Computing The Global Minimum of Benchmark Functions Using Genetic Algorithms

Fiodorov Cristian and Neagu Alexandru

November 2022

Abstract

In this paper, we describe the efficiency of a simple Genetic Algorithm **(GA)** in the context of function optimization. The algorithm will approximate the global minimum for Rastrigin's, Michailewicz's, DeJong's 1, and Schwefel's functions. By using meta-algorithms, gray codes, elitism, and adaptive parameters we will provide an Optimized Genetic Algorithm **(GA OPT)** that will generate far better approximations.
We compare the performance of both versions of the Genetic Algorithm with that of traditional Iterative Hill Climbing Algorithms that are using Best Improvement **(HCBI)**, First Improvement **(HCFI)**, and the Simulated Annealing heuristic **(SA)**. Our experiments demonstrate that Genetic Algorithms perform faster and give more accurate results than a traditional Iterative Hill Climbing Algorithm.

Contents

1 Introduction

One of the most important combinatorial optimization problems is computing the optimal global solutions for multivariable functions. There are several non-deterministic algorithms that are able to compute reasonably good approximations. The Iterative Hill Climbing Algorithm, optimized with Simulated Annealing, is a popular approach for this problem. The bitstring encoding of cartesian points turns out to be an excellent idea when working with a big number of dimensions. Unfortunately, the Iterative Hill Climbing algorithm is not perfect and generates the solution at a very slow pace. Also, for some multimodal functions, this algorithm has a very slow probability to find the global optimal solution. Thus, a faster algorithm that explores the space better is required. For this purpose, a Genetic Algorithm turns out to be the algorithm that meets these criteria.

This paper presents how a **Genetic Algorithm** approximates an optimal solution for the above-mentioned combinatorial optimization problem. Also, in order to strengthen our motivation for using a Genetic Algorithm, we will compare the generated results to those generated by a traditional Hill Climbing Algorithm.

The paper is organized as follows.

Section 2 describes both the benchmark functions used in the problem and the implementation of the Genetic Algorithm

Section 3 describes the parameters used by the Genetic Algorithm

Section 4 compares the Genetic Algorithm with the Iterative Hill Climbing Algorithm by displaying the computed solutions

Section 5 Presents conclusions

1.1 Motivation

Genetic Algorithms are faster and more efficient compared to other traditional methods. Also, they are suited for parallel implementation, optimizing both continuous and discrete functions and delivering good results fast. There are several NP-hard problems that can't be solved in polynomial time complexity using a deterministic algorithm. Unfortunately, even the most efficient computing systems can't solve these problems in a reasonable amount of time. In this scenario, Genetic Algorithms turned out to be a powerful tool for providing good enough approximations in a short period of time.

The main advantage of a Genetic Algorithm is the fact that the search is not biased toward a locally optimal solution. A genetic algorithm uses the **mutation** operator to avoid the convergence to suboptimal solutions. The **crossover** operator is used in order to explore a multitude of regions at a fast enough pace. These features make the Genetic Algorithm to be an obvious choice over most gradient descent algorithms.

1.2 Problem Description

In order to test the genetic algorithm, we need some good, representative benchmark functions. In this paper, we will compute the **global minimum** value of 4 well-known functions: Rastrigin's, Michalewicz's, De Jong's 1, and Schwefel's. We will further describe these functions in the Methodology section. These functions are multi-dimensional, but we will explicitly test the 5, 10, and 30-dimension versions to see how the results change if the number of dimensions is increased.

