Homework T1
Computing The Global Minimum Of A Function Iterated Hill Climbing And Simulated Annealing

Alexandru Neagu

October 25, 2022

Abstract

In this paper we will continue to discuss about algorithms that are able to compute the global minimum of a function in a fast and accurate way. I will present an iterative algorithm called **Hill Climbing**, that encodes the function points using bitmasks. The algorithm will be followed by 3 different ways of implementing it, using First Improvement, Best Improvement and Worst Improvement. The results will show that **Best Improvement** is often the most accurate implementation of the algorithm. Also, we will see that an heuristic called **Simulated Annealing** might optimize the algorithm in a pleasantly unexpected way.

1 Introduction

1.1 Motivation

Creating an algorithm that computes the global minimum of a function is not an easy task. Maybe the hardest part of such an algorithm is to decide how to represent the function **domain** and how to explore it. We might be tempted to use the cartesian form of representation of a point. For example, the natural way of representing a point x in the D^n space is by saying that $x = (x_1, x_2, \cdots, x_n) \mid x_1, \cdots, x_n \in D$. This representation is bad for multiple reasons. Firstly, for each dimension, we don't have so many neighbours to explore. We either subtract an ϵ from x_i or add an ϵ to x_i. Because of this issue, the algorithm will expand very hard and the time required for such an algorithm to find the global minimum will be enormous. Secondly, it will be hard to use this representation in a computer code. Most probably a vector of double values will be required for each point explored in our code, which is both memory and time consuming. Thus, some way of encoding the points that increases the number of explored neighbours and is fairly easy to represent in code is urgently needed. The solution turns out to be easier than we expect, **bitmasks!**

1.2 Bitmask Representation

Let's imagine that we already know a way to easily convert any point x located in the space D^n and represented in the cartesian form $x = (x_1, x_2, \cdots, x_n) \mid x_1, \cdots, x_n \in D$ into a bitmask $b = (b_1, b_2, \cdots, b_m) \mid b_1, \cdots, b_m \in \{0, 1\}$ and vice versa. In this situation, we may represent a neighbour of a point b, as a point b' that differs from b in exactly one bit. For example, all the valid neighbours of the point $(1, 0, 0, 1)$ are $\{(0, 0, 0, 1), (1, 1, 0, 1), (1, 0, 1, 1), (1, 0, 0, 0)\}$. Thus, the number of neighbours of a point b is exactly m - the number of bits in its representation. Depending on the number of bits used, this representation helps to explore the neighbourhood of a point in a faster way. Also, if we think of a computer code, working with a bitmask is faster than working with a vector of doubles. It turns out that such a conversion is possible and I will present it in an elaborate way in the **Methodology** section. That being said, keep in mind that every algorithm that will be discussed in this paper, will use this representation.

1.3 Describing The Problem

In order to test the implemented algorithms, we need some good, representative functions. In this paper, we will compute the global minimum 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, 30 dimension versions to see how the results change if the number of dimensions is increased. In order to solve this problem, I will use the above mentioned bitmask representation of the points. The main algorithm to be discussed is the **Iterated Hill Climbing** algorithm. This algorithm selects some random candidate solution and then iteratively tries to improve the solution. Based on the criterion used by the algorithm to improve the solution, we define 3 different implementations: Best Improvement, First Improvement, Worst Improvement. I will describe and compare these implementations based on the results generated by each of them. Finally, I will present an heuristic called **Simulated Annealing** which I will use along with the First Improvement method to see if the Iterated Hill Climbing algorithm can generate better results.

1.4 Contents

The next sections in this paper are:

- **Methodology** - An elaborate description of the functions and algorithms used
- **Experimental Setup** - The initialization of the parameters
- **Results** - The results produced by the described algorithms, compared with the expected ones

- **Conclusion** - Elaborate discussion about the results

2 Methodology

2.1 Functions

1. **Rastrigin's Function**

   (a) Definition:

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

   (b) Search Domain: $-5.12 \leq x_i \leq 5.12, \forall i = 1, 2, \cdots, n$

   (c) The Global Minima: $x^* = (0, \cdots, 0) \mid f(x^*) = 0$

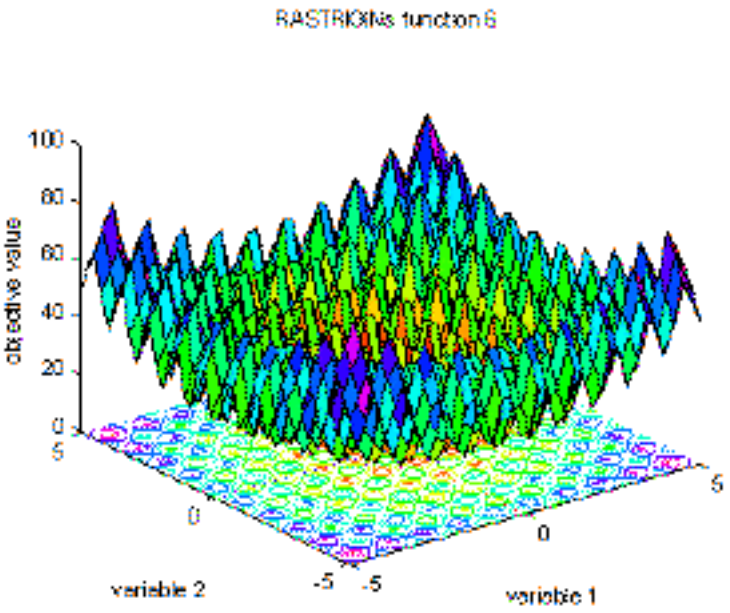

Figure 1: Rastrigin's Function

2. **Michalewicz's Function**

   (a) Definition:

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

   (b) Search Domain: $0 \leq x_i \leq \pi, \forall i = 1, 2, \cdots, n$

   (c) The Global Minima:
   at $n = 5$, $f(x^*) = -4.687658$
   at $n = 10$, $f(x^*) = -9.66015$
   at $n = 30$, $f(x^*)$ is unknown

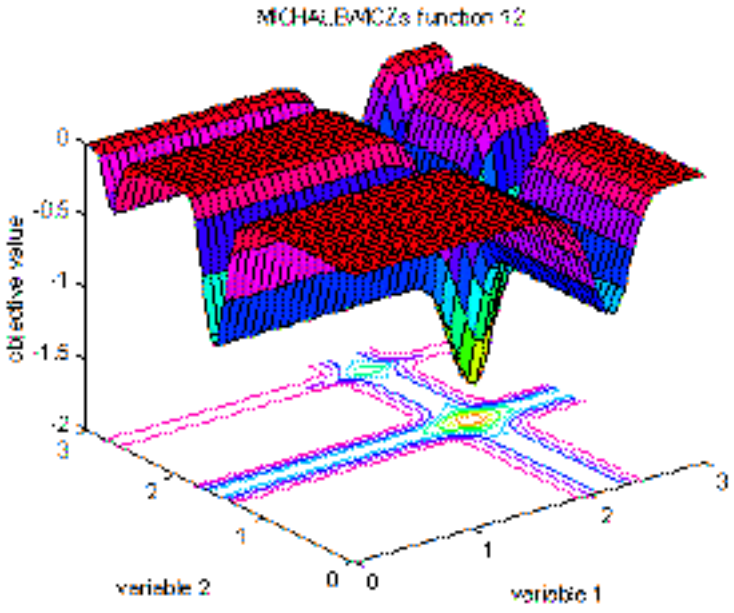

Figure 2: Michalewicz's Function

3. **DeJong's 1 Function**

(a) Definition:

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

(b) Search Domain: $-5.12 \leq x_i \leq 5.12, \forall i = 1, 2, \cdots, n$

(c) The Global Minima: $x^* = (0, \cdots, 0) \mid f(x^*) = 0$

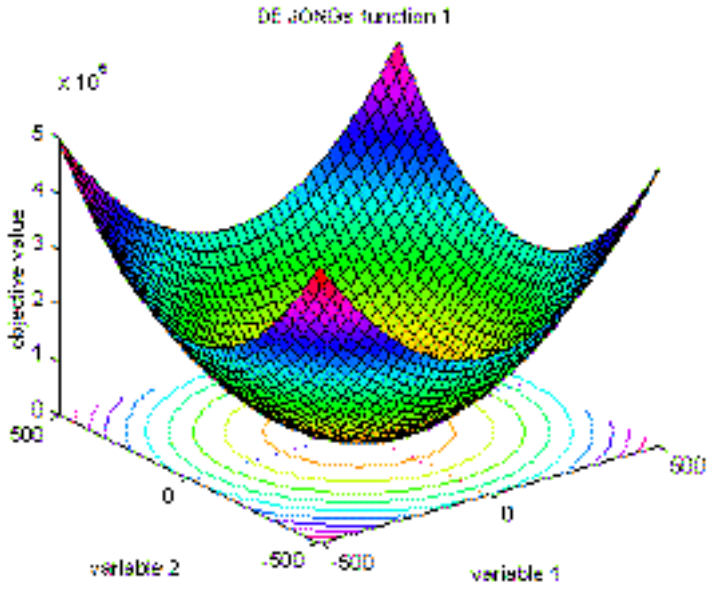

Figure 3: DeJongs's 1 Function

4. **Schwefel's function**

   (a) Definition:

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

   (b) Search Domain: $-500 \leq x_i \leq 500, \forall i = 1, 2, \cdots, n$

   (c) The Global Minima:
   $x^* = (420.9687, \cdots, 420.9687) \mid f(x^*) = 0$

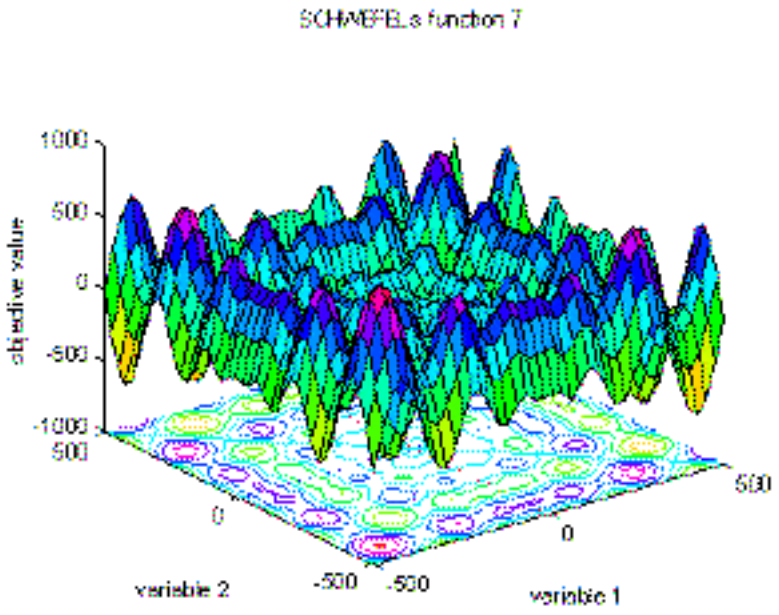

Figure 4: Schwefel's function

2.2 Algorithms

- Before explaining the main algorithm, we need to clarify the exact way of **encoding** the function points from the cartesian form to bitmasks. In order to fully grasp the encoding formulae, let's generalize the functions. Firstly, let's say that we are working with the n dimensions representation of some function f. For the sake of our example, let's say that each dimension is defined on some fixed interval of real numbers $[l, r]$.

To summarize, we may define our function in the following way

$$f : [l, r]^n \to R$$

There is one more question we need to answer before trying to encode the function points: how many of these points does the domain of the function contain? Because we are working with the real space, the answer to our question is, unfortunately, **infinitely** many. Obviously, our algorithm is able to investigate only a finite number of points, so we should find a way to solve this issue. The solution is simple, we should use some precision value ϵ. For each real value x, we should consider $x + \epsilon$ as it's successor and $x - \epsilon$ as it's predecessor. In this way, the algorithm will analyze a finite number of points. In our abstract example, let's fix the precision value as $e = 10^{-d}$. Using such a precision, it's trivial to see that the total number of points contained in our domain is

$$N = \left((r - l) * 10^d\right)^n$$

Now, how many bits do we need in order to represent any of these N points? The response to this question is

$$B = \lceil \log_2(N) \rceil = n * \left\lceil \log_2((r - l) * 10^d) \right\rceil$$

Using this formula for finding the number of bits needed to encode the points, we may assign a bitmask $b = (b_0, b_1, \cdots, b_{B-1}) \mid b_0, \cdots, b_{B-1} \in \{0, 1\}$ for each point explored by the algorithm. All good, but how can we convert this bitmask b into a valid function point x in order to compute $f(x)$? Let $ld = \frac{B}{n}$. Given the fact that each segment of ld bits corresponds to one dimension of our function, we can analyze them separately. Let $b_k = [b_{ld*(k-1)}, \cdots, b_{ld*k-1}]$ be the submask coresponding to the k-th dimension of our function. Also, let x_{int} be the decimal representation of the bitmask b_k using the trivial trivial algorithm

$$x_{int} = \sum_{i=0}^{ld-1} 2^i * b_{ki}$$

We should now find a way to convert this integer number into a coresponding real number. Please notice the following observations:

$x_{int} \in [0, 2^{ld} - 1]$
$\frac{x_{int}}{2^{ld}-1} \in [0, 1]$
$\frac{x_{int}}{2^{ld}-1} * (r - l) \in [0, r - l]$
$\frac{x_{int}}{2^{ld}-1} * (r - l) + l \in [l, r]$

Thus, we may conclude that for some dimension k, the real value x_k and the submask b_k of length ld that correspond to this dimension are correlated by the following formula

$$x_k = \frac{\sum_{i=0}^{ld-1} 2^i * b_{ki}}{2^{ld} - 1} * (r - l) + l$$

If we compute this formula once for each segment of size ld bits, we wil get the value of the entire point $x \in R^n$.

- Now that we have a very good intuition of how to represent the points in the bitmask form, let's see what exactly is the **Iterated Hill Climbing** algorithm. From now on, when I will mention a bitmask point b it will correspond to the bitmask representation of some cartesian point. When $f(b)$ will be mentioned, it will refer to the value of the function in the point x, where x is the point obtained by decoding the bitmask b using the formulae mentioned above.

  The main idea of the algorithm is to generate a bitmask point b and iteratively go to its better neighbours until there is no neighbour better than the current point. In this case, a local minimum has been reached and we can update the global answer. By repeating this process I times, where I is a parameter as big as possible, we may get some very good results. Before enumerating the steps of the algorithm, let's clarify some definitions

  > The bitmask b' is the neighbour of the bitmask b if and only if there is **exactly one** $i \mid 0 \leq i \leq B-1$, where $b_i \neq b'_i$.
  >
  > The neighbour b' is better than the current bitmask b if and only if $f(b') < f(b)$

  That being said, these are the steps of the algorithm:

  1. Repeat the following steps I times
  2. Select a random bitmask point b
  3. Pick a neighbour b' that is better than b, assign $b = b'$, and repeat the same process
  4. If step 3. is not possible anymore, then we reached a local minimum, so if $f(b)$ is better than our best answer so far, we update the answer

  This algorithm looks good, but there is one very important thing to clarify, what if there are multiple neighbours better than out current bitmask point b? Which one should we choose? Based on these questions, I will present 3 different criterias to choose the better neighbour. Let's have C the list of all the neighbours b' such that $f(b') < f(b)$.

  - The **Best Improvement** method picks the better neighbour b' with the minimum value of $f(b')$. So, we choose such a neighbour $b' \in C \mid f(b') <= f(c), \forall c \in C$. This is the most intuitive thing to do and works quite well in a lot of cases. This method requires a relatively big amount of time to run and might be predictable. Another thing to point out, is the fact that this implementation is, indeed, a greedy strategy. For some special looking functions, this greedy method might not generate the best result.

- The **First Improvement** method shuffles all the neighbours of the current bitmask point b and then picks the **first** better neighbour. Even though this method might seem a little bit risky, it has a lot of advantages. Firstly, the first improvement method doesn't iterate through all the good neighbours. This feature makes the method very convenient to use if we want the elapsed time to be as small as possible. It might not generate as accurate results as the best improvement method, but it will surely generate decent results at a considerable **faster** pace. Also, the unpredictability of this method, makes it harder for someone to design such a function that will cause bad generated results.
- The **Worst Improvement** method picks the better neighbour b' that has the maximum value of $f(b')$. To say it better, we pick such a neighbour $b' \in C \mid (f(b) - f(b') <= f(b) - f(c)), \forall c \in C$. This method is by far the most time consuming one, because it iterates through all the better neighbours every time and is approaching the local minimum in a very slow pace. Despite all of this, this method might help a lot when working with some complicated functions, because it gives the opportunity to explore some unusual points, that the other methods wouldn't normally explore.

In the **Results** section, we will see how close are the minimum values generated by each of these methods to the actual global minimums. Also, we will analyze how much time is consumed in each case and draw some conclusions.

- All of the above mentioned algorithms have an important characteristic in common. When we analyzed all the neighbours of a given bitmask point b, we were only focused on the **better** ones. If the algorithm doesn't find any better neighbours, all of the above 3 discussed methods stop there. But what if we also consider neighbours worse than the current point? Maybe a worse neighbour at the moment will then generate a path to an unexpectedly better local minimum? This is what the **Simulated Annealing** heuristic tries to fix. When we reach a local minimum and there is no better neighbour, the heuristic doesn't stop there, but tries to continue its way to a worse neighbour, hoping that in the future a better local minimum will be discovered. This idea sounds interesting, but there are some things that should be discussed. Firstly, in the case of reaching a local minimum, we can't just go to a worse neighbour every time! If we are doing this, the algorithm will simply enter an infinite loop. We should fix a parameter W - the number of times the algorithm can go to a worse neighbour. Also, to improve our algorithm, we should have a probability function that will tell us whether to explore the worse neighbour or not. In a real life scenario, this is very similar to solving some puzzle and deciding to worsen the current progress by executing a risky move, hoping that this will generate a more promising and intuitive situation to find the solution.

Obviously, we don't want to take such risks all the time, but only when the move looks promising. We should think of a way to update our probability function every time we go to a worse neighbour. The ultimate goal is to make it less probable to repeat such a move in the future. This is exactly why the simulated annealing heuristic defines the temperature parameter T. The algorithm decreases T from time to time and this directly causes our probabilities to become smaller and smaller. Now that we understand what the temperature is, let's imagine that we are located at point b and think to go to the worse neighbour w. We may decide to explore the point w with the following probability

$$P = e^{\frac{-|f(b)-f(w)|}{T}}$$

This formula works particularly well, because the probability of exploring a worse neighbour is getting smaller only when T decreases or $|f(b) - f(w)|$ increases. This is exactly the behaviour we want to obtain. Let's fix another parameter for out algorithm: R the number of times we are allowed decrease the temperature T. Now that we understand these concepts, let me present how I implemented the algorithm.

1. For a fixed number of iterations I, repeat the following steps

   Choose an initial temperature T

   Select a random bitmask b

2. As long as we didn't decrease T more than R times repeat the following steps

   Fix a variable that counts the number of times we choose to explore a worse neighbour: $cnt = 0$

3. Repeat the following steps as long as $cnt < W$

   Pick a neighbour b' of the current point b

   If $f(b') < f(b)$, assign $b = b'$, exactly how we did in the Iterated Hill Climbing Algorithm

   If $f(b') > f(b)$, compute the probability P as described above in this subsection. After this, generate a random real value $p \in [0, 1]$ and check whether $p \leq P$.

   - If $p \leq P$, it means that the algorithm decides to explore the neighbour b'. Firstly, we should process if this local minimum $f(b)$ is better than our answer so far. Then, we assign $b = b'$ and go to the worse neighbour. Also, we should increase the variable $cnt = cnt + 1$
   - If $p > P$, it means that the algorithm decides not to explore the worse neighbour and stops at the current local minimum b. If this happens, we should break the loop from step 3.

4. If step 3 is not possible anymore or we exited from it, check whether the local minimum achieved $f(b)$ is better than the current answer.

Decrease the temperature T using a function g defined by us $T = g(T)$. After decreasing the temperature, go back to step 2.

There are two more things to clarify in my implementation. Firstly, I defined the temperature decreasing function g as $g(T) = T * 0.5$. There might be some better choices for picking this function and it's left to the programmer to make this decision. Lastly, I didn't specify the way I am picking a neighbour b' for the current point b. This pick can be done in a multitude of ways and it's really flexible. The way I decided to do it is to pick a neighbour using the **First Improvement** method. If I successfully find a better neighbour than my current point, I apply the exact same technique as in the Iterated Hill Climbing Algorithm. Otherwise, I decide to pick a worse neighbour b', whose $f(b')$ is the closest to $f(b)$. In other words, I pick $b' \mid (f(b') - f(b) \leq f(c) - f(b)), \forall c$ - neighbour of b.

That being said, this is how the simulated annealing heuristic works. We sometimes allow ourselves to go to a worse solution, in order to obtain a better solution later. This heuristic might be incredibly strong in some cases, because it gives us the chance to explore some unusual points in our function.

