Comparing genetic optimisation algorithms: scratch developed algorithm against published algorithm

André Sebastián Cóndor [*]

September 2022

Abstract

Genetic algorithms are a metaheuristic tool for finding solutions to optimisation problems. There are several methods and approaches to develop and implement a genetic algorithm. The aim of this report is to explain the creation of a genetic algorithm from scratch and to make a comparison with a published algorithm.

Keywords: Metaheuristics, genetic algorithm, compare genetic algorithms.

1. Introduction

Formerly, processes had to be run through a real system to test prototypes and make improvements. As computers developed, interested people started to simulate models on computers. This meant that a real experiment was no longer necessary, thus cost and time were reduced. Examination of the simulations results conceded a better understanding of the models and provided insights into locating optimal model parameter values.

These modelling methods gained popularity and came to be applied in various fields, such as physics, industry production and engineering. Real-world optimisation problems necessitate consideration of complex issues such as: computational cost, constraints, multiple objectives and uncertainty. In order to achieve these requisites, heuristic algorithms approximate a solution when classical methods fail finding any optimal solution. Likewise, metaheuristic algorithms perform better than heuristic algorithms by applying a trade-off between local search and global exploration (Gandomi et al., 2013).

Metaheuristic algorithms are generally divided into single-solution and population-based algorithms. Single-solution algorithms use one candidate solution, which is improved by local search. Population-based algorithms utilize multiple candidate solutions, promoting diversity to avoid local optima (Katoch et al., 2021). Evolutionary Algorithms (EA) are part of the population-based metaheuristic algorithms.

EAs became popular tools in finding the optimal solutions for optimisation problems (Mirjalili, 2019). In broad terms, EAs are initialized with a set of solutions, which is subsequently evaluated by the objective function(s) and modified to maintain the best solutions minimising or maximising the objective(s).

Genetic Algorithms (GA) are a branch of EAs. GAs were inspired from the Darwinian theory of evolutionary. GAs simulate the survivance of fittest individuals in a population via processes of selection, crossover and mutation (Mirjalili, 2019).

This research aims to compare a GA created from scratch with a GA already implemented and published. To achieve this, the basic flowchart of a genetic algorithm will be presented. Subsequently, the functions of the algorithm to be developed will be explained. Finally, the algorithm produced will be compared with the published algorithm.

Both algorithms will be implemented in Python. The Distributed Evolutionary Algorithms in Python (DEAP) package will be used as source for the published algorithm.

2. Background

2.1. Genetic algorithm flowchart

Fig. 1 illustrates the flowchart of a GA. Once the algorithm begins, the first step is to randomly select a starting population, a chromosome-like data group of individuals (Mathew, 2012). Each individual corresponds to a chromosome. Concomitantly, each chromosome holds the individual's information in an encoded format consisting of various genes (Lambora et al., 2019). Chromosomes are considered as points in the solution space $\mathcal{X}$ (Katoch et al., 2021).

Second, the set of individuals is evaluated using the objective function(s). The fitness of each chromosome is computed.

Third, the algorithm enters a buckle conditioned by the stopping criteria. The steps listed inside the loop will be performed until the criteria is satisfied. Every time the algorithm goes through the loop, a new generation is constructed. Generally, the stopping criteria is defined as the desired total number of generations.

Fourth, the GA chooses the parents: some of the best individuals of a certain population. The goal is to assure the predominance of the best individual's features in the following generation, looking forward to even better individuals. The parents can be selected through several mechanisms. This procedure is extremely important as the best solutions should be more likely to be chosen as their pick probability is directly proportional to their fitness/objective values (Kumar et al., 2010). The local optima issues are overcomed by the possibility of choosing poor solutions as well, and the diversity of individuals that this entails.

Fifth, the crossover operator is employed to generate offspring from the selected parents. In this step, parents' chromosomes exchange information, resulting in a new gene combined chromosome (Katoch et al., 2021). A new population is stated as a group of children solutions.

Sixth, mutation introduces new random information for each child's chromosome. This enhances escaping from a local optima, since information diversity expands the solution space. Technically, mutation aims to propose new chromosomes inside the current chromosome's neighbourhood (Mathew, 2012). Mutation is the second operator of a GA.

[*]*E-mail address:* a.sebastian.condor@gmail.com (A.S. Cóndor).

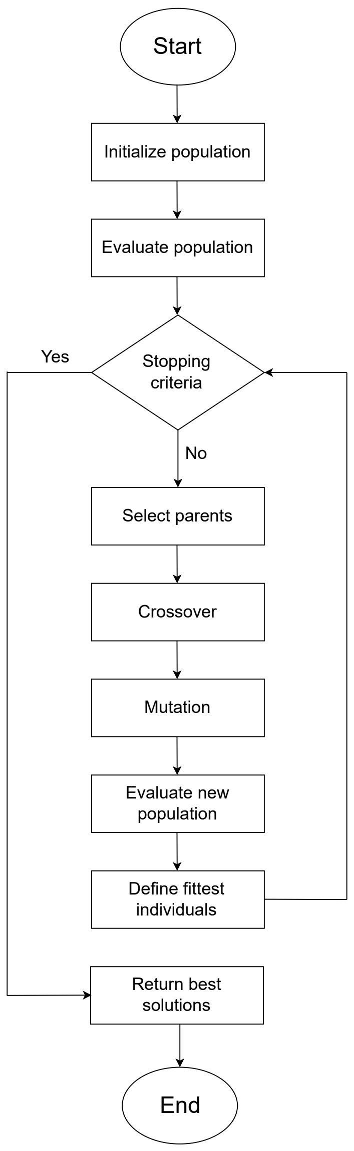

Fig. 1. The GA starts with an initial populated and evaluates it. Then, the loop steps will be executed until the stopping criteria is met. The best individuals are returned at the end.

Seventh, the new mutated population is evaluated on the objective function(s).

Eighth, the objective values from the previous step function as basis to pick the greatest individuals. The fittest individuals are identified and assembled into another new best population of the corresponding generation.

From the second to the eighth, the looped flowchart steps are repeated while the stopping criteria is unsatisfied. Finally, when this constraint is accomplished, the GA returns the best individuals found through the whole process.

3. Optimisation problem

For the sake of comparing the algorithms, it is essential to propose an optimisation problem and use both of the algorithms to solve it. Then, the solutions given after running the algorithms will be contrasted.

3.1. *Product storage problem*

A hypothetical furniture shop has to determine which of the furnishings will be stored in a small humidity-controlled warehouse. Products left outside the warehouse are damaged by complications caused by the weather. All the furnitures have an assigned price and volume. The stored products will be sold.

The priority of the problem is to maximize the sales profit, deliberating that the warehouse has a maximum space capacity of $6.5m^3$.

There are 11 products. Table 1 states the attributes of each product.

	Volume (m^3)	Price
Sofa	1.045	955.33
Armchair	0.833	839.7
Bed	3.672	1029.51
Mattress	0.017	1440.12
Wardrobe	1.193	1245.85
Dressing table	0.356	695.47
Light pendant	0.053	435.33
Picture	0.007	121.99
Mirror	0.024	185.77
Rug	0.012	54.12
Bookcase	0.248	615.17

Table 1. This table contains the respective price and volume for each product. The total volume of the inventory is of $7.46m^3$ and its total price is 7618.36.

Indeed, the optimisation problem is a single objective maximisation problem with the warehouse capacity as a unique constraint.

The subsequent section is fully oriented in the creation of the GA from scratch. The different algorithms designed for each of the flowchart steps are broken down.

4. From flowchart to implemented algorithm

This section expands the description of the designed algorithm. The section will be divided by each step of the flowchart of Section 2.1.

4.1. *Initialize population*

An individual is a solution to the optimisation problem. Recalling that an individual is represented as a chromosome γ, each chromosome will have j number of genes α. Hence, a chromosome is defined as:

$$\gamma_j = \begin{bmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_j \end{bmatrix} \tag{1}$$

Having A as the event that certain product is stored, γ will have a binary α cipher where $1 \implies A$ and $0 \implies \bar{A}$, i.e.:

$$\alpha_j = \begin{cases} 1 & \text{if } A \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

A set of various γ_j conforms to a population $\delta_{i,j}$; with j as the number of genes of each γ, and i as the number of individuals in a specific δ. Therefore, a population is constructed as follows:

$$\delta_{i,j} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \dots \\ \gamma_j \end{bmatrix} = \begin{bmatrix} \alpha_{1,1} & \alpha_{2,1} & \dots & \alpha_{i,1} \\ \alpha_{1,2} & \alpha_{2,2} & \dots & \alpha_{i,2} \\ \dots & \dots & \dots & \dots \\ \alpha_{i,1} & \alpha_{i,2} & \dots & \alpha_{i,j} \end{bmatrix} \tag{3}$$

Since there are 11 products, and it is possible to formulate A or $\bar{A}$ for each of these, j is equal to 11. The binary value of each α_j is randomly determined. The population size i is a hyperparameter defined by the user.

4.2. *Evaluate population*

The objective function f is crucial for rating how well a certain γ solves the problem. f is also used later in the GA to choose parent solutions.

Algorithm 1 receives a chromosome γ, its genes $\alpha_{j,\gamma}$, and its respectives prices $\rho_{j,\gamma}$ and volumes $\phi_{j,\gamma}$. When certain $\alpha_{j,\gamma}$ happens to be elected for A, the objective value of γ is added $\rho_{j,\gamma}$ and the used space ϕ is also added $\phi_{j,\gamma}$.

For accounting the warehouse space limitation, $\theta = 6.5$ and behaves as a constraint. If the space used by the chromosome is grater than the limit, $\iota_\gamma > \theta$, the chromosome receives a penalization on its objective values as a decrement, i.e. $\eta_\gamma = 1$.

Algorithm 1: Objective function f.

```
Input  : $\gamma$: A chromosome
         $\alpha_{j,\gamma}$: All $j$ genes in $\gamma$
         $j$: Number of genes in $\gamma$
         $\theta$: Warehouse maximum capacity as a constraint
         $\rho$: Corresponding price of $\alpha_{j,\gamma}$
         $\phi$: Corresponding volume of $\alpha_{j,\gamma}$
Output: Objective value of $\gamma$.
1  $\eta_\gamma \leftarrow 0$                    // Initialize objective value to zero
2  $\iota_\gamma \leftarrow 0$                    // Initialize used space to zero
3  $\theta \leftarrow 6.5$                        // Capacity constraint
4  for $n \leftarrow 0$ to $j$ do
5      if $\alpha_{n,\gamma} == 1$ then
6          $\eta_\gamma \leftarrow \eta_\gamma + \rho_n$
7          $\iota_\gamma \leftarrow \iota_\gamma + \phi_n$
8  if $\iota_\gamma > \theta$ then      // Penalization for constraint violation
9      $\eta_\gamma \leftarrow 1$
10 return $\eta_\gamma \wedge \iota_\gamma$
```

4.3. *Stopping criteria*

As explained in Section 2.1, each time the loop is performed, a new generation arises. For this GA in particular, the number of generations ω will serve as the stopping criteria. ω should be stated by the user.

4.4. *Select parents*

Parents are used to generate better individuals over the generations. For selecting those γ that will be converted into parents, this GA uses the *Roulette Wheel Method*. This approach seeks for weighting the like hood of a γ to be chosen proportionally to its objective value η_γ (Chudasama et al., 2011). As the optimisation problem is concerning maximisation, the higher the objective value, the more likely the chromosome is to be chosen as a parent. However, the other individuals also have a small chance of being chosen, thus embracing the diversity of solutions and avoiding falling into a local optima.

For example, considering a population of i individuals, we will have i objective values. Assuming that the best chromosome is the first one and that the objective values go in descending order[1]; the first chromosome γ_1 will have the highest probability of being chosen, and the last chromosome γ_i will have the lowest probability of being parent.

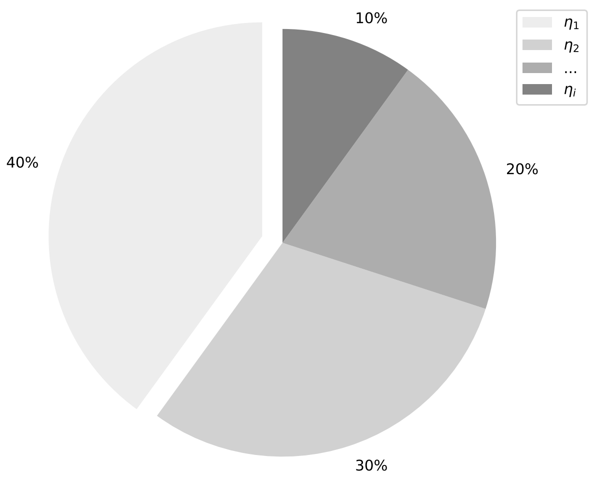

Fig. 2. The pie chart represents the probability of γ_i being a parent. This probability is proportional to η_i.

Fig 2 shows the roulette of the above hypothetical case. Each time the roulette wheel is spun, the best individual is more likely to be a parent. The roulette wheel helps diversity by giving all chromosomes a probability to be chosen.

4.5. *Crossover*

Once the parents are available, the crossover genetic operator is applied. Crossover combines genes from two parents to form a better child. The offspring chromosomes are pooled and a new fitter population is formed.

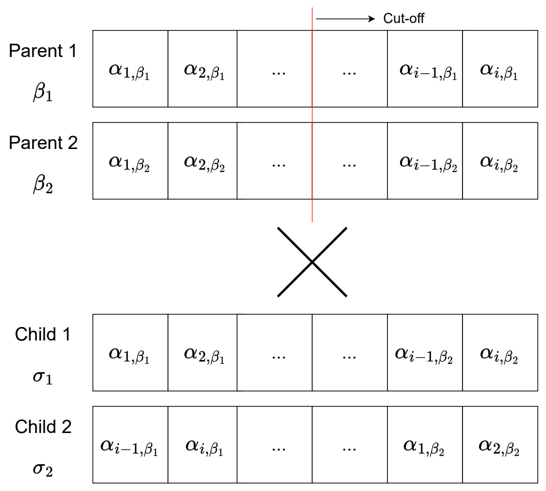
Cut-off
Parent 1
β_1
α_{1,β_1} α_{2,β_1} ... ... α_{i-1,β_1} α_{i,β_1}
Parent 2
β_2
α_{1,β_2} α_{2,β_2} ... ... α_{i-1,β_2} α_{i,β_2}
Child 1
σ_1
α_{1,β_1} α_{2,β_1} ... ... α_{i-1,β_2} α_{i,β_2}
Child 2
σ_2
α_{i-1,β_1} α_{i,β_1} ... ... α_{1,β_2} α_{2,β_2}

Fig. 3. Crossover graphical description. Children are generated crossing the parents' genes divided by the cut-off point.

It takes two parents $\beta_1 \wedge \beta_2$ to father a daughter chromosome σ. A point or index at which the two parental chromosomes will be split is chosen at random, known as a cut-off ν. The chromosomes of the parents are split in two each, it

[1] $\eta 1 > \eta 2 > ... > \eta i$.

does not matter if the division does not split the chromosome symmetrically.

This GA will create two new individuals $\sigma_1 \wedge \sigma_2$ from each pair of parents. The crossover will combine the left genes of β_1 with the right genes from β_2 in order to craft σ_1. On the other hand, right genes from β_1 and left genes from β_2 will conform σ_2.

Fig 3 illustrates the process outlined above. It is important to emphasise that the cut-off is determined randomly and must be within the range of number of genes, i.e. $\nu \in [0, i]$.

4.6. *Mutation*

Mutation m is the second genetic operator. Mutation aggregates diversity when randomly changes genes in a specific chromosome (Mathew, 2012). As in nature, mutation is very less common than crossover. m is governed by a mutation probability p_m. This probability is usually a low value, for instance $p_m < 0.1$.

Algorithm 2: Mutation m.

```
Input  : $\gamma$: A chromosome
         $p_m$: Mutation probability
         $\alpha_{j,\gamma}$: All $j$ genes in $\gamma$
         $j$: Number of genes in $\gamma$
Output: Mutated chromosome $\gamma^*$.
1 for $k \leftarrow 0$ to $j$ do
2     if $\zeta < p_m$ then                    // $\zeta$: random generated number
3         if $\alpha_{k,\gamma} == 1$ then
4             $\alpha_{k,\gamma} = 0$
5         else
6             $\alpha_{k,\gamma} = 1$
7 return $\gamma^*$
```

Algorithm 2 describes the m in the developed GA. When a random number ζ is less than p_m, a 1-valued α_i will mutate and take on a value of 0.

4.7. *Evaluate new population*

After crossover and mutation, a new population δ_j^* has been created as a group of changed γ^*'s. The new population is as follows:

$$\delta_j^* = \begin{bmatrix} \gamma_1^* \\ \gamma_2^* \\ \dots \\ \gamma_j^* \end{bmatrix} \tag{4}$$

Algorithm 1 is executed again, but with δ_j^* as input. Naturally, this will return a new objective value η_γ^* and used space ι_γ^* of δ_j^*.

4.8. *Define fittest individuals*

Once having δ_j^*, the algorithm will choose the fittest γ^*'s looking for the highest η_γ^*'s as this is a maximisation problem. This improved population ξ will then return to the loop and, if ω has not been satisfied, ξ will pass through all the four previous steps.

4.9. *Return best solutions*

Finally, when ω is met, the GA will return the best population $\xi_{\omega,j}$ found after the evolution of ω generations. The best population decipher by the GA is defined as:

$$\xi_\omega = \begin{bmatrix} \gamma_{1,\omega} \\ \gamma_{2,\omega} \\ \dots \\ \gamma_{j,\omega} \end{bmatrix} \tag{5}$$

5. DEAP algorithm

DEAP stands for Distributed Evolutionary Algorithms in Python. DEAP was presented at The Genetic and Evolutionary Computation Conference 2012, this package is commonly used among researchers from different branches (Kim and Yoo, 2019). This library requires prior knowledge of Python.

Since the library is built on top of Python, it can be used in combination with other Python libraries. In addition, it becomes a highly adaptable tool to any specific need the user may have.

The general structure handled by DEAP is divided between the *toolbox* and the *creator* (Fortin et al., 2012). The *creator*, broadly speaking, it receives data and functions that are used as primary resources for the problem to be optimised.On the other side, the *toolbox* saves all the operators the user will be applying to the GA.

5.1. *Toolbox*

The first feature that was saved in the toolbox is the representation of γ_j. As explained in Section 4.1, γ_j will be a set of 11 binary numbers (1 or 0); where 1 means that the product was stored and 0 means that the product was not stored.

A population was initialised and subsequently evaluated with the f function explained at Algorithm 1. The mutation probability for this distributed GA example was set to 0.1 and the tools.mutFlipBit mutation method was used. As stated at Section 4.4, this distributed GA example will also use the roulette method to select parents, this was achieved by passing the tools.selRoulette attribute to the toolbox.

5.2. *Creator*

For the creator, the problem was explicitly defined as a maximisation problem. Individuals were also created and will be tested according to the f objective function.

The DEAP algorithm was configured to perform the GA using a 200 population size and 5000 generations.

6. Comparing algorithms

The comparison of the algorithms was based on the results of the objective function and the space occupied for a certain profit.

It is known that there are certain hyperparameters set by the user. Table 2 lists the operators and parameters used for each of the GA's.As can be seen, the two algorithms were run using the same hyperparameters.

	Scratch GA	Distributed GA
Population size (i)	200	200
Mutation probability (p_m)	0.1	0.1
Number of generations (ω)	5000	5000
Constraint (θ)	≤ 6.5	≤ 6.5

Table 2. This table shows that the same hyperparameters and the constraint used for both algorithms.

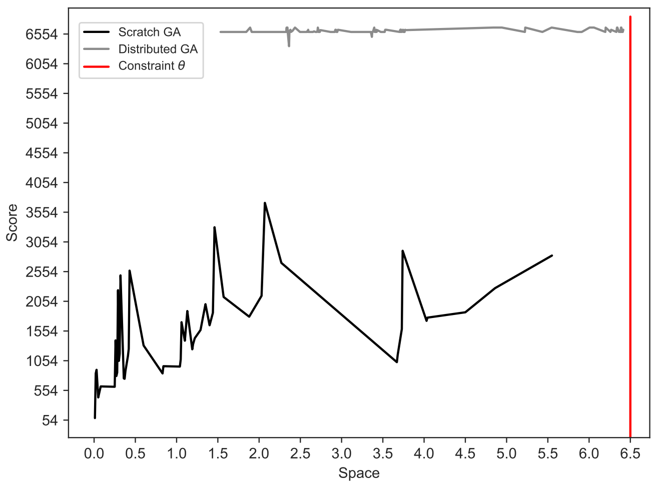

Fig. 4. Line plot showing the objective value of the bests populations of each generation for both the scratch and distributed algorithm. The red line represents θ, i.e. the maximum space of the warehouse.

In order to make the comparison of results more intuitive and understandable, a line plot was made with the horizontal axis as the space used and the ordinate axis as the target value of the best individuals found by each of the algorithms.

Fig 4 illustrates the results of the two algorithms. Clearly, the distributed algorithm performs better compared to the algorithm developed from scratch, in terms of the target value. The algorithm written from scratch tends to vary its target values considerably, which is why its movement within the graph generates peaks followed by sharp rises or falls. DEAP maintains regularity in its results, this may be due to the fact that its method of population initialisation is better and therefore, from the first individuals, solutions with a high objective value are extracted. In terms of the constraint, the developed algorithm does not come close to the constraint, hence it does not use all the available space in the warehouse. In contrast, the distributed algorithm does tend towards the constraint and, therefore, makes more efficient use of the available space.

The solutions found by DEAP range between 6300. On the contrary, the results of the developed algorithm find their solutions within a range of gains between 54 and 3500 approximately.

Fig 5 is a box plot which allows for a comparison of the objective values produced by each of the algorithms. The algorithm created in this report provides responses to the f function from 100 to a maximum outlier of approximately 4000. The median of this algorithm is 1200. The DEAP algorithm has better individuals. Its answers range between 6300 and 6700. The median is towards the third quartile. There are no outliers.

Reproducibility. All codes, datasets and scripts used for this report are fully available at repository.

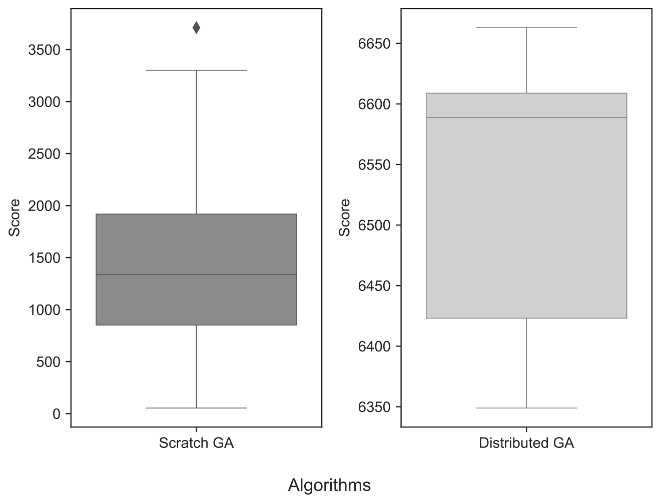

Fig. 5. Boxplot of objective values from the two compared GA's.

7. Conclusions and future work

After observing the performance of both algorithms, it can be concluded that the DEAP algorithm generates individuals with a better performance than those generated by the algorithm developed from scratch. This may be due to a higher complexity behind the *toolbox* and the *creator* provided in DEAP.

It is important to note that the hyperparameters were the same for both algorithms. The configurations made in the *toolbox* and the *creator* were chosen based on the similarity of the processes detailed in Section 4.

Although the results are not similar, it is valuable that these differences have been found. This work emphasises the importance of the use of distributed algorithms due to their quality but, at the same time, proposes new research topics such as a sensitivity analysis when changing hyperparameters or even the methods of genetic operators.

Acknowledgement

Thanks are due to all those who contributed to making this report possible. Manuel López-Ibañez, Carlos M. Fonseca and Leslie Pérez Cáceres for their knowledge and inspiration provided at the SIGEVO Summer School 2022. Special thanks to Juan Esteban Diaz for his mentoring in the field of optimisation.

References

Chudasama, C., Shah, S., and Panchal, M. (2011). Comparison of parents selection methods of genetic algorithm for tsp. *International conference on computer communication and networks CSI-COMNET-2011, Proceedings*, 85(1):87. https://www.ijcaonline.org/proceedings/comnet/number1/5431-1019.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M., and Gagné, C. (2012). Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175. https://dl.acm.org/doi/10.5555/2503308.2503311.

Gandomi, A. H., Yang, X.-S., Talatahari, S., and Alavi, A. H. (2013). Metaheuristic algorithms in modeling and optimization. *Metaheuristic applications in structures and infrastructures*, 1(1):1–5. http://dx.doi.org/10.1016/B978-0-12-398364-0.00001-2.

Katoch, S., Chauhan, S. S., and Kumar, V. (2021). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126. https://doi.org/10.1007/s11042-020-10139-6.

Kim, J. and Yoo, S. (2019). Software review: Deap (distributed evolutionary algorithm in python) library. *Genetic Programming and Evolvable Machines*, 20(1):139–142. https://doi.org/10.1007/s10710-018-9341-4.

Kumar, M., Husain, D., Upreti, N., Gupta, D., et al. (2010). Genetic algorithm: Review and application. *International Journal of Information Technology and Knowledge Management*, 2(2):451–454. https://dx.doi.org/10.2139/ssrn.3529843.

Lambora, A., Gupta, K., and Chopra, K. (2019). Genetic algorithm-a literature review. *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, 1:380–384. https://doi.org/10.1109/COMITCon.2019.8862255.

Mathew, T. V. (2012). Genetic algorithm. *Report submitted at IIT Bombay*. http://datajobstest.com/data-science-repo/Genetic-Algorithm-Guide-[Tom-Mathew].pdf.

Mirjalili, S. (2019). Genetic algorithm. *Evolutionary Algorithms and Neural Networks*, 780:43–55. https://doi.org/10.1007/978-3-319-93025-1_4.