also, with an addition and a multiplication as operator functions.

$$J_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ J_{n-1} + 2J_{n-2} & \text{if } n > 1. \end{cases}$$

Figure 8: Jacobsthal numbers

To carry out this genetic programming algorithm, the following operators and leaves have been defined:

functions = ['add', 'sub', 'mul', 'div'] terminals = ['f(n-1)','f(n-2)', -2, -1, 1, 2].

Also, she following parameters have been defined:

population size =100 maximum depth =5 mutation rate = 0.01 generations = 5000

Then a random tree has been generated for each of the individuals of the population with maximum depth 5 so that the solution space is reduced since it is been tacked into account a not very long recurrence. The random tree generates leaves with a probability of 0.3 at each level before reaching its maximum depth. This tree, has been encoded as a list of lists given the simplicity of doing it this way in Python.

Once the population is defined, the individuals have been evaluated by squaring the difference between the expected value and the expected value. Then it is adjusted (1/(1+rawError)) and finally normalized by dividing the adjusted value by the sum of all the adjusted values of the population. To evaluate the error, the function evaluateTree has been defined in which the values f(n-1) and f(n-2) are passed as a parameter (to cover the case in which the recurrences are needed).

After defining the probabilistic distribution, the crossover function allows us to randomly exchange subtrees between parents to generate the children. The crossover function has been defined as follows to generate the desired subtree ex-

change randomness. First, the minimum depth between parents 'd' is calculated. Then, a probability (1/'d') of crossover is generated which will be called each time a tree function is iterated over until the crossover is performed or the tree reaches terminals. If both children are functions, it randomly (uniform distribution) branches through either of them. Otherwise it branches through the node that is a function if there is one.

On the other hand, the mutation has been designed so that in case the node is a function with a certain probability it is pruned with a terminal node. On in contrary, if the node is initially terminal, a subtree with maximum depth 2 is generated. In this way it generates adequate variability of the solutions to be able to adequately diversify the population.

Functions have been defined that would allow other recurrences to be found. Although it could be extended by adding operations such as *ifelse* condition functions that would change the logic a bit since it would be necessary to generate 3 nodes for the children of these functions. One node to compare, another to execute in case the condition is met and the last node to execute in case the condition is not met. However, any terminal or function involving two child nodes could easily be incorporated to identify other sequences. For example, quadratic recurrences could be found by means of the multiplication operator, multiplying f(n-1) * f(n-1).

To measure the performance of the algorithm, 5 runs have been performed and the average relative errors have been computed. As well as the average generations in which a sufficiently good solution is found (a relative error limit of 0.1 has been set to consider a correct solution). The first 18 numbers of the sequence were used because larger values slowed down the execution of the program too much.

Although it does not always find the exact sequence, it can be observed that it is quite close (Figures 9 and 10).

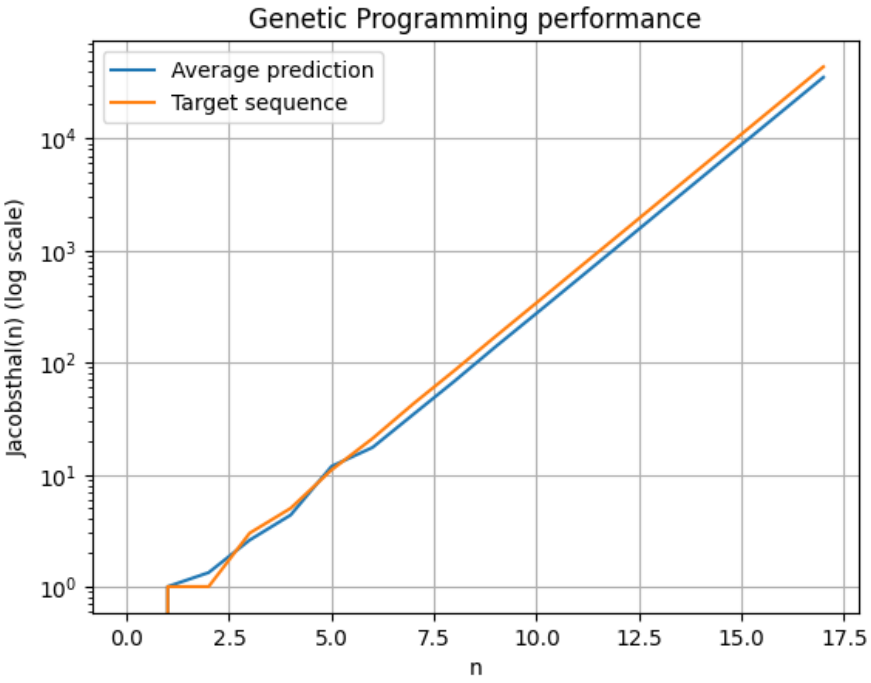

Figure 9: Genetic programming performance

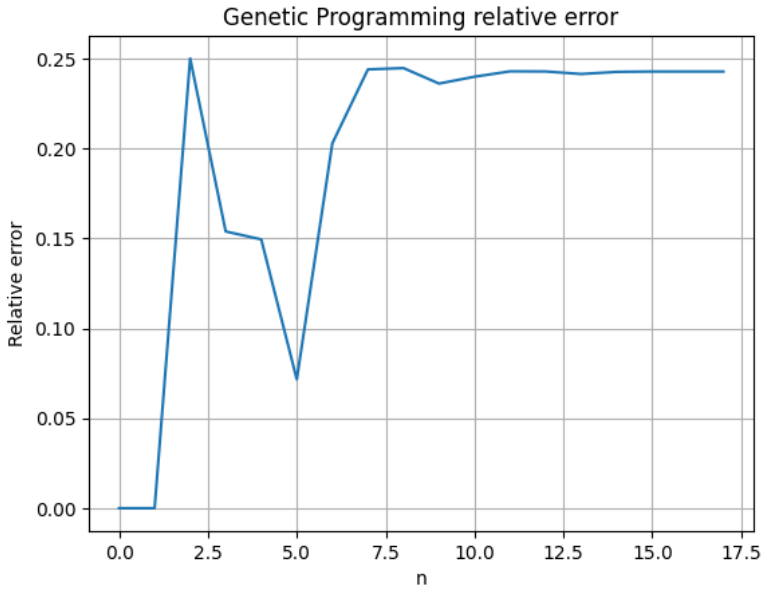

Figure 10: GP relative error

Average generations: 3944.6

References

[1] *https://en.wikipedia.org/wiki/Jacobsthal*$_n$*umber*

[2] *16.4 Basic PSO Parameters Carnegie Mellon University- https://web2.qatar.cmu.edu/ gdicaro/15382-Spring18/hw/hw3-files/pso-book-extract.pdf*

[3] Magnus Erik Hvass Pedersen *TGood Parameters Hvass Laboratories Particle Swarm Optimization Technical Report no. HL1001 2010*

4 Appendix

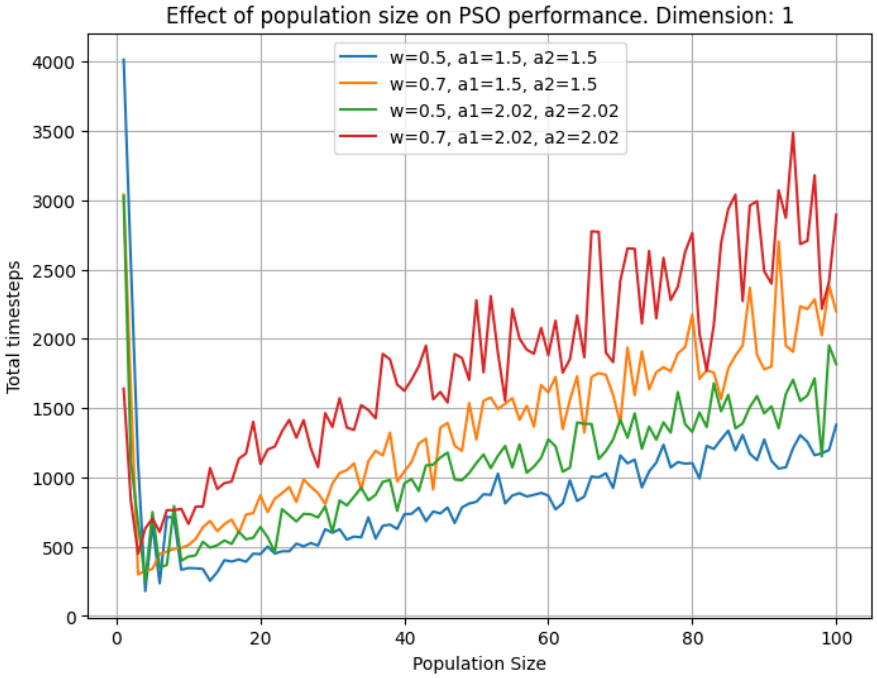

Figure 11: Effect of population size on PSO performance (dimension 1)

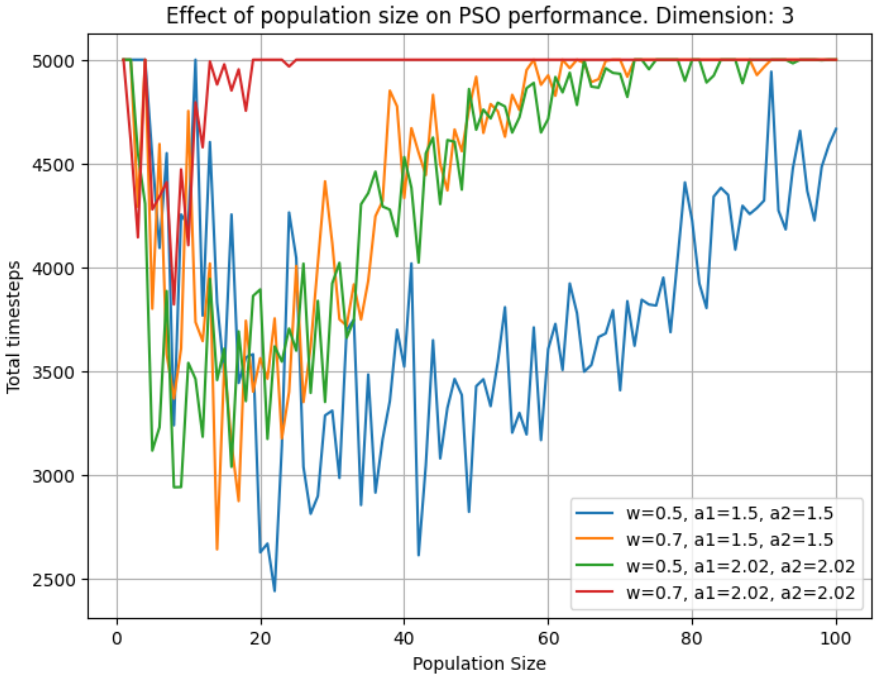

Figure 12: Effect of population size on PSO performance (dimension 3)

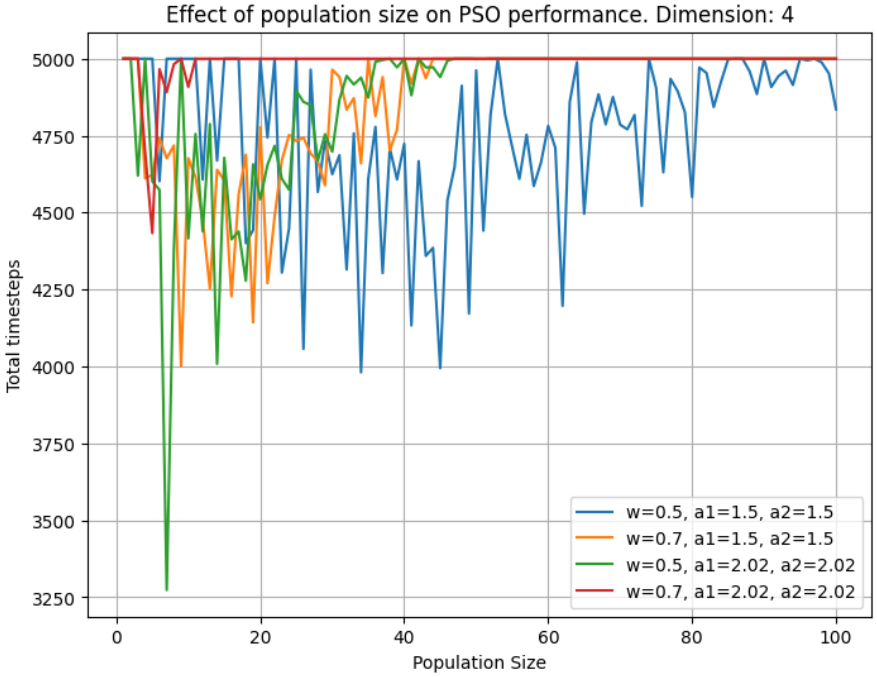

Figure 13: Effect of population size on PSO performance (dimension 4)

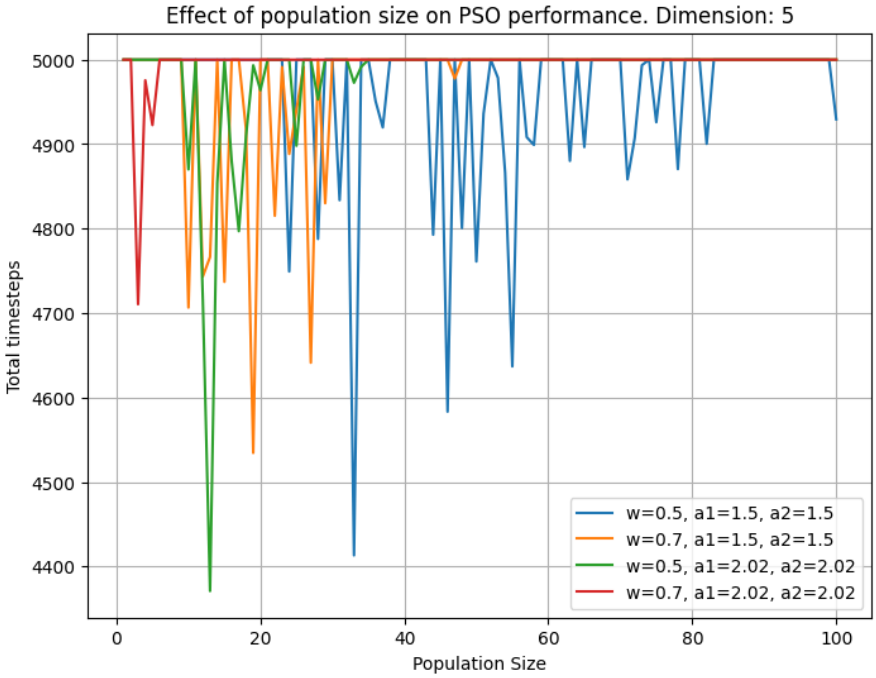

Figure 14: Effect of population size on PSO performance (dimension 5)