UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Evolving models with Estimation of Distribution Algorithms

propusă de

Adrian Tiron

Sesiunea: *iulie, 2019*

Coordonator științific

Prof.dr. Henri Luchian

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Evolving models with Estimation of Distribution Algorithms

Adrian Tiron

Sesiunea: *iulie, 2019*

Coordonator științific

Prof.dr. Henri Luchian

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conţinutului lucrării de licenţă

Subsemntatul(a) ……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …….. specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul: _____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „ *Evolving models with Estimation of Distribution Algorithms*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iaşi, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi, *data*

Absolvent *Adrian Tiron*

_____

(semnătura în original)

I Table of Contents

II Introduction

Evolution is a process of optimization. The idea comes from Darwin, who was amazed by 'organs of extreme perfection', such as the eye. *The Darwinian theory of evolution* explains the concept of adaptation of organisms by the principle of natural selection, which implies "survival of the fittest", meaning that the individuals best adapted to their environments and utilizing it to the full extent are surely going to contribute most to the future generations. Darwin does not say that evolution is perfect, and it isn't, but it sure does discover solutions to the organism's problems that, in some cases, are highly functional. Given these concepts, we find it quite natural to describe evolution as an algorithm that can be adapted into our own problems posed by our own environment.

Evolutionary computation is a family of population-based trial and error algorithms for global optimization inspired by biological evolution, with a metaheuristic or stochastic optimization character. Generally speaking, a set of chromosomes is generated and updated through iterations. Every new generation is engineered in such a way as to keep the more promising individuals and change them in small ways (just enough to keep their good attributes) and remove entirely the bad ones. The changes are made by the use of mutation and crossover. The population, with each iteration, increases in fitness, getting closer to an acceptable solution.

The subset of evolutionary computation which concerns us is *evolutionary algorithms* (EAs). Genetic algorithms (GA), evolution strategy (ES), genetic programming (GP) and evolutionary programming (EP) are very popular EAs. They model the strategy of optimization within a solutions population, each of them representing a particular solution in the search space of solutions to a mathematical problem. The starting population is initialized by an algorithm-dependent method, and evolves to better regions of the search space, using a combination of genetic operators. The environment delivers a *fitness* value (measurement of quality) for new search points, and the selection favors those individuals more likely to successfully pass on their good characteristics. The crossover operator combines features of parents and passes them on to future generations, and mutation introduces diversity to the process.

III Genetic algorithms

Genetic algorithms are a metaheuristic that belong to the class of *evolutionary algorithms* (EA) described above. They were proposed by John Holland in the 1970s after many years of studying the idea of simulating evolution. These algorithms shape genetic inheritance and the Darwinian struggle for survival. GAs are probabilistic in the sense that they maintain a population of candidate solutions, evolve in the course of generations / iterations and measure individual merit under the control of a fitness function.

GAs use a vocabulary borrowed from genetics:

- evolution is simulated by a succession of generations of a **population** of candidate solutions;
- a candidate solution is called the **chromosome** and is represented as a sequence of genes;
- the **gene** is the atomic information of a chromosome;
- the position occupied by a gene is called **locus**;
- all the possible values for a gene form the **allele** set of the gene;
- the population evolves through the application of genetic operators: **mutation** and **crossover**;
- the chromosome on which a genetic operator is applied is called a **parent** and the resulting chromosome is called a **descendant**;
- **selection** is the procedure by which the chromosomes that will survive in the next generation are chosen; better adapted individuals will be given greater chances;
- the degree of adaptation to the environment is measured by the **fitness function**;
- the **solution** returned by a genetic algorithm is the best individual of the last generation.

At each step, a **GA** three rules to help shape the next generation as a better version of the current one:

1. In selection, parents are selected through a method that assures us of the good fitness of the chosen individuals.
2. In crossover, two parents are interbred to create children that will go in the next generation.
3. In mutation, small random changes are applied to chromosomes to enhance exploration.

III.1 Selection

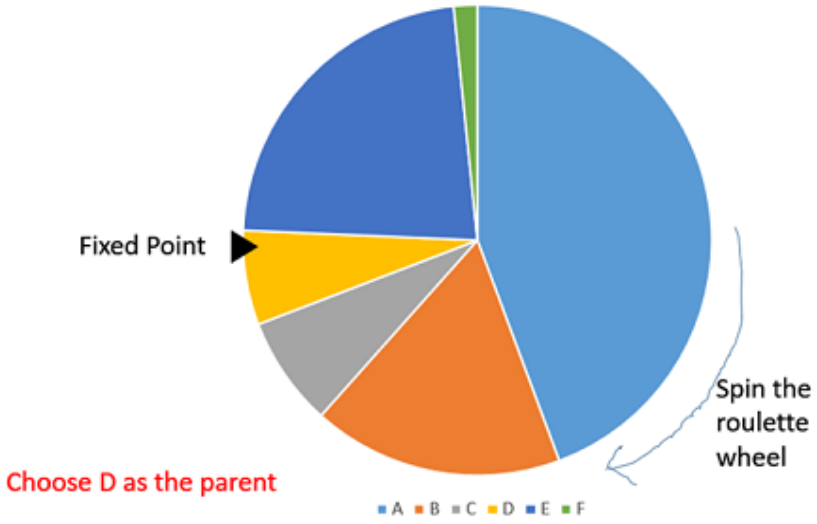
The selection operators used are completely deterministic. In each iteration, we select a number of chromosomes from the population through a method that relies on fitness, meaning that fitter individuals have a bigger chance to be selected. Some methods rate the whole population, while others rate only a random portion, as the first one can get quite time-consuming.

III.1.1 Fitness Proportionate Selection

As mentioned above, every chromosome has a chance to be selected based on its fitness. These individuals will then propagate their good features in the following generation. Fitness Proportionate Selection puts more pressure on fitter individuals, evolving them over the course of the iterations.

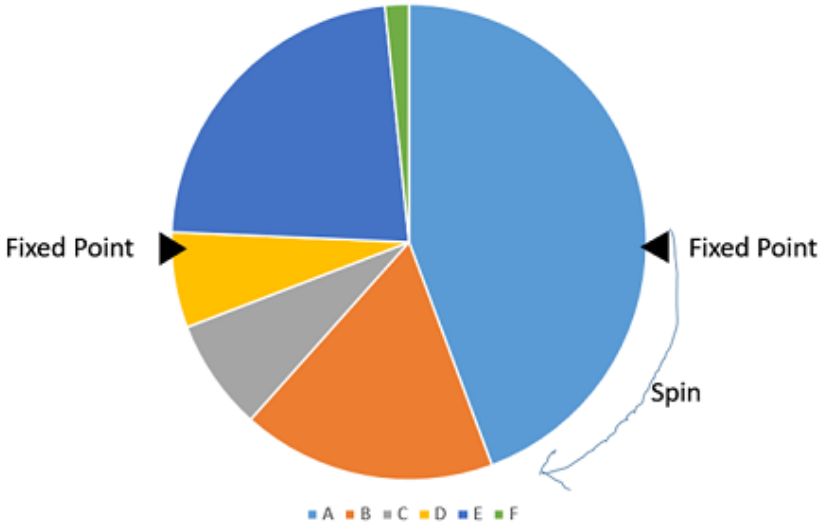
Say we have a wheel divided in *n* parts. Each part represents an individual. The bigger the part, the higher the change that, when spun, the arrow will land on that individual.

Fig. 1 – Roulette Wheel



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Fig. 2 - Stochastic Universal Sampling



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

If we want to choose two parents for crossover, the first is chosen normally by spinning the wheel, and the exact same process is repeated for the second parent.

Stochastic Universal Sampling employs the exact same ideas as Roulette Wheel, the difference being that multiple fixed points are chosen (as shown in the image above) and when the wheel is spun, more chromosomes are chosen at once, improving on time performance.

III.1.2 Tournament Selection

In tournament selection, K individuals are randomly selected and, just like in a tournament, choose the one with the best fitness value. Just like in Roulette Wheel, if we want to find two parents, the process is repeated. A big advantage of this method is that it can work with negative fitness values.

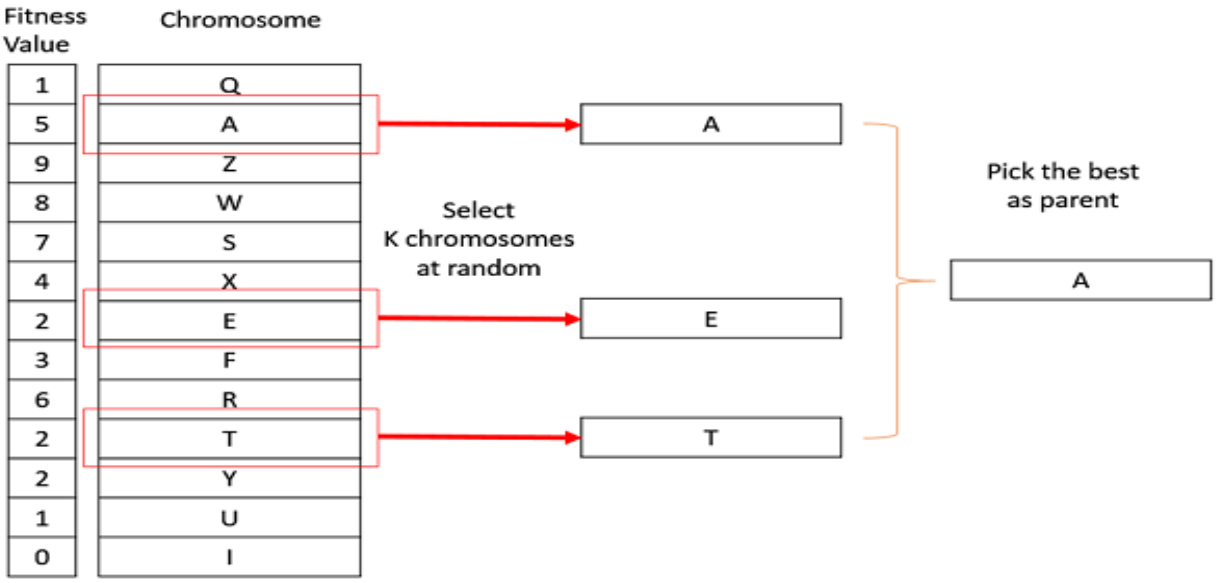


Fig. 3 - Tournament Selection

III.1.3 Rank Selection

This method also works with negative fitness values and is efficient near the end of the algorithm, where the chromosomes have very close fitness values. If we were to use Roulette Wheel, each individual would have an equal share of the pie and so it wouldn't matter how better one chromosome is that another, because they would all have the almost same probability to be chosen. Given this problem, it can be solved by ditching the concept of choosing absolute fitness and introducing relative fitness.

