Study of function minima using Hill Climbing, Simulated Annealing and Genetic algorithms

Mihalache Radu Stefan

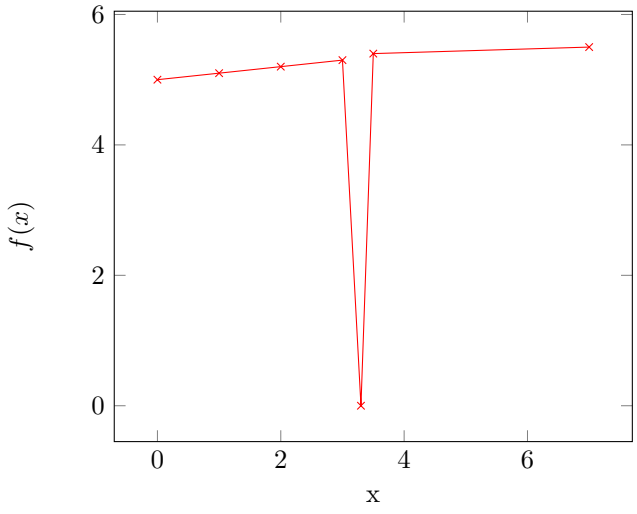
Abstract

In this paper I will approach the problem of solving the global minima with Hill Climbing, Simulated Annealing and Genetic algorithms, use my implementation to compare their performance and draw some conclusions regarding their efficacy.

Introduction

Motivation

The problem of exploring the values of functions and finding the global minimum of said function for a specified domain has useful applications, yet it is difficult to solve with a deterministic algorithm. That is because some functions have a very small accumulation basin compared with the size of the domain for the global minimum.



To solve this problem we will explore the following nondeterministic approaches:

- The Hill Climbing algorithm which is a greedy method
- The Simulated Annealing algorithm which is a meta-heuristic method that better explores the function's domain
- The Genetic algorithm which takes inspiration from nature and adapts its solution population to get closer to the global minimum

The motivation of this experiment is to determine which one of the approaches listed above gives the best results for a set of functions in the shortest amount of time and examine why.

Method

The representation of the input variables will be a string of n bits such that they can adequately represent the function domain.

$$x = a + decimal_{represenation}(bit_{str}) \cdot (b-a)/(2^n - 1), x \in [a, b]$$

Hill Climbing and Simulated annealing make use of this representation by generating a random input called the candidate solution, its vicinity can be explored by negating one bit, such that the hamming distance between the candidate solution and the vicinity is one.

Hill Climbing:

Selects a candidate solution for each iteration and try to improve it using either the first better vicinity (first improvement) or the best vicinity (best improvement). This algorithm finds the minimum by exploring the basin of the candidate solution.

Simulated annealing:

Selects a candidate solution at the start and explore its vicinity. This algorithm better explores the domain of the function by choosing worse vecinities based on the probability given by this expression:

$$random.uniform(0, 1) < math.exp(-abs((evaln - evalc)/temperature))$$

This algorithm makes use of the hot iron concept. At the beginning the temperature is high and the chance to choose a worse solution is high, but it decreases over each iteration (which makes it more greedy) based on this formula:

$$temperature = temperature \cdot 0.9$$

Genetic:

The genetic algorithm takes inspiration from nature, by holding a population of chromosomes, which are selected and evolve under the pressure a a fitness function. A chromosome represents a potential solution and its chances to be selected for the next generation depend on the chromosome's evaluation by the fitness function. As the solution is set of bits, a gene is an individual bit in the solution. The evolution process is accomplished by performing mutation and cross over on the chromosomes of the population.

Mutation is performed by flipping a random bit of a chromosome.

