COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION

PRACTICAL PROJECT REPORT

MASTER IN ADVANCED ANALYTICS

AKINYEMI SADEEQ AKINTOLA(M20170002),
DENNIS CROON (M20170053),
PIERO MAGGI (M20170252),
ALEXANDER OBENAUFF (M20170724)

ACADEMIC YEAR 2017/2018

Table of Contents

1. PROJECT SETTINGS

Setting for final experiments:

- 100 triangles
- 2000 generations
- 25 individuals in the population
- 3 runs

2. PROJECT INTRODUCTION

This project seeks to approximate a target image (the Mona Lisa Image) with a given number of triangles using the minimize Euclidean distance.

Every solution stores t triangles.

$\boldsymbol{Triangle_0}$	…	$\boldsymbol{Triangle_{t-1}}$

The triangles in a solution represent one raster (pixeled image) and it has the same size as the target raster.

Index	0	1	2	3	4	5	6	7	8	9
Value	$[\mathbf{0}, \mathbf{256}]$	$[\mathbf{0}, \mathbf{256}]$	$[\mathbf{0}, \mathbf{256}]$	$[\mathbf{0}, \mathbf{256}]$	$[\mathbf{0}, \mathbf{200}]$	$[\mathbf{0}, \mathbf{200}]$	$[\mathbf{0}, \mathbf{200}]$	$[\mathbf{0}, \mathbf{200}]$	$[\mathbf{0}, \mathbf{200}]$	$[\mathbf{0}, \mathbf{200}]$

Each triangle is represented by 10 values (one at each index):

Index
0: hue (color) of triangle
1: saturation (intensity)
2: brightness (light/dark)
3: alpha (transparency)

Indexes 4-9:

correspond to the coordinates of each vertex of triangle in the raster (values from 0 to 200).

Vertex 1: has indexes 4 and 5
Vertex 2: has indexes 6 and 7
Vertex 3: has indexes 8 and 9

3. PROBLEM ANALYSIS

The solution to this problem could be an application of the Knapsack Problem that we studied during the course, more precisely: for the Knapsack problem, as an instance class, we had a maximum number of items and the corresponding weights and values for each of them; in this case, instead, we have a target image, the Mona Lisa's Paint, and the number of triangles that are used to approximate the target image.

4. SOLUTION APPROACH

A baseline implementation that accompanied this project problem has an initial solution that was meant to be optimized. Our solution required that better implementation of some classes was to be made. Hence, we have made significant changes to some classes and also added new ones. We also imported external resources that are not included within the regular java programming environment, in order to perform specific functions such as writing program results into csv. Should our solution be implemented on another machine, these external libraries would have to be referenced from the location on the host on which the library files reside.

Classes

CreateExcel.java – This new class is used generate an output Microsoft Excel File
GeneticAlgorithm.java – This modified class houses the entire Genetic Algorithm Implementations
Main.java – This modified class is the main class from which the Project is instantiated and runs
Problem.java – This class defines all the default characteristics of the problem in question
SearchMethod.java – This is an embodiment which the GeneticAlgorithm class extends
Solution.java – The class houses the solution/methods (functions) with which the problem was solved
Statistics.java – This class calculates (and displays) intermittent solution results at runtime.

Imported Resources

We made use of the POI Library interface to allow us to easily read and write all kinds of Excel files (XLS and XLSX). Additionally, there is a specialized SXSSF implementation which allows to write very large Excel (XLSX) files in a memory optimized way.

5. METHODS IMPLEMENTED

The table below shows a comprehensive summary of new and updated functions/methods implemented in the project (the more advanced methods are described below the table in more detail):

Process	Type	Method	Class	Description
Initialization	Initialization	initializeFillingBorders	Solution	Fill borders with triangles first, after randomly; Colors: choice between MidPoint color, ColorTargetPicture, default
Selection	Selection	rouletteWheelSelection	Genetic Algorithm	Returns the individual's index based on the probabilities proportional to the (inversed) individual finesses
Selection	Selection	rankingSelection	Genetic Algorithm	1st ranks the population and then every individual receives fitness from this ranking. The worst (max. fitness) will have fitness 1, second worst 2 etc. and the best will have fitness N (number of individuals in population).
Variation	Crossover	multiPointCrossover	Genetic Algorithm	randomly choose 2 crossover points and alternating segments are swapped
Variation	Crossover	UniformCrossover	Genetic Algorithm	for loop, each index decides if will be included in the off-spring, setting parameter
Variation	Crossover	CycleCrossover	Genetic Algorithm	Cycle Crossover operator identifies "cycles" between two parents. Then, to form offspring 1, cycle one is copied from parent 1, cycle 2 from parent 2, cycle 3 from parent 1, and so on
Variation	Crossover	TriangleCrossover	Genetic Algorithm	Triangle crossover checks the fitness of triangles and swaps the bad triangles with the better ones to create a better offspring.

Variation	Crossover	PMX (Partially Matched Crossover	Genetic Algorithm	The partially matched crossover is using the multipoint crossover in the first place (after re-ordering the individuals) and order them back by using the matching by indices.
Variation	Mutation	applyMutationWithAmount	Solution	amount [0.0 – 1.0], determines number of indices to be mutated by random resetting
Variation	Mutation	applyScrambleMutation	Solution	select a subset of indices and shuffle them randomly
Variation	Mutation	applyMutationTargetPictureColor	Solution	random resetting of Triangle colors, only chooses Colors of the target Picture
Variation	Mutation	applyMutationTriangleColorTargetPicture	Solution	random resetting of Triangle colors, only chooses from Colors of triangle area
Variation	Mutation	applyColorTriangleAmount	Solution	applies colors of triangle area to specified amount
Variation	Mutation	applyColorTriangleAll	Solution	applies colors of triangle area to all triangles
Variation	Mutation	applyMutationConstantAlpha	Solution	same as applyMutation but applies a constant alpha (60)
Variation	Mutation	applySwapMutation	Solution	select 2 random Value indices and interchange the values within a Triangle
Variation	Mutation	applyInversionMutation	Solution	select a subset of indices by choosing 2 random value indices and invert the subset (Colors or Vertex)
Variation	Mutation	applyRandomTriangleLineColor	Solution	Random resetting of Triangle colors, only chooses Colors of the Triangle lines (see slope, intercept, Coordinates method)
Variation	Mutation	applyMidpointColor		applies Midpoint Color of a Triangle
Parameters				
	double	MUTATION_AMOUNT	Main	amount of indices values to be mutated
	double	decayrate	Genetic Algorithm	see **

	boolean	CheckFitnessXOMut	Main	if true, checks if fitness of XO offspring is better than mutation offspring, keeps better one.
	boolean	BestXO	Main	evaluates different crossover operators based on fitness and takes best one
	boolean	BestMutation	Main	evaluates different mutation operators based on fitness and takes best one
	boolean	CheckBestParentsXO	Main	Parents fitness < XO fitness? Parent (1 or 2): XO offspring
	boolean	CheckFitnessXOMut	Main	Parents fitness < mut. fitness? Parent (1 or 2): mut. offspring
	boolean	BestXO	Main	evaluate various xo operators take best one
	boolean	BestMutation	Main	evaluate various xo operators take best one
	boolean	CheckBest2XO	Main	apply xo multiple times to same offspring
	boolean	CheckBest2Mut	Main	apply mutation multiple times to same offspring
Methods				
		TargetPictureColors	Solution	store all Colors from targetPicture into an array
		GetColorOfPixel	Solution	A function to check the color of a pixel in the target picture. Searching by the index of a pixel instead of using the coordinates.
		loctoindex	Solution	Function to calculate the index of a pixel based on the coordinates.
		indextoloc	Solution	Function to calculate the coordinates of a pixel based on the index.
		getFitnessofTriangle	Solution	calculates the fitness of specified triangle
	Fitness Sharing	getdifference	Solution	calculate difference (distance) between individuals
	Fitness Sharing	getsharedfitnesses	Solution	calculates and returns all shared fitnesses
		getmiddlepointindex	Solution	retrieves middle point index from a specified triangle

		isintriangle	Solution	checks if an index/pixel is in a triangle
		getImageIndices	Solution	gets Indices inside a Triangle
		ordersolution	Solution	order coordinates of triangles
		orderwholesolution	Solution	order triangle by midpointindex
		slope	Solution	Slope of 2 Vertices
		intercept	Solution	Intercept of 2 Vertices
		Coordinates	Solution	Retrieve all Indices which are on the line/ graph between 2 Vertices using y = slope * x + intercept
		IsInArray	Genetic Algorithm	check if value is in array
		totalmFitness	Genetic Algorithm	calculate modfied total fitnesses
		totalFitness	Genetic Algorithm	calculate total fitness

** $DecayRate(cg) = (1 - mutationProbability)^{\left(\frac{\text{numberOfGenerations-currentGeneration}}{\text{numberOfGenerations}}\right)}$

The following are the more complex methods implemented/improvised to achieve a better solution in this project

Initialization Methods

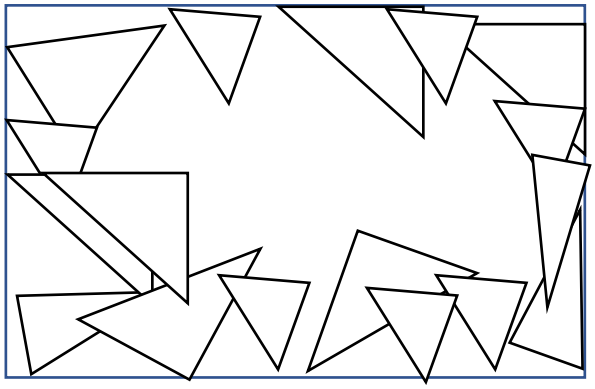


Image 1. Initialization sample

For the first twenty triangles (a sample is shown in the above figure), our approach required that we generate the outside of the boarders first, to ensure that the outside is pre-filled first. After generating the location of the triangles, we calculated the centroids. Based on the centroids, we were able to determine the color of the target picture.

However, to use this, the target picture format had to be converted from RGB-format to HSB(A) format. Even though, there was a drawback from this approach - The value for the Alpha(A) could not be determined through this conversion. Hence, we assumed a reasonable constant value of 60. We eventually stored the calculated values of the target image into the array housing the values of Col1 - Col3 for the 100 individuals.

We also created two arrays (x and y) with some x and y coordinates where we would be choosing location values from. This was done for the first twenty triangles. Therefore, each individual of the class solution was represented as shown below:

Individual 1	Col1	Col2	Col3	Col4	x1	y1	x2	y2	x3	y3
Individual 2	Col1	Col2	Col3	Col4	x1	y1	x2	y2	x3	y3
…	…	…	…	…	…	…	…	…	…	…
Individual 100	Col1	Col2	Col3	Col4	x1	y1	x2	y2	x3	y3

At any point in the solutions space, depending on the respective values that make up the array-set of the individual, any one individual is represented as shown below.

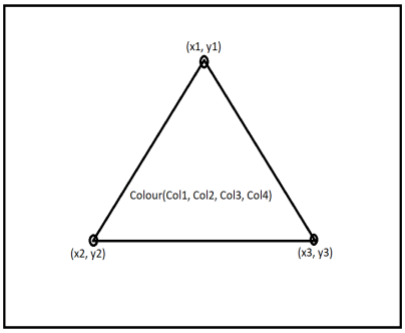


Image 2. Individual in the solution space

Crossover operators

Cycle Crossover

1. Create 2 arrays y to keep initial Order for both parents (OrderArrays)
2. Loop through index to invert OrderArrays randomly
3. Create “OffspringArray” with -1 as values to distinct between parents
4. OffspringArray[0] equal to OrderArrayOffspring[0]
5. Store value of first index as int called “Idx”
6. Do until termination condition
    a. check if Idx is already in OffspringArray
    b. if Idx not in OffspringArray check each value in OrderArrayOffspring, and find the index (j) where its value = Idx;
    c. OffspringArray[j] = Idx
    d. Idx = OrderArrayOffspring2 [j] – next a.
7. set offspring values from 1st and 2nd parent by checking OffspringArray, if index value != -1, then 1st parent; else 2nd parent

Partially Matched Crossover

1. Create 2 arrays to keep initial Order for both parents (OrderArrays)

2. Loop through index to invert OrderArrays and the two parents randomly
3. Swap a part of the first parent with a part of the second parent, the from i1 to i2 (these integers are both randomly chosen)
4. Go through the numbers in the OrderArrays array of the second parent and check which number is double (excepting the numbers between i1 and i2, they should be like this). If a number is double:
    a. Find it in the same (second) parent
    b. Check the number of the same index in the first parent (call this "relatable")
    c. Find this number in the second parent and check it's relatable
    d. Continue this until you can't find the relatable in the second parent
    e. The last relatable is the one that should be swapped with the original number checked in the second parent.
5. Re-order the first and second parent again based on the OrderArrays arrays.
6. Set offspring is equal to first parent.

Triangle Crossover

1. Order both parents with the **ordersolution**- and the **orderwholesolution** methods.
2. Calculation the fitnesses of all the triangles of both solutions by using the **getfitnessoftriangle** function.
3. Comparing per index (1 to 100) the fitness of the triangles in the two arrays with fitnesses and take the best triangle.

Advanced methods

Ordersolution

This method is used to order the coordinates of all the triangles in the same way. It is an important step before using the fitness sharing. Fitness sharing is calculating the differences between individuals and this may lead to an incorrect difference. I.e. the two individuals below are exactly the same, but if we calculate the difference, by comparing all the indices, we think that there is a difference.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	230	89	34	78	19	39	65	130	40	90

Individual 1

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	230	89	34	78	65	130	19	39	40	90

Individual 2

To avoid this incorrect difference, we order all the coordinates of the triangles by the value of the three x-indices. Both individuals will be set as below:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	230	89	34	78	19	39	40	90	65	130

Individual 1

Index	0	1	2	3	4	5	6	7	8	9
Value	230	89	34	78	19	39	40	90	65	130

Individual 2

As pictured above, there is no difference anymore (which is the right approach).

Orderwholesolution

In the previous method, the coordinates are ordered in another way. In this method, the whole individual will be ordered in such a way that there is less change to calculate a difference which is not there. The whole individual will be ordered by the middlepoint index of the triangles. To calculate this index, we made the function **getmiddlepointindex**. After this it is just ordering the triangles from small index to big index.

GetFitnessOfTriangle

In order to find the worst triangle and to improve this specific one, the fitness of this triangle should be calculated to compare with others. To calculate the fitness of one single triangle, we should figure out which pixels are inside this triangle. So, we created an array of size 40.000 (height of 200 x width of 200) to note the pixels. This means a for-loop over all these pixels and we have to determine if it is inside the triangle or not. If it is, we apply a 1 to this index. If it is not inside the triangle, it gets a 0.

To check if a pixel is in the triangle, we have a method called **isintriangle**. This method receives the coordinates of the triangle and the target point (point to determine if it is in- or outside the triangle). The used method is creating a line between point 1 and 2 and checks if point 3 and target point are on the same side of the line. This will be done for the three combinations of points and if it is all true, the target point is inside the triangle (return true).

After doing this for all the pixels in the solution, we have an array with length 40.000 to calculate the fitness with. The old method to calculate the fitness is used, but in this case, it will only be calculated for the indices with a "1" (which means that it is inside the triangle).

Getsharedfitnesses

After preparing the data with the two order-methods, it is possible to calculate the shared fitness for all the individuals in the population. To get the shared fitness, you should constantly calculate the difference between two individuals, so we created a method for that called **getdifference**. As the name says, it gets the difference between two individuals. This difference is a number between 0 and 1. The absolute difference between the two individuals will be calculated per index. The maximum difference between two individuals is: 100 (number of triangles) * (4 * 256 (maximum color) + 3 * 200 (maximum heigth) + 3 * 200 (maximum width))

= 222.000. All the differences per index will be divided by this number and summed up for all the 1000 indices.

To calculate all the sharedfitnesses in the population, one individual per time should be compared with all the others. These differences between all the other individuals multiplied by the fitness of the original individual, gives us the shared fitness.

6. Impact of the Methods after Application

Excel Report

We felt the need to have an Excel file with the finesses for generation at every run, in order to have a general overview about the trend through the runs by creating a line graph and making decisions on which methods to finally use and what set of parameters are best combined.

So, we created a new class, CreateExcel, we defined a method to write, after each run, the results in a new array. Then we created a new method to be run at the end of the very last run, in order to write the content of the arrays in the Excel file. In order to do all the implementations above, we used the Apache POI Java Libraries.

Attention: a change of the destination file path has to be done, in order to let the program write the Excel file in the local machine; the change can be made in the CreateExcel class, within the try {} at the very last lines.

Initialization (Impact)

We implemented a new initialization method which we called initializeFillingBorders (Average). This method is slightly similar to the baseline method initialize except for that fact that a certain number of triangles have been used to fill the boarders from the outset. We compared the results of both initialization stages which shows that the initializeFillingBorders produced better solutions than the baseline. The results comparison is shown below:

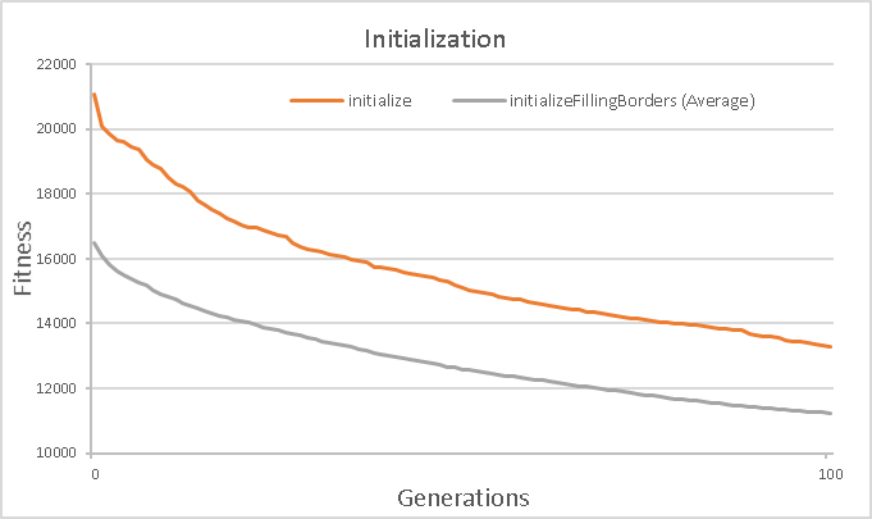


Image 3. Comparison between the two initializations

The results show that the baseline fitness level dropped from 21,000 (at starting stage) to about 13,500 (at finish stage). However, the initializeFillingBorders method produced better results by dropping from about 16,300 (at starting stage) to about 5,000 (at finish stage). Hence the need to apply different settings/parameters to the initializeFillingBorders for improved results as shown below:

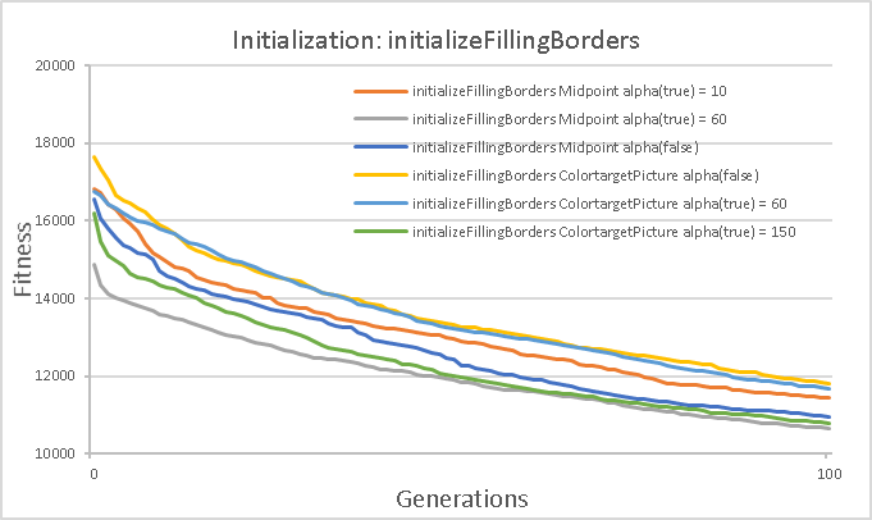


Image 4. Initialization with different parameters and settings

Before drawing conclusions on our own implementation, we experimented on the key elements of the solution by changing parameter values of the baseline implementation of Crossover probability, Mutation probability and selection methods. The results are as seen below:

a. Changing parameters on Baseline Implementation of Mutation Probability

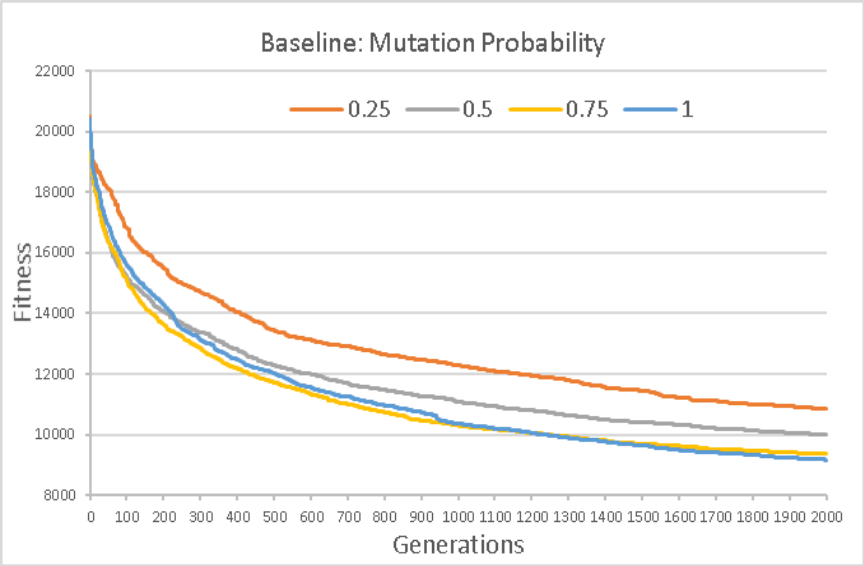


Image 5. Comparison between different values of Mutation Probability

Here, we arrived at the conclusion that a Mutation Probability value of 1 gives a better fitness.

b. Changing parameters on Baseline Implementation of Crossover Probability

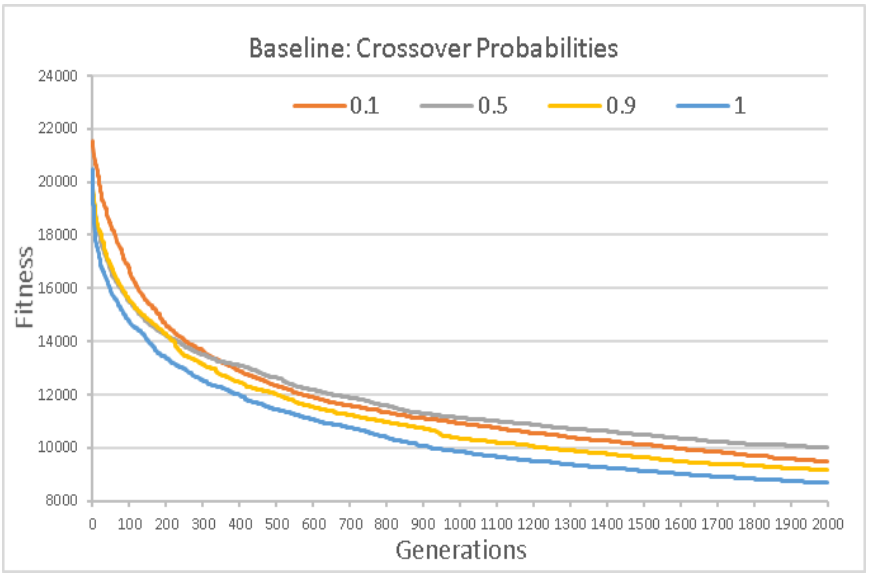


Image 6. Crossover Probablity with different values

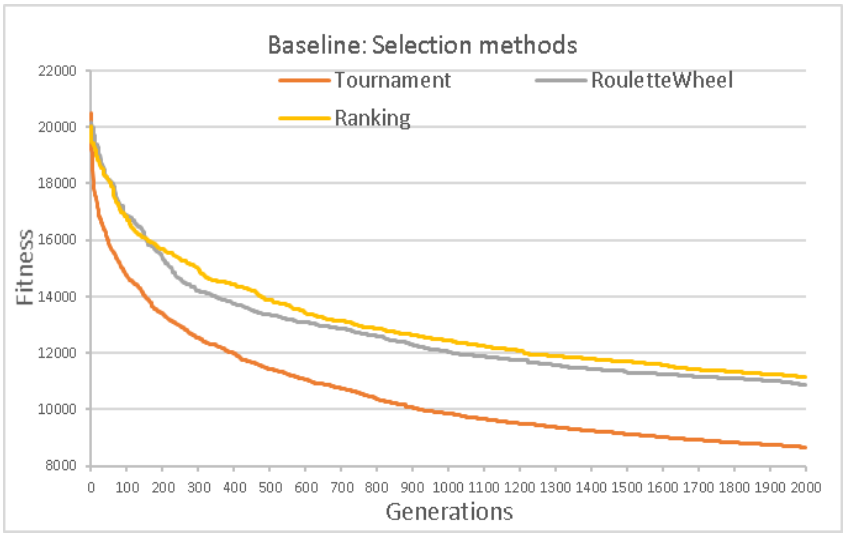


Image 7. Selection methods

Again, we arrived at the conclusion that a Crossover Probability value of 1 gives a better fitness value.

Having established that both Mutation Probability and Crossover Probability values of 1 give better fitness value, we run the Baseline implementation for the Selection Methods:

Tournament, Roulette Wheel and Ranking (using Mutation Probability = 1.0, Crossover Probability = 1.0 and Tournament size = 3)

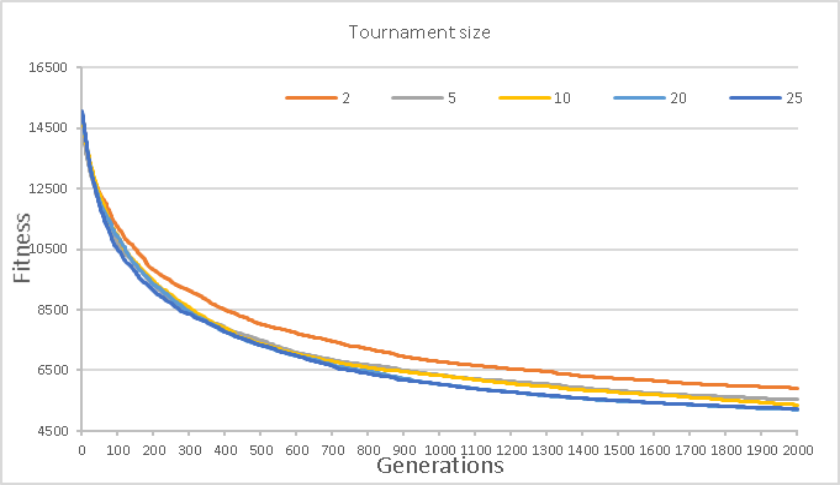


Image 8. Comparison between different tournament sizes

As seen above, the results showed that the Tournament Selection method was the best approach to this problem. We explored further on the Tournament Method by experimenting on Tournament Sizes of 2, 5, 10, 20 and 25. The result is as shown:

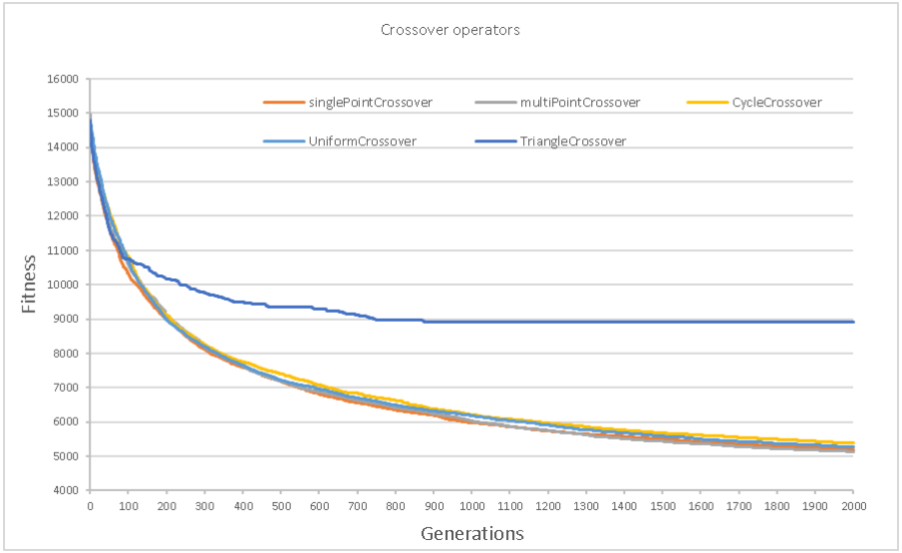
In all our runs, the Tournament size of 25 gave us the overall best fitness value. A high tournament size puts high pressure on the selection process as only the strong individuals survive.

Crossover (Impact)

Following our tests during the initialization phase, we were able to establish the best combination of settings and parameters that would give us the best fitness possible. These settings are:

i. initializeFillingBorders (Midpoint),
ii. Tournament Size (25),
iii. applyMutationTargetPictureColor,
iv. Mutation Probability = 1.0 and
v. Crossover Probability = 1.0

a. We applied these settings over a couple of different crossover methods which we implemented. The results are as shown below; singlePointCrossover and multiPointCrossover method gives us "Best" Count result:



Row Labels	Count "Best"
singlePointCrossover	1077
multiPointCrossover	868
UniformCrossover	44
TriangleCrossover	12
CycleCrossover	0
Grand Total	**2001**

Image 9. Different crossover operators

The Triangle Crossover didn't perform as expected, but this probably due to complexity and terminology of the method. The following graph is showing us the best crossover operator per generation (excluding TriangleCrossover) based on the average fitness, showing that the singlePointCrossover is giving a better fitness until generation 1100 after multiPointCrossover returns a better fitness.

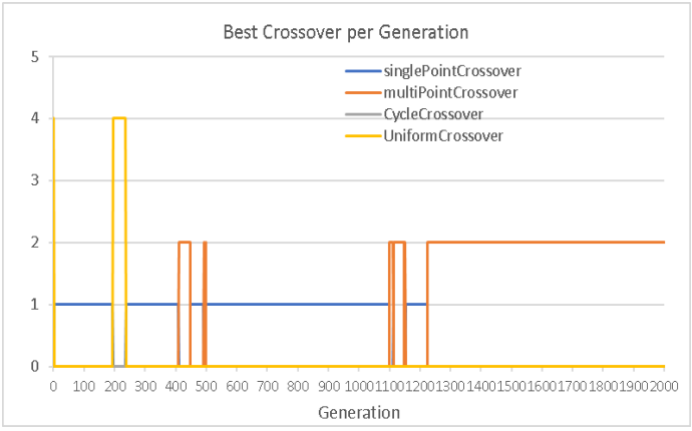
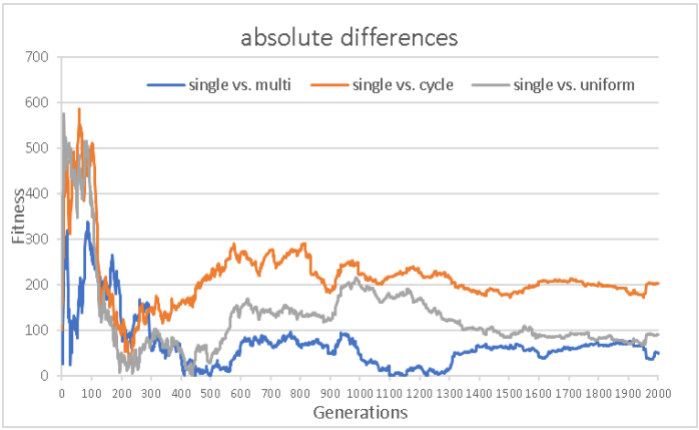


Image 10. Best crossover per generation



The graph above (absolute differences) shows the absolute differences between crossover operators, showing the lowest difference for single and multipoint crossover, showing the least between single and multipoint crossover.

Compare different mutation operators

The different mutation operators have been evaluated by using the following settings and parameter:

initializeFillingBorders (Midpoint), Crossover prob. 1.0, Mutation prob. = 1.0, Tournament, multiPointCrossover/UniformCrossover

Overall, we were able to achieve the best results with the applyMutation and applyMutationTargetPicture. applyInversionMutation or applyScrambleMutation converge much quicker during the first 100 iterations, but converge between a fitness of 7000 and 9000.

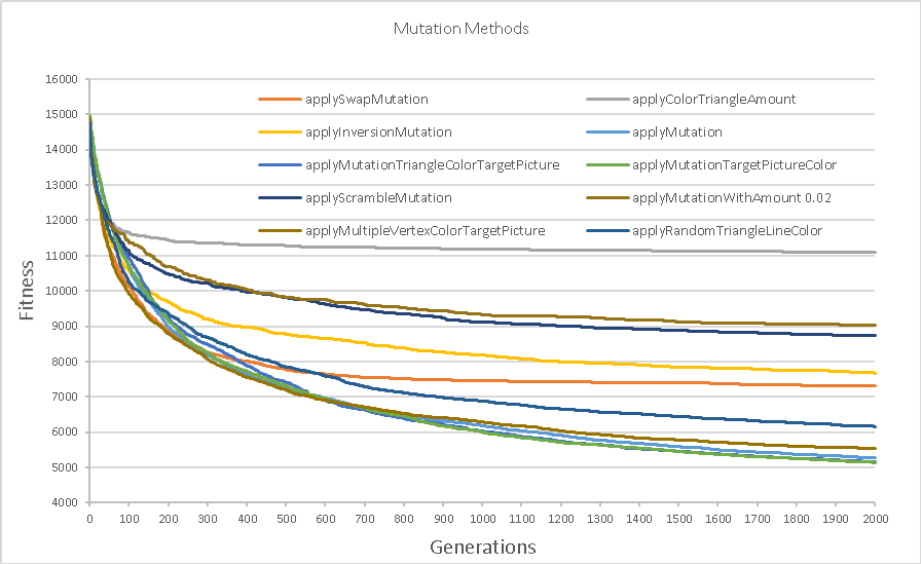


Image 11. All the different mutation methods implemented

Instead of just changing one index value of an individual the applyMutationwithAmount operator is given a specified “amount” as described before. Here we can see the comparison of different amounts used and see as higher the amount chosen the worse the fitness gets. Therefore, a lower mutation amount is proven to be more efficient for this optimization problem.

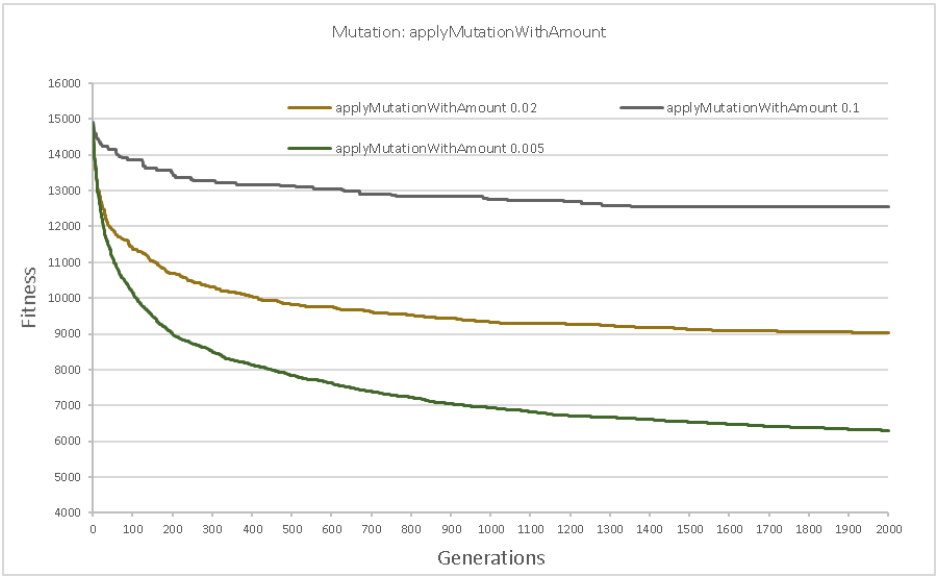


Image 12. Mutation with different amounts

Instead of applying a flat mutation rate (default) over the all generation we implemented a decayed mutation rate and randomly selected mutation rate:

- default: apply constant Mutation rate
- decayed: apply exponentially decreasing decayed rate
- random: apply random probability

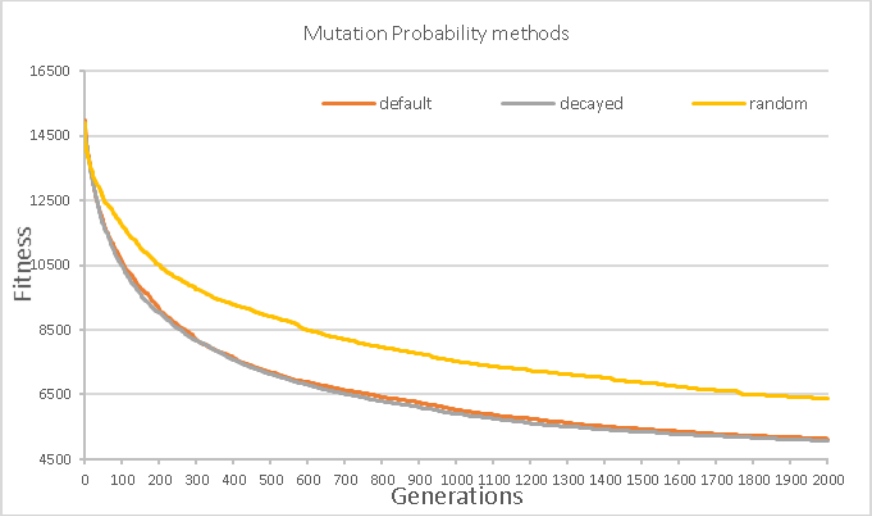


Image 13. Comparison between different probability methods

Settings: Crossover Prob 1.0, Mutation Prob 1.0, initializeFillingBorders, Midpoint, Tournament, multiPointCrossover, applyMutationTargetPictureColor
This shows that the decayed and default mutation rate achieve quite similar fitness, whereas a randomly chosen rate converges around 6500.

The best fitness we achieved with the following setting:

Initialization: *initializeFillingBorders*
InitializeColor: *Midpoint*
Crossover: *mulitPointCrossover;*
Mutation: *applyMutationTargetPictureColor*
MutationRate: *decayed*
CheckFitnessXOMut: *false*
BestXO: *false*
BestMut: *false*

Got 5041,35 as a result for run 1			
All runs:	5041,35		
Got 5117,44 as a result for run 2			
All runs:	5041,35	5117,44	
Got 5048,87 as a result for run 3			
All runs:	5041,35	5117,44	5048,87
	Mean +- std dev	Best	Worst
Results	**5069,22 +- 34,24**	5041,35	5117,44

Fitness Sharing

At the end, we implemented a fitness sharing method to find another local optimum. The methods mentioned before are used to realize the fitness sharing. Besides those methods, we made a new replacement method to apply the fitness sharing based on the shared fitness instead of the calculated fitness. To give an indication of the effect of the fitness sharing function, we applied the settings below for fitness sharing and for no fitness sharing.
This test is done with the same setting as mentioned above:

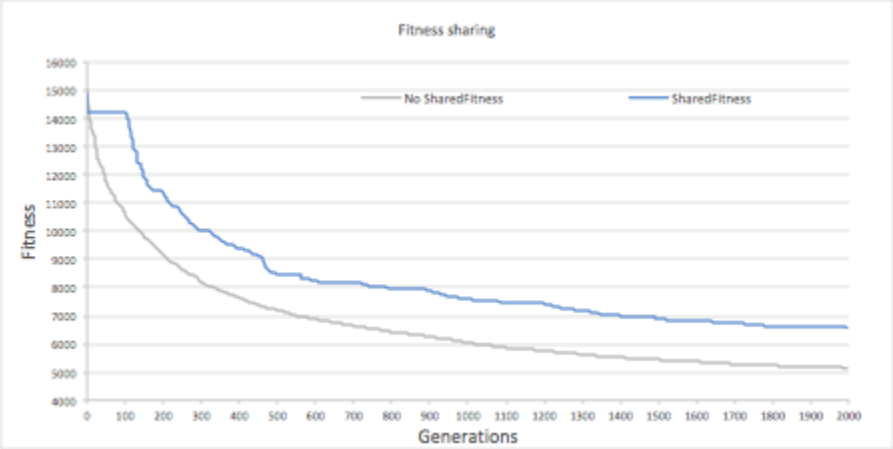


Image 14. Comparison between FitnessSharing and Non-Fitness sharing

It is obvious that there is no improvement after applying the fitness sharing. After comparing with this crossover method, we can conclude that there is no effect of the fitness sharing or the implementation must be reviewed to consider further improvements. There could be a further research in a following project.

Another approach

The idea behind this approach is to use the operators which fits best to a certain generation. We have implemented three approaches, which can be enabled by setting the parameters in (bracket) to true (Main class). Approaches should be used separately due to performance and NOT all at once.

a. Choose best crossover or mutation operator (BestXO and/or BestMut)

Instead of choosing a crossover or mutation operator based on the evaluation above we suggest comparing the different crossover and mutation operators during each iteration choosing the one with the best fitness.
for (int k = 0; k < population.length; k++) {

Crossover

1. generate an offspring for each crossover operator using the same parents.
2. Then evaluate the fitness of those offsprings e.g. ZOffspring (operator) has best fitness
3. Take the best fitness, offspring[k] = ZOffspring;

Mutation (Best Mut)

1. Clone the offspring and use it to apply mutation operator to each clone separately
2. evaluate fitness of those offsprings (mutated clones)
3. take offspring (mutated clone) with best fitness and replace offspring[k]
4. offsprings[k].evaluate(); next k

b. Check if Parents, xo offsprings have better fitness (CheckBestParentsXO and CheckFitnessXOMut)

for (int k = 0; k < population.length; k++) {

Crossover (CheckBestParentsXO)

1. checks if Parents fitness is better than xo offspring fitness, take best one

Mutation (CheckFitnessXOMut)

1. checks if xo offspring fitness better than mutation offspring fitness, take best one

c. Apply several crossover and mutation operators (CheckBest2XO, CheckBest2Mut)

for (int k = 0; k < population.length; k++) {

Crossover (CheckBest2XO)

1. apply crossover as usual

2. until termination condition is met: apply several xo operators to the same offspring[k] randomly and check if fitness improves, if offspring's fitness is worse than at the start, take the offspring from 1.

Mutation (CheckBest2Mut)

1. apply mutation as usual

2. until termination condition is met: apply several mutation operators to the same offspring; after multiple mutations of the offspring check if fitness is worse than the 1.; take better one

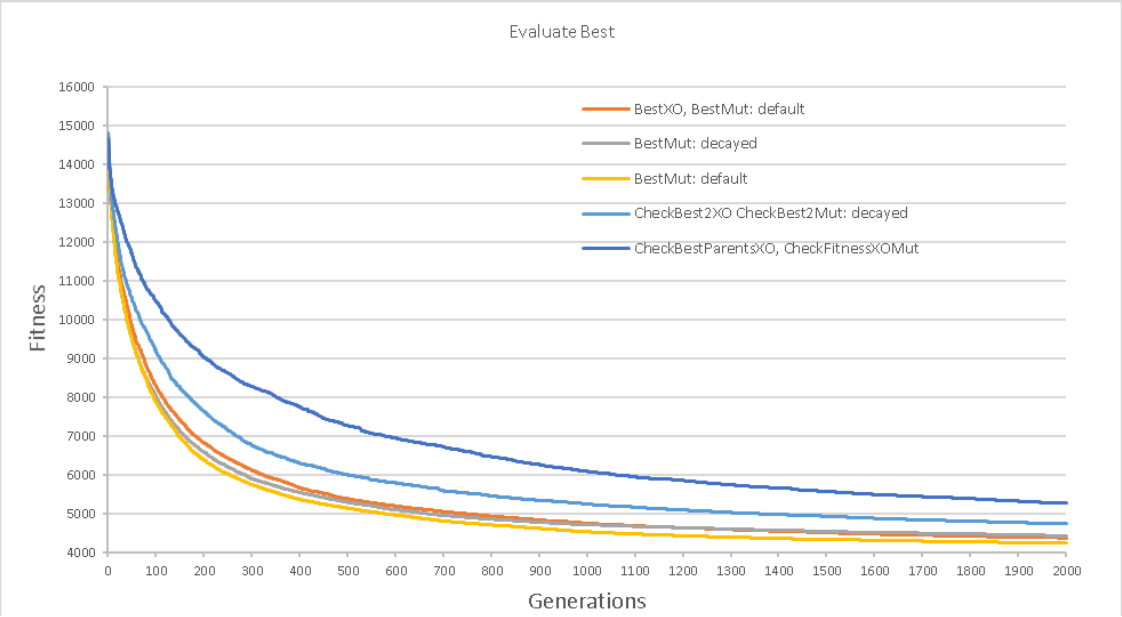


Image 15. Comparison between the different mutations

Best fitness:

We achieved the best fitness by using BestMut (=true), default as shown in the graph above. To summarize the result, see as follows:

Got 4344.09 as a result for run 1			
All runs:			
Got 4130.93 as a result for run 2			
All runs:			
Got 4254.66 as a result for run 3			
All runs:	4344.09	4130.93	4254.66
	Mean +- std dev	Best	Worst
Results	**4243.23 +- 87.40**	4130.93	4344.09

7. Conclusion and Recommendation

We hereby draw some basic conclusions from what where have done. Summarily, after having:

- Explored the baseline implementation,
- Explored influence of different parameter sets for the problem class,
- Implemented and tracked the effect of other algorithmic features,
- Tried other variation policies,
- Tried other variation operators (mutation and crossover),
- Implemented population diversity measures e.tc,

We believe the baseline version of the provided Genetic Algorithm problem has been outperformed. And we recommend that for any Genetic Algorithm Problem, indeed, several of the above noted points (and sometimes, more) are necessary to be carried out for an optimal solution to be achieved.