Basically, we rank the individuals by their fitness and instead of using their fitness to form probabilities of choosing we use their ranks. This way, chances are further apart and we get closer to a more precise result.

III.2 Genetic operators

After the selection of two parents from the current population using one of the methods described above, they can now have genetic operators applied on them, such as crossover and mutation. The resulting children will bear their parents characteristics. For each new child, a new pair of parents is selected and so on until we have the desired number of individuals. Generally, a higher average fitness is desired and, in a lot of cases, it will happen so.

III.2.1 Mutation

Mutation is used to "explore" the search space by tweaking individuals randomly and in small ways. By applying mutation, we ensure that diversity is maintained and so, it is essential to the convergence of the optimum.

The probability used for mutation has to be low because otherwise, too much random is introduced and we lose track of solutions.

Some of the most commonly used mutation operators are:

- Bit Flip - select one or more random bits and flip them;

- Swap - select two positions on the chromosome at random, and swap the values;
- Scramble - a number of genes is chosen and they get shuffled randomly.
- Inversion – we choose a subset just like in scramble mutation, but instead of scrambling them, we invert all the genes.

III.2.2 Crossover

Generally, two parents are chosen and one or more off-springs are created. A high probability is usually used for crossover.

Some of the most used crossover operators are:

- One Point - a random point is selected and the same side of its two parents are interchanged;
- Multi Point – same idea as **One Point**, but we interchange alternating parts;
- Uniform – each gene is treated independently, meaning that each of them is tested against a probability to see if we flip it or not.

III.3 Termination conditions

All of these steps are followed until we meet one of the following termination conditions:

- No improvement in the population for *N* generations.
- A defined number of generations has been surpassed.
- The current best chromosome has an acceptable fitness value to stop.

A genetic algorithm produces very good solutions in every few generations, but in the later part, improvements are not worth it. Having said that, by combining two or more methods in the same category we could achieve marginally better

results (for example, use roulette selection as the main and then swap it for rank selection near the end of the algorithm).

III.4 Conclusions

Natural evolution is just like a game, where good players (organisms which have continued survival and are efficient in their environment) receive a hefty prize: propagation of their genetic material to their successors. Every game has, evidently, more players, and these other players can come in the way of some, this concept being modelled by genetic operators. Sometimes, evolution can be random, we cannot know what the next generation will bring. Optimization is a very interesting and promising solution to problems anywhere and this domain can always *evolve* to expand and improve on its current techniques.

IV Iterated local search

Iterated local search (ILS) is a simple search method for solving discrete optimization problems. The iterative part of this search is made up of several disturbances produced to the current chromosome that will form a starting point for the improvement method. A big disadvantage is that these methods have the possibility to get stuck in a local minimum, where neighbors with better fitness are non-existent.

Iterated Local Search has two simple steps

(i) perturb the current individual;

(ii) improve on this individual to find more promising neighbor.

IV.2 Hill Climbing

Hill Climbing is heuristic search used for mathematical optimization problems. These problems imply that an optimum of a function has to be found. The

heuristic part implies that the algorithm may not necessarily find the best solution, but it will find an acceptable one in good time.

Most important features of HC are:

1. Generation and testing, in the sense that it generates a possible solution, tests it to see if it is the expected solution, in which case it finishes, else it starts over.
2. Usage of the Greedy approach, meaning that no matter the current position in the search space the algorithm goes in the direction that improves the fitness, even if it leads to a local minimum.

Types of Hill Climbing algorithms:

- Simple Hill climbing: iteratively look through the neighbors and select the first one as the next chromosome.
- Steepest-Ascent Hill climbing: examines all of the neighbors and chooses the one with the best fitness value.
- Stochastic hill climbing: choose one neighbor at random and decide, based on a probability, if it is worth jumping over to that neighbor instead of repeating this process with another one.

IV.3 Simulated Annealing

Simulated Annealing (SA) is useful in optimization problems which can have a big number of local optima, where it can find a reasonable solution (instead of a precise one) in good time. This algorithm refers to thermodynamics, more exactly the way metals anneal (or cool), where a controlled cooling method reduces metal defects.

Implementing a simulated annealing algorithm has the same difficulty factor as Hill Climbing. Instead of choosing the best neighbor, it chooses a random one. If this individual has a better fitness value that the current one, we select it. If not, we

compare a random number between 0 and 1 with a probability which decreases exponentially with the "badness" of the move.

$$Prob\ (accepting\ uphill\ move) \sim 1\ -\ exp(deltaE\ /\ kT))$$

The parameter T means the temperature of the annealing system. At higher values of T, exploration is increased, in the sense that we can also choose relatively worse solutions for the sake of diversity. For values of T close to 0, these changes are less likely to occur, selecting only better solutions for the sake of exploitation, just like Hill Climbing does. Typically, T is initially given a large value and, using a cooling schedule, it is decreased gradually. The parameter k is a constant that connects energy to temperature (Boltzmann's constant).

V Estimation of distribution algorithms

EDAs belong to the evolutionary computation superclass and are non-deterministic search strategies, employing heuristic approaches. Just like in genetic algorithms, a number of individuals is generated and are evolved until an acceptable solution is found. What differentiates EDAs from GAs, though, is their ability to modify their implicit distribution, which dictates the probabilities of apparition of each bit in every position. This modification is done on the fittest individuals, and the sampling is done on the resulted model. By doing this, we eliminate the need for genetic operators, and, also, the number of parameters (that would eventually need to be adaptive) is reduced.

In EDAs, each individual has genes that were generated using probabilities in the current sampling distribution, which is initially the uniform distribution. These genes are also called decision variables which have dependencies between them. The dependencies can be simple or complex, depending on the level of implementation we want to accomplish. Simple dependencies can be univariate modeled, meaning that each variable is independent and the algorithm relies only on statistics. Some examples here are Univariate marginal distribution algorithm (UMDA), Population-based incremental learning (PBIL) and Compact genetic algorithm (cGA). Complex dependencies are

modeled using multivariate distributions, usually represented using Probabilistic Graphical Models (graphs) in which vertices are variables and edges are statistical dependencies.

In an EDA, we distinguish four main steps:

1) At the beginning, the first population $P0$ of N individuals is generated, usually by assuming a uniform on each of the genes (or variables) and afterwards applying the fitness function on the individuals to evaluate them.

2) A number Se ($Se <= N$) of individuals are selected, usually the fittest ones.

3) The chosen probabilistic model (univariate or multivariate) is induced. In this part of the algorithm, we compute the probabilities of each gene to appear in the positions of chromosomes and it is important to represent the dependencies properly.

4) The new population of *N* new individuals is obtained by sampling the model resulted from the previous step.

If a termination condition has not yet been met, steps 2 to 4 are repeated. Examples of stopping conditions are: achieving a fixed number of populations or a fixed number of different evaluated individuals, uniformity in the generated population, and the fact of not obtaining an individual with a better fitness value after a certain number of generations.

V.1 Univariate Marginal Distribution Algorithm

UMDA starts from the central probability vector that has value of *0.5* for each locus and falls in the central point of the search space. Sampling this probability vector creates random solutions because the probability of creating a 1 or 0 on each locus is equal. Without loss of generality, a binary-encoded solution is sampled from a probability vector $p(t)$. At iteration t, a population $S(t)$ of n individuals are

sampled from the probability vector $p(t)$. The samples are evaluated and an interim population $D(t)$ is formed by selecting μ ($\mu < n$) best individuals. Then the probability vector is updated by extracting statistics information from $D(t)$ as such: the probability for a particular position is the sum of all values on that position for all chromosomes, divided by the number of chromosomes. After the new probabilities are computed, we sample this distribution and create a new, better fitted, population.

The main advantage of UMDA is that the algorithm converges on a global optimum in a much smaller number of generations than all other evolutionary algorithms. This is because each new population's average fitness is better than the previous ones.

V.2 Population-based incremental learning

PBIL employs the same idea as UMDA, but it uses a different population scheme, meaning that, at each iteration, a number of individuals are sampled instead of storing an entire population. Another difference is that a learning rate is used (a small value is preferred, such as 0.1).

The difference is that there is less memory used by using only a attribute vector from which we can produce candidate solutions. It is preferred that genetic operations are applied on this vector, rather than the solutions.

The PBIL algorithm is as follows:

1. Generate a population from the probability vector.
2. Evaluate the fitness of each chromosome.
3. Based on the most promising chromosomes, update probability vector.
4. Apply genetic operator.
5. Repeat steps 1 to 4

VI Detailed description of the algorithms

All of the algorithms were written in the C++ programming language. No external libraries were used.

For all algorithms, a **chromosome** (individual of the population) is represented by a *struct* data type, which contains a *string* of bits and a fitness value with *double* precision. Also, the **fitness** function is the computation of the value resulted from an individual and the mathematical formula. For each implementation, some problems have special particularities. For example, all of the mathematical functions (DeJong, Rastrigin etc.) require a number of variables and a precision, an interval for the variables, with which the length of a single variable is computed using the formula : $n = \lceil log_2((u - l) * 10^p) \rceil$. After this, the length of a chromosome is the number of variables multiplied by n. Also, for the fitness, the chromosome has to be decoded using: $\mathrm{l} + \mathrm{decimal(bits)} * \frac{(\mathrm{u}-\mathrm{l})}{(2^n-1)}$.

VI.1 Classic genetic algorithm

Firstly, the **selection** is done using the *Roulette Wheel* approach and the genetic operators applied on chromosomes are **mutation** (Bit Flip) and **crossover** (One Point). As for the parameters, there are 3 fixed ones (MAX_RUNS = no. of times the main algorithm is run; MAX_ITER = max generations permitted per run; POP_SIZE = population size) and 2 adaptive ones, crossover and mutation rates.

The adaptiveness of the rates consist of the usage of a formula for each chromosome: $\frac{(\mathrm{population_{best}} - \mathrm{current_{fitness}})}{(\mathrm{population_{best}} - \mathrm{population_{avg}})} * \mathrm{k}$, where k is 1.0 for crossover and 0.5 for mutation. For crossover, $\mathrm{current_{fitness}}$ is the worst fitness of the 2 parents. This formula is only applied if the fitness of the individual we want to operate on is better that the average because the formula will give it a lower rate. On the else branch, the given rate is k, such that the individual is exploited more (it is less promising than the others).

Following the initializations, the main algorithms is run MAX_RUNS times and the steps are:

1. Initialize a random population and assign their fitness values;
2. Create a temporary population by selecting 2 parents at a time, applying crossover on them, resulting in 2 children, which are mutated (stops when we reached POP_SIZE chromosomes);
3. Replace main population with the temporary one and stop the run after MAX_ITER generations.

