Optimization Exam
Enhancing the simple evolutionary algorithm

PBA Software Development 2020

Lars Søndergaard Petersen

Table of content

1. Your experiments in general.

I started out trying to loop all the different parameters for P1, like I did when analysing the iterated hillclimber. The enhanced simple evolutionary algorithm has more parameters, resulting in six nested loops needed to try all combinations. To increase data quality, I wanted to keep the step size small, this resulted in a too big amount of data. It is not that relevant but if interested see *iterateParams()*.

Instead I elected to follow the suggested approach and created a different method, locking the values to the defaults and varying one parameter at a time. *See: iterateParamsIndividually().*

All the raw data from the experiments can be found in an excel file, no appendix have been added, instead look in the file: Optimization_Exam_Lars_Experiments.xlsx

Common for all the experiments is that one parameter is varied, and the others are locked, these are the default parameters:

Generations	PopulationSize	TournamentSize	Elite	CrossoverRate	MutationRate
100	100	5	5	0.5	0.025

The following experiments have been run:

	Min	Max	StepSize
Generations	0	500	20
PopulationSize	20	200	20
TournamentSize	1	15	1
Elite	0	20	1
CrossoverRate	0.1	1	0.1
MutationRate	0	0.51	0.025

Each experiment has been run 1000 times, the average genes and evaluation has been calculated for the supplied problems problems (P1, P2, RevAckley, RevSphere and RevRosenbrock).

Experimenting with collecting statistics about in what generation improvement occurred proved very useful for comparing different sets of parameters, see section 7.

Evolving parameters for each problem was a fun challenge, even thou it was very easy to plug into the existing code by extending the problem class. It gave some useful insight and even improved improvement for some problems, see section 8.

When the deadline has been passed all the code, report and experiment data will be made public here: https://github.com/trezum/OptimizationExam2

2. The effectiveness in terms of number of evaluations, compared to the iterated hill – climber?

The best solution found for RevAckley in my previous hillclimber experimentation was -0,00003440397255, it took 7577600 calls to the eval function.

The enhanced evolutionary algorithm (EEA) gets better solution quality at 13305 evaluations using the default parameters and 140 generations. That is about 569 times less computation needed to reach the same level of precision. What a difference!

RevAckley is relatively complex, maybe the iterated hillclimber has an advantage on simpler problems like P1.

From my previous experimentation I had a run with 2068 evaluations on P1 by the iterated hillclimber. This resulted in the evaluation 0,99999851386049.

With 1905 evaluations using default parameters, but only 20 generations, EAA reaches 0,999999931158378, so two more decimals of precision. So even thou the problem is simple the EEA gives better precision per evaluation.

Some experiments could be made stopping the algorithms after the same number of evaluations, but judging from the above examples it is pretty clear EAA is the more effective of the two.

3. Where do you think improvement happens in the evolutionary algorithm?

Evolutionary algorithms generally have the following parts where improvement happens.

Elite

The elite is some number of the best individuals in a population, that are preserved unchanged and added to the next generation. Having an elite is not improving the solution directly but can be beneficial depending on the problem being evolved on.

Selection – Mating

This part of the algorithm is where parents are selected for recombination.

In the code we are working with, mating selection is done by tournament, some individuals are selected randomly and the fittest one is the winner of the tournament. By selecting fitter individuals for recombination, the overall fitness of the population is likely to improve. If an individual is not selected as parent its genes die out, unless it good enough to be an elite.

This can be found in the method *tournamentSelection(Population pop)*.

Selection – Environment

Removing the least fit individuals from the population could be environmental selection. One can imagine many other ways of doing this. Could be changes in the environment, introduction of predators and so on.

In the SimpleGeneticAlgorithm class I would argue that we do not have any environmental selection, because no individuals are killed outside of not being selected as parents.

Variation – Recombination

The genes of the selected parents are combined in some way to make up the genes of the child.

In the code for this project it is done in the method *Individual crossover(Individual parent1, Individual parent2)*. Even thou we are working with a crossover rate in this project the children are recombined every time. But the rate decides how often they are set to the average values of the parents instead of just inheriting the genes.

Variation – Mutation

Mutation is adding some random change to the population. This helps with not getting stuck in a local optimum by exploring solutions not based on parents.

In our code the mutation based variation can be found in the method *mutate(Individual indiv)*.

We mutate at some rate, and make sure to stay within the min and max of the given problem.

The random variation is based on a gaussian probability distribution, a distribution that is very prevalent in biology. Practically it means we have a high chance for a small change and a low change for a big change of the genes.

