AutoOptLib User Guide

Qi Zhao*

May 18, 2023

1 Quick Start

AutoOptLib is a MATLAB library for automatically designing metaheuristic algorithms for solving optimization problems. Users can use, redistribute, and modify it under the terms of the GNU General Public License v3.0. Please reference the following papers if using AutoOptLib in your research:

- Zhao Q, Yan B, Chen X, et al. AutoOpt: A General Framework for Automatically Designing Metaheuristic Optimization Algorithms with Diverse Structures. arXiv preprint arXiv:2204.00998, 2022.
- Zhao Q, Yan B, Hu T, et al. AutoOptLib: A Library of Automatically Designing Metaheuristic Optimization Algorithms in MATLAB. arXiv preprint arXiv:2303.06536, 2023

AutoOptLib is downloadable at `https://github.com/qz89/AutoOpt`. MATLAB R2018 or higher versions are recommended for using AutoOptLib. MATLAB R2020a or higher versions are required for invoking AutoOptLib's graphic user interface (GUI). Following the steps below to use AutoOptLib:

1. Download AutoOptLib and add it to MATLAB path.
2. Implement the targeted optimization problem.
3. Define the space for designing algorithms.
4. Run AutoOptLib by command or GUI.

2 Implement Problem

Users can implement their targeted optimization problem according to the template `prob_template.m` in the `/Problems` folder. `prob_template.m` has three main cases. Case `'construct'` is for setting problem properties and loading the input data. In particular, line 7 defines the problem type, e.g., `Problem.type = {'continuous','static','certain'}` refers to a continuous static problem without uncertainty in the objective function. Lines 10 and 11 define the lower and upper bounds of the solution space. Lines 18 and 21 offer specific settings as indicated in the comments of lines 14-17 and 20, respectively. Line 25 or 26 is for loading the input data. As a result, problem proprieties and data are saved in the `Problem` and `Data` structs, respectively.

```
1 case 'construct' % define problem properties
2     Problem = varargin{1};
3     % define problem type in the following three cells.
4     % first cell : 'continuous'\'discrete'\'permutation'
5     % second cell: 'static'\'sequential'
6     % third cell : 'certain'\'uncertain'
7     Problem.type = {'','',''};
8
9     % define the bound of solution space
```

*Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China. Email: zhaoq@sustech.edu.cn.

```
    lower = []; % 1*D, lower bound of the D-dimension decision space
    upper = []; % 1*D, upper bound of the D-dimension decision space
    Problem.bound = [lower;upper];

    % define specific settings (optional), options:
    % 'dec_diff'          : elements of the solution should be different w.r.t each other for discrete problems
    % 'uncertain_average': averaging the fitness over multiple fitness evaluations for uncertain problems
    % 'uncertain_worst'   : use the worse fitness among multiple fitness evaluations as the fitness for uncertain problems
    Problem.setting = {''}; % put choice(s) into the cell

    % set the number of samples for uncertain problems (optional)
    Problem.sampleN = [];
    output1 = Problem;

    % load/construct data file in the following
    Data = load(''); % for .mat format
    % Data = readmatrix('','Sheet',1); % for .xlsx format
    output2 = Data;
```

Case `'repair'` is for repairing solutions to keep them feasible, e.g., keeping the solutions within the box constraint. Lines 2 and 3 input the problem data and solutions (decision variables). Programs for repairing solutions should be written from line 5. Finally, the repaired solutions will be returned.

```
1 case 'repair' % repair solutions
2     Data = varargin{1};
3     Decs = varargin{2};
4     % define methods for repairing solutions in the following
5 
6     output1 = Decs;
```

Case `'evaluate'` is for evaluating solutions' fitness (objective values penalized by constraint violations). In detail, lines 2 and 3 input the problem data and solutions. The targeted problem's objective function should be written from line 6. Constraint functions should be written from line 8. Constraint violation can be calculated in line 10 by [JD13]:

$$CV(\mathbf{x}) = \sum_{j=1}^{J} \langle \overline{g}_j(\mathbf{x}) \rangle + \sum_{k=1}^{K} |\overline{h}_k(\mathbf{x})|,$$

where $CV(\mathbf{x})$ is the constraint violation of solution $\mathbf{x}$; $\overline{g}_j(\mathbf{x})$ and $\overline{h}_k(\mathbf{x})$ are the jth normalized inequality constraint and kth normalized equality constraint, respectively, in which the normalization can be done by dividing the constraint functions by the constant in this constraint present (i.e., for $g_j(\mathbf{x}) \geq b_j$, the normalized constraint function becomes $\overline{g}_j(\mathbf{x}) = g_j(\mathbf{x})/b_j \geq 0$, and similarly $\overline{h}_k(\mathbf{x})$ can be normalized equality constraint); the bracket operator $\langle \overline{g}_j(\mathbf{x}) \rangle$ returns the negative of $\overline{g}_j(\mathbf{x})$, if $\overline{g}_j(\mathbf{x}) < 0$ and returns zeros, otherwise. During solution evaluation, accessory (intermediate) data for understanding the solutions may be produced. This can be written from line 12. Finally, the objective values, constraint violations, and accessory data will be returned by lines 13-15.

```
1 case 'evaluate' % evaluate solution's fitness
2     Data = varargin{1}; % load problem data
3     Decs = varargin{2}; % load the current solution(s)
4 
5     % define the objective function in the following
6 
7     % define the inequal constraint(s) in the following, equal constraints should be transformed to inequal ones
8 
9     % calculate the constraint violation in the following
10 
```

```
    % collect accessory data for understanding the solutions in the following (optional)

    output1 = ; % matrix for saving objective function values
    output2 = ; % matrix for saving constraint violation values (optional)
    output3 = ; % matrix or cells for saving accessory data (optional), a solution's accessory data should be saved in a row
```

Examples of problem implementation can be seen in the CEC 2005 benchmark problem files in the `/Problems/CEC2005 Benchmarks` folder. The implementation of a real constrained problem `beamforming.m` is given in the `/Problems/Real-World/Beanforming` folder.

3 Define Design Space

AutoOptLib provides over 40 widely-used algorithmic components for designing algorithms for continuous, discrete, and permutation problems. Each component is packaged in an independent `.m` file in the `/Components` folder. The included components are listed in Table 1.

The default design space for each type of problems covers all the involved components for this type. Users can either employ the default space with the furthest potential to discover novelty or define a narrow space in `Space.m` in the `/Utilities` folder according to interest. For example, when designing algorithms for continuous problems, the candidate Search components can be set by collecting the string of component file name in line 3. More components can be added, which will be detailed in Section 1.

```
1  case 'continuous'
2      Choose = {'choose_traverse';'choose_tournament';'choose_roulette_wheel';'choose_brainstorm';'choose_nich'};
3      Search = {'search_pso';'search_de_current';'search_de_current_best';'search_de_random';'cross_arithmetic';'cross_sim_binary';'cross_point_one';'cross_point_two';'cross_point_uniform';'search_mu_gaussian';'search_mu_cauchy';'search_mu_polynomial';'search_mu_uniform';'search_eda';'search_cma';'reinit_continuous'};
4      Update = {'update_greedy';'update_round_robin';'update_pairwise';'update_always';'update_simulated_annealing'};
5
6  case 'discrete'
7      Choose = {'choose_traverse';'choose_tournament';'choose_roulette_wheel';'choose_nich'};
8      Search = {'cross_point_one';'cross_point_two';'cross_point_uniform';'search_reset_one';'search_reset_rand';'reinit_discrete'};
9      Update = {'update_greedy';'update_round_robin';'update_pairwise';'update_always';'update_simulated_annealing'};
10
11 case 'permutation'
12     Choose = {'choose_traverse';'choose_tournament';'choose_roulette_wheel';'choose_nich'};
13     Search = {'cross_order_two';'cross_order_n';'search_swap';'search_swap_multi';'search_scramble';'search_insert';'reinit_permutation'};
14     Update = {'update_greedy';'update_round_robin';'update_pairwise';'update_always';'update_simulated_annealing'};
```

4 Run AutoOptLib

Users can run AutoOptLib either by MATLAB command or GUI.

Table 1: Algorithmic components included in the current AutoOptLib version.

Component	Description
Continuous search:	
`cross_arithmetic`	Whole arithmetic crossover
`cross_sim_binary`	Simulated binary crossover [DA+95]
`cross_point_one`	One-point crossover
`cross_point_two`	Two-point crossover
`cross_point_n`	n-point crossover
`cross_point_uniform`	Uniform crossover
`search_cma`	The evolution strategy with covariance matrix adaption
`search_eda`	The estimation of distribution
`search_mu_cauchy`	Cauchy mutation [YLL99]
`search_mu_gaussian`	Gaussian mutation [Fog98]
`search_mu_polynomial`	Polynomial mutation [DG+96]
`search_mu_uniform`	Uniform mutation
`search_pso`	Particle swarm optimization's particle fly and update [SE98]
`search_de_random`	The "random/1" differential mutation [SP97]
`search_de_current`	The "current/1" differential mutation
`search_de_current_best`	The "current-to-best/1" differential mutation
`reinit_continuous`	Random reinitialization for continuous problems
Discrete search:	
`cross_point_one`	One-point crossover
`cross_point_two`	Two-point crossover
`cross_point_n`	n-point crossover
`cross_point_uniform`	Uniform crossover
`search_reset_one`	Reset a randomly selected entity to a random value
`search_reset_rand`	Reset each entity to a random value with a probability
`search_reset_creep`	Add a small positive or negative value to each entity with a probability, for problems with ordinal attributes
`reinit_discrete`	Random reinitialization for discrete problems
Permutation search:	
`cross_order_two`	Two-order crossover
`cross_order_n`	n-order crossover
`search_swap`	Swap two randomly selected entities
`search_swap_multi`	Swap each pair of entities between two randomly selected indices
`search_scramble`	Scramble all the entities between two randomly selected indices
`search_insert`	Randomly select two entities, insert the second entity to the position following the first one
`reinit_permutation`	Random reinitialization for permutation problems
Choose where to search from:	
`choose_cluster`	Brain storm optimization's idea picking up for choosing solutions to search from [Shi15]
`choose_roulette_wheel`	Roulette wheel selection
`choose_tournament`	K-tournament selection
`choose_traverse`	Choose each of the current solutions to search from
Select promising solutions:	
`update_always`	Always select new solutions
`update_greedy`	Select the best solutions
`update_pairwise`	Select the better solution from each pair of old and new solutions
`update_round_robin`	Select solutions by round-robin tournament
`update_simulated_annealing`	Simulated annealing's update mechanism, i.e., accept worse solution with a probability [KGJV83]
Archive:	
`archive_best`	Collect the best solutions found so far
`archive_diversity`	Collect most diversified solutions found so far
`archive_tabu`	The tabu list [Glo89]

4.1 Run by Command

Users can run AutoOptLib by typing the following command in MATLAB command window:

`AutoOpt('name1',value1,'name2',value2,...)`,

where name and value refer to the input parameter's name and value, respectively. The parameters are introduced in Table 2. In particular, parameters `Metric` and `Evaluate` define the design objective and algorithm performance evaluation method, respectively. They are detailed in Tables 3 and 4, respectively.

Parameters `Problem`, `InstanceTrain`, `InstanceTest`, and `Mode` are mandatory to input into the command. For other parameters, users can either use their default values without input to the command or input by themselves for sophisticated functionality. The default parameter values can be seen in `AutoOpt.m`. As an example, `AutoOpt('Mode', 'design', 'Problem', 'CEC2005_f1', 'InstanceTrain', [1,2], 'InstanceTest', 3, 'Metric', 'quality'` is for designing algorithms with the best solution quality on the `CEC2005_f1` problem.

There are also conditional parameters when certain options of the main parameters are chosen. For example, setting `Metric` to `runtimeFE` incurs conditional parameter `Thres` to define the algorithm performance threshold for counting the runtime. All conditional parameters have default values and are unnecessary to set in the command.

After AutoOptLib running terminates, results will be saved as follows:

- If running the `design` mode,
  1. The designed algorithms' graph representations, phenotypes, parameter values, and performance will be saved as `.mat` table in the root dictionary. Algorithms in the `.mat` table can later be called by the `solve` mode to apply to solve the targeted problem or make experimental comparisons with other algorithms.
  2. A report of the designed algorithms' pseudocode and performance will be saved as `.xlsx` table. Users can read, analyze, and compare the algorithms through the report.
  3. The convergence curve of the design process (algorithms' performance versus the iteration of design) will be depicted and saved as `.fig` figure. Users can visually analyze the design process and compare different design techniques through the figure.
- If running the `solve` mode,
  1. Solutions to the targeted problem will be saved as `.mat` table and `.xlsx` table.
  2. Convergence curves of the problem-solving process (solution quality versus algorithm execution) will be plotted in `.fig` figure.

4.2 Run by GUI

The GUI can be invoked by the command `AutoOpt()` without inputting parameters. It is shown in Figure 1. The GUI has three panels, i.e., Design, Solve, and Results:

- The Design panel is for designing algorithms for a targeted problem. It has two subpanels, i.e., Input Problem and Set Parameters:
  - Users should load the function of their targeted problem and set the indexes of training and test instances in the Input Problem subpanel.
  - Users can set the main and conditional parameters related to the design in the Set Parameters subpanel. All parameters have default values for non-expert users' convenience. The objective of design, the method for comparing the designed algorithms, and the method for evaluating the algorithms can be chosen by the pop-up menus of the Metric, Compare, and Evaluate fields, respectively.

  After setting the problem and parameters, users can start the run by clicking the RUN bottom.

Table 2: Parameters in the commands for running AutoOptLib.

Parameter	Type	Description
Parameters related to the targeted problem:		
`Problem`	character string	Function of the targeted problem
`InstanceTrain`	positive integer	Indexes of training instances of the targeted problem
`InstanceTest`	positive integer	Indexes of test instances of the targeted problem
Parameters related to the designed algorithms:		
`Mode`	character string	Run mode. Options: `design` - design algorithms for the targeted problem, `solve` - solve the targeted problem by a designed algorithm or an existing algorithm.
`AlgP`	positive integer	Number of search pathways in a designed algorithm
`AlgQ`	positive integer	Maximum number of search operators in a search pathway
`Archive`	character string	Name of the archive(s) that will be used in the designed algorithms
`LSRange`	$[0, 1]$ real number	Range of inner parameter values that make the component perform local search*.
`IncRate`	$[0, 1]$ real number	Minimum rate of solutions' fitness improvement during 3 consecutive iterations
`InnerFE`	positive integer	Maximum number of function evaluations for each call of local search
Parameters controlling the design process:		
`AlgN`	positive integer	Number of algorithms to be designed
`AlgRuns`	positive integer	Number of algorithm runs on each problem instance
`ProbN`	positive integer	Population size of the designed algorithms on the targeted problem instances
`ProbFE`	positive integer	Number of fitness evaluations of the designed algorithms on the targeted problem instances
`Metric`	character string	Metric for evaluating algorithms' performance (the objective of design). Options: `quality`, `runtimeFE`, `runtimeSec`, `auc`.
`Evaluate`	character string	Method for evaluating algorithms' performance. Options: `exact`, `intensification`, `racing`, `surrogate`.
`Compare`	character string	Method for comparing the performance of algorithms. Options: `average`, `statistic`
`AlgFE`	positive integer	Maximum number of algorithm evaluations during the design process (termination condition of the design process)
`Tmax`	positive integer	Maximum running time measured by the number of function evaluations or wall clock time
`Thres`	real number	The lowest acceptable performance of the designed algorithms. The performance can be solution quality.
`RacingK`	positive integer	Number of instances evaluated before the first round of racing
`Surro`	real number	Number of exact performance evaluations when using surrogate
Parameters related to solving the targeted problem:		
`Alg`	character string	Algorithm file name, e.g., Algs

$*$: Some search operators have inner parameters to control performing global or local search. For example, a large mutation probability of the uniform mutation operator indicates a global search, while a small probability indicates a local search over neighborhood region. As an example, in cases with `LSRange`$= 0.2$, the uniform mutation with probability lower than 0.2 is regarded as performing local search, and the probability equals or higher than 0.2 performs global search.

Table 3: Design objectives involved in AutoOptLib.

Objective	Description
`quality`	The designed algorithm's solution quality on the targeted problem within a fixed computational budget.
`runtimeFE`	The designed algorithm's running time (number of function evaluations) till reaching a performance threshold on solving the targeted problem.
`runtimeSec`	The designed algorithm's running time (wall clock time, in second) till reaching a performance threshold on solving the targeted problem.
`auc`	The area under the curve (AUC) of empirical cumulative distribution function of running time, measuring the anytime performance [YDWB22].

Table 4: Algorithm performance evaluation methods provided in AutoOptLib.

Method	Description
`exact`	Exactly run all the designed algorithms on all test problem instances.
`approximate`	Use low complexity surrogate to approximate the designed algorithms' performance without full evaluation.
`racing` [LIDLC+16]	Save algorithm evaluations by stopping evaluating on the next instance if performance is statistically worse than at least another algorithm.
`intensification` [HHLBS09]	Save algorithm evaluations by stopping evaluating on the next instance if performance is worse than the incumbent.

- When the running starts, warnings and corresponding solutions to incorrect uses (if any) will be displayed in the text area at the top of the Results panel. The real-time stage and progress of the run will also be shown in the area. After the run terminates, results will be saved in the same format as done by running by commands. Results will also be displayed on the GUI as follows:
  - The convergence curve of the design process will be plotted in the axes area of the Results panel.
  - The pseudocode of the best algorithm found during the run will be written in the text area below the axes, as shown in Figure 1.
  - Users can use the pop-up menu at the bottom of the Results panel to export more results, e.g., other algorithms found during the run, and detailed performance of the algorithms on different problem instances.
- The Solve panel is for solving the targeted problem by an algorithm. It follows a similar scheme to the Design panel. In particular, users can load an algorithm designed by AutoOptLib in the Algorithm File field to solve the targeted problem. Alternatively, users can choose a classic algorithm as a comparison baseline through the pop-up menu of the Specify Algorithm field. AutoOptLib now provides 17 classic metaheuristic algorithms in the menu. After the problem-solving terminates, the convergence curve and best solutions will be displayed in the axes and table areas of the Results panel, respectively; detailed results can be exported by the pop-up menu at the bottom.

5 Extend AutoOptLib

AutoOptLib follows the open-closed principle [Mey97, Lar01]. Users can implement their algorithmic components, design objectives, and algorithm performance evaluation techniques based on the current sources, and add the implementations to the library by a uniform interface. Taking Listing 1 as an example, new algorithmic components can be added as follows.

Listing 1: Implementation of the uniform mutation operator.

```
function [output1,output2] = search_mu_uniform(varargin)
mode = varargin{end};
switch mode
    case 'execute'
        Parent  = varargin{1};
        Problem = varargin{2};
        Para    = varargin{3};
        Aux     = varargin{4};
        if ~isnumeric(Parent)
            Offspring = Parent.decs;
        else
            Offspring = Parent;
        end
        Prob   = Para;
        [N,D]  = size(Offspring);
```

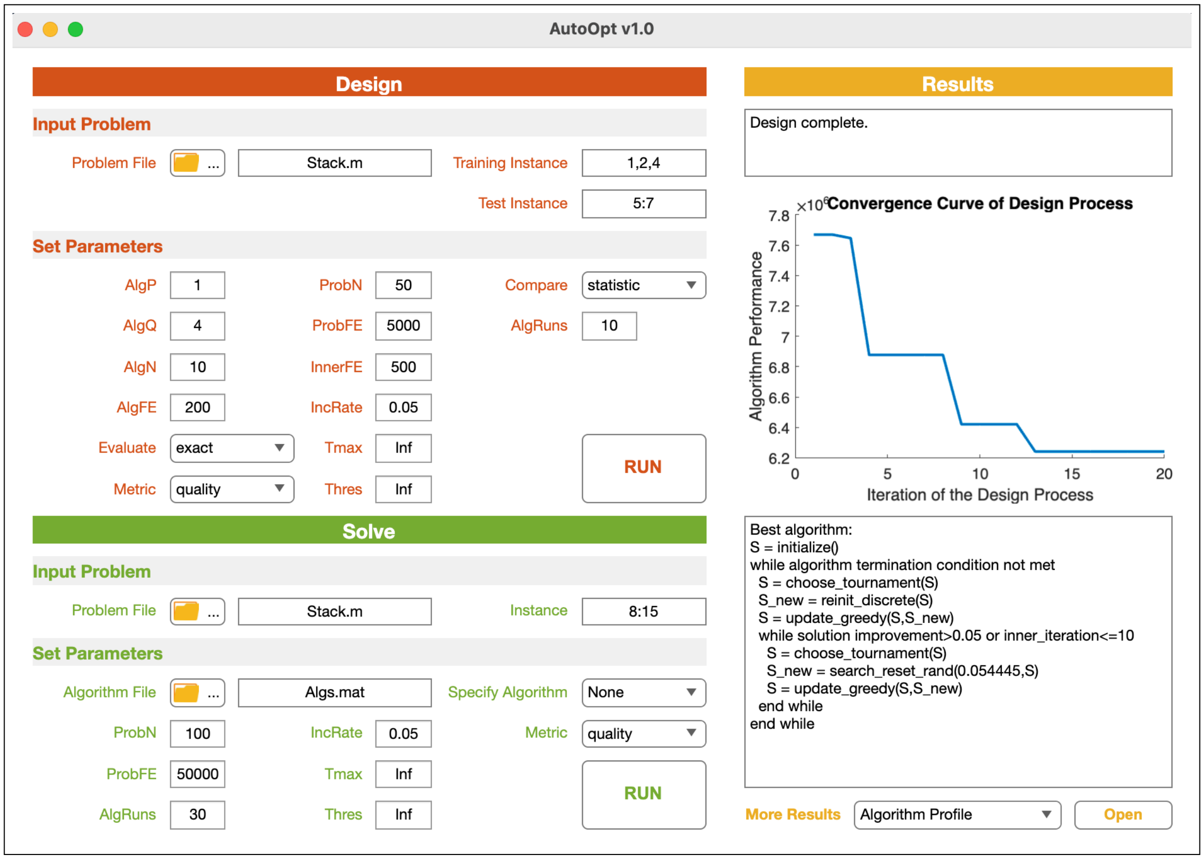

Figure 1: GUI of AutoOptLib.

```
16            Lower = Problem.bound(1,:);
17            Upper = Problem.bound(2,:);
18            ind = rand(N,D) < Prob;
19            Temp = unifrnd(repmat(Lower,N,1),repmat(Upper,N,1));
20            Offspring(ind) = Temp(ind);
21            output1 = Offspring;
22            output2 = Aux;
23        case 'parameter'
24            output1 = [0,0.3]; % mutation probability
25        case 'behavior'
26            output1 = {'LS','small';'GS','large'}; % small probabilities perform
                  local search
27    end
28    if ~exist('output1','var')
29        output1 = [];
30    end
31    if ~exist('output2','var')
32        output2 = [];
33    end
```

An algorithmic component is implemented in an independent `function` with three main cases. Case `execute` refers to executing the component. There are seven optional inputs:

1. Current solutions, i.e., `Parent` in line 5.
2. The problem proprieties, i.e., `Problem` in line 6.
3. The component's inner parameters, i.e., `Para` in line 7.

4. An auxiliary structure array for saving the component's inner parameters that are changed during iteration (e.g., the velocity in particle swarm optimization (PSO)), i.e., `Aux` in line 8.
5. The algorithm's generation counter `G`.
6. The algorithm's inner local search iteration counter `innerG`.
7. The targeted problem's input data `Data`.

The component should be implemented from line 9. The outputs of lines 21 and 22 are mandatory, in which `output1` returns solutions processed by the component, and `output2` returns the `Aux` structure array. If the component has inner parameters that are changed during iteration, `Aux` is updated (e.g., update PSO's velocity and save it in `Aux`); otherwise, `Aux` will be the same as that in line 8.

Case `parameter` defines the lower and upper bounds of the component's inner parameter values. For example, the mutation probability is bounded within $[0, 0.3]$ in line 24. For components with multiple inner parameters, each parameter's lower and upper bounds should be saved in an independent row of the matrix `output1`.

For search operators (components) with inner parameters controlling the search behavior, case `behavior` defines how the inner parameters control the search behavior. For example, line 26 indicates that the uniform mutation with smaller mutation probabilities performs local search and that with larger probabilities performs global search. For other operators, `output1` in case `behavior` is left empty, i.e., `output1={};`.

References

[DA+95] Kalyanmoy Deb, Ram Bhushan Agrawal, et al. Simulated binary crossover for continuous search space. *Complex Systems*, 9(2):115–148, 1995.

[DG+96] Kalyanmoy Deb, Mayank Goyal, et al. A combined genetic adaptive search (GeneAS) for engineering design. *Computer Science and Informatics*, 26:30–45, Jan. 1996.

[Fog98] David B Fogel. *Artificial intelligence through simulated evolution.* Wiley-IEEE Press, 1998.

[Glo89] Fred Glover. Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206, Aug. 1989.

[HHLBS09] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, Oct. 2009.

[JD13] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *IEEE Transactions on evolutionary computation*, 18(4):602–622, Aug. 2013.

[KGJV83] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Lar01] Craig Larman. Protected variation: The importance of being closed. *IEEE Software*, 18(3):89–91, 2001.

[LIDLC+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, Sept. 2016.

[Mey97] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[SE98] Yuhui Shi and Russell Eberhart. A modified particle swarm optimizer. In *IEEE International Conference on Evolutionary Computation*, pages 69–73, Anchorage, AK, USA, 1998. IEEE.

[Shi15] Yuhui Shi. An optimization algorithm based on brainstorming process. In *Emerging Research on Swarm Intelligence and Algorithm Optimization*, pages 1–35. IGI Global, 2015.

[SP97] Rainer Storn and Kenneth Price. Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, Dec. 1997.

[YDWB22] Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation*, 26(6):1526–1538, Dec. 2022.

[YLL99] Xin Yao, Yong Liu, and Guangming Lin. Evolutionary programming made faster. *IEEE Transactions on Evolutionary Computation*, 3(2):82–102, July 1999.