VI.2 Hill Climbing

We have here only 2 parameters, which are fixed, MAX_RUNS and MAX_ITER (same meaning as for the GA). The implementation for this algorithm is simple:

1. A single chromosome is created randomly and is given its fitness value;
2. A single neighbor is chosen by flipping a random bit and is saved if it's more promising;
3. Stop after MAX_ITER generations

VI.3 Simulated Annealing

This algorithm required a few more parameters, initial/final temperatures (T0 / Tn) and a number of cycles. In a way, SA is similar to HC. The starting chromosome is random and for a no. of steps we choose a neighbor by flipping a random bit and saving it if its fitness is better. The difference, though, is with the additional probability of accepting the neighbor: $e^{(-1 * \frac{|\text{neighbor.fitness} - \text{current.fitness}|}{t})}$, where t is the temperature. The annealing of the temperature is done with a formula

based on the 3 parameters described above and the current cycle: $\mathrm{Tn} + (\mathrm{T0} - \mathrm{Tn}) * (\frac{\mathrm{cycles} - t}{\mathrm{cycles}})^2$.

VI.4 Univariate Marginal Distribution Algorithm

Besides POP_SIZE, UMDA also has a SELECT_SIZE, which represents how many chromosomes will be chosen for computing the probabilities for the next distribution. It is similar in structure to the classic GA, but the difference lies in generating the next population. In GA, we use operators to do modify chromosomes generated with a normal distribution (0.5 chance for each bit), but UMDA's whole purpose is to change that distribution with each step, leading to better solutions in much less time. Having said that, after generating an initial population with the normal distribution, the individuals are sorted by fitness (best to worst) and SELECT_SIZE are chosen. All of these chromosomes contribute to the computing of the appearance probability of the bit 1 in each position (no. of 1's in that position divided by the SELECT_SIZE). Following this is the generation of the new population using the new distribution.

VII Experimental results

In any algorithm, the main part was run 4 times, and 100 generations were allowed per run. Where it is necessary, the population size is 120.

VII.1 De Jong

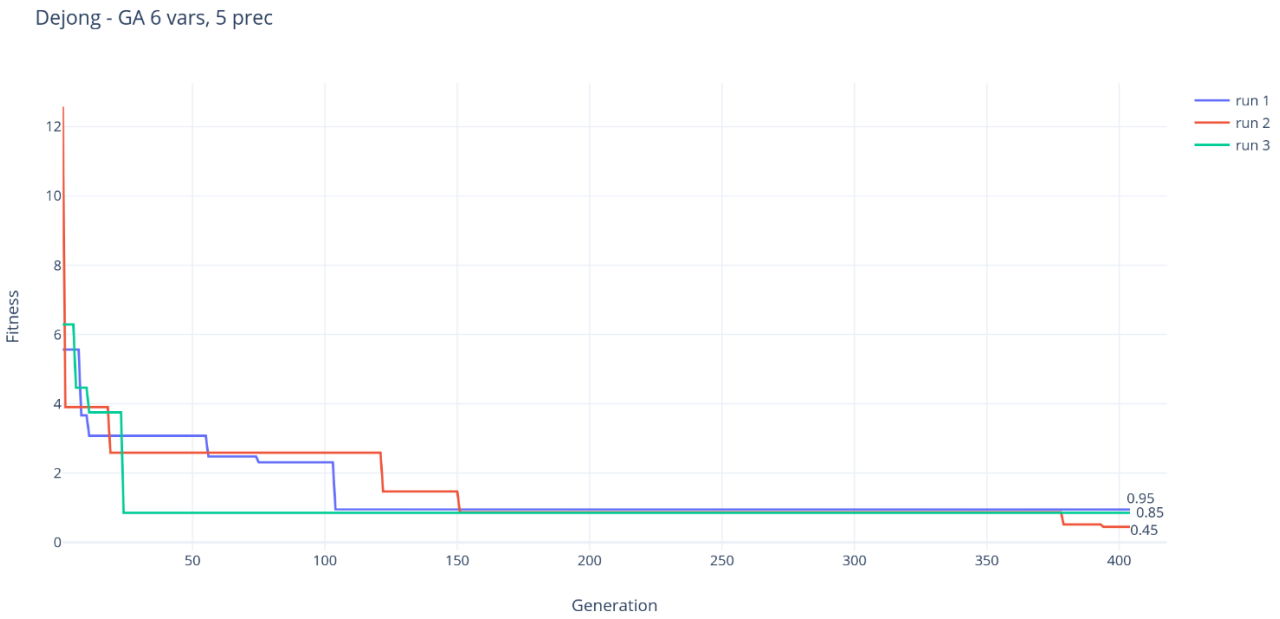
De Jong's function (sphere model) is continuous, convex and unimodal. Definition of function:

$$f(x) = \sum_{i=1}^{N} x_i^2, \qquad -5.12 \leq x_i \leq 5.12$$

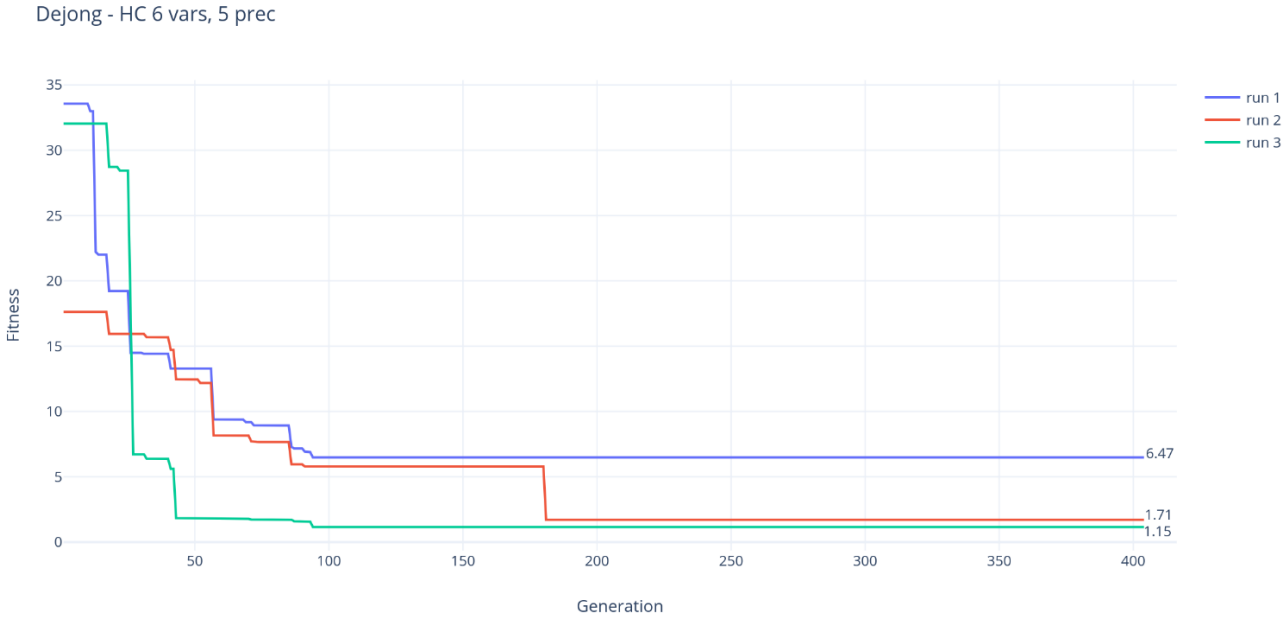
Global minimum: $f(x) = 0,\ x_i = 0\ ,\ i = 1{:}N$

For every variation, $N = 6$:

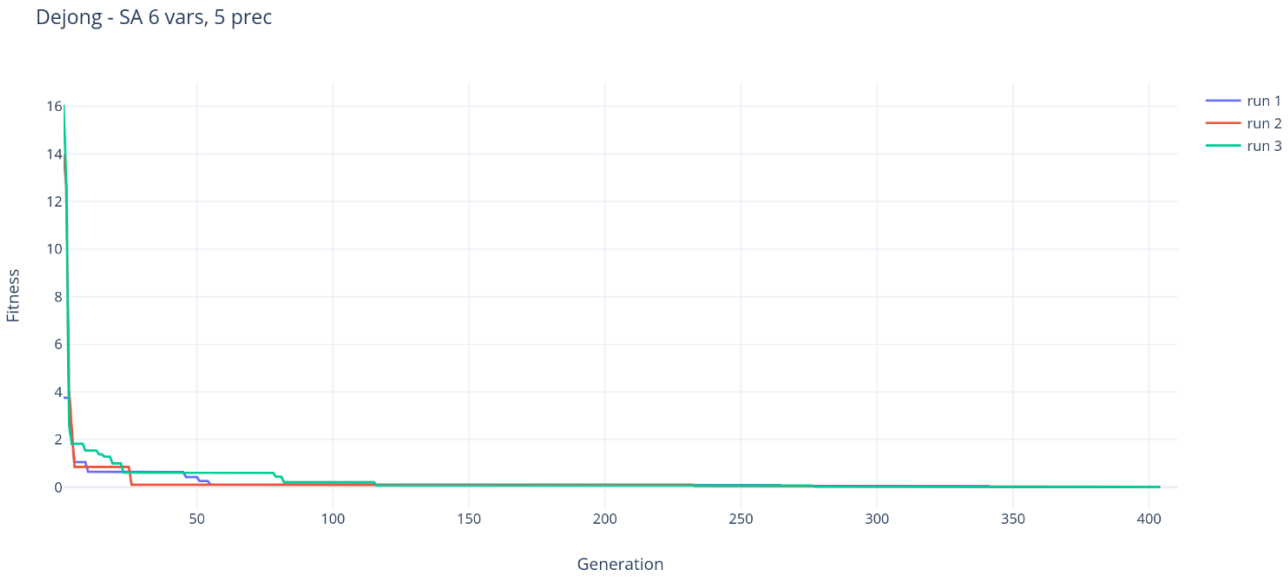
- Genetic Algorithm



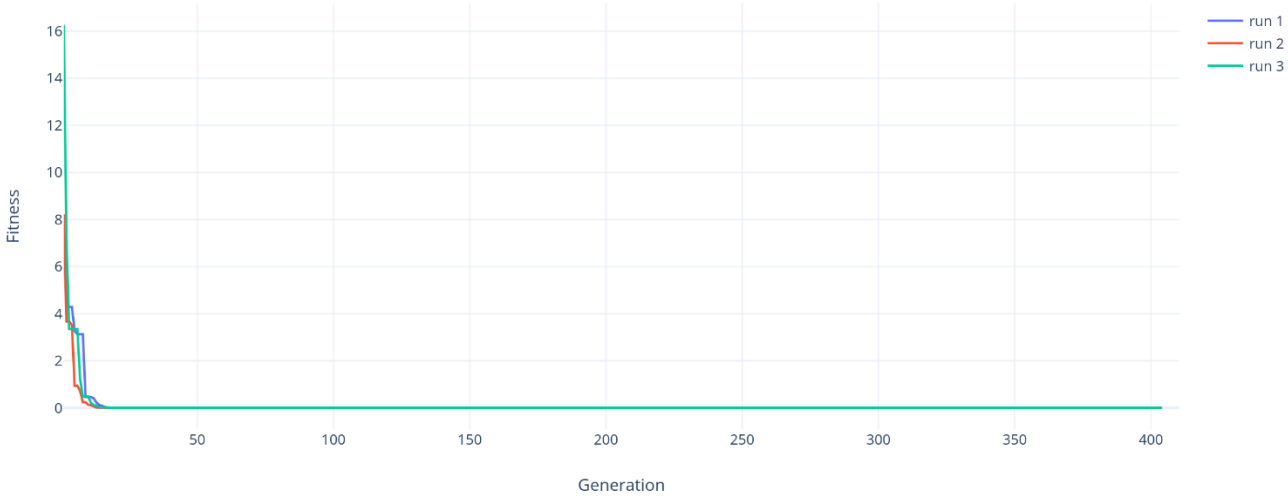
- Hill Climbing



- Simulated Annealing



- UMDA



VII.2 HalfMin

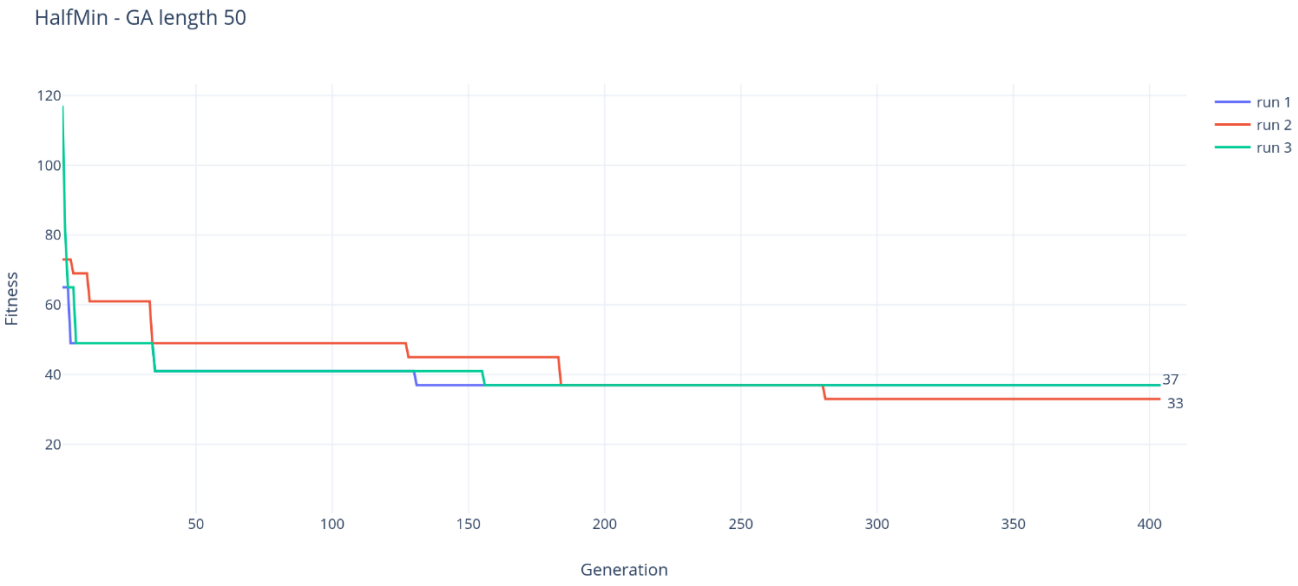
Seemingly inexistent in any document I searched in, this function asks for N (preferably even) x_i bits and which are transformed into '-1' or '1' values, using the formula: $y_i = 2 * x_i - 1$. With these y_i values, the following function has to be minimized:

$$f(x) = \sum_{i=1}^{N} \left(\sum_{j=1}^{N-i} y_i * y_{i+j} \right)^2$$

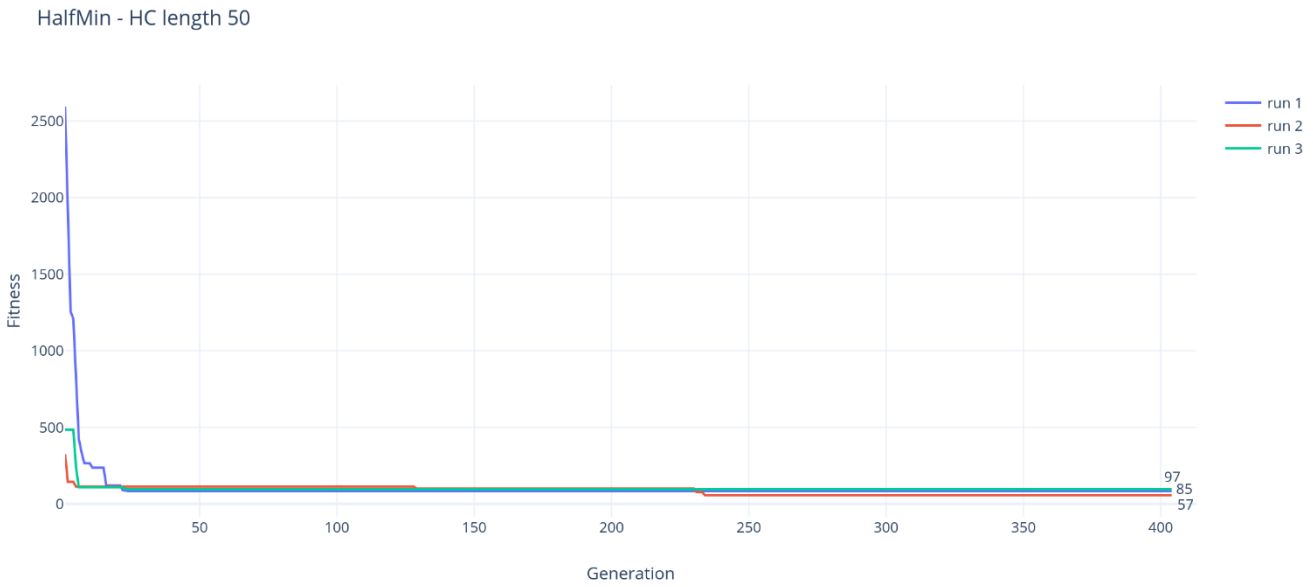
Global minimum: $f(x) = \frac{\mathrm{N}}{2}$, $\mathrm{x_i} \neq x_{i+1}, i = 1{:}\,N-1$ (hence the name, HalfMin), the corresponding bit string being an alternating series of 1's and 0's.

For every variation, $N = 50$:

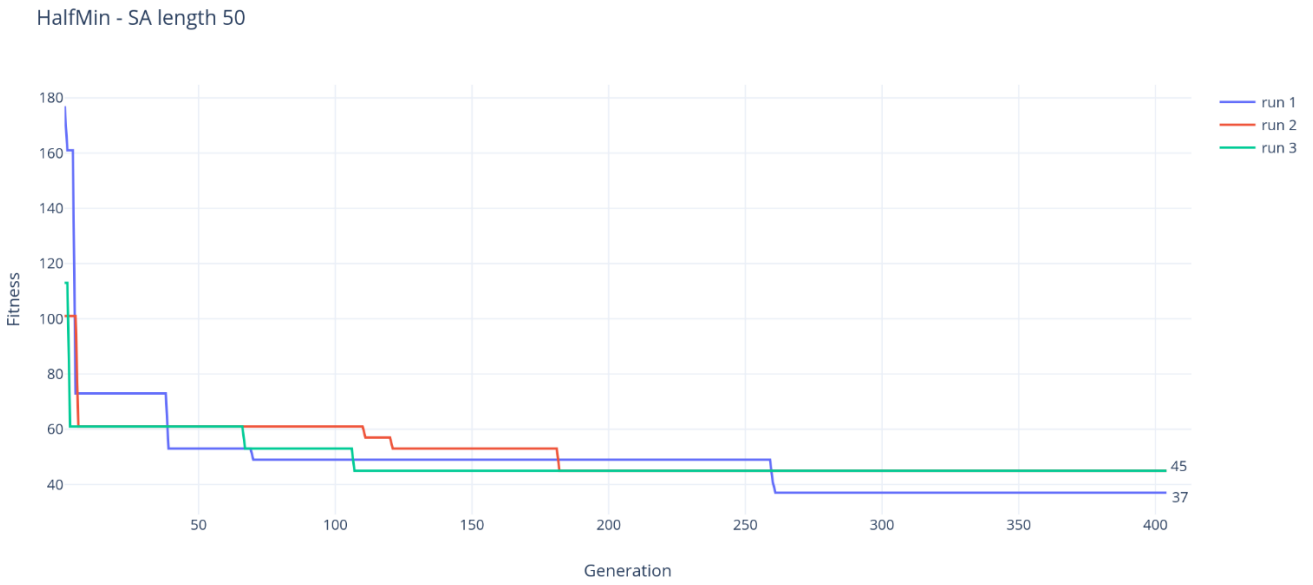
- Genetic Algorithm



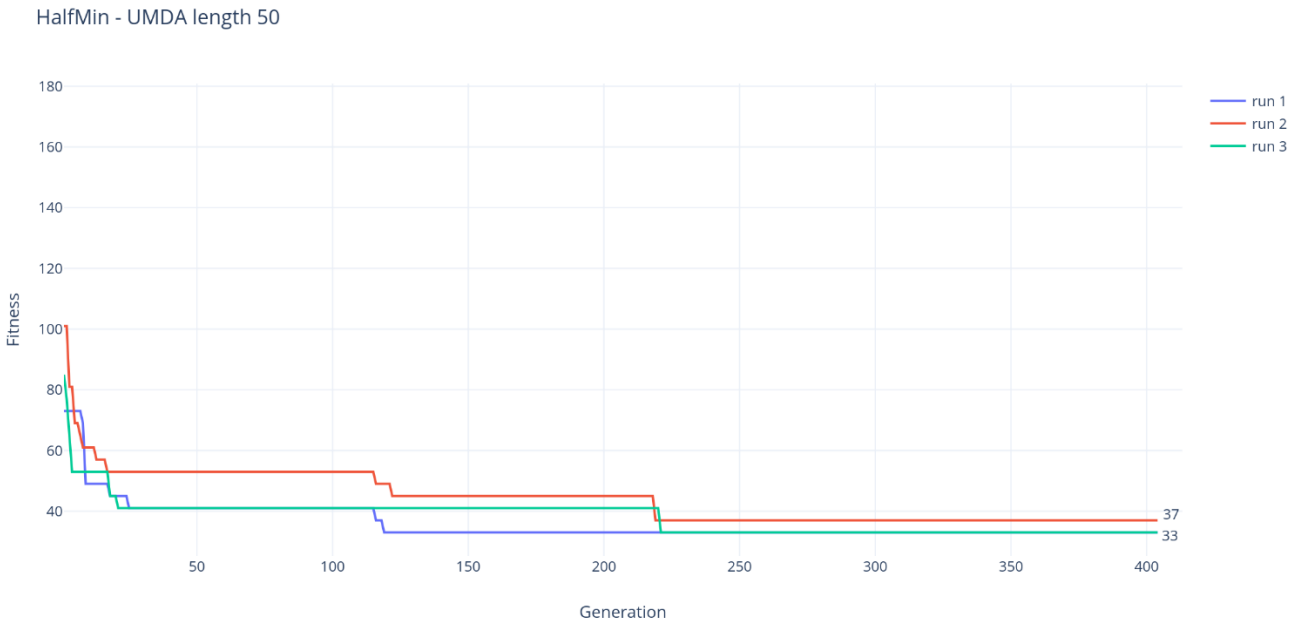
- Hill Climbing



- Simulated Annealing



- UMDA



VII.3 Rastrigin

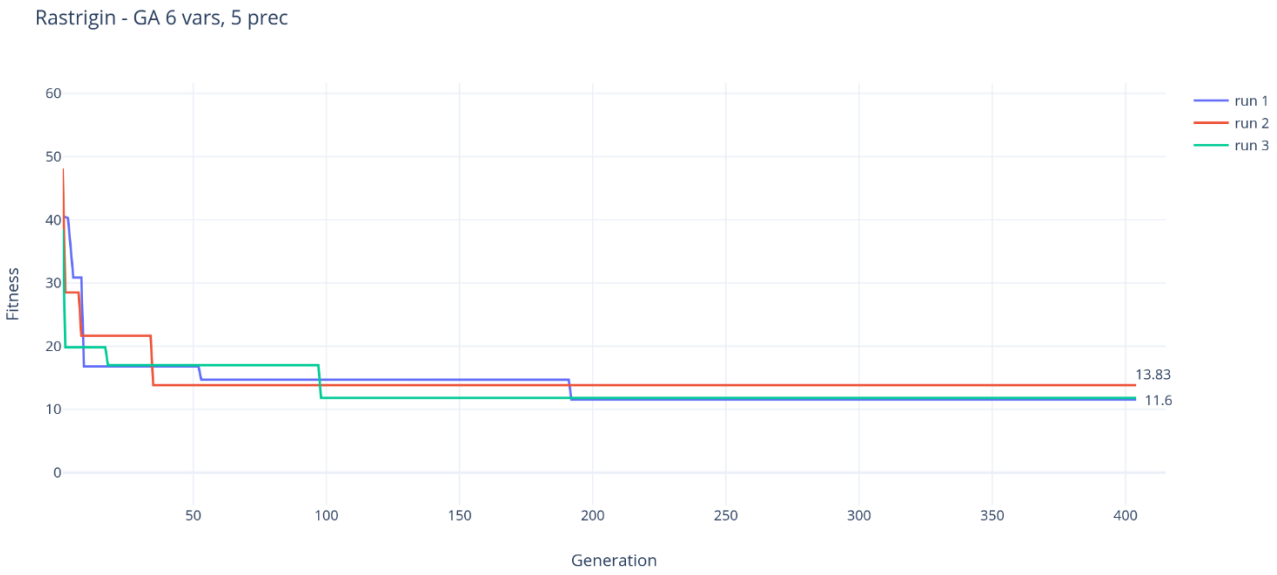
Rastrigin's function is based on the De Jong function, but it adds cosine modulation to produce more local minima. Even if the function is highly multimodal, the location of the minima are regularly distributed. Definition of function:

$$f(x) = 10 * n + \sum_{i=1}^{N} (x_i^2 - 10 * \cos(2 * \pi * x_i)), \qquad -5.12 \leq x_i \leq 5.12$$

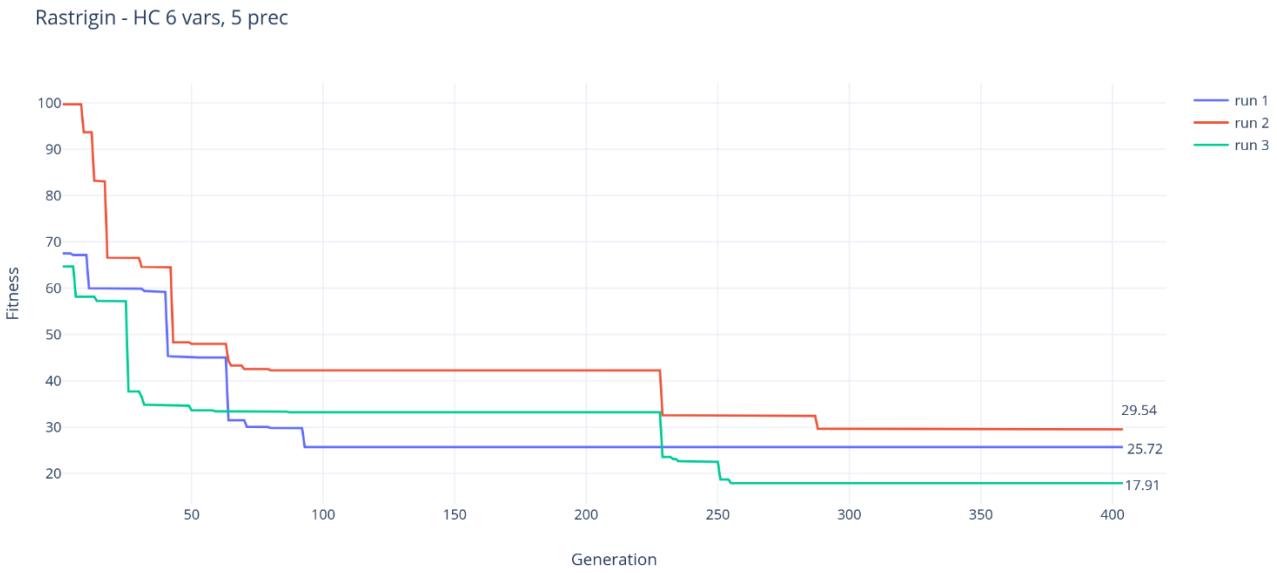
Global minimum: $f(x) = 0,\ x_i = 0\ ,\ i = 1{:}\,N$

For every variation, $N = 6$:

- Genetic Algorithm

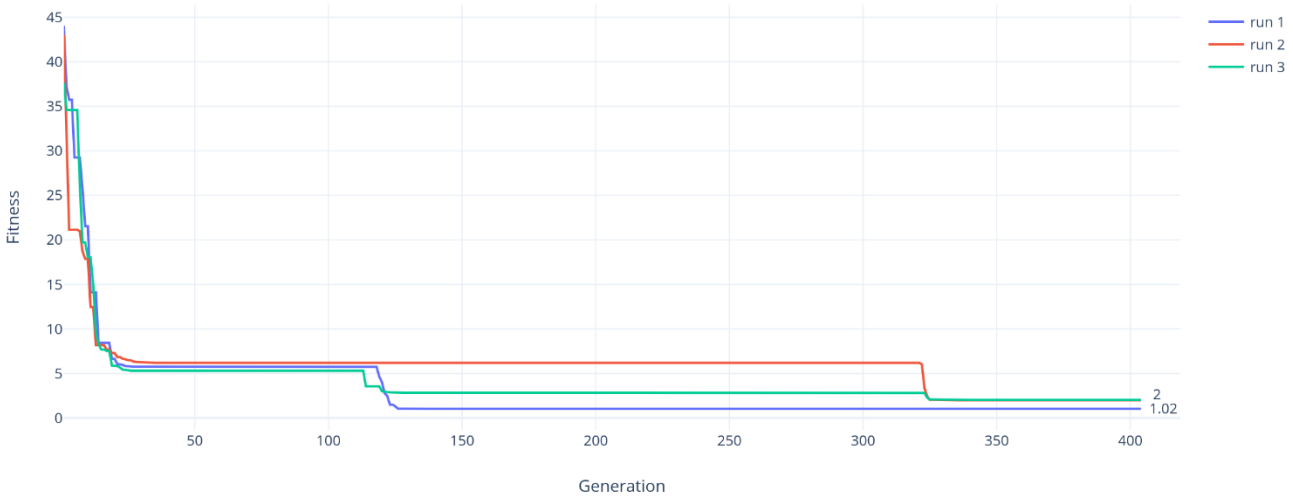


- Hill Climbing



- Simulated Annealing

- UMDA



VII.4 Rosenbrock

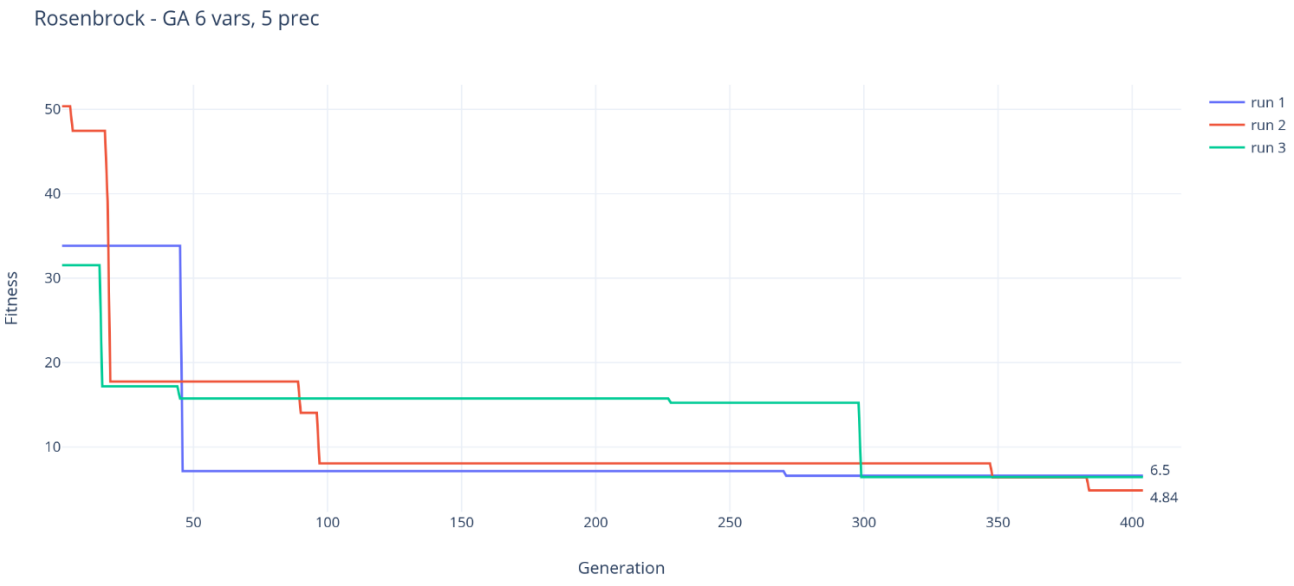
Rosenbrock's valley (commonly known as Banana function) has its global optimum is inside a parabolic shaped flat valley. Finding this valley is easy, but convergence to the global optimum is hard and so this problem has been used extensively to assess the performance of optimization algorithms. Definition of function:

$$f(x) = \sum_{i=1}^{N-1} [100 * (x_{i+1} - x_i^2)^2 + (1 - x_i)^2], \qquad -2.048 \leq x_i \leq 2.048$$