2 Methodology

2.1 Benchmark Functions

Rastrigin's Function

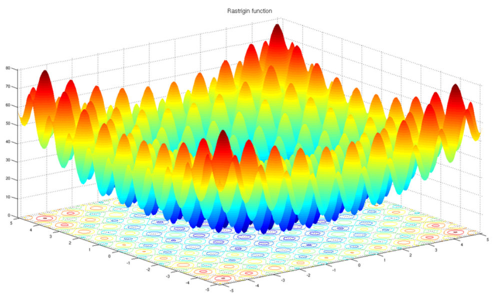


Figure 1: Rastrigin's Function

Definition

$$\mathbf{f}(\mathbf{x_1}, \cdots, \mathbf{x_n}) = \mathbf{10n} + \sum_{\mathbf{i=1}}^{\mathbf{n}} \left(\mathbf{x_i^2} - \mathbf{10}\cos\left(\mathbf{2}\pi\mathbf{x_i}\right)\right)$$

Search Domain $-\mathbf{5.12} \leq \mathbf{x_i} \leq \mathbf{5.12}, \forall \mathbf{i} = \mathbf{1}, \mathbf{2}, \cdots, \mathbf{n}$

(a) **The Global Minima** $\mathbf{x}^* = (\mathbf{0}, \cdots, \mathbf{0}) \mid \mathbf{f}(\mathbf{x}^*) = \mathbf{0}$

Michalewicz's Function

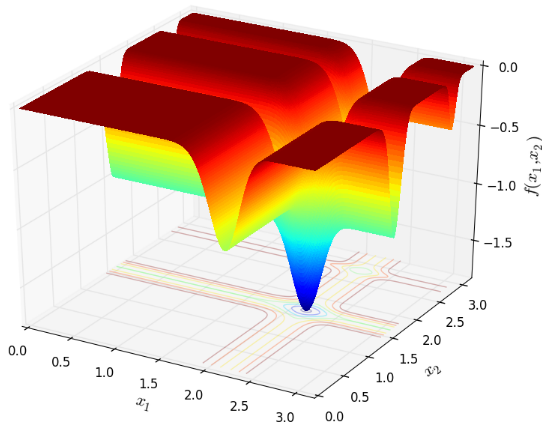


Figure 2: Michalewicz's Function

Definition

$$\mathbf{f}(\mathbf{x_1}, \cdots, \mathbf{x_n}) = -\sum_{\mathbf{i=1}}^{\mathbf{n}} \sin(\mathbf{x_i}) \cdot \sin^{\mathbf{20}}\left(\mathbf{i} \cdot \frac{\mathbf{x_i^2}}{\pi}\right)$$

Search Domain $\mathbf{0} \leq \mathbf{x_i} \leq \pi, \forall \mathbf{i} = \mathbf{1}, \mathbf{2}, \cdots, \mathbf{n}$

The Global Minima
at $\mathbf{n} = \mathbf{5}$, $\mathbf{f}(\mathbf{x}^*) = -\mathbf{4.687658}$
at $\mathbf{n} = \mathbf{10}$, $\mathbf{f}(\mathbf{x}^*) = -\mathbf{9.66015}$
at $\mathbf{n} = \mathbf{30}$, $\mathbf{f}(\mathbf{x}^*) = -\mathbf{29.63088}$

DeJong's 1 Function

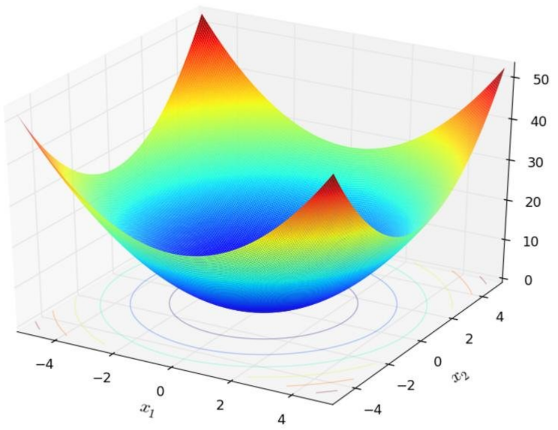


Figure 3: DeJongs's 1 Function

Definition

$$\mathbf{f(x_1, \cdots, x_n) = \sum_{i=1}^{n} x_i^2}$$

Search Domain $\mathbf{-5.12 \leq x_i \leq 5.12, \forall i = 1, 2, \cdots, n}$

The Global Minima $\mathbf{x^* = (0, \cdots, 0) \mid f(x^*) = 0}$

Schwefel's function

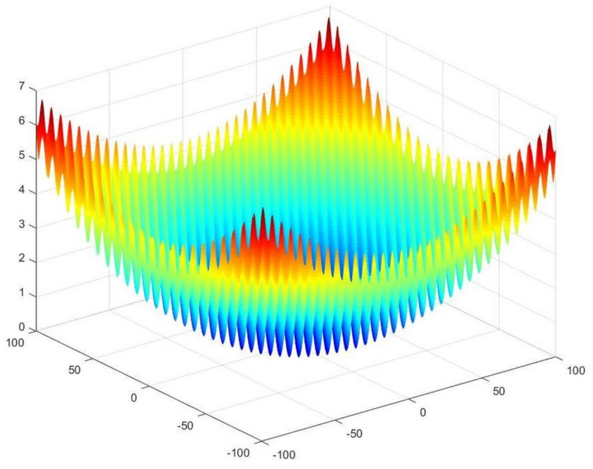


Figure 4: Schwefel's function

Definition

$$\mathbf{f(x_1, \cdots, x_n) = 418.9829n - \sum_{i=1}^{n} x_i \sin(\sqrt{|x_i|})}$$

Search Domain $\mathbf{-500 \leq x_i \leq 500, \forall i = 1, 2, \cdots, n}$

The Global Minima
$\mathbf{x^* = (420.9687, \cdots, 420.9687) \mid f(x^*) = 0}$

2.2 Algorithm

General Description

A traditional Genetic Algorithm is usually characterized by the following components

- A feasible encoding method of a candidate solution called **chromosome** (individual)
  - This encoding mechanism is determined by the programmer. Usually, an array of bits is preferred as the solution representation. If we think of a chromosome as an array, every element of this array is called a **gene**, and its corresponding position is called **locus**
- A population of encoded solutions (chromosomes) that evolve through a multitude of **generations**
  - The generations might be viewed as **iterations** made by the Genetic Algorithm
- A **fitness function** that evaluates how good (fit) a candidate solution is
  - The fitness function takes a candidate solution (individual) as the input and outputs a real number called fitness value. Genetic Algorithms are considered "maximization" algorithms, so the optimal solutions are considered those with the **biggest** fitness values. In this paper, we want to compute the global minimum of a benchmark function (let's call this function E). In such a case where minimization is required, we should think of a fitness function F that is based on the initial function and respects the following property for all pairs of candidate solutions

$$E(x) < E(y) \Rightarrow F(x) > F(y)$$

- A rigorous **selection** mechanism that implements the concept of "survival of the fittest"
  - In other words, the selection procedure generates a new population based on the existing one
  - Each chromosome is selected in the new population according to a fixed probability which is **directly proportional** to its fitness value
  - A good selection mechanism isn't always biased toward the fittest individuals. Each individual has a chance (even a very small one) to be selected by the algorithm
- Specialized genetic operators such as **mutation** and **crossover** that alter the existing population and generate fitter individuals
  - These operators shouldn't be applied every time. Instead, a **mutation probability** and a **crossover probability** should be chosen carefully. The bigger these probabilities are, the more often the corresponding genetic operator will be applied

Implementation

The Genetic Algorithm designed to optimize the benchmark functions described in this paper consists of the following steps

1. Generate a starting population **P[0]** at random
2. Repeat the following steps **G** times where G is a fixed number of generations
   3. Use the selection mechanism on the previous generation **P[i - 1]** to generate a new population **P[i]**
   4. Apply the mutation operator on each individual from population **P[i]**, respecting the mutation probability
   5. Apply the crossover operator on each individual from population **P[i]**, respecting the crossover probability
6. The final solution is the **fittest** individual from the last population **P[G]**

Solution Representation

The Genetic Algorithm implemented in our paper uses arrays of bits called **chromosomes** to encode the candidate solutions. To calculate the length required by the array of bits we use the domain function $[a, b]$, the precision we discretize the searching domain 10^{-d}, and the number of dimensions the function has (that we will note with D).

The domain function interval $[a, b]$ is split in $N = (b - a) \cdot 10^d$ values. In order to represent those values we need $L = \lceil \log_2 N \rceil$ bits. Therefore, the length of the array of bits that is used to represent a candidate solution is $L \cdot D$.

At the moment we want to evaluate the solution it is necessary to decode this array of bits (chromosomes) that represents the real values for all the function parameters. Let's consider Dec - the decimal representation of the array of bits c for one function parameter (this is not the final representation).

$$Dec = \sum_{i=0}^{L-1} 2^i * c_i$$

If we want to compute the real representation R, the following formula will decode the bits into a real value that belongs to the function domain: $R = a + \frac{Dec \cdot (b - a)}{2^L - 1}$

Another method to represent candidate solutions is by using gray codes, discusses in the **Gray Coding** subsection 2.3

Initial Population

The initial population is an array of N randomly generated individuals, where each individual is represented as an array of bits. When we decode an individual we will get an array with D real numbers in the interval $[a, b]$, where D is the number of dimensions the function has. The size of the population will always remain the same ($N = 200$ in our case), even if the individuals from the population will change.

Fitness Function

The fitness function is used to measure the quality of a chromosome (individual) and is designed based on the numerical optimization function.
The value for our fitness function is calculated after the formula

$$F_i = \frac{MAX - E_i}{MAX - MIN + C} + 0.01,\ 0 \leq i \leq N - 1$$

where E_i is the result for the numerical optimization function for the chromosome i, MAX and MIN are the maximum and minimum value of all the individuals from the population, and C is a small constant (in our case is equal with 0.000001) that will exclude the case of zero division when $MAX = MIN$. In the end, we add 0.01 in order to give some chances, for even the individual that has $E_i = MAX$, to be selected in the next generation.
The higher the F_i value is, the better the individual i is, with a better chance to be selected in the next generation.

Selection

The selection mechanism is the most complex part of our algorithm. There are several selection methods that might be used for an optimization problem. In this paper, the **wheel of fortune** method is preferred.

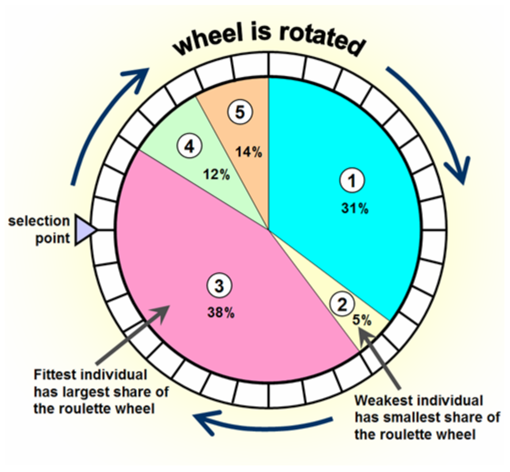

wheel is rotated
selection point
5
14%
4
12%
1
31%
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

Figure 5: Roulette wheel selection

Given a population P, the algorithm selects N individuals and adds them to a new population. The individual P_i from population P is selected according to a probability p_i that is directly proportional to its fitness value F_i. The algorithm might choose the **same** individual multiple times. These probabilities are computed in the following way

- Let **T** be the sum of all the fitness values in the population

$$T = \sum_{i=0}^{N-1} F_i$$

- We compute the probability of selection for each individual using the following formula

$$p_i = \frac{F_i}{T},\ 0 \leq i \leq N-1$$

  We can see that for each individual i, $p_i \in (0, 1]$ and p_i is directly proportional to F_i

- In order to simulate the selection at each step, with respect to the probability computed for each individual, we should aggregate the array p in the following way

$$q_0 = 0 \wedge q_i = q_{i-1} + p_i,\ 1 \leq i \leq N$$

(*) We generate a random number r and check every two consecutive values q_i and q_{i+1} from the aggregated array q if they satisfy the following condition

$$q_i < r \leq q_{i+1}$$

When an index i that satisfies the above condition is met, the individual P_i is added to the new population, and step (*) is repeated until we reach the desired number of individuals N added to the new population.

Mutation

The mutation operator is implemented in a simple way. We iterate through the array of bits, that is the representation of an individual, and flip a bit from 1 to 0 or vice versa with a specific probability. We generate a random number and if that number is below that probability we flip it and if not - we do not.

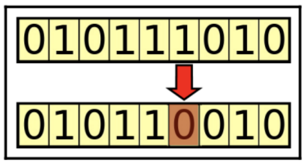


Figure 6: Mutation Example

Crossover

The version we have chosen for the crossover operator is the single-point crossover. This type of crossover is implemented in the following way: at the begging, we have two "*parents*" and we generate a random number between 1 and $Lenght(parent) - 2$ which will be the cutting point. From these two parents, we will generate two new descendants. The first one will have the bits of the first parent until the cutting point and the bits from the second parent after the cutting point, and the second - vice versa.

Before crossing individuals, we have to choose which of them will undergo crossing and we have to make the pairs. As with mutation, we have a certain probability for crossover. We generate a random number $n \in [0, 1]$ for each chromosome, which is their probability to cross, and sort them in ascending order with respect to that number. Afterward, we loop through them and get two individuals at each iteration. If their crossing probabilities are below the global crossing probability, they will generate two new descendants that will be added to the new population. If we have a pair that contains an individual that has the crossing probability below the global probability and the other above, they will have a 50% probability to cross. If both individuals from the pair have the crossing probability below the global probability they will not cross.

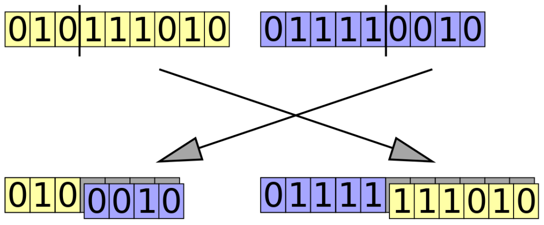


Figure 7: Crossover Example

Termination Condition

The termination condition is the condition that must be reached in order to stop generating a new population and return the best individual from the last generation. Common termination conditions are:

- Fixed number of generations reached (This is the termination condition that we used in our algorithm, and we set the max number of generations equal to 2000);
- A individual is found that returns a solution that satisfies minimum criteria set by us;
- The best individual has reached a plateau such that successive iterations no longer can produce a better result;
- Manual inspection;
- Combination of all described above.

2.3 Optimizations

Meta-optimization

A Genetic Algorithm has several parameters that govern its behavior and efficiency in optimizing a given problem. These parameters must be chosen by a specialist to achieve satisfactory results. Selecting the parameters by hand is a laborious task for a person that does not know what makes the optimizer perform well. For this kind of task, the meta-genetic algorithm was invented.

In our case, we used the meta-genetic algorithm to select the best mutation and crossover probabilities from a range imposed by us. The range for mutation probability was from 0.00001 to 0.01 and the range for crossover probability was from 0.5 to 1.0.

The meta-genetic algorithm has the same structure as a simple genetic algorithm.

In the beginning, we initialize our population with some random individuals (each individual is represented as an array of bits, so our population is an array of arrays of bits). After, we evaluate the population. This means that we start a new genetic algorithm for each set of parameters encoded in each individual from the population. The results from those algorithms are stored in an array of doubles. Afterward, we enter a while loop that has the number of generations that we want to be evaluated as the termination condition. We select the individuals for the new generation based on how they performed during the evaluation. After, we mutate and cross the population based on the same algorithms as at a simple Genetic Algorithm, and, at the end of the while loop, we evaluate the new population. While we have not reached the termination condition we undergo the same steps as before. When we exit the loop we select the mutation and crossover probability that performed the best and return them.

Elitism

Elitism is an additional schema that can be implemented with any mechanism of selection. The scope of elitism is to move the best k individuals from a generation to the next generation. Without elitism, a fit individual has the chance to disappear because it was not selected to survive or it was destroyed while applying the mutation and crossover operators.

We implemented the elitism in the following way: before selecting the individuals for the next generation, we order the individuals descendingly after their fitness values and get the first 5% of them and move to the next generation.

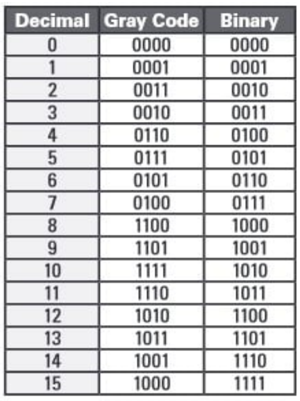
Gray Coding

As we discussed in the **Solution Representation** subsection 2.2, an individual (chromosome) is represented by an array of bits of size $L \cdot dim$. Each argument of the multidimensional function has its real value encoded in a subarray c of length L. When decoding the array of bits c, we used the following formula to calculate the corresponding decimal value

$$Dec = \sum_{i=0}^{L-1} 2^i * c_i$$

In this representation, we consider that the array of bits c is a binary representation of a decimal number Dec. Instead of computing Dec using the above-mentioned formula in a binary system, we may try to consider c as the **gray code representation** of a decimal number.

The gray code is another popular numeral system where two successive values differ in **only one** bit. Thus, a decimal number Dec might have its binary representation different from its gray code representation as we can see in the image below



Decimal	Gray Code	Binary
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Figure 8: Comparation of Binary and Gray Code representations

Obviously we can't use the same formula to compute the corresponding decimal value for a gray representation. Let g be an array of bits of length L that represents an argument from our multidimensional function encoded in the gray system. As we already saw above, we should find the corresponding decimal value Dec encoded by the array of bits b. For the gray system, the decoding algorithm is based on the following steps

- Transform g into an array of bits c that corresponds to the binary system and encodes the same decimal number as g. This conversion is done using the following formulae

$$
\begin{aligned}
&c_L = g_L \\
&c_{L-1} = g_{L-1} \oplus c_L = g_{L-1} \oplus g_L \\
&c_{L-2} = g_{L-2} \oplus c_{L-1} = g_{L-2} \oplus g_{L-1} \oplus g_L \\
&. \\
&. \\
&c_1 = g_1 \oplus c_2 = g_1 \oplus g_2 \oplus \cdots \oplus g_L
\end{aligned}
$$

- Apply the same formula presented at the beginning of this subsection to decode an array of bits from the binary system to a decimal number

Adaptive Parameters

As stated in the **General Description** subsection 2.2, when we use the **mutation** genetic operator, we should choose a constant probability at the beginning of our algorithm as our mutation probability. Each time we iterate through the individuals of a population, we negate each bit according to the mutation probability as described in the **Mutation** subsection 2.2.

Unfortunately, using the same mutation probability everywhere might disadvantage the performance of the algorithm. Sometimes, we need a larger mutation probability for better and faster exploration of different regions. However, there are situations when exploration should be neglected and further exploitation of the current region should be promoted. In these cases, lower mutation probabilities are needed. Thus, we can try to make the mutation probability **adaptive** for a better balance between **exploration** and **exploitation**.

We compute a different mutation probability for each individual from the current population P in the following manner

Let F_{max} be the maximum fitness value from the current population

Let $\overline{F}$ be the average fitness value based on each individual from the current population

For each individual P_i we compute its mutation probability pm_i using the following formula

$$
pm_i = \begin{cases} k_1 \cdot \dfrac{F_{max} - F_i}{F_{max} - \overline{F}}, & F_i \geq \overline{F} \\ \\ k_2, & F_i < \overline{F} \end{cases} \tag{1}
$$

The constant k_2 is a big mutation probability we set for individuals with below-average fitness values. The constant k_1 directly controls how big our adaptive mutation probability might be.

We may observe that in case of $F_i = F_{max}$ the mutation probability pm_i will be equal to 0. In case this happens, we should assign pm_i to a small enough probability (we used 0.0008).

3 Experimental Setup

Each of the four functions described above (including 5, 10, and 30-dimensional variants of them) will be tested first with the genetic algorithm without any optimization and after with the one that has all the optimizations included. While testing each of the methods with any of the parameters and functions described, we will use the precision of 5 numbers after the dot.
The probabilities displayed in the tables below were chosen with the help of the Meta Genetic Algorithm that had the Mutation Probability set to 0.0001, and the Crossover Probability set to 0.65, which are some well-known parameters that perform well together.
We ran each set of parameters, which were chosen by Meta Genetic Algorithm, 50 times, in order to collect relevant data to make an average value.

Genetic Algorithm without optimizations

Function	Mutation Probability	Crossover Probability
Rastrigin	0.000263900293255132	0.885984251968504
Michalewics	0.00109395894428153	0.871968503937008
De Jong 1	0.0000490615	0.696771653543307
Schwefel	0.0015724633431085	0.794881889763779

Table 1: GA without optimizations

Genetic Algorithm with optimizations

Genetic Algorithm with optimizations is the same basic algorithm that has some features implemented to make it perform better. Our includes Adaptive Parameters, Elitism and, for some functions, Gray Coding.

Function	K1 Probability	K2 Probability	Crossover Probability	Gray Encoding
Rastrigin	0.0058	0.1	1.0	+
Michalewics				-
De Jong 1				+
Schwefel				+

Table 2: GA with optimizations

4 Results

4.1 Rastrigin's Function

Dimensions	Algorithm	Average eval.	Eval. SD	Actual min	Average time (ms)	Time SD (ms)
	HCFI	0.862	0.541		8449.24	0.073
	HCBI	0.542	0.501		33394.88	0.07
5	SA	2.525	1.547		27425.88	0.124
	GA	0.94	0.99		3946.52	115.51
	GA OPT	0	0		4763.82	135.46
	HCFI	6.375	1.343		60084.3	0.211
	HCBI	4.302	0.927		272354.8(6)	0.175
10	SA	5.542	2.440	0	57603.57	0.156
	GA	6.28	2.88		6997.04	155.34
	GA OPT	0	0		9179.18	213.27
	HCFI	40.377	3.637		2803105.3	0.603
	HCBI	28.27	3.307		843354.9	0.575
30	SA	19.987	5.439		154159.4	0.233
	GA	35.71	8.81		23112.28	4383.77
	GA OPT	5.69	3.11		22782.7	423.4

Table 3: Rastrigin test results

4.2 Michailewicz's Function

Dimensions	Algorithm	Average eval.	Eval. SD	Actual min	Average time (ms)	Time SD (ms)
	HCFI	-4.683	0.006		7927.17	0.008
	HCBI	-4.686	0.0009		35723.66	0.003
5	SA	-4.578	0.1004	-4.687658	30582.9	0.031
	GA	-4.635	0.0591		3853.86	469.552
	GA OPT	-4.687	0.0013		4178.34	467.232
	HCFI	-9.2019	0.145		57181.9(3)	0.069
	HCBI	-9.4054	0.0825		285602.6	0.0524
10	SA	-9.2885	0.1953	-9.66015	64618.56	0.0441
	GA	-9.277	0.149		7345.0	938.512
	GA OPT	-9.523	0.099		8338.52	1196.218
	HCFI	-25.65	0.3394		2570176.2	0.1842
	HCBI	-26.829	0.1522		2860254.5	0.1233
30	SA	-28.2486	0.3376	-29.63088	175053.54	0.0581
	GA	-27.129	0.458		21353.6	2178.804
	GA OPT	-27.766	0.530		23146.12	3029.156

Table 4: Michalewics test results

4.3 DeJong's Function

Dimensions	Algorithm	Average eval.	Eval. SD	Actual min	Average time (ms)	Time SD (ms)
5	HCFI	0	0	0	6402.12	0
	HCBI				37879.04	
	SA				26723.34	
	GA	0	0		3761.06	119.57
	GA OPT				4736.18	151.94
10	HCFI	0	0		45670.7(6)	0
	HCBI				309311.1	
	SA				55497.19	
	GA	0	0		6895.88	1429.22
	GA OPT				9148.1	193.11
30	HCFI	0	0		2145668.5	0
	HCBI				1014093.8	
	SA				150090.86	
	GA	0.08462	0.03277		20716.68	4001.21
	GA OPT	0.0004	0.0002		24705.58	3838.66

Table 5: De Jong 1 test results

4.4 Schwefel's Function

Dimensions	Algorithm	Average eval.	Eval. SD	Actual min	Average time (ms)	Time SD (ms)
5	HCFI	19.40	16.29	0	10277.04	0.403
	HCBI	0.109	0.07		46673.99	0.027
	SA	6.089	19.82		27813.03	0.445
	GA	5.03	6.27		6239.7	804.94
	GA OPT	0	0		5930.34	203.67
10	HCFI	272.509	61.51		72377.4	1.431
	HCBI	130.75	49.12		379807.9	1.279
	SA	14.54	31.84		58885.5	0.564
	GA	33.30	34.53		9966.18	1861.91
	GA OPT	0	0		9339.42	128.33
30	HCFI	1774.13	154.55		3322499.7	3.931
	HCBI	1259.65	149.44		1666915.8	3.865
	SA	86.96	87.13		156960.71	0.933
	GA	426.76	106.88		26217.5	798.84
	GA OPT	73.75	81.09		34086.34	4500.27

Table 6: Schwefel test results

5 Conclusion

After analyzing the results generated by both variants of our genetic algorithm (with and without optimizations), we may draw the following conclusions

- A Genetic Algorithm doesn't operate with a single candidate solution at a time. Instead, it is using a population of **multiple** candidate solutions. Because of this feature, our Genetic Algorithm is way faster than the traditional Hill Climbing Algorithm (approximately 5 times faster in the worst case).
- For unimodal functions (eg DeJong's 1 function) a Genetic Algorithm generates slightly worse solutions than a Hill Climbing Algorithm.
- A genetic algorithm is able to obtain outstanding results for Rastrigin's and Schwefel's functions. As we can see in the results section, the optimized Genetic Algorithm generated the best results for these two particular functions. A powerful Simulated Annealing heuristic designed for a Hill Climbing Algorithm wasn't able to outperform the optimized Genetic Algorithm.
- For some special multimodal functions (eg. Michailewicz's function) our Genetic Algorithm wasn't able to generate the best approximations. However, it was still able to generate fairly good solutions **way faster** than the other algorithms.
- There are a lot of possibilities to implement a Genetic Algorithm. We could pick other selection methods, crossover probabilities, and genetic operators in order to generate better solutions for particular functions. Testing every combination of such functions and parameters is a hard and long process.
- By using a meta-algorithm we automated the process of picking the perfect combination of parameters, which later helped us to generate better solutions.

6 Bibliography

Croitoru Eugen
Laboratory page:
`https://profs.info.uaic.ro/~eugennc/teaching/ga/`

M. Srinivas, and L. M. Patnaik
"Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms"

Charlie Vanaret, Jean-Baptiste Gotteland, Nicolas Durand, Jean-Marc Alliot
"Certified Global Minima for a Benchmark of Difficult Optimization Problems"

Wikipedia
Meta-optimization:
`https://en.wikipedia.org/wiki/Meta-optimization`

Wikipedia
Genetic algorithm:
`https://en.wikipedia.org/wiki/Genetic_algorithm`

Rastrigin's Function:
`https://en.wikipedia.org/wiki/Rastrigin_function#/media/File:Rastrigin_function.png`

Michalewicz's Function:
`https://infinity77.net/global_optimization/test_functions_nd_M.html`

DeJong's 1 Function:
`https://xloptimizer.com/projects/toy-problems/sphere-function-pure-excel`

Schwefel's function:
https://www.researchgate.net/figure/Three-dimensional-image-of-Schwefel-function

Gray Code:
`https://cp-algorithms.com/algebra/gray-code.html`