Analysis of the Efficiency of Genetic Algortihms on Different Numerical Benchmark Functions

Harabagiu Ștefan, Teodorescu Călin

December 2022

Abstract

Minimizing a function can be done using different types of algorithms. This article compares two local-search algorithms (Simulated Annealing and Iterative Hill-Climbing) and a genetic algorithm on different benchmark numerical functions. Based on the obtained results, genetic algorithms seem to give better results than local-searches, but require specific optimization to be pushed even further. For the experiments, the search dimensions were 5, 10 and 30, while the starting population size and generations number were set to 200, respectively 2000.

1 Introduction

The aim of this report is to point out the differences in accuracy and efficiency between two local-search algorithms(Iterative Hill-Climbing and Simulated Annealing) and a genetic algorithm, based on finding the global minimum of four different functions.

1.1 Motivation

For some functions, finding the exact solution using a deterministic algorithm would be too costly in terms of duration. For these cases, using a genetic algorithm, in order to find an approximation to the solution would be more efficient.

1.2 Problem description

Minimizing a function is the process of finding a value within an allowed range of parameters for which the function has the lowest value possible.

2 Experiments

The experiment was tested on 4 functions:

1. De Jong's function 1
2. Schwefel's function
3. Rastrigin's function
4. Michalewicz's function

For all the functions, the algorithms were executed 30 times, with a precision of 5 decimals. The search dimensions were 5, 10 and 30.
The starting population size was set to 200, while the number of generations was 2000.

2.1 De Jong's function 1

$$\sum_{i=1}^{n} x_i^2 \qquad -5.12 \leq x_i \leq 5.12$$

Global minimum: $f(x) = 0, \quad x_i = 0, \quad i = 1 : n$

Size	Mean	SD	Min	Max
5	0	0	0	0
10	0	0	0	0
30	0.0064	0.	0	0.1435

Table 1: Genetic Algorithm

Size	Mean	SD	Min	Max
5	0	0	0	0
10	0	0	0	0
30	0	0	0	0

Table 2: Optimized Genetic Algorithm

2.2 Schwefel's function

$$\sum_{i=1}^{n} -x_i \cdot \sin\left(\sqrt{|x_i|}\right) \qquad -500 \leq x_i \leq 500$$

Global minimum: $f(x) = -n \cdot 418.9829, \quad x_i = 420.9687, \quad i = 1 : n$

Dimensions	Global Minimum
5	-2094.9145
10	-4189.8290
30	-12569.4870

Size	Mean	SD	Min	Max
5	-2094.815	0.08318	-2094.09	-2094.621
10	-4183.167	22.39287	-4189.714	-4070.870
30	-11618.52	388.7251	-12088.663	-10389.93

Table 3: Genetic Algorithm

Size	Mean	SD	Min	Max
5	-2094.914	0.07281757	-2094.9145	-2094.8353
10	-4189.527	0.13598492	-4189.7733	-4189.3056
30	-12299.92	242.60094	-12560.684	-11566.657

Table 4: Optimized Genetic Algorithm

2.3 Rastrigin's function

$$10 \cdot n + \sum_{i=1}^{n} \left(x_i^2 - 10 \cdot \cos\left(2 \cdot \pi \cdot x_i\right) \right) \qquad -5.12 \leq x_i \leq 5.12$$

Global minimum: $f(x) = 0, \quad x_i = 0, \quad i = 1 : n$

Size	Mean	SD	Min	Max
5	0.57672	0.76404	0	2.4716
10	2.58255	2.6397	0	12.216
30	26.009	7.3583	18.626	46.583

Table 5: Genetic Algorithm

Size	Mean	SD	Min	Max
5	0	0	0	0
10	0.980629	1.0845366	0	4.1432911
30	14.1555	6.2070611	7.1690325	35.073326

Table 6: Optimized Genetic Algorithm

2.4 Michalewicz's functions

$$-\sum_{i=1}^{n} \sin(x_i) \cdot \left(\sin\left(\frac{i \cdot x_i^2}{\pi} \right) \right)^{2 \cdot m} \qquad i = 1 : n, m = 10 \quad 0 \leq x_i \leq \pi$$

Dimensions	Global Minimum
5	-4.687
10	-9.660
30	-29.630

Size	Mean	SD	Min	Max
5	-4.6619	0.055576	-4.6877	-4.5376
10	-9.5068	0.16828	-9.6367	-8.9432
30	-27.273	0.721991	-28.561	-24.412

Table 7: Genetic Algorithm

Size	Mean	SD	Min	Max
5	-4.6875	0.00093733	-4.6876582	-4.6832540
10	-9.54521	0.14090694	-9.6597871	-9.2275067
30	-28.4807	0.64980132	-29.482166	-26.494752

Table 8: Optimized Genetic Algorithm

3 Methods

3.1 Search-Space

Each minimum search is limited within a space $[a, b]$, predefined for each function. The precision of our tests is the same for all functions, that being 10^{-5} because 5 decimals after 0 is good compromise between precision and running speed since increasing the precision greatly increases the running time of the algorithms. The given space can then be divided into $(b - a) \cdot 10^5$ subintervals. The parameters of a function are represented as an array of bits, each parameter being $\lceil log_2\big((b-a) \cdot 10^5\big)\rceil$ bits long.

3.2 Fitness Function

Fitness functions are a type of function which allow us to approximate how good chromosomes are compared to each other. Getting an accurate fitness function is very difficult and is the main challenge of most genetic algorithms. For our purposes, we decided to go with some basic mathematical functions since we only needed to minimize the function. We will denote f as the fitness function and g as the benchmark function. For De Jong's and Rastrigin's function, since they are always positive we can simply invert the result using the function $f(x) = \frac{1}{g(x)}$. For Schwefel's and Michalewicz's functions we have decided to deduct the maximum value so all results are negative, and then multiply by -1, resulting in the function $f(x) = -(g(x) - max(g(x)))$.

3.3 Selection

Selection is done using a fitness proportionate selection or wheel of fortune selection. Each individual is given an individual selection probability equal to its fitness value divided by the total fitness of the population. We then divide the interval $[0, 1]$ into "buckets", each bucket ranging from the end of the last bucket to the value of the last bucket + the individual selection probability of the individual. We then generate random numbers between 0 and 1 each time we want to select a chromosome, and according we select the individual whose "bucket" contains the random number generated. Selected chromosomes will be ordered randomly, and the same chromosome can be selected multiple times. Selected chromosomes are kept for the next generation.

3.4 Cross-Over

Chromosomes will be kept in a vector, and with probability $crossOverP$ will have a chance to exchange genes with the next chromosome in the vector. The cross-over is done using a single-point cut, meaning one children will have the genes of the first parent up to the cut point and the genes of the second parent after that, and the other will have the opposite.

Parent 1:	Child 1:
0110|1010	0110|0101
Parent 2:	Child 2:
1101|0101	1101|1010

Example of Cross-Over

3.5 Mutation

Child chromosomes produced from cross-over have a chance to mutate. Mutation is generally done by going over each bit of a chromosome, generating a random number, and if that number is lower than a probability mutation, the bit gets flipped. This has considerably slowed our algorithm down, so we decided to opt for a new method. We chose a probability p, then generated random numbers until one is greater than p. Each time we generate a number, we flip a random bit. This dramatically speeds up our algorithm, but comes with 2 disadvantages. First, the actual probability of mutation is now different from p, which has to be calculated separately if we wish to know the real p. This also means that the amount of genes mutated is not proportional to the length of the chromosome, but is a constant number instead. This can be offset by using a different p for each combination of function and dimension. Secondly, the same bit can be selected twice for mutation, resulting in no actual changes in the chromosome even though it passed the mutation check twice. This means that our mutation probability is lower than we actually calculate. This can be offset by simply increasing p, or by keeping track of which bits have been mutated already. We went with the first version for this algorithm, since we wanted to maximize speed. From this point on, $BitsToBeMutated$ will be used as the number of bits our algorithm will modify on average for each children created through selection, not including overlap.

4 Improvements

4.1 Elitism

By sorting the chromosomes by fitness, we can be sure the best chromosomes always survive. If a chromosome that is in the best 5 has not been selected, elitism will make sure to add it to the next population, although without the possibility of cross-over. This makes our algorithm always keep the best chromosomes, making it more likely to converge on a good solution, but reducing the amount it explores the search space.

4.2 Hill-Climbing

To optimize our solution, whenever we find a new best chromosome we keep it, and at the end of our algorithm we run a Best-Improvement Hill-Climbing, in order to make sure it reaches a local minimum.

4.3 Meta-Algorithm

Since all the functions are vastly different, and our algorithm has many different parameters, finding the best one using trial and error would be too hard. Running a Best-Improvement Hill-Climbing on the parameters, we can try and approximate these results. Starting with the default parameters: $BitsToBeMutated$: 3, $ChanceToCrossOver$: 1, $ChromosomesToBeSelected$: 100, $NumberOfElites$: 5, we select each parameter one by one and generated it neighbourhood incrementing and decrementing it by 4 predetermined values called steps.

Steps for chanceToMutate and chanceToCrossOver: $0.01, 0.03, 0.05, 0.07$
Steps for chromosomesToBeSelected: $1, 3, 5, 8$
Steps for elitesToBeSelected: $1, 2, 3, 4$

We then run the algorithm 30 times for each neighbour, and if the average is better than our current best average we modify that parameter accordingly. This algorithm is then run for each combination of function and dimension. The percentage of mutated bits in a chromosomes is calculated as following: given $p : chanceToMutate$ $\sum_{i=1}^{\infty}(\frac{1}{p})^i$ is the average number of bits which will be modified. We divide by the length of a chromosome, which is 135 and we get the percentage of bits modified.
Best parameters for 30 dimension version of functions found by this algorithm were:

Function	Percentage Mutated	Cross-Over Chance	Selected Chromosomes	Elites
Schwefel	0.0102	0.99	117	25
Rastrigin	0.0744	0.96	108	16
Michalewicz	0.0197	1	99	16

We can see that the algorithm leans into heavy elitism and 100% chance to cross-over. This is due to the fact that once a chromosomes has a very good parameter it can then pass it to other chromosomes.

4.4 Final Results

All of the results have been structured in tables for each function, for an easier comparison.

Function	Algorithm	Size	Expected minimum	Average found
De Jong	HC-first	5	0	0
	HC-best			0
	HC-worst			0
	SA			0
	GA			0
	GA opt			0
	HC-first	10	0	0
	HC-best			0
	HC-worst			0
	SA			0
	GA			0
	GA opt			0
	HC-first	30	0	0
	HC-best			0
	HC-worst			0
	SA			0
	GA			0.0064
	GA opt			0

Function	Algorithm	Size	Expected minimum	Average found
Schwefel	HC-first	5	-2094.9145	-2094.782
	HC-best			-2094.913
	HC-worst			-2031.407
	SA			-2038.963
	GA			-2094.815
	GA opt			-2094.914
	HC-first	10	-4189.829	-3982.711
	HC-best			-4139.763
	HC-worst			-3947.281
	SA			-4139.536
	GA			-4183.167
	GA opt			-4189.527
	HC-first	30	-12569.487	-11004.53
	HC-best			-11534.59
	HC-worst			-6261.155
	SA			-12491.29
	GA			-11618.52
	GA opt			-12299.92

Function	Algorithm	Size	Expected minimum	Average found
Rastrigin	HC-first	5	0	0.1326613
	HC-best			0
	HC-worst			1.041193
	SA			3.657982
	GA			0.57672
	GA opt			0
	HC-first	10	0	4.430919
	HC-best			2.655754
	HC-worst			6.226605
	SA			6.133923
	GA			2.58255
	GA opt			0.980629
	HC-first	30	0	32.64126
	HC-best			24.62465
	HC-worst			17.64222
	SA			16.03733
	GA			26.009
	GA opt			14.1555

Function	Algorithm	Size	Expected minimum	Average found
Michalewicz	HC-first	5	-4.687	-4.687544
	HC-best			-4.687649
	HC-worst			-4.679329
	SA			-4.390999
	GA			-4.6619
	GA opt			-4.6875
	HC-first	10	-9.66	-9.422776
	HC-best			-9.503494
	HC-worst			-9.057041
	SA			-9.275028
	GA			-9.5068
	GA opt			-9.54521
	HC-first	30	-29.63	-26.67321
	HC-best			-27.28958
	HC-worst			-14.16651
	SA			-28.31273
	GA			-27.273
	GA opt			-28.4807

5 Conclusions

We can see that genetic algorithms provide a clear improvement over local-search algorithms while also allowing custom tailoring towards specific problems. Elitism provided the best improvement out of all the improvements, probably due to the nature of the functions. The meta-algorithm would've probably provided the best improvements, but due to the random nature of running an iteration, it would very often converge to local minima, which later would find that they were not that good. The run time of the meta-algorithm was very high, even though the normal algorithm was very quick (approximately 2-3 seconds).
The algorithm could be improved by running the meta-algorithm with a higher precision(running each neighbour for about 100-200 iterations before concluding its average), but this would take a very long time. Allowing more generations or greater average population sizes may also improve the effectiveness of this algorithm. Other specific improvements like gray encoding or simulating annealing could improve the result of some functions.

References

[1] Eugen Croitoru, Teaching: Genetic Algorithms
`https://profs.info.uaic.ro/~eugennc/teaching/ga/`

[2] Functions Information
`http://www.geatbx.com/docu/fcnindex-01.html#P89_3085`

[3] Global minimum for Michalewicz's function
`http://www.alliot.fr/papers/gecco2014b.pdf`

[4] Wheel of fortune selection
`https://en.wikipedia.org/wiki/Fitness_proportionate_selection`

[5] Mersenne-Twister Random Number Generator `https://www.learncpp.com/cpp-tutorial/generating-random-numbers-using-mersenne-twister/`

[6] Point-Crossover `https://www.sciencedirect.com/topics/computer-science/point-crossover`