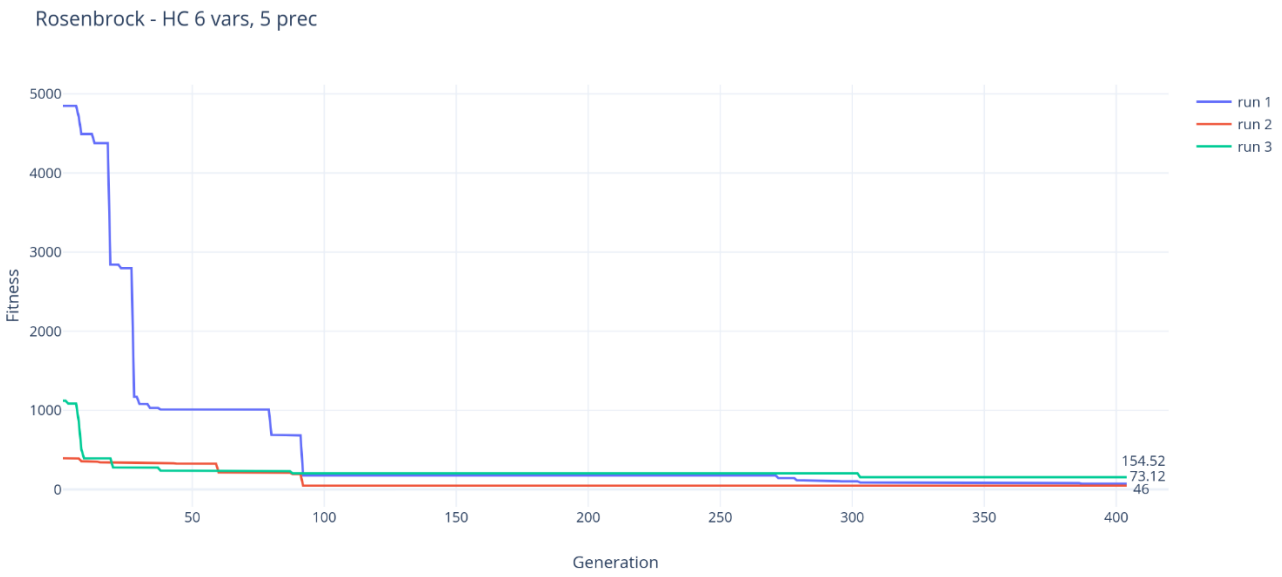
Global minimum: $f(x) = 0,\ x_i = 1\ ,\ i = 1{:}N$

For every variation, $N = 6$:

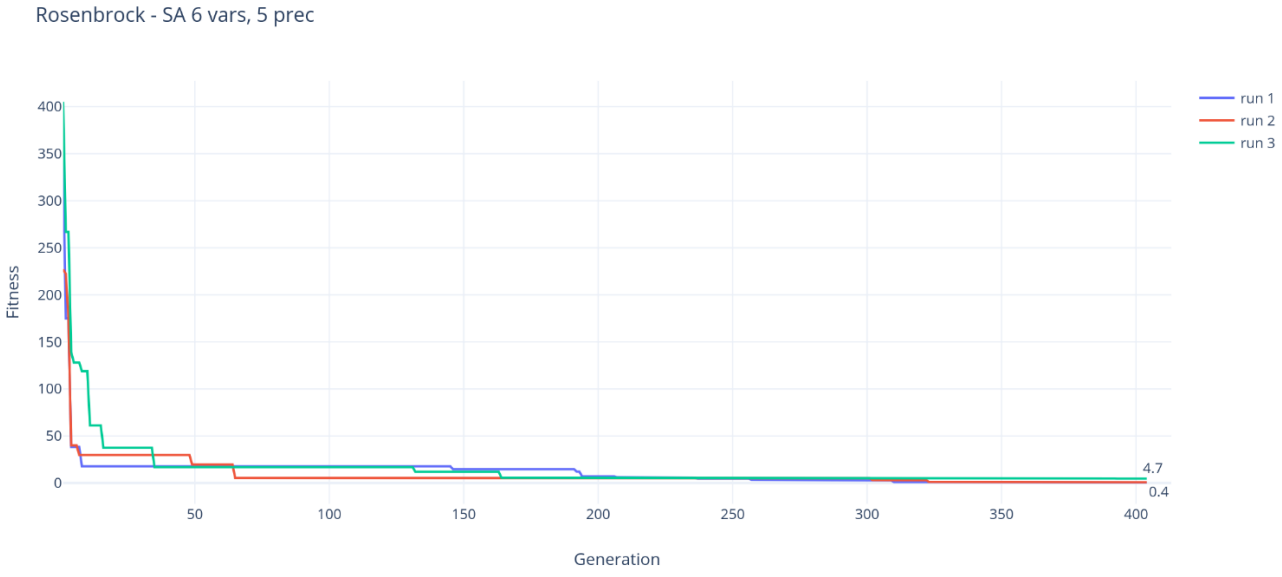
- Genetic Algorithm



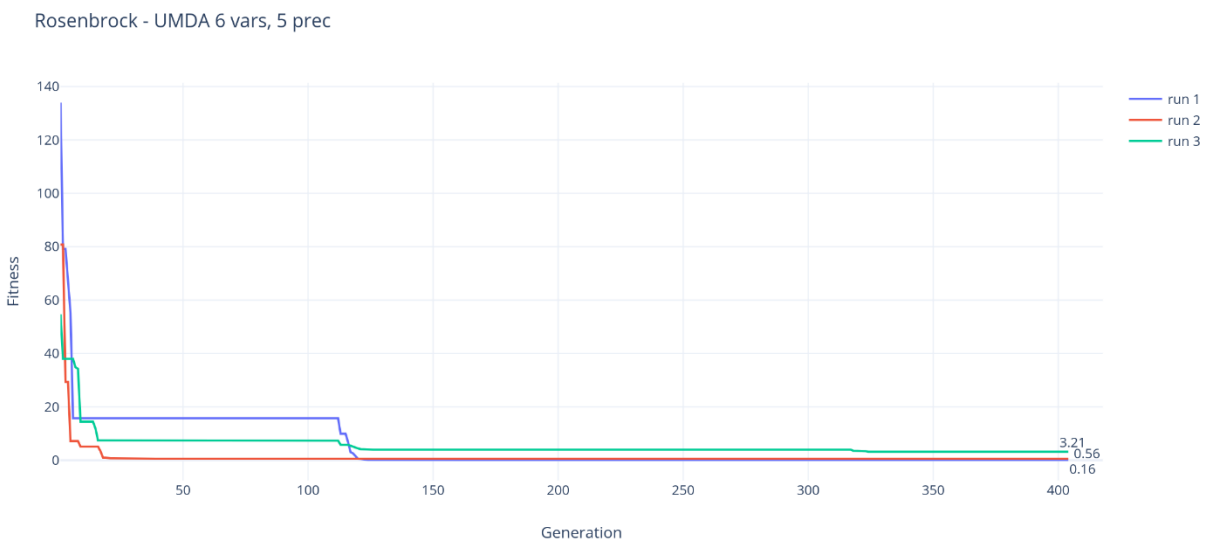
- Hill Climbing



- Simulated Annealing



- UMDA



VII.5 Royal Roads R1

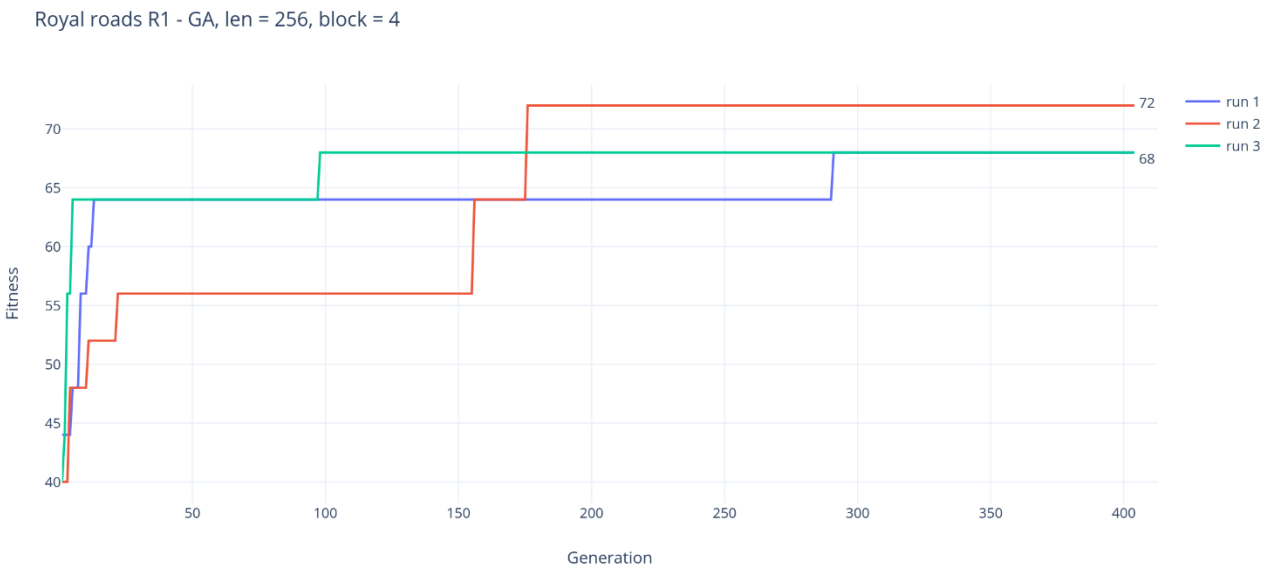
This maximizable function asks for a length N of the bit string and block size M. It's preferable that we choose the length as a power of 2 (ex: 128, 256, 512 etc.). The block size can be any even value, but we choose only values that are powers of 2 (2, 4, 8 etc.) because this way we can assure that the bit string is evenly divided in N/M equal parts.

Definition of function: $F(x) = \sum_{s \in S} c_s * \sigma_s(x)$, where s is a schema, x is a bit string, $c_s = order(s)$, and $\sigma_s(x) = 1$ if x is an instance of s, 0 otherwise. So, basically, we sum M whenever we find a bit string that matches the current schema.

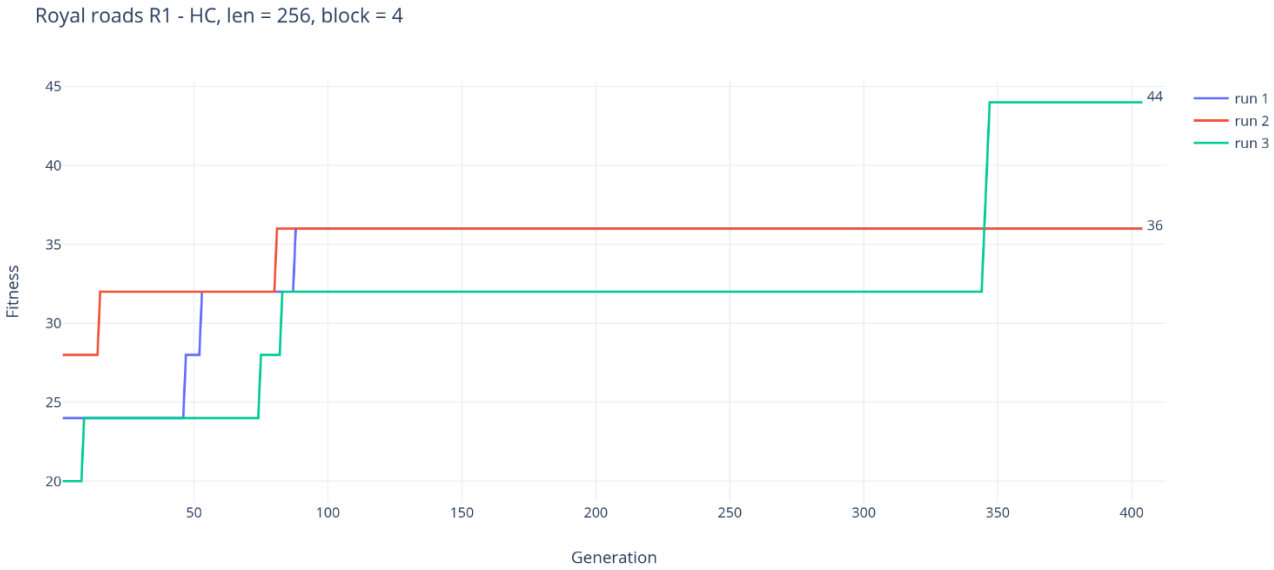
Global maximum: $F(x) = N,\ x_i = 1,\ i = 1{:}N$. The optimal bit string is 'all ones', meaning '11111…'.

For every variation, $N = 256,\ M = 4$:

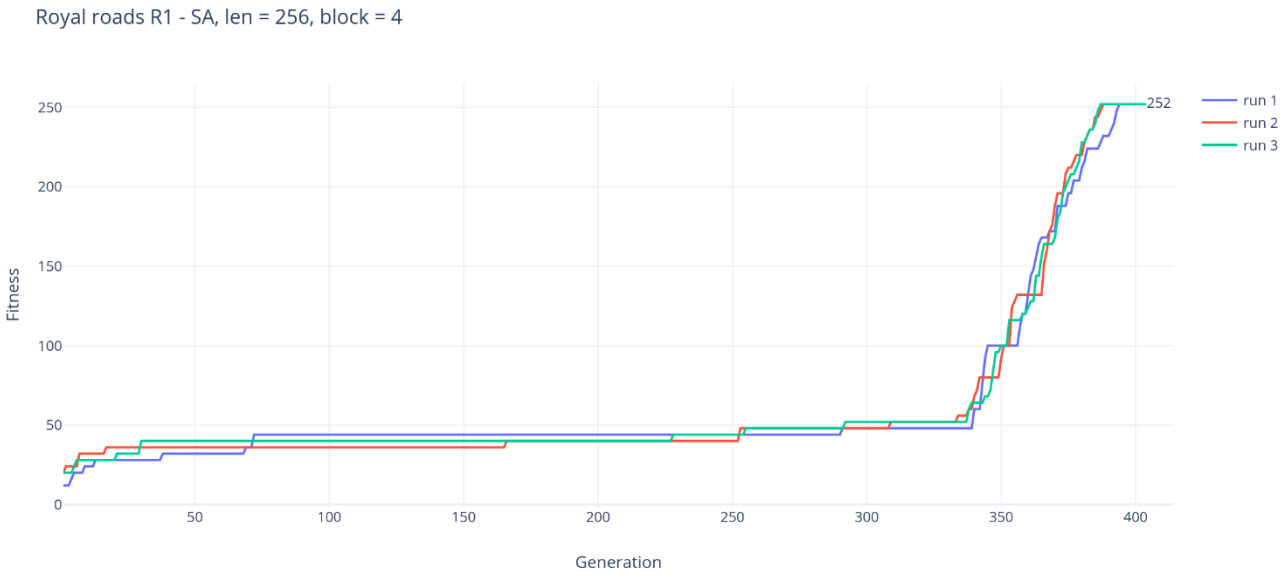
- Genetic Algorithm



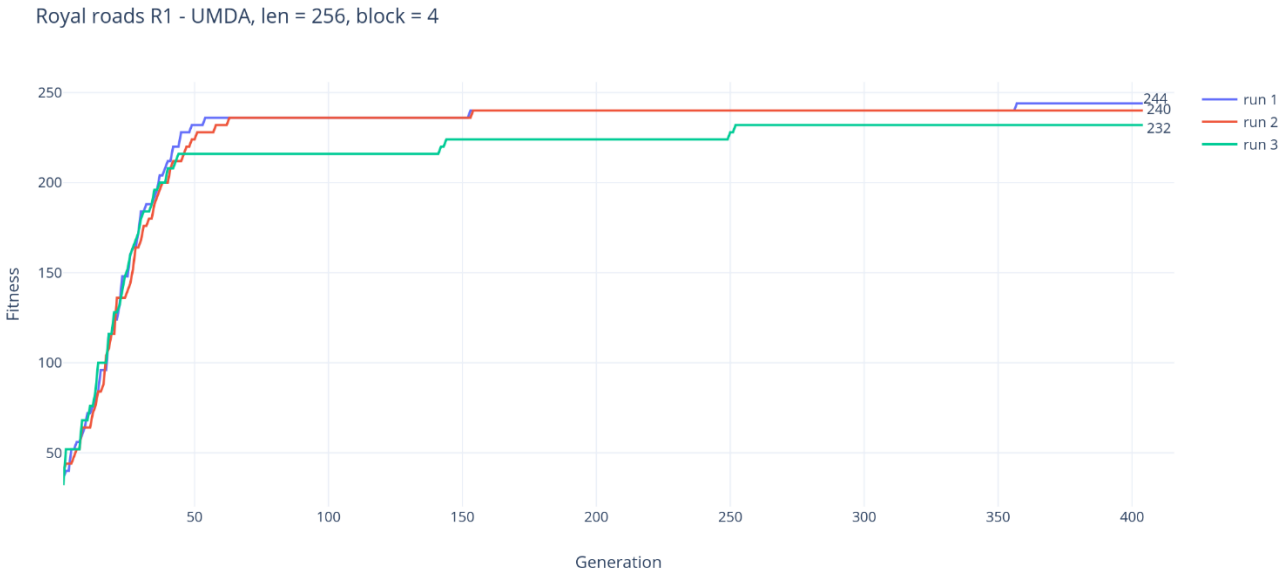
- Hill Climbing



- Simulated Annealing



- UMDA



VII.6 Six-Hump Camel Back

The 2-D Six-hump camel back function is a global optimization test function. It has two global minima out of the six local ones. Definition of function:

$$f(x_1, x_2) = \left(4 - 2.1 * x_1^2 + x_1^{\frac{4}{3}}\right) * x_1^2 + x_1 * x_2 + (-4 + 4 * x_2^2) * x_2^2,$$

$$-3 \leq x_1 \leq 3, -2 \leq x_2 \leq 2$$

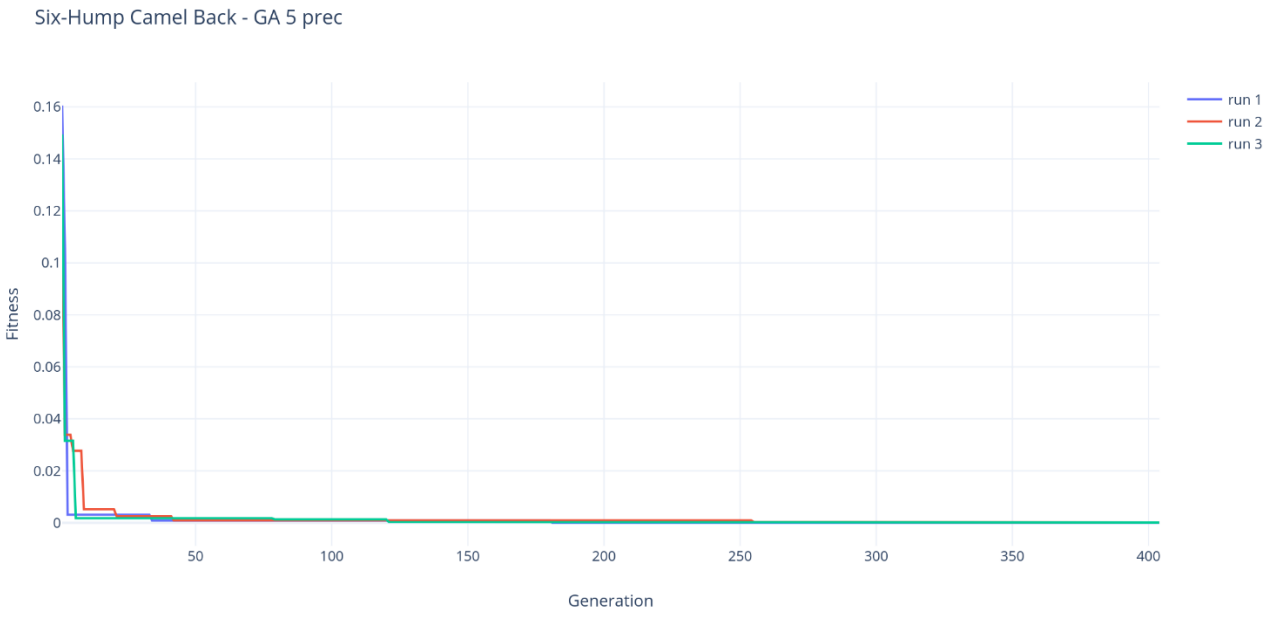
Global minimum:

$$f(x1, x2) = -1.0316;\ (x1, x2) = (-0.0898, 0.7126), (0.0898, -0.7126).$$

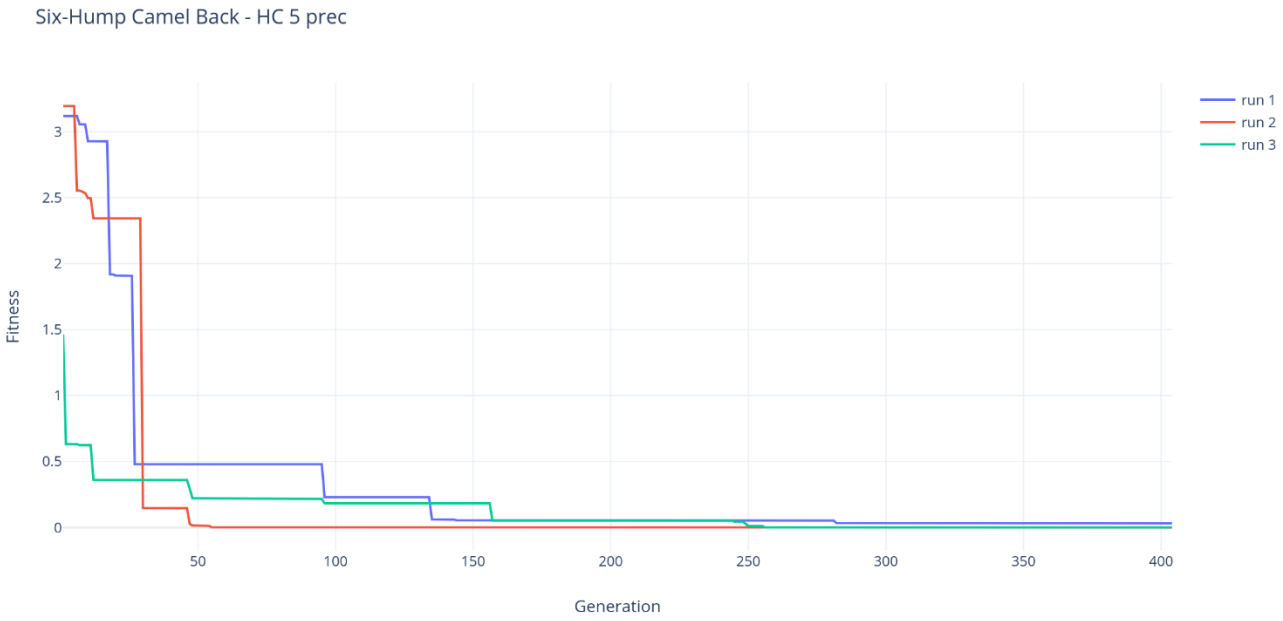
I added 1.0316 to the result of the fitness function to compare the final best found minimum to 0.

For every variation, the implicit number of variables is 2:

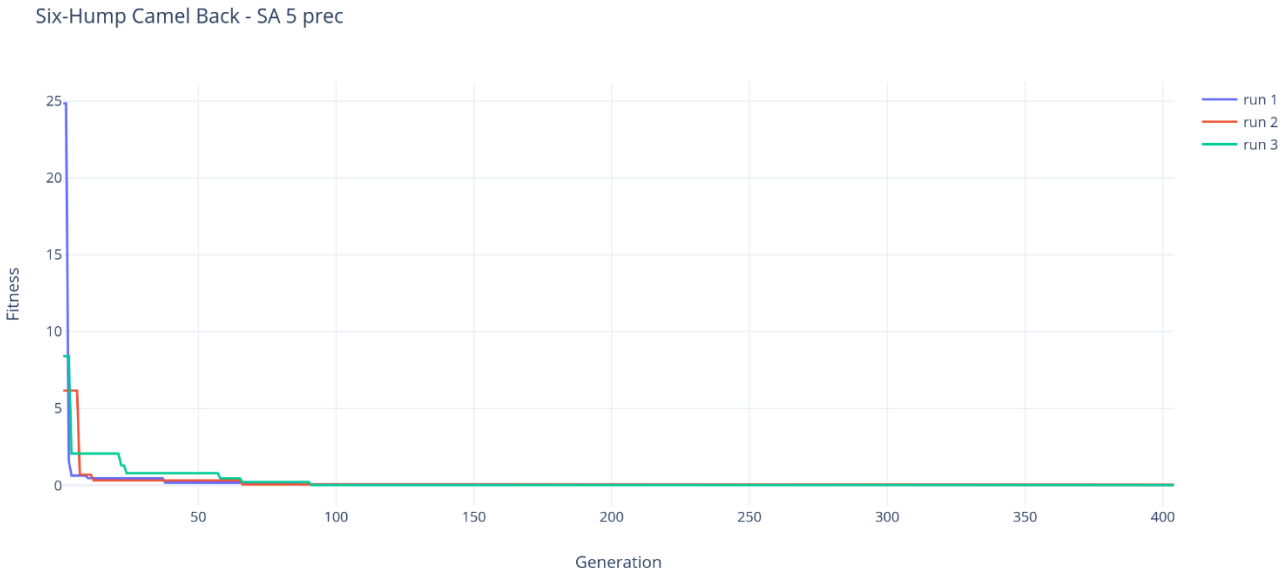
- Genetic Algorithm



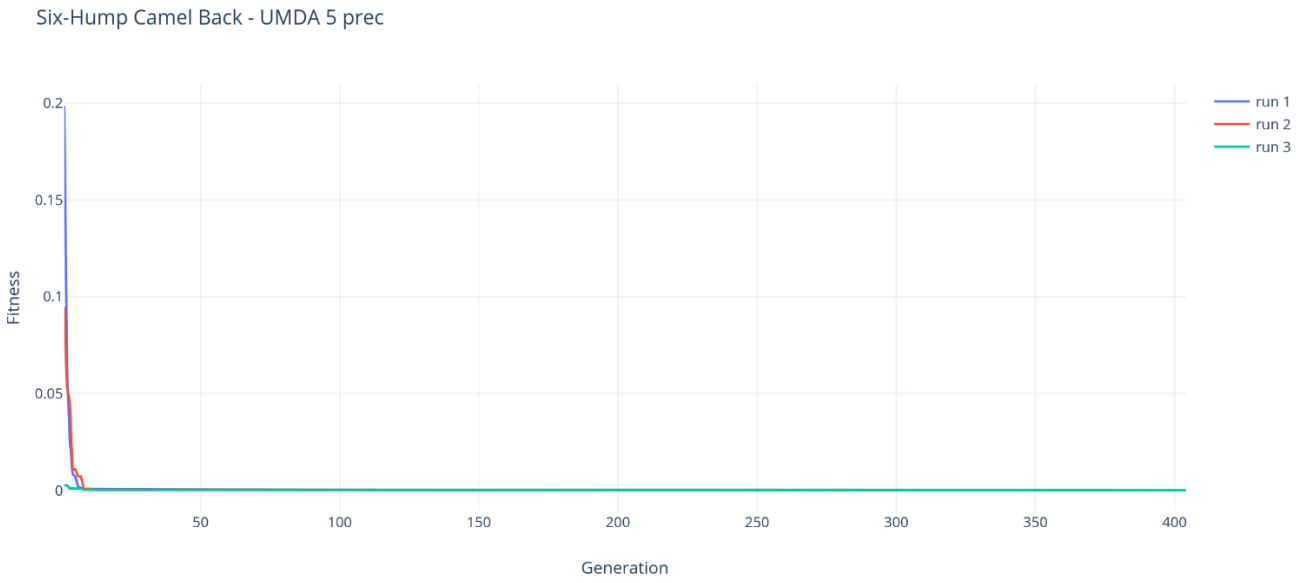
- Hill Climbing



- Simulated Annealing



- UMDA



VIII Final Conclusions

As we can see in the experimental results, the EDA algorithm, UMDA, either outright performs better than the rest or the same as another algorithm, but in fewer iterations, which means a faster convergence. From this we can draw a simple conclusion: EDAs can be used in any situation another algorithm would be used, the losses in performance being close to 0. More so, the implementation is relatively easy and doesn't imply (in the univariate model) any complicated theory or code technique. To achieve a similar performance as EDA with other types of algorithms, we would have to adapt the parameters and optimize the code, and that could prove to become difficult. In EDAs, very few parameters are used, so the performance does not depend on them.

VIII.1 Personal contributions

The bulk of the work comes from designing the algorithms and adapting each one to every single optimization function. The implemented algorithms were: a classic genetic algorithm (GA), hill climbing (HC), simulated annealing (SA) and

univariate marginal distribution algorithm (UMDA). I researched adaptive parameters to make the GA and SA perform closer to optimal (to not be influenced by pre-defined parameters) and applied that to my code. Finally, the experimental results plots were also compiled by me.

VIII.2 Future work

EDAs can definitely be improved upon by using a more complex dependencies model which would put us in the multivariate category. Also, we could improve the adaptation of parameters and techniques in the other algorithms to achieve an even closer, more precise comparison. More problems could be introduced to diversify the experiments and, with more time, bigger dimensions for the problems could be used to see how the algorithms do in those situations.

IX Bibliography

Back, T. (1996). *Evolutionary algorithms in theory and practice.* Oxford University Press.

Bengoetxea, E. (2002). Estimation of Distribution Algorithms. In *PhD Thesis.*

Breaban, M. (n.d.). Retrieved from https://profs.info.uaic.ro/~pmihaela/GA/

Estimation of distribution algorithm - Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Estimation_of_distribution_algorithm

Genetic Algorithms Tutorial. (n.d.). Retrieved from https://www.tutorialspoint.com/genetic_algorithms/index.htm

Simionescu, P., Beale, D., & Dozier, G. (2004). Constrained Optimization Problem Solving Using.

Simulated Annealing - Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Simulated_annealing