4. What was the closest you got to the optimal point for each problem, during all your runs?

During the experiments the best solutions where the following:

	Closest eval	Experiment:	Evaluations
P1	0,999999999999934	PopulationSize: 200	19505
P2	2,00311674600683	MutationRate: 0.45	9505
RevAckley	-0,0000011760893856931	PopulationSize: 200	19505
RevSphere	-0,00000054370121629839	Generations: 500	47505
RevRosenbrock	-0,058328591645830	MutationRate: 0.5	9505

The fact that the generation experiment is not always the closest shows that compute is not king here, even thou other experiments ran with more eval calls. Some of the problems need optimized settings to perform better.

5. What was the best parameter setting for your evolutionary algorithm for each problem?

Here are the best parameters when optimized individually:

	Generations	Population size	Tournament size	Elite	Crossover	Mutation
P1	500	200	2	0	0.9	0.475
P2	500	200	5	4	0.3	0.450
RevAckley	500	200	2	0	0.9	0.475
RevSphere	500	200	15	3	0.2	0.075
RevRosenbrock	480	200	6	6	0.6	0.500

It looks like generations and population size just need to be as big as possible, besides for the RevRosenborck problem for some reason. Still I would say it is a good general conclusion as it does increase evaluations. Tournament size has been capped on the RevSphere problem, so it could potentially be optimized more. The same goes for mutation on RevRosenbrock.

Running with the settings found by experimentation the solutions averaged over 1000 runs where:

	Evaluation	Eval calls
P1	0,99998260989367700000	100000
P2	2,00311674606299000000	98004
RevAckley	-0,00030393192483468300	99002
RevSphere	-0,00000008153160405188	98503
RevRosenbrock	-0,00006538570684559180	93126

For P1 and RevAckley the solution was worse than the one found in the experimentations, with default values and a population sized of 200. This is likely because of not optimizing combinations of the parameters but one at a time. Better solutions can be found with a smaller step size, larger intervals and by testing combinations like I wanted to do with the mentioned method *iterateParams()*. I have a different idea that can be found in the section named Evolving the parameters.

6. What was the average closeness to the optimal points for each problem, over 20 runs?

Reducing the number of runs to 20 and using best found parameters give the following solutions:

Problem		Eval calls	Optimum	Closeness to optimum
P1	1,00000000000000000000	19505	1	0,00000000000000000000
P2	2,00311674606312000000	98004	2,00311674606316	0,00000000000003996803
RevAckley	-0,00026777177016603600	99002	0	0,00026777177016603600
RevSphere	-0,00000008176832904147	98503	0	0,00000008176832904147
RevRosenbrock	-0,00006841995171256370	93126	0	0,00006841995171256370

For P1 we get so close to the optimum that ither java does not support handling the difference or we got “lucky” and hit exactly on the optimum.

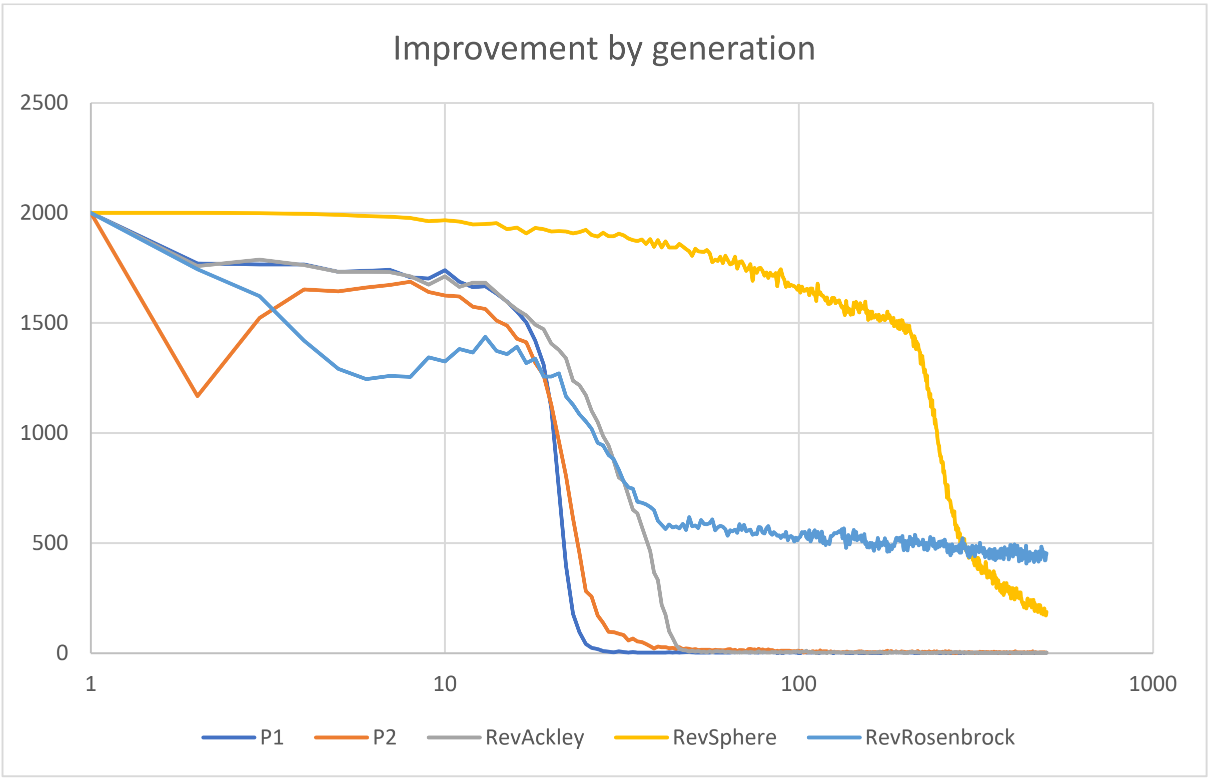
RevAckley is also closer than the previous experiments with 1000 runs. My guess is we have had some lucky runs this time around.
The rest of the problems are further off than before, but all in all we are getting pretty close.

