T1 - Numerical optimization using single solution metaheuristics

Oloieri Alexandru

November 4, 2019

Abstract

This paper analyzes and compares the behaviour and performance of three single solution metaheuristics (Best / First Improvement Hill Climbing and Simulated Annealing) when dealing with a complex problem: finding the global minima/maxima of a mathematical function.

1 Introduction

In computer science, a hard problem is a problem which requires a lot of resources (time and memory) in order to be solved. For this kind of problems, metaheuristics are used in practice, because even if they don't produce an optimal result on each run, the results they produce are usable and their running time is low compared to other algorithms that would find the optimal result, but realistically speaking, they would never end for large instances of the problem.

> "In computer science and mathematical optimization, a metaheuristic is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity. Metaheuristics sample a set of solutions which is too large to be completely sampled."

A well known hard problem is finding the global minimum/maximum of a mathematical function which is defined on an interval. In this experiment we'll see how good three metaheuristics (Best / First Improvement Hill Climbing and Simulated Annealing) are behaving in practice, when dealing with this problem, more exactly, when trying to find the global minimum of some functions with more than one dimensios, more exactly, functions which have arguments of type $x = (x_1, x_2, ..., x_n)$.

2 Algorithms

There are a lot of metaheuristics, and they have a lot of properties based on which we can classify them. We'll list here just two of those classifications, the ones I think are the most relevant for our experiment. After that, we will describe how the three algorithms we'll use in this experiment work.

2.1 Single-solution vs population-based

Single-solution metaheuristics focus on modifying and improving a single candidate solution, while population-based approaches "maintain and improve multiple candidate solutions, often using population characteristics to guide the search". As stated in the introduction of this paper, for this experiment we will focus only on single-solution algorithms.

2.2 Local search vs global search

Local search algorithms are algorithms which get stuck in a local minima, as their strategy is to never take a solution that's worse than the current one. Sometimes, that global minima is also the global maxima, but for highly multimodal functions that doesn't happen very often. A well known local search algorithm is the Hill Climbing method, and for this experiment we'll use two of its variations.

Global search metaheuristics don't get stuck in the local minima of a function because they will take a worse solution with a probability that is strictly greater than 0. In this category we usually include population-based metaheuristics, for example Ant Colony Optimization or Genetic Algorithms, but the analysis of these algorithms is beyond the scope of this paper.

There exist some metaheuristics which can be both classified as local search-based or global search metaheuristics, and in this experiment we will use one of these methods, more exactly, we'll use Simulated Annealing. This algorithm is very similar to Hill Climbing methods, but in has an optimization which ensures that the algorithm will not always get stuck in a global minima, and it does that by giving a small chance to worse solutions to be considered as the next solution.

2.3 Common details

2.3.1 Representation of the solutions

The solutions will be represented as binary arrays. Let's say we want to represent every real number with d digits for precision. Then we will divide the function domain $[leftMargin, rightMargin]$ in $Cnt = (rightMargin - leftMargin) * 10^d$ equal intervals. For a dimension, to represent Cnt different values we need $len = \lceil log2(Cnt) \rceil$ bits, so for N dimensions we will need $len \cdot N$ bits.

2.3.2 Decodification of the solutions

We will have to decode the solutions to be able to evaluate them. If we represented real numbers in the manner described at 2.3.1, then for a function with N dimensions, the following domain: $[leftMargin, rightMargin]$, and $binary_i$ being the binary representation of the i^{th} dimension, the real value for each dimension will be:

$$realValue_i = leftMargin + decimal(binary_i)/(2^{len} - 1) \cdot (rightMargin - leftMargin)$$

2.4 Neighbors of a solution

The neighbors of a solution S with a binary representation having length len will be all binary arrays BA of length len with the following property: $hammingDistance(S, BA) = 1$.

2.5 Hill Climbing

The algorithm has some simple steps:

Step 1: Generate a random solution.

Step 2: While the current solution has a better neighbor, choose it as the new candidate. If it doesn't have one, go to Step 3.

Step 3: Return the best found candidate.

The difference between the two variations of this algorithm are only related to how we choose the next candidate.

2.5.1 Best Improvement

In this variation of the algorithm we will choose as the next candidate the neighbor which has the best value.

2.5.2 First Improvement

In this variation of the algorithm we will choose as the next candidate the first neighbor which has a value better than the value of the current candidate.

2.6 Simulated Annealing

This algorithm is very similar to the Hill Climbing, only that we can also choose as the next candidate a worse solution (with a probability that decreases over time, we'll do that using a variable T representing a temperature):

Step 1: Generate a random solution, initialize T.

Step 2: $selected = false$; Check each neighbor of the current candidate in order: if the current neighbor is a better solution or if $random[0,1) < exp(-(abs(currentValue - neighborValue))/T)$ then $selected = true$ and stop checking neighbors; If $selected == true$ then decrease T and repeat Step 2, otherwise go to Step 3.

Step 3: Return current solution.

3 Functions

Rastrigin's function:

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

Schwefel function:

$$f(x) = 418.9829 \cdot n - \sum_{i=1}^{n} \left[x_i \cdot sin(\sqrt{|x_i|})\right], x_i \in [-500.0, 500.0]$$

De Jong's function:

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

Six-Hump Camel Back function:

$$f(x) = (4 - 2.1 \cdot x_1^2 + 4 \cdot x_1^4/3) \cdot x_1^2 + x_1 \cdot x_2 + (-4 + 4 \cdot x_2^2) \cdot x_2^2, x_1 \in [-3, 3], x_2 \in [-2, 2]$$

For our experiment to have varied and relevant results we have to choose functions with diffent types of graphs and different properties.

The Rastrigin and the Schwefel functions are non-convex, multimodal functions that have a lot of local minima, which should cause the Hill Climbing algorithms to get stuck in such a local minima. While the Simulated Annealing method is designed to get out of local minimums a number of times, I believe it has undefined behaviour when facing such complex functions.

De Jong's function is continuous, convex and unimodal, so all three algorithms should find the global minima on each run, the only thing that will vary will be the running time.

The Six-Hump Camel Back function has six local minima, two of which are global, so for a decent number of runs all three algorithm should find the global minima.

4 Experiment

Let's call an "iteration" running one of the algorithms described at 2.5.1, 2.5.2 or 2.6 once. A run will have 100 iterations, and for a function with fixed number of dimensions we will have 30 runs (so we will get 3000 results). For every run we will represent real numbers with 5 digits for precision.

5 Results

	Best Improvement Hill Climbing			First Improvement Hill Climbing			Simulated Annealing		
dim	5	10	30	5	10	30	5	10	30
best	0.00000	2.23583	26.26305	0.00000	3.46650	33.71223	0.99496	3.70747	35.19001
worst	2.23078	8.70245	40.83755	4.22569	12.92795	50.62044	3.23079	11.39477	51.67900
mean	1.27748	6.24778	34.04548	2.18285	8.53429	42.01656	2.23335	8.28342	42.90162
stDev	0.62882	1.68922	3.92445	0.8787	1.88110	4.28573	0.56593	1.75221	4.46615
time (s)	2.156	14.544	399.787	2.889	18.931	495.159	3.374	12.390	231.494

Table 1: Rastrigin's function Analysis

	Best Improvement Hill Climbing			First Improvement Hill Climbing			Simulated Annealing		
dim	5	10	30	5	10	30	5	10	30
best	0.103	68.886	723.326	26.987	233.830	1736.136	0.104	187.918	2048.135
worst	118.652	421.385	1817.148	249.664	674.637	2406.667	249.773	590.661	3299.346
mean	30.182	262.262	1498.298	125.542	472.583	2137.520	120.241	438.037	2716.953
stDev	31.698	86.028	216.901	49.197	97.693	155.370	60.708	101.559	253.647
time (s)	5.077	43.371	974.664	2.980	21.508	508.136	5.53	25.279	330.391

Table 2: Schwefel's function Analysis

	Best Improvement Hill Climbing			First Improvement Hill Climbing			Simulated Annealing		
dim	5	10	30	5	10	30	5	10	30
best	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.00000	0.00000
worst	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.00000	0.00000
mean	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.00000	0.00000
stDev	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.00000	0.00000
time (s)	1.531	10.365	314.747	0.968	5.576	149.911	2.036	7.969	179.553

Table 3: De Jong's function Analysis

	Best Improvement Hill Climbing	First Improvement Hill Climbing	Simulated Annealing
best	-1.03163	-1.03163	-1.03163
worst	-1.03163	-1.03159	-1.03161
mean	-1.03163	-1.03162	-1.03162
stDev	0.00000	0.00005	0.00001
time	0.854	0.327	0.765

Table 4: Six-Hump Camel Back function Analysis

5.1 Interpretation

5.1.1 Rastrigin's and Schwefel's functions

For both functions, for each number of dimensions, the Best Improvement Hill Climbing method has the best results, but the differences are not that big.

5.1.2 De Jong's function

As expected, each algorithm got to the global minima on every run. First Improvement Hill Climbing method has by far the best running times, as it always chooses the first neighbor which improves the current solution, and since the function is unimodal, each choice will get us closer to the optimal solution. Simulated Annealing method also chooses the first neighbor, but it does some additional computations, which take time. The strategy that gave us the best results for the other functions, Best Improvement Hill Climbing, gave us the highest running times for this continuous function.

5.1.3 Six-Hump Camel Back function

Again, Best Improvement Hill Climbing has the best results, but all the methods got very close to the global minima.

5.1.4 Remarks

To get better results, we can increase the number of digits used for precision, or we could increase the number of runs.

Even though in theory the Simulated Annealing method can find more basins of attraction, so it has a better chance to reach the global minima/maxima of a function, we can observe that the strategy used by the Hill Climbing Best Improvement method can get us better results for a large number of iterations (in our experiment we had 3000 starting points for each function), but that could come with a higher running time (as we have seen in 5.1.3).

6 Conclusions

Any of the three metaheuristics used in this experiment can get us good results in practice, as each of them explores much of the function domain, so we cannot state that one of them is the best. Each one can behave better than the others in different situations, and with a little bit of analysis and experience we could do the right choices.

References

[1] More about metaheuristics in computer science: `https://en.wikipedia.org/wiki/Metaheuristic`

[2] Information about functions used in this experiment: `https://www.sfu.ca/~ssurjano/optimization.html`

[3] More details about the implementation of the mentioned algorithms: `https://profs.info.uaic.ro/~pmihaela/GA/laborator2.html`

[4] The github repository of the application used in this experiment: `https://github.com/OloieriAlexandru/Numerical-Optimization-Platform/tree/T1`