3 Experimental Setup

Let's try to run the presented algorithms several times using the following parameters:

- For the **Iterated Hill Climbing** algorithm, I ran the program with the following sets of parameters (They are the same for all of the 3 methods of improvement)
  1. $n = 5$, $\epsilon = 10^{-5}$, $I = 10000$
  2. $n = 10$, $\epsilon = 10^{-5}$, $I = 5000$
  3. $n = 30$, $\epsilon = 10^{-5}$, $I = 500$
- For the **Simulated Annealing** heuristic hybridised with **First Improvement Hill Climbing**, I ran the program with the following sets of parameters
  1. $n = 5$, $\epsilon = 10^{-5}$, $I = 5000$, $T = 10000$, $R = 20$, $W = 50$
  2. $n = 10$, $\epsilon = 10^{-5}$, $I = 1000$, $T = 10000$, $R = 20$, $W = 50$
  3. $n = 30$, $\epsilon = 10^{-5}$, $I = 200$, $T = 10000$, $R = 20$, $W = 50$

4 Results

Table 1. ***Test results for the Rastrigin's Function***

Climbing Type	Dimensions	Result Avg.	Best Result	Worst Result	Result StDev	Time Avg.	Time StDev
Best Improvement	5	0.0002459	0.0000000	0.0018560	0.0005084	44.0741925s	27.5494025
	10	2.8207561	1.2358228	4.4615646	0.8006324	140.9025228s	65.7189051
	30	30.2583399	24.7734728	34.0193247	2.5813554	286.2131309s	81.1851112
First Improvement	5	0.0000000	0.0000000	0.0000000	0.0000000	14.2292166s	7.4299210
	10	4.5115113	1.9949708	6.6973981	0.9228425	26.9961050s	14.7687734
	30	42.8713310	34.6581218	47.7256115	3.3517488	20.7945305s	0.3150737
Worst Improvement	5	1.1123281	0.9998405	1.9998522	0.3027413	122.0061366	7.4706728
	10	5.7938076	4.2358578	7.0000818	0.8826613	390.1538057s	15.0079332
	30	47.8227027	44.2340219	52.6274274	2.8507345	887.3357939s	8.4559049
Simulated Annealing	5	0.0000000	0.0000000	0.0000000	0.0000000	183.2585846s	4.9544791
	10	0.0000000	0.0000000	0.0000000	0.0000000	107.3117944s	12.1012343
	30	14.5344338	7.4565471	19.4203586	2.5160594	85.3967830s	7.6239335

Table 2. ***Test results for the Michalewicz's Function***

Climbing Type	Dimensions	Result Avg.	Best Result	Worst Result	Result StDev	Time Avg.	Time StDev
Best Improvement	5	-4.6874874	-4.6876582	-4.6859568	0.0003667	57.9769114s	38.6050802
	10	-9.4974878	-9.6044050	-9.3330536	0.0690328	166.3546440s	103.6549853
	30	-26.8906492	-27.6377749	-26.3411384	0.3204304	306.4876233s	72.5498590
First Improvement	5	-4.6875534	-4.6876574	-4.6869817	0.0002088	16.7280764s	11.0437759
	10	-9.3989878	-9.6150330	-9.2677398	0.0916765	27.5201241s	7.9269803
	30	-25.4420376	-26.1340301	-24.5496476	0.3665461	29.4078169s	14.3523534
Worst Improvement	5	-4.6806965	-4.6857131	-4.6704742	0.0065958	89.7826130s	5.5708986
	10	-9.0600153	-9.4663903	-8.9197239	0.1654873	291.1534901s	2.5694653
	30	-23.6177287	-24.0603078	-23.2475967	0.2703261	724.8148897s	5.6968327
Simulated Annealing	5	-4.6876573	-4.6876581	-4.6876513	0.0000016	280.7034987s	11.5369865
	10	-9.4397865	-9.5697537	-9.3272650	0.0691054	203.9754449s	22.8609702
	30	-26.0261762	-26.2653553	-25.5338401	0.2477484	330.0825645s	13.5313598

Table 3. ***Test results for the DeJong's 1 Function***

Climbing Type	Dimensions	Result Avg.	Best Result	Worst Result	Result StDev	Time Avg.	Time StDev
Best Improvement	5	0.0000160	0.0000000	0.0000883	0.0000309	48.3180702s	43.1675385
	10	0.0001885	0.0000000	0.0009898	0.0003485	126.4207512s	51.6574075
	30	0.0015527	0.0000000	0.0072237	0.0028215	265.5213486s	105.0822592
First Improvement	5	0.0000000	0.0000000	0.0000000	0.0000000	11.8479844s	7.7243945
	10	0.0000000	0.0000000	0.0000000	0.0000000	19.1047824s	4.5721925
	30	0.0000000	0.0000000	0.0000000	0.0000000	17.4413414s	5.8445059
Worst Improvement	5	0.0000000	0.0000000	0.0000000	0.0000000	41.5928400s	4.7939308
	10	0.0000000	0.0000000	0.0000000	0.0000000	123.0810994s	0.9094083
	30	0.0000000	0.0000000	0.0000000	0.0000000	276.0422139s	2.9743852
Simulated Annealing	5	0.0000000	0.0000000	0.0000000	0.0000000	182.7429397s	5.6386803
	10	0.0000000	0.0000000	0.0000000	0.0000000	107.3625583s	11.2653827
	30	0.0000000	0.0000000	0.0000000	0.0000000	83.7853707s	6.9133600

Table 4. ***Test results for the Schwefel's Function***

Climbing Type	Dimensions	Result Avg.	Best Result	Worst Result	Result StDev	Time Avg.	Time StDev
Best Improvement	5	0.0077816	0.0000640	0.1037528	0.0260933	46.4842882s	2.7787772
	10	38.7977906	0.3127369	103.1213980	29.1157193	151.9471409s	4.6055647
	30	1310.7159984	897.5911435	1610.2674949	171.2289460	352.0059316s	2.7493362
First Improvement	5	1.0318057	0.0001007	27.0904036	4.9225732	18.3493665s	2.5946418
	10	176.8034623	27.4030464	287.8192888	58.0894840	34.5210891s	0.9505940
	30	1803.6222791	1511.6099937	2029.0853291	152.1381304	31.3664636s	1.2021989
Worst Improvement	5	75.7713656	53.7670702	80.6497281	10.8536958	302.2959056s	6.6490251
	10	259.8100053	222.4173058	319.3039644	31.4261751	992.2623599s	8.9823694
	30	1826.8175412	1294.9669161	2111.1675225	227.2825598	2363.1978634s	56.1215827
Simulated Annealing	5	2.3215095	0.0010346	27.0907372	7.4553140	31.3524896s	1.6975224
	10	236.1349039	118.8612354	345.0193405	59.6496586	20.9339730s	3.7590683
	30	1984.3125984	1632.3667276	2157.0141555	141.9525654	27.6957207s	0.9984566

5 Conclusion

From the results showed above, we may conclude that all 4 implementations behave differently. The most important conclusions are

- The most accurate version of the Iterated Hill Climbing algorithm was the **Best Improvement** one. Except for the Rastrigin's function, where the simulated annealing heuristic worked incredibly well, the best improvement method generated the most accurate results.
- The fastest version of the Iterated Hill Climbing algorithm was the **First Improvement** one. It's worth to notice that besides the excellent speed of the algorithm, the results generated by this method are reasonably close to the best improvement method. Thus, it might worth to use this algorithm if we want decent results, generated with blazing fast speed.
- The **Worst Improvement** method was by far the slowest one. For these particular functions, the method didn't show off in any way. Still, it will be a good idea to implement this method in the future, because it might behave well on some other examples.
- The most impressing fact observed in the results tables, is the average result generated by **Simulated Annealing** for the **Rastrigin's** function. Somehow, this heuristic helped us generating results 2 times more accurate than the other methods. Even though the results generated for the rest of the functions aren't as impressive, the simulated annealing heuristic is definitely able to obtain incredible results for a lot of functions.
- It isn't the best practice to set the parameters exactly the same for all the functions and algorithms. The optimal set of parameters is different on a case-by-case basis and depends on a various of things. It is hard to guarantee that all the methods will work well with the exact same set of parameters.

References

[1] Schwefel's Function
https://www.sfu.ca/~ssurjano/schwef.html

[2] Rastrigin's Function
http://www.geatbx.com/docu/fcnindex-01.html#P140_6155

[3] Michalewicz's function
http://www.geatbx.com/docu/fcnindex-01.html#P204_10395

[4] DeJong's 1 Function
http://www.geatbx.com/docu/fcnindex-01.html#P89_3085

[5] LateX Table Generator
https://www.tablesgenerator.com/

[6] Simulated Annealing Analogy
https://www.youtube.com/watch?v=eBmU1ONJ-os&ab_channel=BadriAdhikari

[7] Simulated Annealing Description
https://www.scirp.org/journal/PaperInformation.aspx?PaperID=78834

[8] Algorithm Descriptions and Implementation Details
https://profs.info.uaic.ro/~eugennc/teaching/ga/

[9] LateX Math Symbols
https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols