ISSN: 1803-3814 (Printed), 2571-3701 (Online)
https://doi.org/10.13164/mendel.2023.2.255

Time Complexity of Population-Based Metaheuristics

Mahamed G. H. Omran[1,3,✉], **Andries Engelbrecht**[2,3]

[1]Computer Science Department, Gulf University for Science & Technology, Kuwait
[2]Department of Industrial Engineering and Computer Science Division, Stellenbosch University, South Africa
[3]Centre for Applied Mathematics and Bioinformatics, Gulf University for Science & Technology, Kuwait
omran.m@gust.edu.kw✉, engel@sun.ac.za

Abstract

This paper is a brief guide aimed at evaluating the time complexity of metaheuristic algorithms both mathematically and empirically. Starting with the mathematical foundational principles of time complexity analysis, key notations and fundamental concepts necessary for computing the time efficiency of a metaheuristic are introduced. The paper then applies these principles on three well-known metaheuristics, i.e. differential evolution, harmony search and the firefly algorithm. A procedure for the empirical analysis of metaheuristics' time efficiency is then presented. The procedure is then used to empirically analyze the computational cost of the three aforementioned metaheuristics. The pros and cons of the two approaches, i.e. mathematical and empirical analysis, are discussed.

Keywords: *Metaheuristics, Optimization, Time Complexity, Time Efficiency, Big-Oh, Big-Theta.*

Received: 19 November 2023
Accepted: 08 December 2023
Online: 11 December 2023
Published: 20 December 2023

1 Introduction

Metaheuristics are general problem-solving techniques used to find good solutions for complex optimization problems where classical, calculus-based algorithms might be impractical or infeasible [2, 3, 13].

One example of the optimization problems solved by metaheuristic algorithms is the following:

$$\min_{\boldsymbol{x}\in\mathbb{R}^D, L_j\leq x_j\leq U_j; \forall j\in\{1,2,\ldots,D\}} f(\boldsymbol{x}), \tag{1}$$

where $f(\boldsymbol{x})$ is the objective function to be minimized, $\boldsymbol{x}$ is a candidate solution in $\mathbb{R}^D$, and L_j and U_j are the lower and upper bounds, respectively, for the jth component of $\boldsymbol{x}$. The objective is to find $\boldsymbol{x}$ that minimizes $f(\boldsymbol{x})$ while satisfying the boundary constraints.

Metaheuristics have the following advantages:

1. They can be applied to a wide range of optimization problems.
2. They often find reasonable solutions within an acceptable amount of time.
3. They can be used to solve multimodal problems.
4. They are often easy to implement.
5. They do not require optimization problems to be convex and do not make use of derivatives.

Notwithstanding their advantages, they have some major drawbacks too:

1. They do not guarantee finding an optimal solution.
2. Their parameters need fine tuning to the specific problem at hand.
3. Sometimes they take a long time to converge to a good solution.

There are many metaheuristic algorithms for solving optimization problems in the literature. One way to compare them is by the *quality* of their solutions to the optimization problem at hand. Another way is by comparing their *efficiency*. The latter is the main focus of this paper. The computational cost of metaheuristics can be compared before implementing them, i.e. as algorithms, or after implementing them, i.e. as program code. In the case of after implementing the algorithms, researchers need first to implement their algorithm in a programming language (typically Matlab or Python), then run it and study its running time. In the former case where the computational cost is estimated before implementing the algorithm, researchers study instead what lead to the programs, i.e. their algorithms. Thus, the computational cost of a metaheuristic algorithm could be analyzed before even implementing it. Hence, the purpose of analyzing algorithms is not to exactly determine how many seconds (or computer cycles) an algorithm will take. This is not a good measure of the efficiency of an algorithm since running it on a faster computer will not make it, as an algorithm, "better" [8]. The process for obtaining estimates of the time and space required to execute an algorithm is called ***algorithm analysis*** [1].

Time complexity studies the time needed by an algorithm to solve the problem as a function of the size of its input [9]. To estimate the time complexity of an algorithm (a metaheuristic approach in our case), the input size must be identified and the basic operation of the algorithm must be counted. This generally results in a reasonable estimate of the efficiency

of an algorithm and helps in comparing different algorithms with respect to their computational cost. On the other hand, estimation of the actual execution time of an algorithm depends on the hardware being used, the quality of the algorithm implementation and how the program is translated into machine language [1].

This paper provides a guide on:

1. how to determine the time complexity of a metaheuristic,
2. how to empirically determine the computational cost of a metaheuristic implementation,
3. how to compare the efficiency of different metaheuristics, and
4. how to report the above results.

Three well-known metaheursitics are used as representative examples and test cases. Mathematical analysis of the time complexity of metaheuristics is discussed in Section 2. Next, an empirical analysis of the computational cost of the implementation of the algorithms is presented in Section 3. Section 4. concludes the paper.

2 Mathematical Time Complexity Analysis

Time complexity, also called *time efficiency*, is used to quantify how fast an algorithm runs [6]. According to [6], the steps needed to analyze the time complexity of an algorithm are:

1. decide on the parameter(s) that indicate(s) the size of the input(s);
2. identify the basic operation of the algorithm (typically found in the inner-most loop). The ***basic operation*** is the operation contributing the most to the total running time (typically *comparison* or *arithmetic* [8]);
3. check whether the number of times that the basic operation of the algorithm is executed depends only on the size of the input; and
4. set up the sum expressing the number of times that the basic operation of the algorithm is executed.

The above steps are applied to the differential evolution (DE) [10] algorithm (listed in Algorithm 1). DE is a powerful optimization algorithm that belongs to the class of evolutionary algorithms. The algorithm operates by maintaining a population of candidate solutions and iteratively improving them via a process of mutation, crossover and greedy selection. DE employs differential operators to create trial solutions by combining information from different solutions within the population. The simplicity of DE, coupled with its ability to handle both continuous and discrete optimization problems, makes it a popular choice in various domains, including engineering, finance, and machine learning [3]. The steps of computing the time complexity of DE are as follows:

Step 1: There are three main inputs to DE, i.e.

1. G, the number of generations,
2. N, the population size, and
3. D, the problem dimensionality.

Thus, these are the sizes of inputs of the algorithm.

In Step 2, the basic operation needs to be identified. There are two possible candidates:

1. The comparison, i.e. $rand < CR \;||\; i = i_{rand}$ (or the $rand$ operation since it may be more computationally expensive than the comparison operation); or
2. function evaluation.

The first one requires D iterations. The other candidate, i.e. function evaluation, is more difficult to estimate since it depends on the problem (sometimes it needs D iterations, sometimes more). To make the analysis simpler, assume that the function evaluation step also requires D iterations.

Regarding Step 3, the number of basic operations are the same for all inputs of the same size; therefore, in terms of this metric, there is no need to distinguish between the worst, average, and best cases here.

Finally, in Step 4, the time complexity is estimated by setting up a summation:

$$\sum_{k=1}^{G}\sum_{i=1}^{N}\sum_{j=1}^{D} 1 = G \cdot N \cdot D$$

Hence, the time complexity of DE is $\Theta(G \cdot N \cdot D)$. The *population initialization* step requires $\Theta(N \cdot D)$ and thus can be ignored as stated by the following Theorem:

Theorem. If $t_1(n) \in \Theta(g_1(n))$ and $t_2(n) \in \Theta(g_2(n))$, then $t_1(n) + t_2(n) \in \Theta(\max\{g_1(n), g_2(n)\})$. ■

For a proof of the above Theorem, please refer to [6] (p. 56).

Let n be the maximum of G, N and D, then $G \cdot N \cdot D \leq n^3$. In that case, the time complexity of DE is cubic, i.e. $\mathcal{O}(n^3)$. Even though this is less precise than $\Theta(G \cdot N \cdot D)$, it gives an upper bound estimation on the time efficiency of DE, which can be very useful when comparing it with other metaheuristics.

The second example is the harmony search (HS) algorithm [5] (listed in Algorithm 2). HS draws inspiration from the musical improvisation process. It mimics the process of musicians searching for harmonious melodies. In HS, a population of candidate solutions evolves over iterations to improve the harmony or balance between decision variables, ultimately converging towards an "optimal" solution. The search process is guided by three main operators: harmony memory

considering rate, pitch adjusting rate and bandwidth. Harmony search has demonstrated efficacy in solving various optimization problems [4]. Application of the above steps to HS yields the following:

1. The sizes of the inputs are the same as those of DE, i.e. G, N and D.
2. The basic operation is the replacement step where $\boldsymbol{v}$ replaces a member of X.
3. The time complexity depends only the inputs, i.e. the values of the solutions in X.
4. The sum series is

$$\sum_{k=1}^{G}\sum_{i=1}^{N}\sum_{j=1}^{D} 1 = G \cdot N \cdot D$$

In the worst case, the replacement step requires N comparisons (when $\boldsymbol{v}$ is worse than all the solutions in X except the last one). Once a worse solution is found, $\boldsymbol{v}$ replaces the worse solution, which requires replacing the D components of the worse solution with the D components of $\boldsymbol{v}$. Thus, the time complexity of HS is also $\Theta(G \cdot N \cdot D)$. However, if we assume that $n = \max\{G, N, D\}$, then the time complexity is of $\mathcal{O}(n^3)$, as for DE.

The third metaheuristic is the firefly algorithm (FA) [13], which is a nature-inspired optimization algorithm that takes inspiration from the flashing behavior of fireflies in their mating process. It mimics the interactions between fireflies. Each firefly in the algorithm represents a candidate solution, and the attractiveness between fireflies is determined by their fitness values. The algorithm leverages the movement of fireflies towards brighter, more attractive ones to guide the search for "optimal" solutions in the solution space. The time complexity computation steps are applied to FA, giving the following:

1. The sizes of the inputs are the same as those of DE, i.e. G, N and D.
2. The basic operation is the square root operation used in the calculation of the Euclidean distance.
3. The time complexity depends only on the sizes of the inputs.
4. The sum series is

$$\sum_{k=1}^{G}\sum_{i=1}^{N}\sum_{l=1}^{N}\sum_{j=1}^{D} 1 = G \cdot N \cdot N \cdot D$$

Thus, the time complexity of the firefly algorithm is $\Theta(G \cdot N^2 \cdot D)$. Let $n = \max\{G, N, D\}$, then the time complexity is in $\mathcal{O}(n^4)$, which is significantly less efficient than DE and HS.

Table 1 summarizes the time complexity of the three algorithms. Although the big-$\mathcal{O}$ (third column) is less accurate than the big-Θ (second column), it gives a clearer picture of the difference in efficiency between the three algorithms.

Table 1: The time complexity of DE, HS and FA.

Algorithm	Big-Θ	Big-$\mathcal{O}$
DE	$G \cdot N \cdot D$	n^3
HS	$G \cdot N \cdot D$	n^3
FA	$G \cdot N^2 \cdot D$	n^4

Sometimes, the metaheuristic is more complicated like when the population size is not constant and it changes during a run. In that case, the calculation of the summation (i.e. Step 4) is more difficult. One popular way to dynamically change the population size is the one used in the popular DE variant, called L-SHADE [12], where the population size decreases linearly according to the following equation:

$N_{t+1} = \text{round}[((N^{final} - N^{init})/nfe_{max}) \cdot nfe + N^{init}]$,

where N_{t+1} is the population size in the next generation, N^{init} the initial population size, N^{final} is the final population size, nfe is the current number of function evaluations and nfe_{max} is the maximum number of function evaluations.

To make the calculations simpler, let N decrease from G to 1 (or due to the commutative law of addition from 1 to N [8] (p. 16) as shown below), the summation is

$$\sum_{k=1}^{G}\sum_{l=1}^{k}\sum_{j=1}^{D} 1 = \sum_{k=1}^{G}\sum_{l=1}^{k} D = \sum_{k=1}^{G} D \cdot k$$

$$= D \cdot \sum_{k=1}^{G} k = D \cdot \frac{(G \cdot (G+1))}{2}$$

Thus, using the above simplification, the time complexity is still cubic, i.e. $\mathcal{O}(n^3)$.

For non-population-based metaheuristics (e.g. simulated annealing [11]), the time complexity is typically in $\mathcal{O}(n^2)$, i.e. quadratic.

3 Empirical Analysis of Metaheuristics

This section empirically analyzes the time efficiency of the three metaheuristics. All the experiments were conducted on an HP Desktop with a 3.6 GHz Intel Core i7, 32 GB RAM, running Matlab R2017b on MS Windows 10 Pro [1]. Given the fact that the time of a system is generally not accurate, i.e. you may get different running time on repeated runs of the same program using the same input, each experiment is repeated four times and the average running time is reported.

Given that the focus of this paper is in studying the time complexity of a metaheuristic rather than the problem it solves, we decided to use the sphere function defined as

[1] Code is available at https://www.mathworks.com/matlabcentral/fileexchange/156194-time-complexity-of-population-based-metaheuristics

Algorithm 1: Differential Evolution.

Input: Population size N, maximum number of generations G, problem dimension D, mutation factor F, crossover probability CR, objective function f

Output: Best solution $\boldsymbol{x}_{\text{best}}$ and its objective function value

$X \leftarrow$ `InitializePopulation(`N`)`;
$\boldsymbol{x}_{\text{best}} \leftarrow$ the best individual in X;
for $k \leftarrow 1$ **to** G **do**
  for $i \leftarrow 1$ **to** N **do**
    $r_1, r_2, r_3 \leftarrow$ Randomly select 3 distinct indices from $\{1, 2, \ldots, N\}$;
    for $j \leftarrow 1$ **to** D **do**
      if $rand < CR \;||\; i = i_{rand}$ **then**
        $v_j \leftarrow x_{r_1,j} + F(x_{r_2,j} - x_{r_3,j})$;
      else
        $v_j \leftarrow x_{i,j}$
    if $f(\boldsymbol{v}) < f(\boldsymbol{x}_i)$ **then**
      $\boldsymbol{x}_i \leftarrow \boldsymbol{v}$;
return $\boldsymbol{x}_{best}$ *and* $f(\boldsymbol{x}_{best})$;

Algorithm 2: Harmony Search.

Input: Population size N, maximum number of generations G, problem dimension D, Harmony Memory Considering Rate $HMCR$, Pitch Adjusting Rate PAR, objective function f

Output: Best solution $\boldsymbol{x}_{\text{best}}$ and its objective function value

$X \leftarrow$ `InitializePopulation(`N`)`;
$\boldsymbol{x}_{\text{best}} \leftarrow$ the best individual in X;
for $k \leftarrow 1$ **to** G **do**
  for $j \leftarrow 1$ **to** D **do**
    if $rand < HMCR$ **then**
      Let $x_{r,j}$ be the j-th dimension of a randomly chosen member from the harmony memory X and $r \in \{1, 2, \ldots, N\}$;
      $\boldsymbol{v} \leftarrow \boldsymbol{x}_r$;
      if $rand < PAR$ **then**
        Adjust the value of v_j by adding/subtracting a certain amount;
    else
      Generate a new solution, $\boldsymbol{v}$, randomly;
    Accept $\boldsymbol{v}$ if it is better than a member of X;
return $\boldsymbol{x}_{best}$ *and* $f(\boldsymbol{x}_{best})$;

Algorithm 3: Firefly Algorithm.

Input: Population size N, maximum number of generations G, problem dimension D, objective function f

Output: Best solution $\boldsymbol{x}_{\text{best}}$ and its objective function value

$X \leftarrow$ `InitializePopulation(`N`)`;
$\boldsymbol{x}_{\text{best}} \leftarrow$ the best individual in X;
Calculate light intensities by $I_i = f(\boldsymbol{x}_i)$;
for $k \leftarrow 1$ **to** G **do**
  for $i \leftarrow 1$ **to** N **do**
    for $l \leftarrow 1$ **to** N **do**
      if $I_l > I_i$ **then**
        Move firefly i toward l in D dimension via Levy flights;
      Vary attractiveness with the Euclidean distance between firefly i and l;
    Evaluate new solutions;
  Rank the solutions and update x_{best} (if needed);
return $\boldsymbol{x}_{best}$ *and* $f(\boldsymbol{x}_{best})$;

Table 2: The execution time (in seconds) of DE, HS and FA for different values of n. The "na" means not available given that it takes a very long time to run (it should be close to 331,466 seconds, which is 16 times 20716.5979).

n	DE	HS	FA
100	1.6395	2.7132	89.2039
200	12.1905	17.5208	1343.9512
400	90.9248	120.7102	20716.5979
800	720.1308	912.3457	na

$$f(\mathbf{x}) = \sum_{j=1}^{D} x_j^2$$

as the objective function that the metaheuristic algorithms should solve. Using another function should not change our conclusions.

First, the findings of the previous section are further investigated. Herein, $n = G = N = D$, which is not a common setting. However, this allows the confirmation of the big-$\mathcal{O}$ time complexities computed in Section 2. Different values of n are investigated to study the impact of doubling the size of the input on the performance of a metaheuristic. Table 2 shows the results.

The ratios $t(2n)/t(n)$ can be computed to see how the running time reacts to the doubling of its input size. Table 3 shows that the ratio $t(2n)/t(n)$ for DE and HS approaches 8, which confirms their cubic time complexity, while $t(2n)/t(n)$ approaches 16 for FA, also confirming its $\mathcal{O}(n^4)$ time complexity.

Fig. 3 presents the scatter plot of the execution time of the different metaheuristics. The scatter plots help

Table 3: Ratio $t(2n)/t(n)$ of DE, HS and FA for different values of n. The "na" means not available given that it takes very long time to run.

$t(2n)/t(n)$	DE	HS	FA
200/100	7.4355	6.45761	15.0661
400/200	7.45866	6.88954	15.4147
800/400	7.92007	7.55815	na

Table 4: The execution time (in seconds) of DE, HS and FA for different values of n using typical settings.

n	DE	HS	FA
100	16.2669	25.9594	216.3163
200	29.8226	43.7792	405.1537
400	56.3473	78.4623	789.8385
800	111.1285	145.1426	1588.9069

in ascertaining the probable efficiency class of an algorithm. The plots show that both DE and HA seem to belong to the same efficiency class, i.e. cubic, while FA belongs to a different class, i.e. $\mathcal{O}(n^4)$.

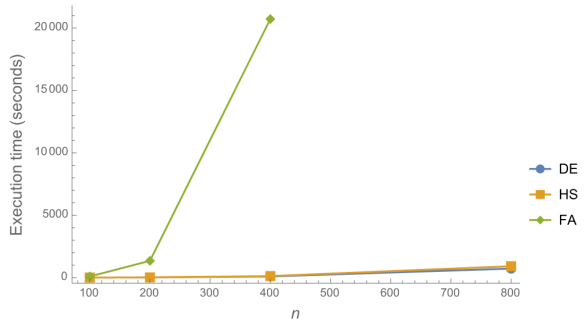

Figure 1: Execution time of DE, HS and FA on different sizes of n.

Next, the performance of the three metaheuristics on typical values of G, N and D is evaluated. For fair comparison, the nfe_{max} is fixed to 100000 function evaluations and $D \in \{100, 200, 400, 800\}$. For DE $G = 1000$, $N = 100$; for HS, $G = 100000$ and $N = 25$; and for FA, $G = 4000$ and $N = 25$.

The running time of the different metaheuristics is reported in Table 4. Table 5 shows that the ratio $t(2n)/t(n)$ for the three algorithms approaches 2. Thus, the three algorithms exhibit linear behavior with increase in D. Fig. 2 confirms this observation.

After studying the individual time efficiency of each algorithm, the way different metaheuristics are compared is explained. The ***speedup*** metric used is defined as $\frac{\text{execution time of } B}{\text{execution time of } A}$, where A is the algorithm we are interested in (typically a newly-proposed metaheuristic) and B a competitive algorithm. If the speedup is greater than 1, then A runs faster, otherwise B is at least as fast as A.

For example, let A be the HS (since it is faster than FA and slower than DE). Table 6 summarizes the results, it shows that HS is slower than DE by a factor of 1.43, i.e. 43% slower. On the other hand, HS is faster than FA by a factor of 9.65. Now, one may ask why

Table 5: Ratio $t(2n)/t(n)$ of DE, HS and FA for different values of n using typical settings.

$t(2n)/t(n)$	DE	HS	FA
200/100	1.83333	1.68645	1.97221
400/200	1.88942	1.79223	1.84984
800/400	1.97221	1.94948	2.01169

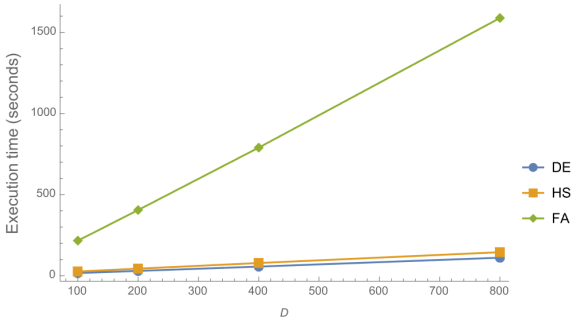

Figure 2: Execution time of DE, HS and FA on different sizes of D using typical values for the parameters.

DE is faster than HS even though both are of the same efficiency class? The answer is because of the constant factor ignored when using the big-$\mathcal{O}$/big-Θ notation.

An alternative way to empirically analyze the time efficiency of a metaheuristic is by determining its ***algorithmic overhead***. The algorithmic overhead is computed as the total running time of the metaheuristic minus the time required to perform nfe_{max} function evaluations. This helps in separating the cost of the objective function calculation from that of the algorithm. Thus, the procedure of computing the algorithmic overhead is:

1. calculate the time required to perform nfe_{max} function evaluations (in our case the function is the sphere function), denoted as T,

2. calculate the running time of a metaheuristic using nfe_{max} function evaluations, referred to as T_m, where m is the metaheuristic name, and then

3. compute $T_m - T$, which is the algorithm overhead.

To obtain more reliable estimations, T and T_m are averaged over several runs (4 runs in our case).

Table 7 summarizes the results of using the above procedure for DE, HS and FA on the sphere function. It can be seen that T is very small compared to T_m; hence, our conclusions do not change from the method

Table 6: The speedup of HS to DE and FA for different values of n using typical settings.

n	DE/HS	FA/HS
100	0.626629	8.33286
200	0.681205	9.25448
400	0.718145	10.0665
800	0.76565	10.9472

Table 7: Comparison of the algorithmic overhead for DE, HS and FA.

n	T	$T_{DE} - T$	$T_{HS} - T$	$T_{FA} - T$
100	0.2327	16.0342	25.7267	216.083
200	0.2455	29.5771	43.5337	404.908
400	0.2792	56.0681	78.1831	789.559
800	0.3445	110.784	144.798	1588.57

that does not exclude the cost of computung the objective function. The algorithmic overhead is visualized in Fig. 3.

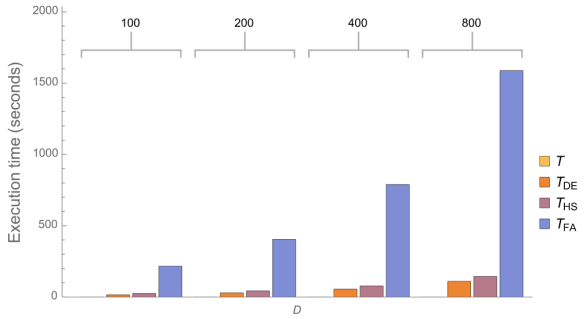

Figure 3: Average algorithmic overhead (in seconds) for DE, HS and FA, on increasing values of D.

4 Conclusions

The mathematical and empirical analysis of the time complexity of three metaheuristic algorithms are outlined and explained. To demonstrate the application of the introduced procedures for analyzing the time efficiency of algorithms, we applied them to three well-known metaheuristics, i.e. differential evolution, harmony search and the firefly algorithm. The main strength of the mathematical analysis is its independence of specific inputs. Its main limitation is its applicability and difficulty of estimating the average case. The main advantage of the empirical analysis is its applicability to any algorithm. However, empirical analysis depends on the implementation, hardware and the specific inputs used [6].

This paper shows the steps needed to compute the time complexity of a metaheuristic. It explains how to empirically determine the time efficiency of a metaheuristic and how to compare the computational cost of different metaheuristic algorithms. Moreover, metaheuristics with a time complexity more than $\mathcal{O}(n^3)$, as in the firefly algorithm, are not appropriate for solving relatively high-dimensional problems. Finally, it is useful to analyze the computational cost of a metaheuristic both mathematically and empirically as demonstrated in the case of DE and HS, where even though both have the same time complexity, the running time of DE is clearly less than that of HS. This shows that although time complexity "*captures the high-level performance of an algorithm, but constants matter too!*" [7].

References

[1] Chartrand, G., and Zhang, P. *Discrete Mathematics*. Waveland Press, Inc., 2011.

[2] Chopard, B., and Tomassini, M. *An Introduction to Metaheuristics for Optimization*. Springer, 2018.

[3] Engelbrecht, A. *Computational Intelligence: An Introduction*, 2 ed. Wiley, 2007.

[4] Gritli, H., Dubey, M., Kumar, V., Kaur, M., and Dao, T.-P. A systematic review on harmony search algorithm: Theory, literature, and applications. *Mathematical Problems in Engineering* (2021).

[5] Kim, J. H. Harmony search algorithm: A unique music-inspired algorithm. *Procedia Engineering 154* (2016), 1401–1405. 12th International Conference on Hydroinformatics (HIC 2016) - Smart Water for the Future.

[6] Levitin, A. *Introduction to the Design and Analysis of Algorithms*, 3 ed. Pearson, 2011.

[7] Liben-Nowell, D. *Connecting Discrete Mathematics and Computer Science*, 2 ed. Cambridge University Press, 2022.

[8] McConnell, J. *Analysis of Algorithms: An Active Learning Approach*, 2 ed. Jones and Bartlett Publishers, 2008.

[9] Rouggarden, T. *Algorithms Illuminated: Omnibus Edition*, 1 ed. SOUNDLIKEYOURSELF PUBLISHING, 2022.

[10] Storn, R., and Price, K. Differential evolution-a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, Berkeley: ICSI, 1995.

[11] Suman, B., and Kumar, P. A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the Operational Research Society 57*, 10 (2021), 1143–1160.

[12] Tanabe, R., and Fukunaga, A. Improving the search performance of SHADE using linear population size reduction. In *Congress on Evolutionary Computation (CEC)* (2014), IEEE, pp. 1658–1665.

[13] Yang, X.-S. *Nature-inspired optimization algorithms*. Elsevier, 2014.