Crossover at one point between two chromosomes is performed by picking a random locus (position of a gene) and swapping the genes after the locus between the two. Example, locus = 4 :

$$10101010 \rightarrow 10101111$$

$$11111111 \rightarrow 11111010$$

The fitness function $fit(chromosome)$ that I have used in this experiment is inversely proportional with the result of the function $f(chromosome)$ for which we want to find the minima. It uses $maximum = max(f(current_{population}))$ and $minimum = min(f(current_{population}))$.

$$fit(chromosome) = (maximum - f(chromosome))/(maximum - minimum)$$

The selection mechanism I have used in this experiment is Wheel of Fortune. It calculates the totalFitness

$$totalFitness = \sum(fit(chromosome)), chromosome \in population$$

in order to compute the individual probability $p_{chromosome} = fit(chromosome)/totalFitness$ and an array of cumulated probabilities $q_{i+1} = q_i + p_{chromosome_i}$ that it uses to a chromosome at position j, $population_{size}$ times such that:

$$r \in (0, 1] \ , \ q_j < r < q_{j+1}$$

An additional mechanism my genetic algorithm implementation make use of is Elitism. It makes sure that the best x solutions are always maintained in the population and mitigates the effects of evolutions that do not result in better solutions.

The analysed functions:

$$f(x) = \sum_{i=1}^{n} \left[x_i^2\right], x_i \in [-5.12, 5.15]$$

$$min = 0$$

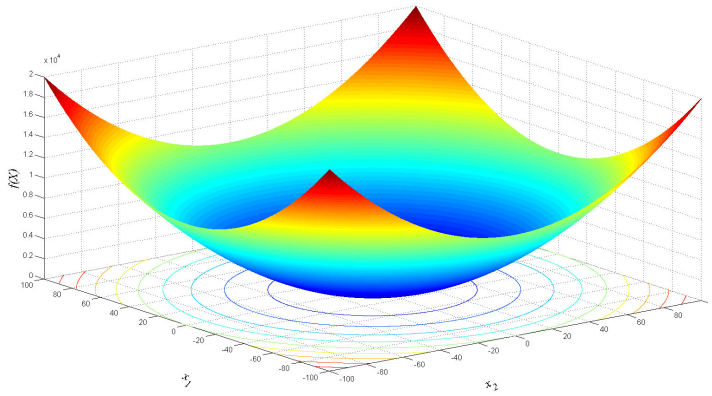


Figure 1: Image De Jong's Function.[1]

[1] https://al-roomi.org/benchmarks/unconstrained/n-dimensions/

$$f(x) = \sum_{i=1}^{n} \left[-x_i \cdot sin(sqrt(|x_i|))\right], x_i \in [-500, 500]$$

$$min = (-n) * 418.98$$

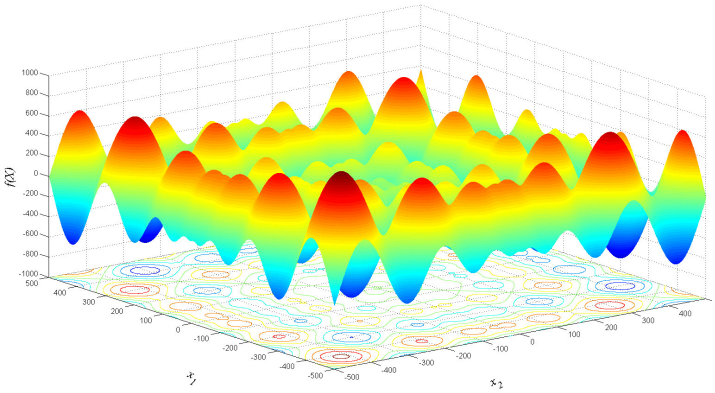


Figure 2: Image Schwefel's Function. [2]

$$f(x) = A \cdot n + \sum_{i=1}^{n} \left[x_i^2 - A \cdot cos(2\pi x_i)\right], A = 10, x_i \in [-5.12, 5.15]$$

$$min = 0$$

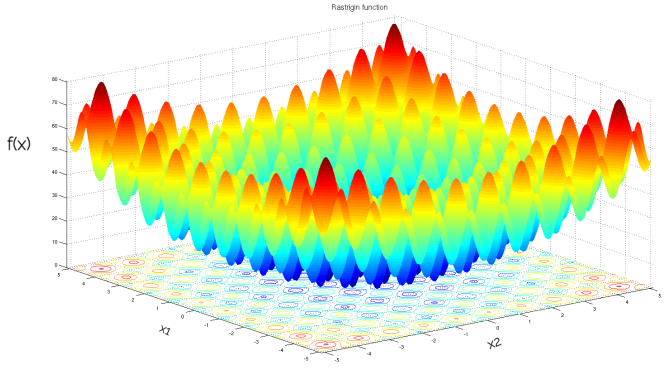


Figure 3: Image Rastrigin's Function. [3]