7. When does improvement stop?

To get a better overview of what generations improve the most on the given problems I made some small additions. The method *int[] runAlgorithmReturnStatistics()* runs the algorithm exactly like the regular *Individual runAlgorithm()* but it collects some simple statistics. When a generation improves the most fit individual in the population, the matching index, in the returned array, is counted one up.

Another method was added to iterate this 2000 times, with the default parameters but for 500 generations and collect multiple arrays into one, see *improvementStatistics()*.

This is graph shows the collected data, it can also be found in the excel file:



Notice that the x-axis is set to logarithmic scale.

When running with the default parameters P1, P2 and RevAckley stagnate on improvement at around 30-50 generations. They might be so simple that they are simply solved, with the precision we have available in java at that point.

RevSphere however is still improving 500 generations in, even thou it looks like it is about to stagnate too.

RevRosenbrock looks like it still has a long way to go.

8. Evolving the parameters

As I discovered writing the *iterateParams()* method the search space to find the optimal settings for evolving a specific problem is pretty big. Hmmm do I know any effective methods for searching large spaces?

Implementing a meta evolution problem proved to be easy, see the class *MetaEvolution*. In order to optimize the needed number of eval calls when possible I added a method to the problem class and implanted it in each problem *Double getOptimumEvalIfKnown()*, and added the method *Individual runAlgorithmWithStopAtOptimum()* to the *SimpleGeneticAlgorithm* class.

This is done to be able to stop the algorithm if the optimum is found. By using the number of evaluation calls required to find the optimum in our optimization we can put pressure on the algorithm to evolve in direction of fewer eval calls.

Additionally to keep the run time workable the evaluation function of MetaEvolution was set to a population size of 50, 100 generations and only averaging out over 20 runs.

I did not evolve generation or population size, because the experiments show that they need to be as high as possible, constraint by time. All the parameters had some values outside my intervals in experimentation. Here is the parameters suggested by evolution:

	Elite	Tournament Size	Crossover rate	Mutation Rate
P1	49	2	0,00744272694806503	0,686219116688251
P2	49	28	0,50352736404037500	0,000518120601988
RevAckley	27	1	0,65349378631203100	0,000000000000000
RevSphere	1	27	0,45338452344530400	0,067058123074578
RevRosenbrock	2	28	0,41709542320557900	0,971687610924064

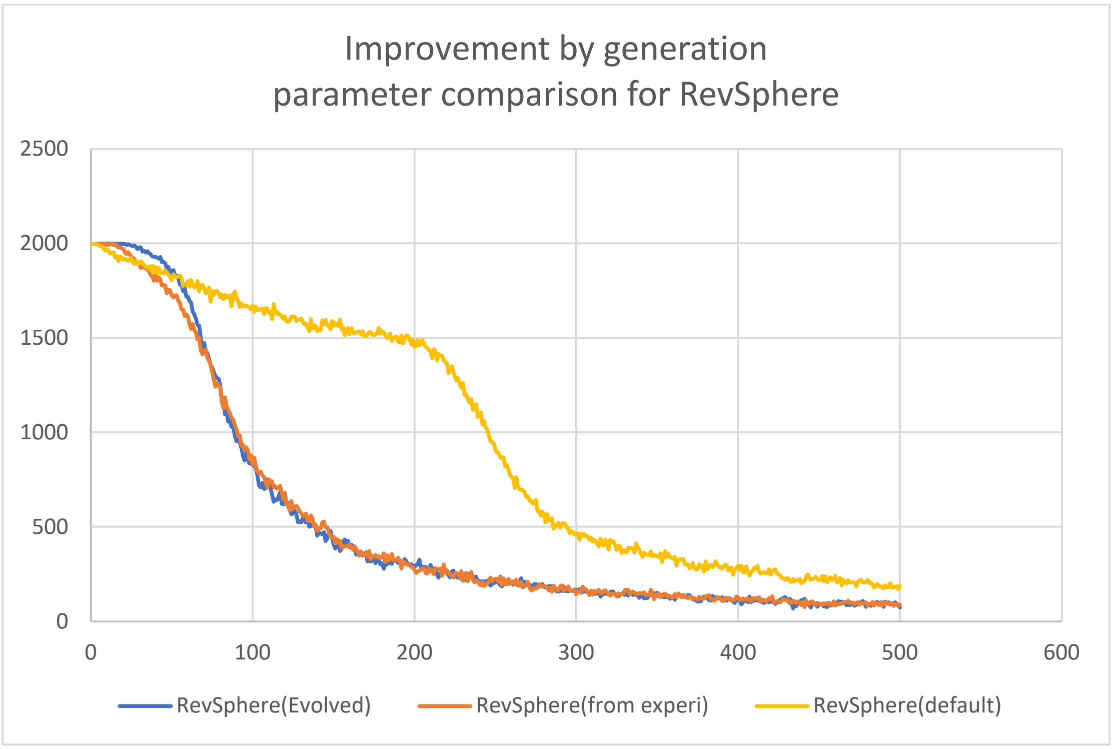
The elite being 49 for P1 and P2 might seem odd, but it is capped by the population size. So P1 and P2 does not benefit much from the evolution part of the algorithm.
The same goes for tournament size, to some degree. These two parameters could be handled as a percentage of the population size. The interactions are not totally clear, so this warrants more experimentation if a conclusion is to be drawn.

Here are the best values found with the evolved parameters:

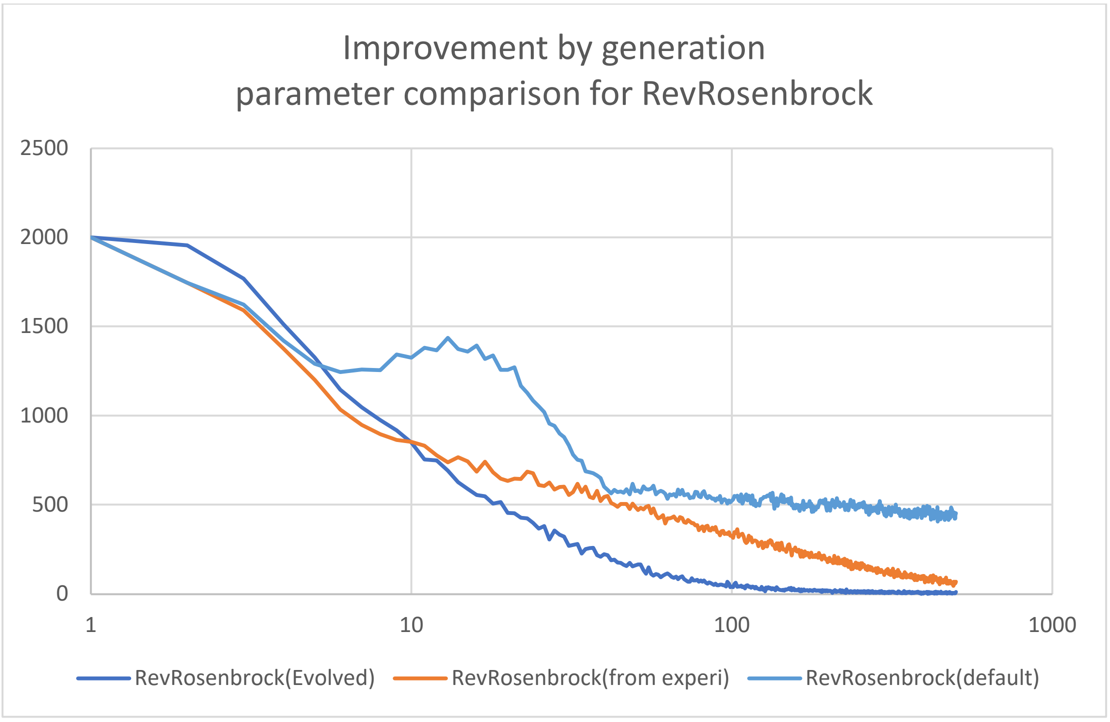
	Evaluation(Evolved)	Evaluation(Experimentation)
P1	0,99999999931160400000	0,99998260989367700000
P2	2,00311029730319000000	2,00311674606299000000
RevAckley	-0,00004878092150243370	-0,00030393192483468300
RevSphere	-0,00000008187862698896	-0,00000008153160405188
RevRosenbrock	-0,00046887376879470500	-0,00006538570684559180

As you can see not all the evaluations where improved, the optimization was to reduce the number of evaluations, so this is to be expected.

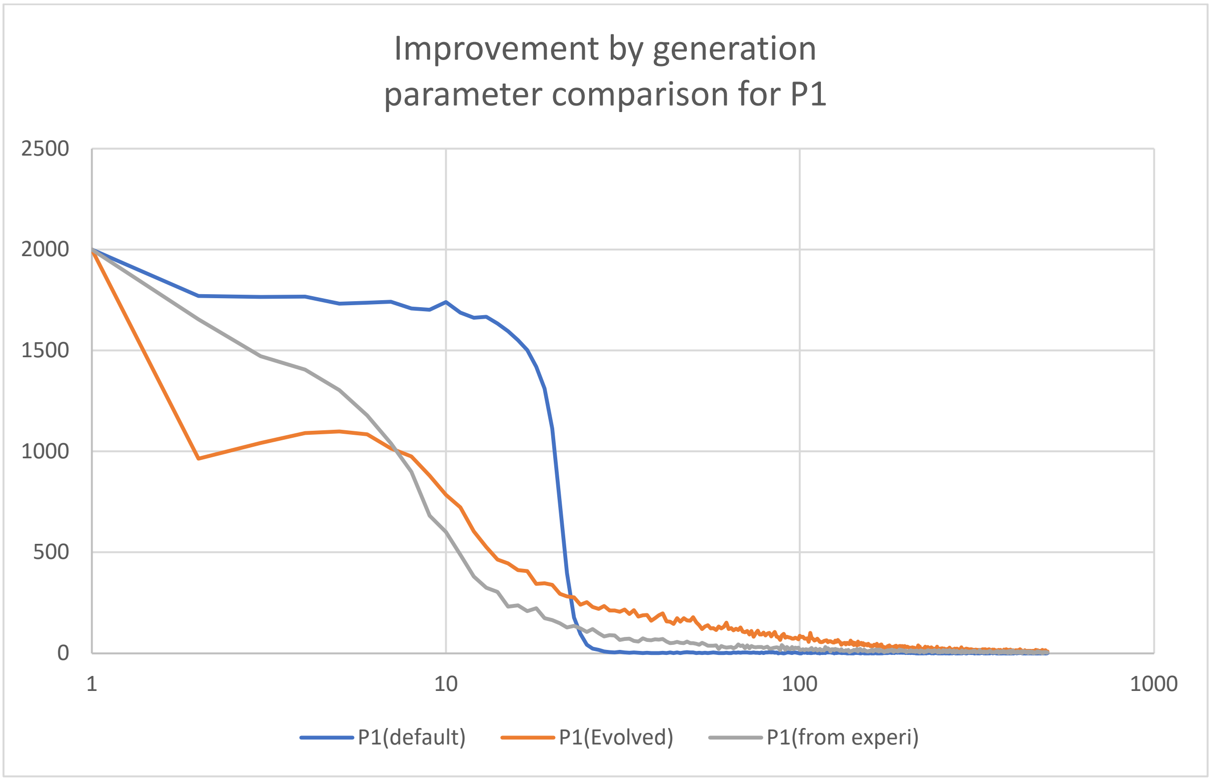
Judging from the graph below it is clear that the original optimization was good enough for the relatively simple RevSphere problem, and that even that was a huge improvement compared to the default values.



The graph below shows that evolving parameters for the RevRosenbrock problem made at difference in improvement by generation. Even compared to the individually optimized parameters. The problems are different, some are harder to optimize than others and evolving parameters clearly helped on this one.



Here is a similar graph for P1.



The improvement is fastest in the evolved case. I believe the reason for crossing back on top of experiment is that it is already much closer to the optimum than experiment at that point.

The parameters here are very similar, but the evolved settings should be more effective if number of generations must be low.

It is very fast to find the optimum for P1 so optimizing it based on the numbers calls to eval was very effective.