Universidade Nova de Lisboa

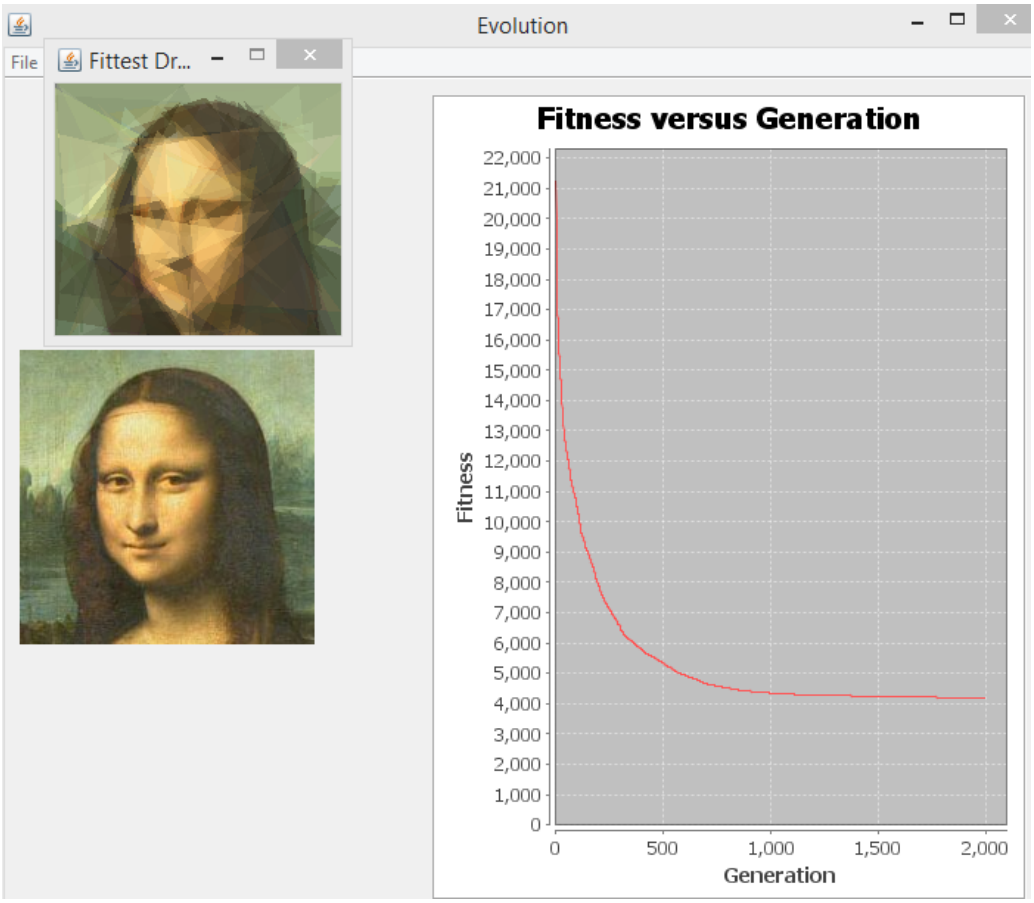
Information Management School

Year 2017/2018

Master degree program in Advanced Analytics

Computational Intelligence for Optimization

Professor Leonardo Vanneschi and Professor Illya Bakurov



Carolina Bellani M20170098

Gonçalo Pereira M20170450

Sofia Jerónimo M20170070

Table of Contents

Introduction

The following project is composed by several source codes containing the inner workings of the Mona Lisa portrait composed by triangles. The modified files were the 'GeneticAlgorithm', in which most of the methods implemented are, the 'Solution' class, composed by the object that will be the solution to our problem, and the 'Statistics' class as an auxiliary.

Solution

The 'solution' object is constituted of 3000 values. The first 1000 are the color and position of the triangles. The values from 1000 to 1999 are the relative fitness of each value that will be used on the relative fitness crossover, or selective crossover. Each gene starts with a relative fitness between 1000 and 1200 and every time it improves the "owner" (the individual that has that gene) it decreases, if it does not improve it will increase. Using this method throughout the generations the good genes will have a better relative fitness and using the selective crossover, a better chance to reproduce (Vekaria and Clark, 1998). To apply this operator, we also set several methods to increase, decrease, get and set these values. The final 1000 values are the indexes of each gene that will be used for the inversion and reorder as a way to keep each value on the correct index with the corresponding get, set, swap and sort (reorder) methods. The 'Solution' class also has a variable 'sharedFitness' used in fitness sharing that it is the redefined fitness.

Genetic Algorithm

Initialization: All the runs start with a fully random initialization. Although in our quest for the global optimum, we added the possibility using the parameter 'GoodInitialization', to start in an already good solution, if necessary.

Selection

We divided the selection in parent 1 and parent 2, and more when necessary, so it would be easy to use multiple selections within the same generation if necessary.

The selection methods implemented were the following:

1) **Tournament**
2) **Roulette**
3) **Ranking**
4) **Annealed:** A mix between selecting the best individual and maintaining the diversity of the population. Uses the following formula: FitnessComputed= Fitness / ((NGen+1) – currentGen) in which 'NGen' represents the number of generations that the algorithm will run and 'currentGen' is the current generation (Mahto and Kumar, 2015).

5) **Elitism:** Directly selects the best individuals. No point on using it as selection with the standard elitism implemented.
6) **RelativeFtournament:** Tournament selection using the average relative fitness as a selection criterion instead of the fitness.
7) **Stochastic**: The stochastic selection combines the features of both exploration and exploitation. It divides the individuals by their fitness similar to the roulette, and then assigns a number of pointers. The individual closer to each pointer will be selected. In our case, since this type of selection will return a number equal to the number of pointers, we selected one random individual from the selected ones to avoid extreme modifications in the code to accommodate this feature (Pencheva, Atanassov, Shannon, 2009).

Crossovers

Applicational independent crossovers

1) **SinglePointCrossover**
2) **TwoPointCrossover:** takes two points randomly. Between the two points it copies the 'Parent2', before and after the 'Parent1'.
3) **AvgCrossover:** the value of the offspring is the average between the value of 'Parent1' and 'Parent2' (Umbarkar and Sheth, 2015).
4) **kPointCrossover:** chooses k points as a parameter, puts them in order (ascendant). It takes two points every time and it uses the logic implemented in 'TwoPointCrossover'.
5) **OnebyoneCrossover:** chooses one value from 'Parent1' and the next value from 'Parent2', until the offspring is populated.

The following crossovers are thought to be used before the Restricted Mating.

6) **MatingCrossover:** selects the parent1 for the triangle that has 'RateSimilarityTriangle' "high", otherwise the triangle of 'Parent2'. Following this pattern, we can maintain the properties common to the good solutions while mixing the remaining ones.

RateSimilarityTriangle: calculates the absolute and normalized for the length of the triangle (for this reason, we called 'RateSimilarityTriangle') difference between the values in 'Parent1' and 'Parent2'. We chose (after view some results that we can define as "high" if the 'RateSimilarityTriangle' is bigger than 2.

7) **MatingCrossoverValue:** works as Mating Crossover but it uses the 'RateSimilarityValue'.

RateSimilarityValue: calculates the absolute, not normalized in this case because we consider all the values, difference between the values in 'Parent1' and 'Parent2'. In this case, we chose (after view some results that we can define as "high" if the RateSimilarityTriangle is bigger than 2.

8) **FourWayCrossover:** uses 4 parents, chooses randomly 4 points. Before the first point, it selects the first parent, progressively the second, third, fourth parents.

9) **DiscreteCrossover:** flips a coin to decide which value to take (Umbarkar and Sheth, 2015).
10) **FlatCrossover:** accepts a boolean parameter. When the parameter's value is "true", it selects the biggest value, for each solution's index, between the one in 'Parent1' and 'Parent2'; when it is "false", it selects the smallest value (Umbarkar and Sheth, 2015).

Applicational dependant crossovers

These crossovers are dependent on some auxiliary methods such inversions and reordering or an implemented ´Solution' object adapted for its purpose.

11) **OrderCrossover**
12) **IndependentCrossover:** Does the crossover only between color values and position values, does not mix them.
13) **CycleCrossover**: The 'CycleCrossoverRF' does the same thing but it also sets the Relative Fitness.
14) **RelativeFitnessCrossover**: It passes each value individually to the offspring flipping a coin, giving the higher chance to the value with the best relative fitness (Vekaria and Clark, 1998).
15) **RelativeF10PointCrossover:** Since the relative fitness is dependent on the values previously computed for each value, this crossover was implemented to be use at the start of the run so it performs a superior number of operations in the solutions and can adapt itself faster to the good values (Vekaria and Clark, 1998).
16) **CycleCrossOverRf:** Same as cycle crossover but is adapted to use relative fitness.
17) **TwoPointFitnessCrossOver:** Same as 'TwoPointCrossover' except it loops for *n* times (parameter) or until it the offspring as a better fitness than both parents.
18) **RFonSteroidsCrossover:** Same as 'RelativeFitnessCrossover' except it loops for *n* times (parameter) or until the offspring has a better fitness than both parents. These last two crossovers use the evaluate method explained later on.

Mutations

Most of the mutation have a parameter in which the user can decide how many values to mutate.

1) **Standard mutation**: Mutates 1 value.
2) **N standard mutation:** Mutates N values passed as a parameter.
3) **Box mutation**: Mutates N values within a certain range.
4) **Difference mutation**: Mutates values if they are more different than the range specified as a parameter.
5) **AllMutation:** A mutation in which is passed as a parameter the probability of mutating each individual, each triangle and each value, with the option to mutate within a certain range (geometric mutation).
6) **TrianglePosMutation:** Mutates only the position of the triangle.

7) **FitnessMutation:** Mutates for *n* times or until the offspring has improved his fitness. Uses the evaluate method which will be mentioned later.
8) **InsertMutation:** Gets 2 random values. Inserts the first random value at the right of the second random value and pushes every element. Not good in this specific problem since it constrains the main logic of the mutation because of the different ranges (Soni and Kumar, 2014).
9) **InversionMutation:** Selects 2 random values and inverts the elements between them. Same problem as the 'InsertMutation' (Soni and Kumar, 2014).
10) **RelativeFitnessMutation:** Standard mutation with an adaptation to use the relative fitness. After mutating to a random value, it lookups in the population for the closest value in the same index and sets the relative fitness of the new random value. It required the adaptation since every time we mutate we lose this property that will be used throughout the generations in the crossover, and using this method we can partially keep it.
11) **ScrambleMutation**: Selects *n* random genes and mixes them randomly. It has a constraint that in can only choose colors or triangles to keep the integrity of the values (Soni and Kumar, 2014).
12) **BoxFitnessMutation:** Same as Box mutation except it loops until *n* iterations or the offspring improved its fitness.
13) **BoxFitnessMutationLooped**: Same as 'BoxFitnessMutation' except instead of selecting *n* random values, it loops through the genes of the individual. The logic was to mutate all the values geometrically approaching the global optimum at each run.

Auxiliary Methods

Fitness sharing: It sets the variable 'sharedFitness' in the Solution to the redefined fitness. If activated, the selection methods select through the 'sharedFitness' as well. It can use the genotypic or the phenotypic variance, but since the genotypic is very computational expensive and thus, takes a long time. In addition to that, this variance is also subject of the curse of dimensionality because of its many attributes while phenotypic variance does not. For these reasons, we will only evaluate the phenotypic variance, which also fits our problem better.

Inversion / reorder: Inversion randomly mixes the values in the solution for *n* times ('nInversions') with the corresponding original indexes. After the individual goes through crossover and at the end the reorder will sort the values to the previous state using the indexes saved in the object solution

Elitism: Elitism selects the number of individuals ('nElites') that are the best from population P and puts them directly in the new population P' replacing the worst individuals by the elites.

We added, as well, several methods to perform calculations on the individuals such as the Euclidian distance, genotypic variance, phenotypic variance, Z-score, Max-Min, and other methods to organize and prepare the individuals for the operators such as sort solutions, lookup value, map relative fitness, among others. For the

parameters, we added a few optional parameters that can change selection, crossover and mutation during the run.

Schema methods to the problem

The triangles that compose the individuals work as whole through their values. That way, the 4 values corresponding to the color could generate the same result with different properties, the same way the coordinates of the triangles can give the same triangle position with different combinations if we rotate them. Because of this feature, it would be impossible to analyse the values individually and then, we need to view them as a whole. For that reason, we decided to check the perimeter which will use the relation between the 6 position values, the color, which will give us the relation between all the color values, and the coordinates, which will give us the specific position in the portrait.

Perimeter: After gathering statistics from the best individuals at the end of the generation, we noticed that, all the triangles have an average size, not too small, not too large, so it can be a good solution. With that in mind, we decided to try and add a constraint that checks the perimeter and mutates its size until it fits the constraint, assuming it will improve the first few generations narrowing the search space.

Color: Following the same logic used in the perimeter, the color should be within a certain range as well. Since different combinations can sum up to the same result we analyzed the color values of the best individuals and noticed they fall within a certain range as well so we added a constraint.

Coordinates: This specific problem as a portrait as different levels of complexity across the pixels. For that reason, it makes sense to assume that some areas will need to be more populated of triangles than others. In the first phase to analyze the distribution on the portrait we divided the painting in 16 quadrants, {0, 50, 100, 150, 200} for both X and Y, and calculated the average point (centroid) of each triangle to verify which are the most populated ones in the best solutions. After gathering those stats, we added a verification to check if the triangles in the individual fall inside that range, if they do not, we “spread” them until they fit the designated schema.

Evaluate method

The evaluate method is used to compute the fitness of a given individual. Usually, it is used at the end of each generation and it is the part of the program that is more computational expensive. But what if we used it to improve the performance of the operators? When used accordingly with mutation, it can allow us to dodge mutations that destroy a good solution, and in the selective crossover, it can vastly improve the algorithm. The selective crossover is dependent on the operations to calculate and recompute the relative fitness each time, if we can allow it to recalculate several times within a generation, it can vastly improve the results with a small cost in performance. The main advantage of combining these two methods is that every time it gets a good or bad result, it updates the relative fitness of the parents so it will do better in the next iteration. For this goal,

we created a parameter called ‘nEvaluates’ that represents the number of times and operator will test the new individual. We also adapted some of the crossovers and mutations to be able to take advantage of this property.

Other Algorithms

Random Search: We did not create a class random search because the ‘AllMutation’ with all the parameters set at 100% will give a random value to every triangle of every individual in the population. Because of that, it felt unnecessary to create a specific class, especially since it will not give a very good result in the generation span we have.

Results

All the results are the average of 10 runs except, in some cases, in which will be mentioned. The graphs, conclusions and decision making for the selected parameters used the median fitness of the 10 runs so it will be more accurate than the best fitness that is more sensible to outliers. In every criterion, there will be a graph with the whole run, the zoomed run for a better visualization with the error bars presented when necessary and, when relevant for the results, the phenotypic variance. Regarding the auxiliary methods they will be compared with a control group running the exact same algorithm except the method that is being tested.

Selection

In the selection methods the best one with significance difference was the tournament, using the fitness or the average of the relative fitness as a selection criterion (Figure 1). The roulette and the ranking selection had a solid result while the annealed selection and elite selection had a better phenotypic variance, the fitness results are much worse. From now on, we will use the tournament selection and determine the best size.

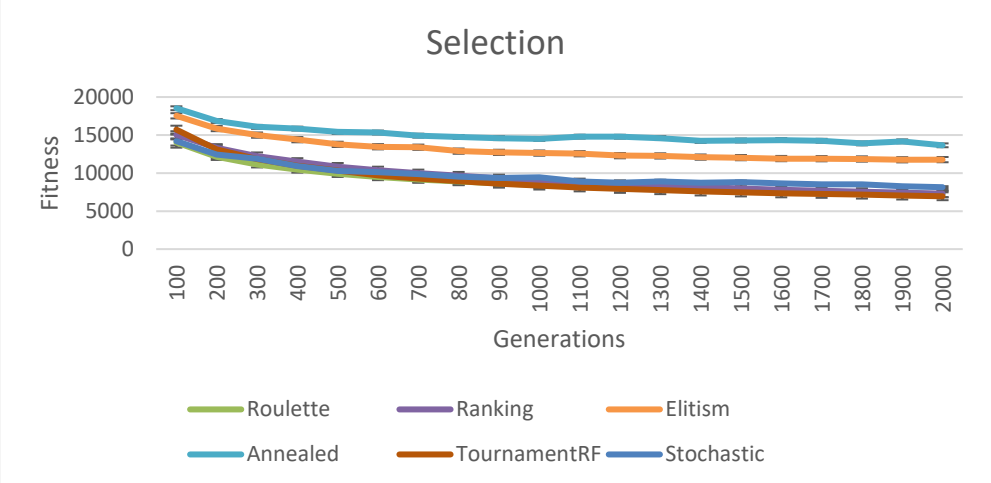


Figure 1. Selection Methods

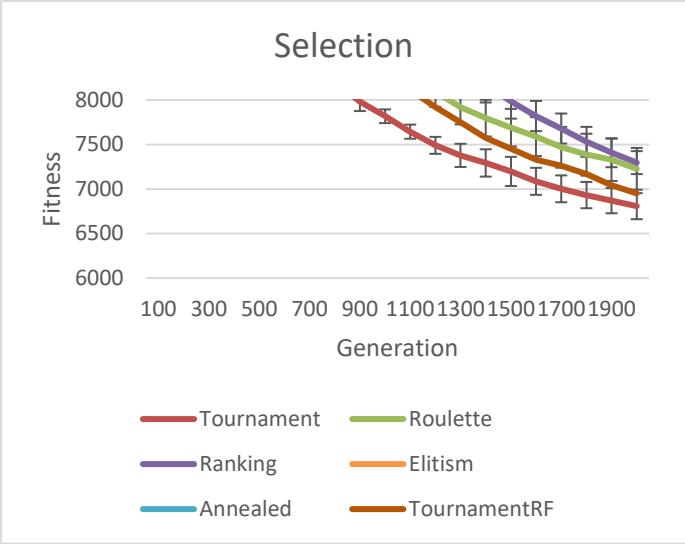


Figure 2. Selection Methods zoomed with error bars

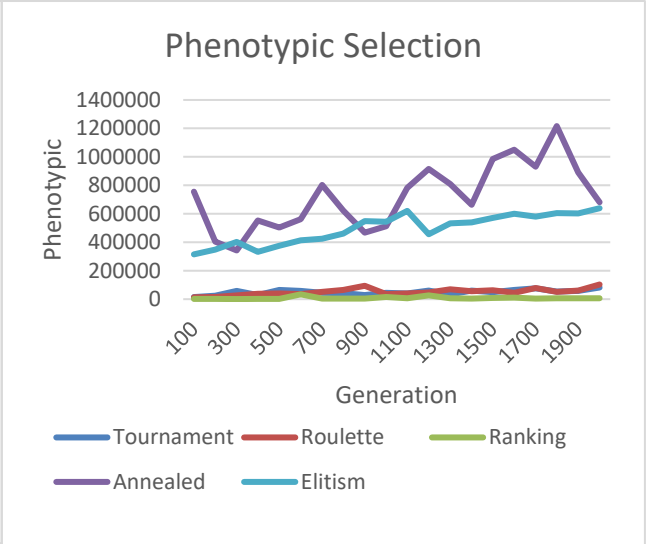


Figure 3. Phenotypic variance for Selection

Tournament Size

Considering the tournament size, there are no significant differences in improving the fitness (Figure 5). The size of 3 tends to be a bit worse than the rest while all the others sizes are on the same level. We decided to choose a tournament size of 12 which tended to have the best median fitness (Figure 4). Relating to the phenotypic variance there were no major differences. The error bars were omitted for readability.

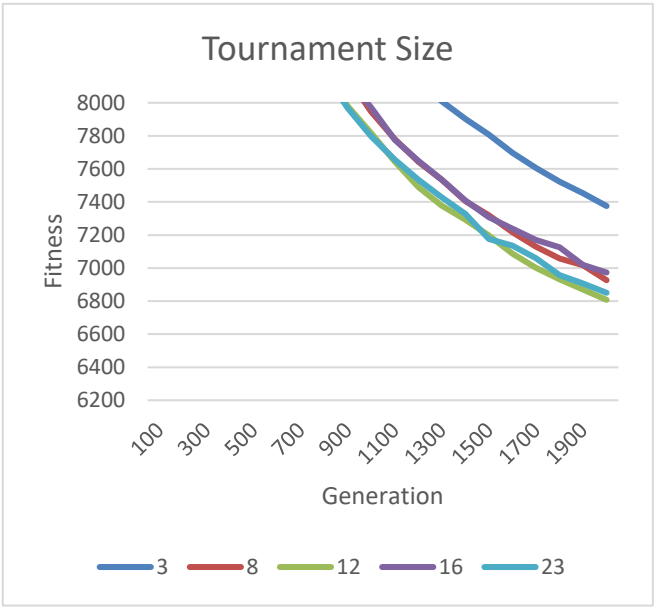


Figure 4. Tournament Size zoomed

Figure 5. Tournament Size with a bigger scale.

Crossover Probability

While evaluating the crossover probability there were no significant differences between any of the probabilities. The 3 best crossovers tend to be with a probability of 0, 0.25 and 0.5, while the worst tend to 0.41 (Figure 6). For the purpose of selecting the best parameters, we will use a probability of 0.5, since it makes no sense to test new crossovers with 0. In the end, when the best parameters are selected, we will rerun and test with the lower probabilities that appear to be the best. Its also important to notice that the phenotypic variance of 0 and 0.25, supposely the best, is too low compared to the others (Figure 8). While approaching better fitness values, this will have a major impact.

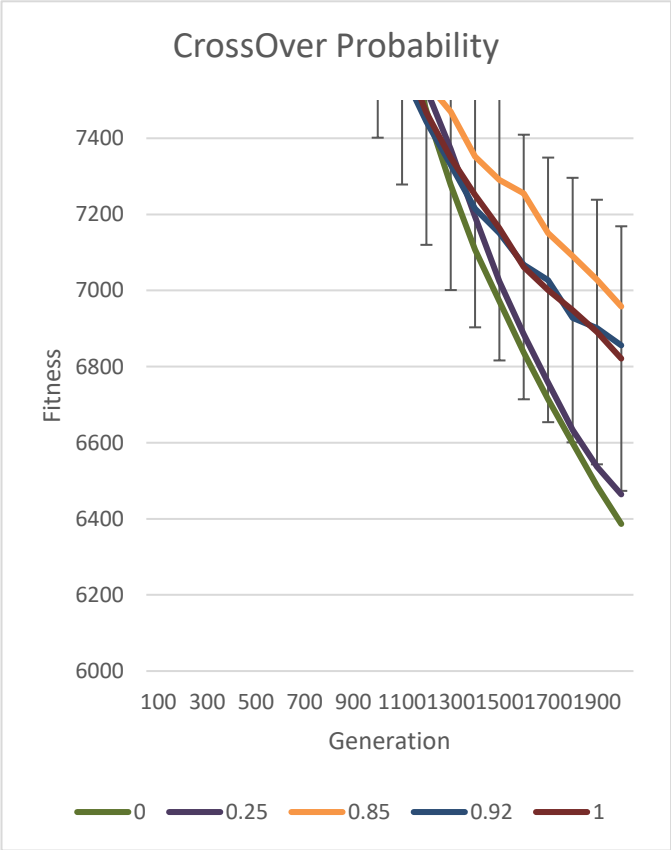


Figure 6. Crossover Probability

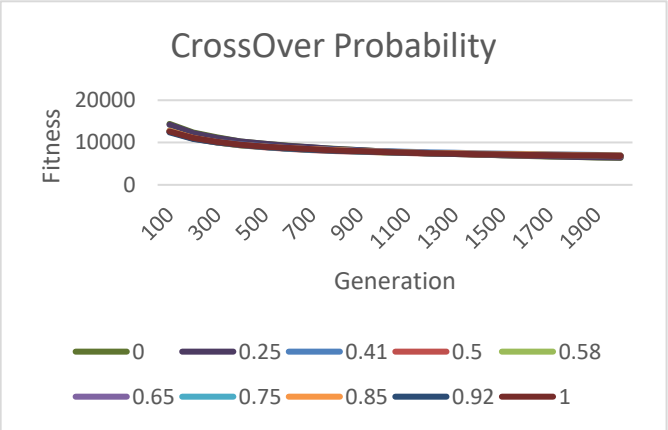


Figure 7. Crossover Probability zoomed

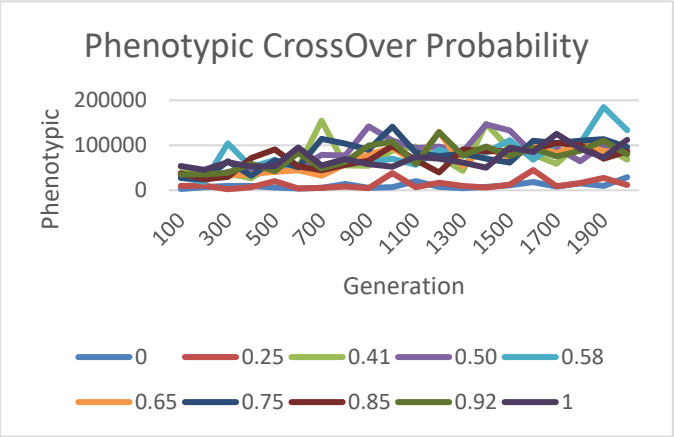


Figure 8. Phenotypic variance Crossover Probability

Mutation Probability

Analysing the mutation probability, there are no significant differences between the values of 0.5 and 1 (Figure 9, error bars omitted for readability). The lower probabilities will have a huge impact in the fitness, with the probability of 0 stagnating the search. This could be explained that the crossover by itself cannot get a decent solution since it cannot change the pre-existing values from the initialization. It seems the best mutation tends

to be 0.9 and it will be the parameter used from now on. With the probability of 1, we have a much better phenotypic variance which can be used later on if we have problems with premature convergence.

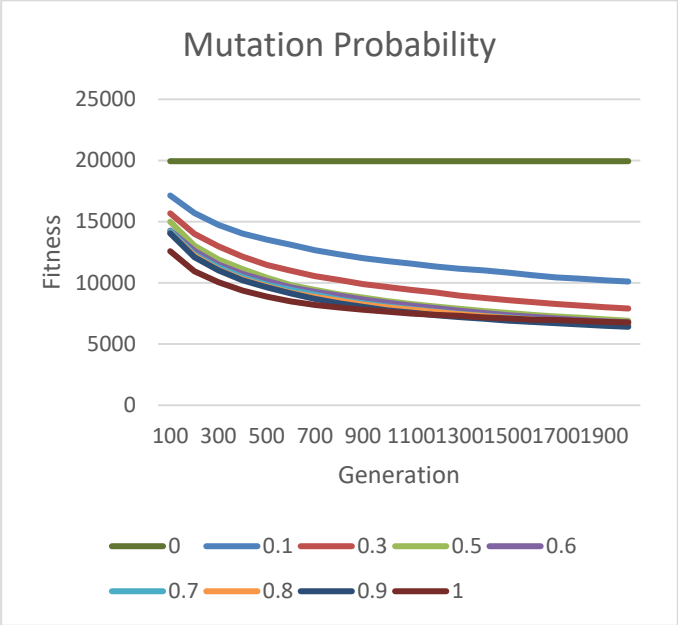


Figure 9. Mutation Probability

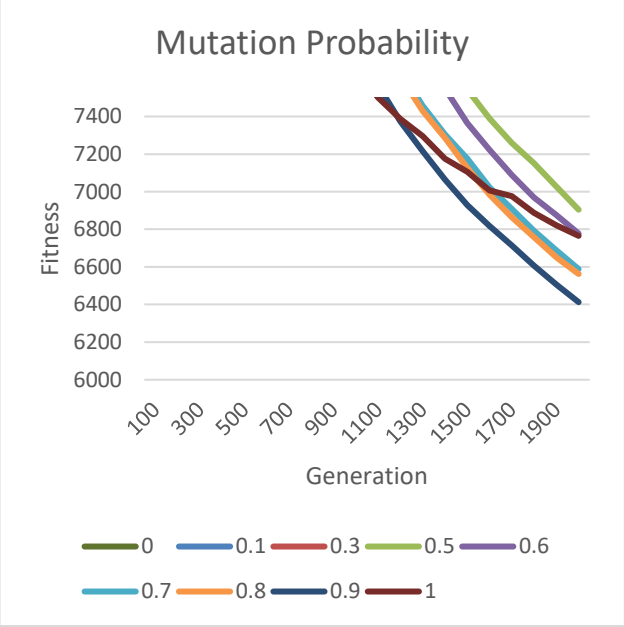


Figure 10. Mutation Probability zoomed

Mutation types

Most mutation seem to be too powerful which makes the performance of the algorithm mostly random, destroying the potential good solutions. The best with statistical significance is the standard mutation (Figure 11), just mutate a *n* number of random values, which will be used from now on.

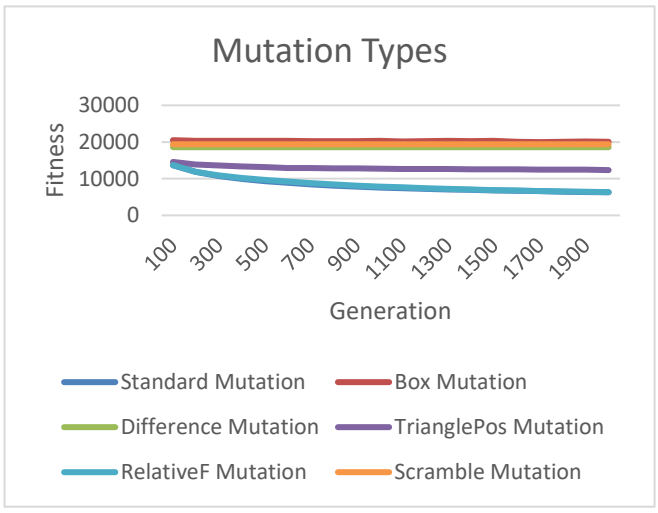


Figure 11. Mutation Types

Number of values mutated

For the number of values mutated it seems the 2 best with significant difference from the rest are 1 and 2 (Figure 13). Everything else seems to be way to random and, although it increases the phenotypical variance makes no sense to apply it since it will also "destroy" the good solutions. From now on the we will use 1 as the number of values mutated and rerun with 2 when we arrive to the final solution.

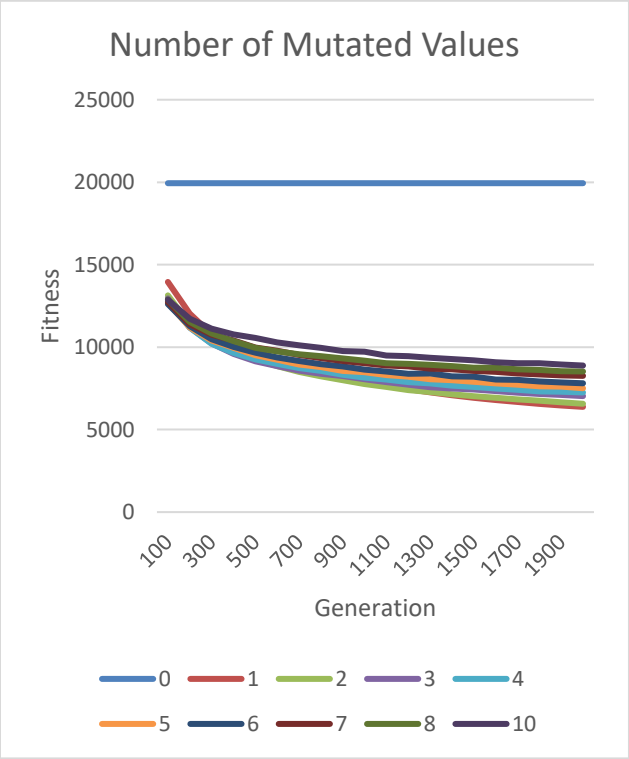


Figure 12. Mutated Values

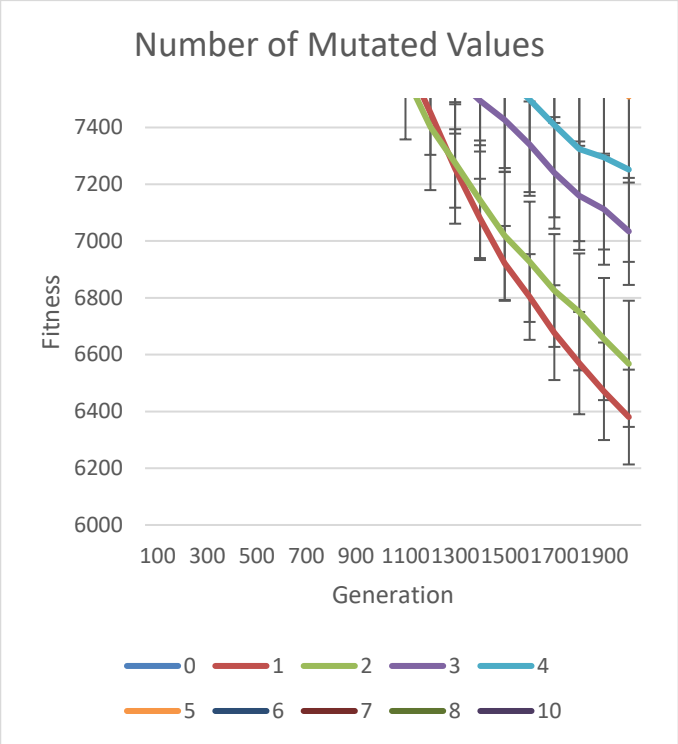


Figure 13.Mutated Values zoomed

Number of elites

The number of elites does not seem to impact the final result (Figure 14). The only major difference is on the phenotypical variance, the higher the number of elites the lower the variance. For that reason, we will use only 1 elite as it will keep the diversity.

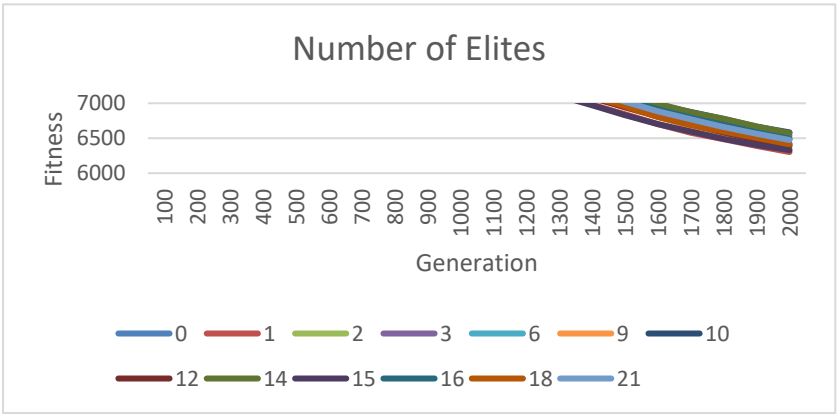


Figure 14. Number of Elites zoomed

Inversion

The number of inversions inside each individual, at first sight, is worst than not using inversion with statistical significance (Figure 16). The important thing to take notice is that is a method that will be used in pair with some others operators like cycle crossover, and it greatly improves the phenotypical variance. With this in mind, we will choose to not use the inversion method throughout the runs. The only cases it will be used is in specific operators that require inversion, in those cases we will use 500 as the number of inversions since is has similar fitness to its competitors and a greater phenotypic variance.

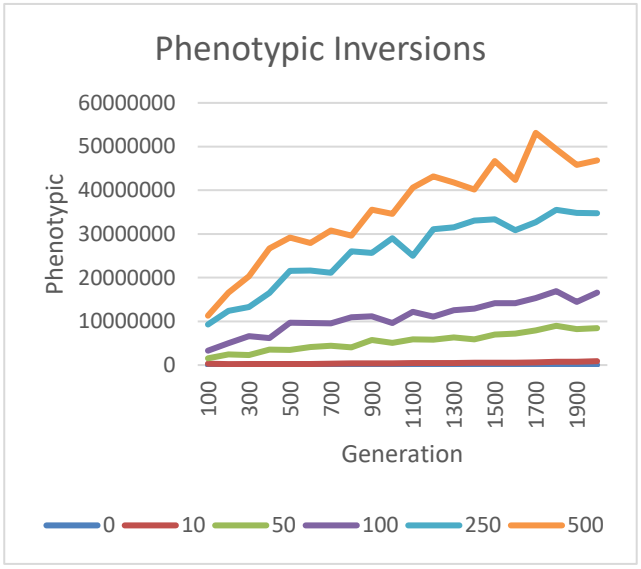


Figure 15. Phenotypic Variance

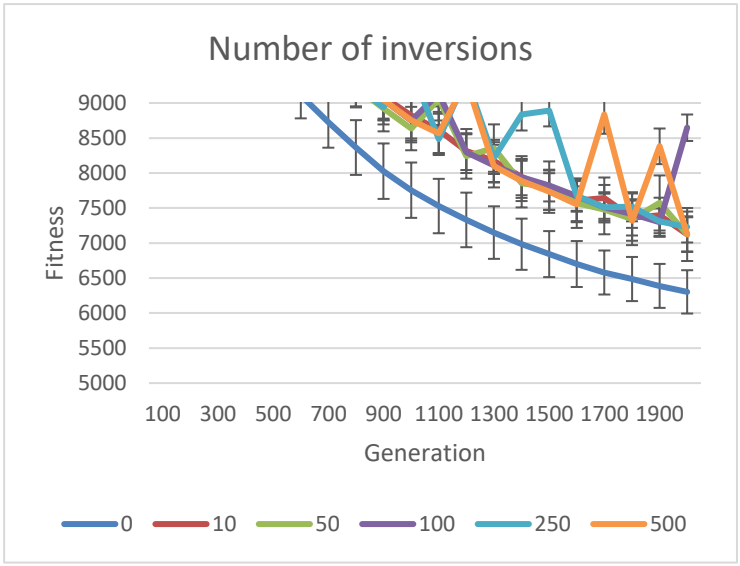


Figure 16. Number of Inversions zoomed

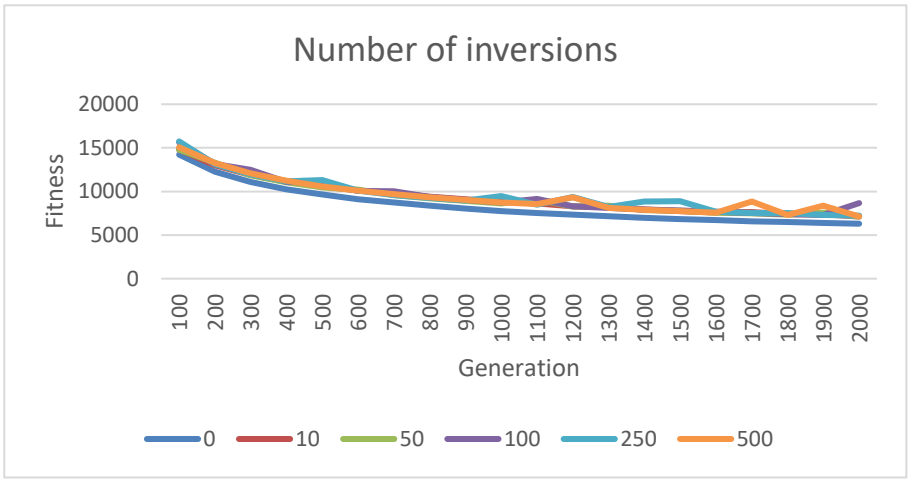


Figure 17. Number of Inversions

Fitness Sharing

We implemented two types of fitness sharing, according to the phenotypic variance and the genotypic variance. Using the genotypic variance, it will cause a huge overload due to the amount of calculations required and it is not feasible in a decent amount of time (for instance, it would take weeks to run 10 times), so we only considered using the phenotypic variance. The goal of fitness sharing is to improve the diversity, which we can see tracking the phenotypic variance, it indeed does to a higher level than any other method. Besides that, it does not give good fitness results compared to the control group so it will no longer be used (Figure 18).

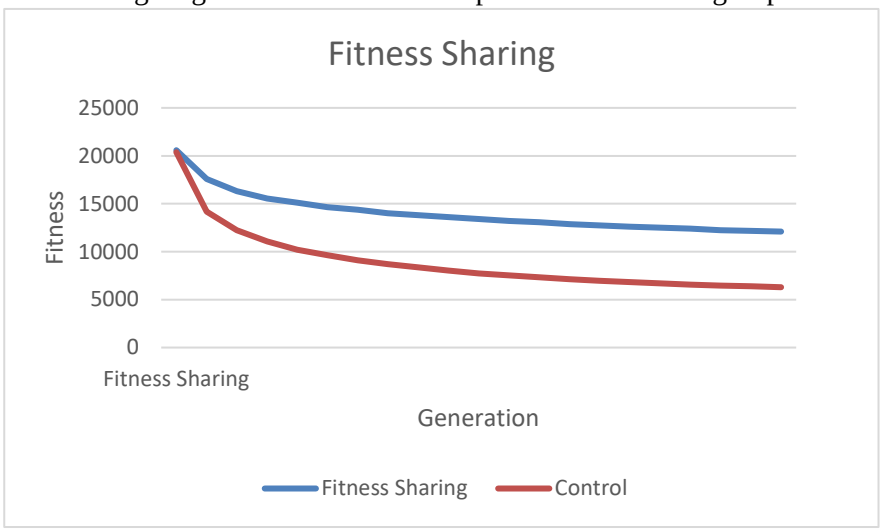


Figure 18. Fitness sharing

Crossover Types

Relating to the type of crossover it seems there are not any major differences (Figure 19, error bars omitted for readability). The cycle and the order crossover used the auxiliary method of inversion and reorder since they are index based crossover with the parameter decided beforehand. Overall, the two best crossover types tend to be the single point crossover and relative fitness or selective crossover without significance difference. Examining the phenotypic variance, it does not seem to make too big of a difference to consider any other type than these two. From now on, we will use the relative fitness crossover and rerun with the final parameters using as well the single point crossover.

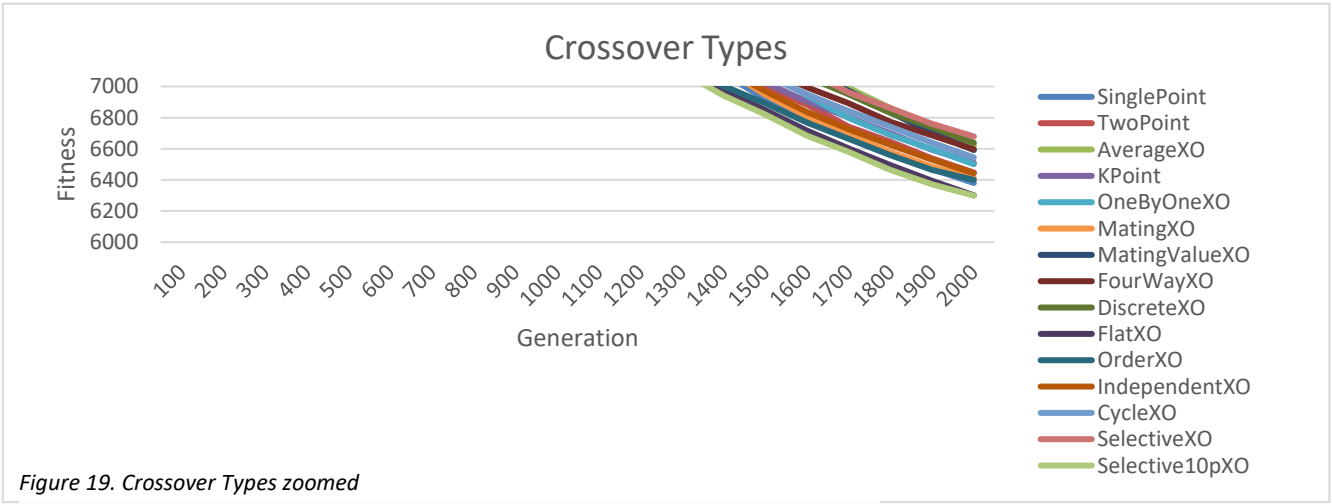


Figure 19. Crossover Types zoomed

Check Color

In the check color constraint there are not significant differences in the fitness results (Figure 20-21), which can be explained, in part, because the narrowing of the search space is not too significative. Relating to the phenotypic variance there are not any major differences between the control and the check color.

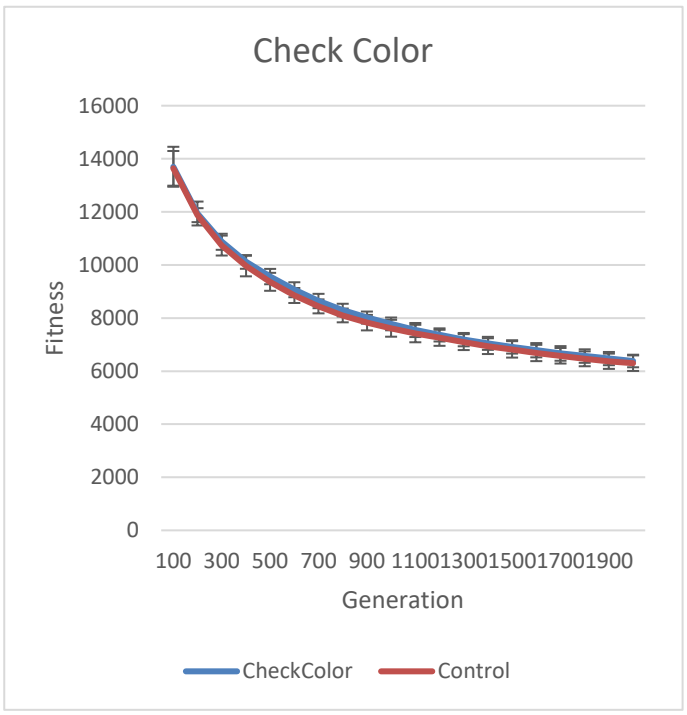


Figure 20. Check Color

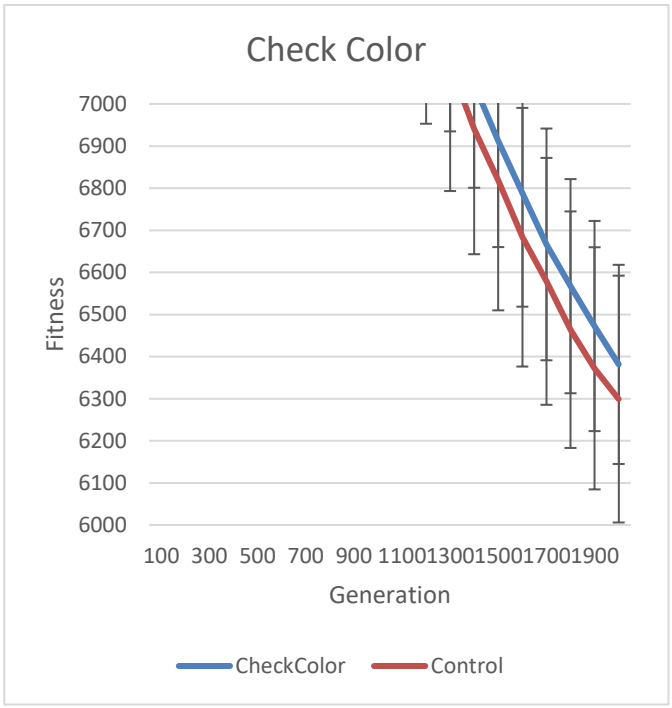


Figure 21. Check Color zoomed

Check Perimeter

Regarding the check perimeter schema, the results were basically the same as in the control group, in fitness (Figure 22) and in phenotypic variance. Seems to be schema too wide that does not really "filter" the individuals. Error bars omitted.

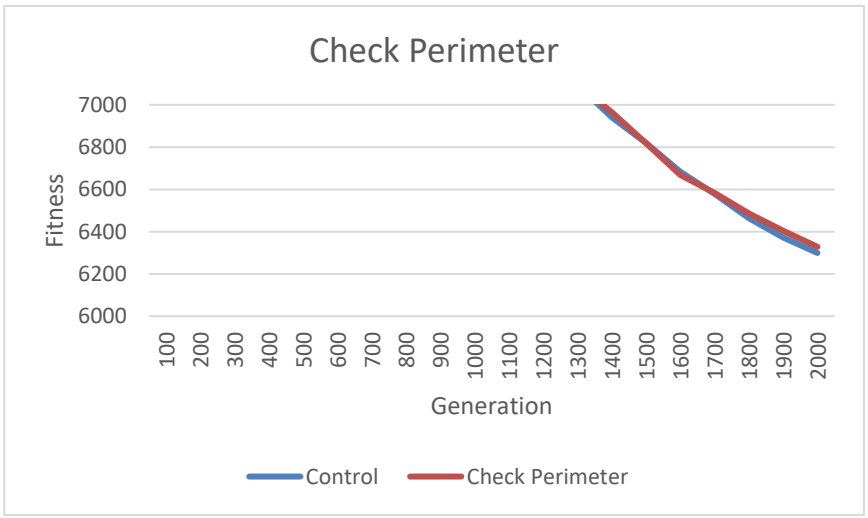


Figure 22. Check Perimeter zoomed

Check Coordinates

Following the pattern, there are is not any significant differences of adding a constraint in the position of the triangles (Figure 23-24). Regarding all the schemas implemented they do not seem to be powerful enough to cause an impact on the execution, we would have to collect more data to understand a very specific pattern that the good solutions could have that effectively narrows the search space.

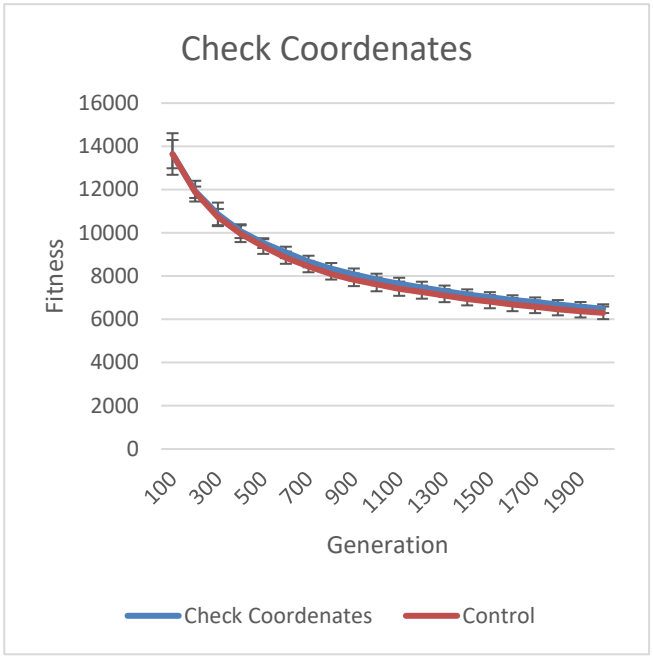


Figure 23. Check Coordenates

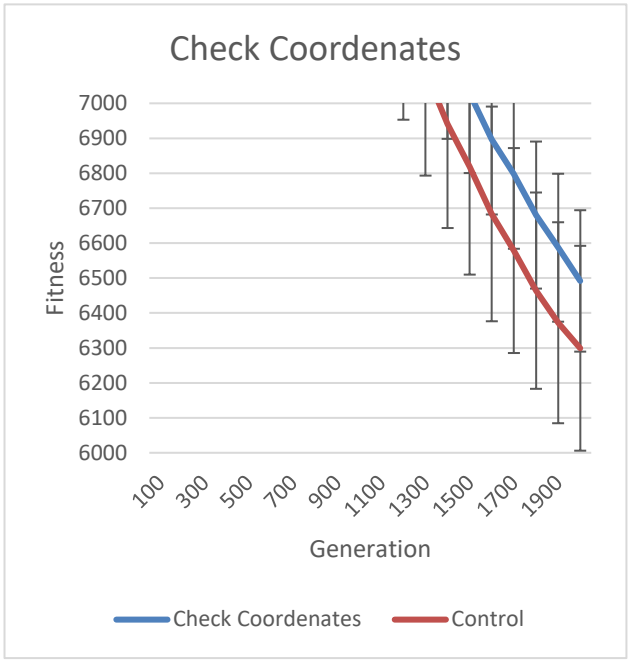


Figure 24. Check Coordenates zoomed

Random search

As expected the random search does not come even close to the remaining results since it is a huge search space and out of the league for this specific algorithm (Figure 25).

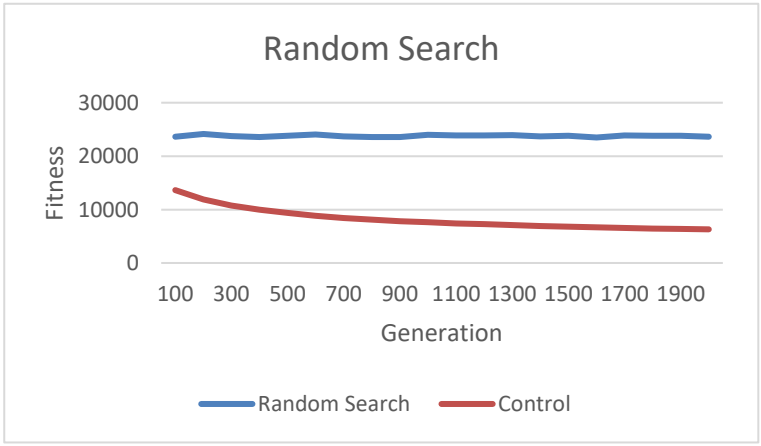


Figure 25. Random Search

Number of Evaluations

The results show that the higher the number of evaluations the best fitness result we can obtain with a tradeoff with the time required to run the algorithm (Figure 26-27). The parameters 25, 50 and 100 were run less than 10 times due to the lack of time. With a median of 4300 the run with all the previously established parameters and 100 evaluations was the best result when taking into account a single run, which represents the problem of premature convergence characteristic of this types of problems since at half the run it already had a close fitness to the final run.

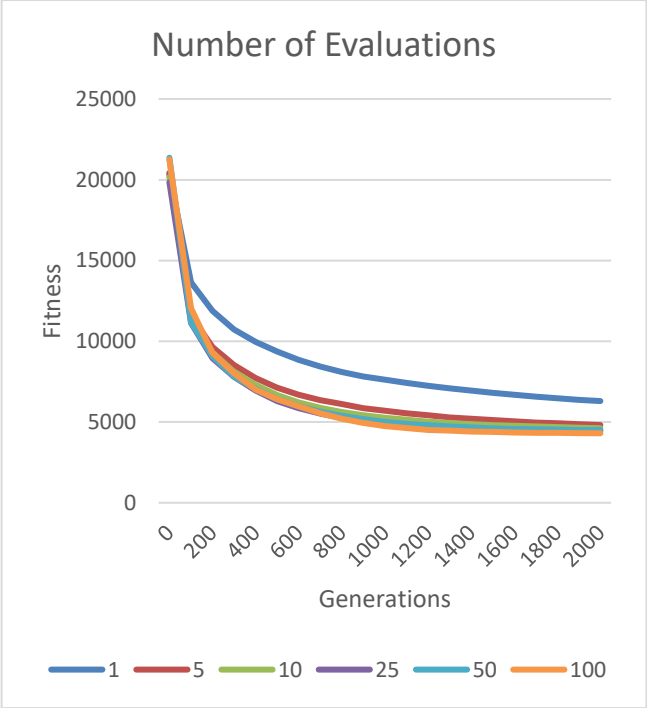


Figure 26. Number Evaluations

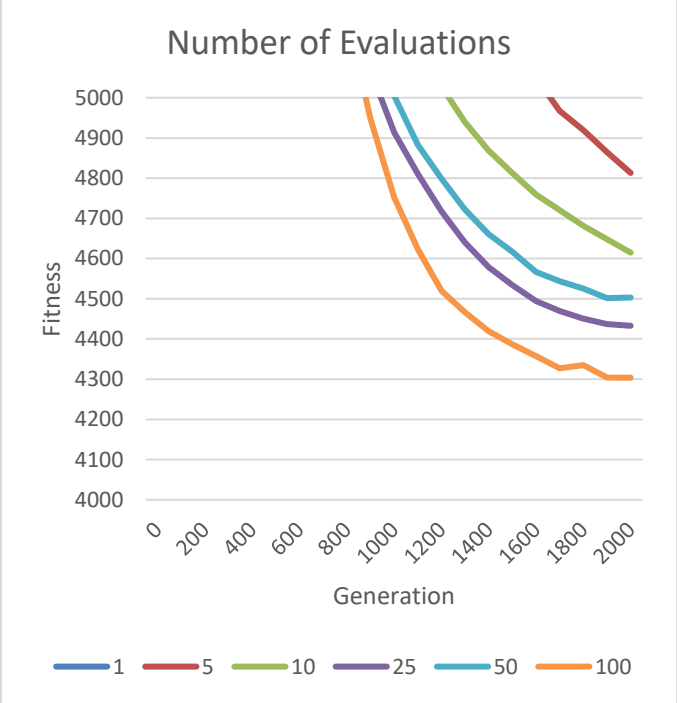


Figure 27. Number Evaluations zoomed

Conclusions

As a final result the best overall fitness was 4193 using the selective crossover, the fitness mutation with 2 mutated values, 1 elite and 500 evaluations (Figure 28). Using a higher computational power, a better result would be obtained. Although, after implementing all the operators and auxiliary items to try and battle the complexity of finding the global optimum in this problem, we can acknowledge that the main issue is the problem of premature convergence in a huge search space. It requires a very specific change of values to decrease the fitness which the algorithm struggles to find even using diversification methods. If these methods increase the diversity, they are too powerful and lose sight of the true objective that is reduce the fitness.

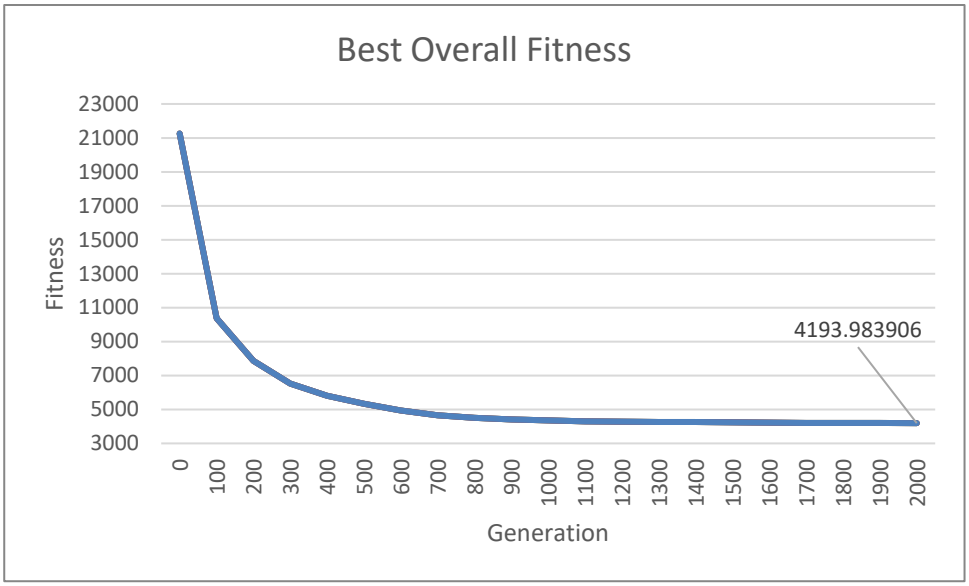


Figure 28. Best Overall Fitness

Bibliography

A.J. Umbarkar and P.D. Sheth (2015) Crossover Operators in Genetic Algorithms: A Review. *Ictact Journal on Soft Computing, October 2015, Volume: 06, Issue:01*

Kanta Vekaria and Chris Clack (1998) Selective Crossover in Genetic Algorithms: An Empirical Study. *A.E. Eiben et al. (Eds.): PPSN V, LNCS 1498, pp. 438-447, 1998. Springer-Verlag Berlin Heidelberg 1998*

Manoj Kr. Mahto,Mr. Lokesh Kumar. (2015) Study of Selection on Performance of Genetic Algorithms. *International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463Vol. 4 Issue 3, March-2015, pp: (264-268)*

Nitasha Soni, Dr Tapas Kumar (2014) Study of Various Mutation Operators in Genetic Algorithms. *Nitasha Soni et al (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (3) , 2014, 4519-4521*

Tania Pencheva, Krassimir Atanassov, Anthony Shannon (2009) Modelling of a Stochastic Universal Sampling Selection Operator in Genetic Algorithms Using Generalized Nets. *Tenth Int. Workshop on Generalized Nets Sofia, 5 December 2009, 1-7*