[2] https://al-roomi.org/component/tags/tag/schwefel-function

$$f(x) = -\sum_{i=1}^{n} \left[sin(x_i) \cdot \left(sin\left(\frac{i \cdot x_i^2}{\pi} \right) \right)^{2 \cdot m} \right], x_i \in [0, \pi], m = 10$$

$$min = -4.68\,,\ n = 5$$

$$min = -9.66\,,\ n = 10$$

$$min < -27.5\,,\ n = 30$$

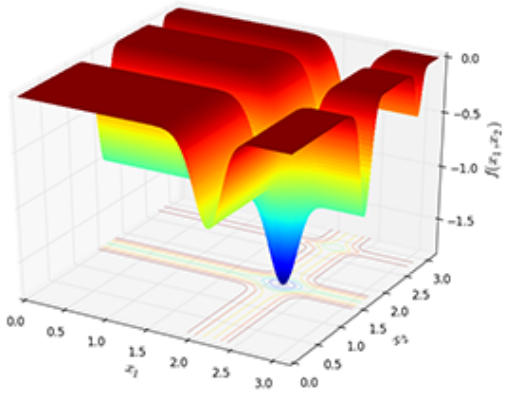


Figure 4: Michalewicz's Function. [4]

[3] https://commons.wikimedia.org/wiki/MainPage
[4] https://www.sfu.ca/ ssurjano/michal.html

Experiment

The Hill Climbing and Simulate Anealing algorithms were implemented in python and analized the functions above with 100 iterations. The temperature for simulated aneealing is initially 2000.
The Genetic algorithm was implemented in C++ and analized the functions above with a population of 200, an elite population of 10 and 2000 iterations. The test was run on the compiled 64 bit program with -O2 compiler flag.
Each test has 10^{-5} precissionn is run 30 times to ensure consistency.

Results

$DeJong\,5D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	0	0	0	0	14
H.C. Best	0	0	0	0	28
Simulated An.	0	0	0	0	4
Genetic	0	0	0	0	0.33

$DeJong\,10D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	0	0	0	0	112
H.C. Best	0	0	0	0	216
Simulated An.	0	0	0	0	8
Genetic	0	0	0	0	0.49

$DeJong\,30D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	0	0	0	0	1946
H.C. Best	0	0	0	0	2570
Simulated An.	0	0	0	0	20
Genetic	0	0	0	0	0.91

$Schwefel\,5D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	-1939	-1992	-1853	-	22
H.C. Best	-1967	-2094	-1893	-	20
Simulated An.	-1825	-1963	-1776	-	6
Genetic	-2094.90312	-2094.91443	-2094.70647	0.04100	0.50

$Schwefel\,10D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	-3652	-3872	-3492	-	262
H.C. Best	-3623	-3772	-3506	-	166
Simulated An.	3642	-3905	-3246	-	14
Genetic	-4189.45055	-4189.82574	-4189.20499	0.15884	0.69

$Schwefel\,30D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	-9453	-9923	-9369	-	3051
H.C. Best	-10153	-10453	-10063	-	3022
Simulated An.	-9746	-11279	-9365	-	40
Genetic	-12495.54426	-12568.20105	-12410.66695	49.48430	1.58773

$Rastrigin\,5D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	2.29497	1.58367	3.39702	-	26
H.C. Best	1.93592	1.279144	1.68702	-	28
Simulated An.	2.23086	1.71005	3.102193	-	6
Genetic	0	0	0	0	0.43

$Rastrigin\,10D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	7.18790	3.93217	15.19234	-	322
H.C. Best	8.18790	3.67834	14.0925	-	242
Simulated An.	9.18790	3.19593	15.2394	-	14
Genetic	0.23970	0	2.24758	0.56087	0.63

Rastrigin $30D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	33.72184	27.12235	48.63187	-	3234
H.C. Best	32.43780	26.97201	47.29053	-	2784
Simulated An.	35.18790	27.67834	56.44140	-	42
Genetic	16.18794	6.32544	29.65766	5.57709	1.47341

Michalevicz $5D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	-4.44109	-4.62793	-3.97123	-	22
H.C. Best	-4.37306	-4.65685	-4.02012	-	20
Simulated An.	-4.28916	-4.6328	-3.26917	-	4
Genetic	-4.69345	-4.69346	-4.69342	0.00013	0.68

Michalevicz $10D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	-8.20880	-9.01029	-6.89132	-	214
H.C. Best	-9.09037	-9.24031	-8.25910	-	158
Simulated An.	-8.70903	-9.22690	-6.15737	-	12
Genetic	-9.56998	-9.66005	-9.30496	0.09151	1.16

Michalevicz $30D$

Algorithms	Solutions				Time Averege
	Averege	Minimum	Maximum	S.D.	
H.C. First	-20.79776	-22.61219	-17.31592	-	1260
H.C. Best	-24.09166	-25.14376	-22.21472	-	846
Simulated An.	-23.66259	-24.29173	-21.53991	-	30
Genetic	-27.09429	-28.04950	-25.58877	0.59147	3.11664

Analysing Results

First of all, in regards to time this is not a fair comparison since the algorithms were implemented in different languages.

In regards to the quality of the results given, the genetic algorithm scores better than all the other algorithms. This may be explained by the fact that it does not "blindly" search the domain on the function, like the others do in some manner, but always constructs the solutions based on what "already works" while still maintaining a great degree of freedom when exploring the function's domain.

Another aspect that can be observed is the fact that S.D. values for low number of axes is extremely low and it exponentially increases as the number of axes increases. While this is true for all algorithms, I have done some testing to see how the S.D. changes with more iterations and found that it decreases significantly.

I believe this signifies the fact that the genetic algorithm has somewhat steady progress towards an optimum solution and that with a certain number of iterations it will reach a good solution with a higher degree of probability than the other algorithms.

Conclusions

First of all, the number of elite individuals assigure that the best persist in the population but also decreeses the genetic diversity as the number of itterations increeses.

I tried to replicate this by lowering the number of cross-overs with the number of iterations, but the only function that was affected positivily was the DeJong. From this, I can conclude that genetic diversity must be balanced with favoring better resaults.

In regards to the implementation details, increesing the efficienty of the algoritm greatly helped with testing and fine-tuning the algorithm. In this sense I propose using std::bitset for representing the solutions and bitwise operations to evolve them and also do operation with the chromosomes of the population on multiple threads.

I believe that, while the genetic algorithm scores the best in my implementation, it can benefit from some thechniques used in the other algorithms, for example after finding an elite member of the population it can be "enhanced" by aplyind one iteration of a Hill Climbing algorithm. Also, to better emulate real conditions, a genetic algorithm may implement more than one population with different fitness functions with a chance for the populations to intersect at a certaint number of iterations.

Bibliography

[1] Wikipedia Commons
Rastrigin's Function rendered image. https://commons.wikimedia.org/wiki/Main_Page

[2] Al-roomi
De Jong's Function rendered image. https://al-roomi.org/benchmarks/unconstrained/n-dimensions/ Al-roomi
Schwefel's Function rendered image. https://al-roomi.org/component/tags/tag/schwefel-function

[3] Sfu
Michalewicz's Function rendered image. https://www.sfu.ca/~ssurjano/michal.html

[4] Geatbxi
Function formulas. http://www.geatbx.com/docu/fcnindex-01.html

[5] Course Page. https://profs.info.uaic.ro/~eugennc/teaching/ga/l

[6] Pycharm. https://www.jetbrains.com/pycharm/

[7] Visual Studio. https://visualstudio.microsoft.com/

[8] Random function. https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution