SoftwareX 12 (2020) 100628

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

CUSTOMHyS: Customising Optimisation Metaheuristics via Hyper-heuristic Search

Jorge M. Cruz-Duarte [a], Ivan Amaya [a,*], José C. Ortiz-Bayliss [a], Hugo Terashima-Marín [a], Yong Shi [b]

[a] School of Engineering and Sciences, Tecnologico de Monterrey, Av. Eugenio Garza Sada 2501 Sur, Monterrey, NL 64849, Mexico
[b] Research Center on Fictitious Economy and Data Science, Chinese Academy of Sciences, Zhongguancun East Road 80, Haidian District, Beijing 100190, China

ARTICLE INFO

Article history:
Received 1 September 2020
Received in revised form 26 October 2020
Accepted 11 November 2020

Keywords:
Metaheuristic
Hyper-heuristic
Search operators
Evolutionary computation

ABSTRACT

There is a colourful palette of metaheuristics for solving continuous optimisation problems in the literature. Unfortunately, it is not easy to pick a suitable one for a specific practical scenario. Moreover, oftentimes the selected metaheuristic must be tuned until finding adequate parameter settings. Therefore, this work presents a framework based on a hyper-heuristic powered by Simulated Annealing for tailoring population-based metaheuristics. To do so, we recognise search operators from well-known techniques as building blocks for new ones. The presented framework comprises six main modules coded in Python, which can be used independently, and which help explore new metaheuristics.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-20-00035
Code Ocean compute capsule	None
Legal Code License	MIT License
Code versioning system used	GitHub
Software code languages, tools, and services used	Python 3.7
Compilation requirements, operating environments & dependencies	NumPy 1.18.5, SciPy 1.5.0, Matplotlib 3.2.2, json 2.0.9, tqdm 4.47.0.
If available Link to developer documentation/manual	https://github.com/jcrvz/customhys/blob/master/README.md
Support email for questions	jorge.cruz@tec.mx, j.m.cruzduarte@ieee.org

1. Motivation and significance

In the current era of information, technological and scientific advances depend on the many tools being developed. Such tools facilitate the workflow of practitioners, allowing them to spend more time thinking about solutions and generating ideas, instead of *reinventing the wheel* at every chance. There are a plethora of tools for countless applications. However, in this work, we focus on metaheuristic (MH) optimisation algorithms. Metaheuristics can be defined as general-purpose methods, and they have been proposed to tackle many real-world problems on different domains [1]. They are characterised by their flexibility, versatility, and algorithmic simplicity when facing a problem. A vast number of MHs claiming to be the best for solving engineering problems have been reported [1,2]. However, in practice, their scope is limited, as stated by the *No-Free-Lunch* theorem [3]. So, researchers and practitioners must select a metaheuristic for a given problem, which naturally requires a certain degree of knowledge about the available MHs. Even when we have picked a metaheuristic to use, we likely need to find an adequate set of values for the MH parameters, which is, in itself, a new problem. Therefore, the question of deciding which MH is worth implementing to solve a defined problem remains open.

Furthermore, in the last three decades, the scientific community has evidenced the appearance of several MHs with curious metaphors (or 'flavours') to face various problems [4,5]. However,

* Corresponding author.
E-mail addresses: jorge.cruz@tec.mx (J.M. Cruz-Duarte), iamaya2@tec.mx (I. Amaya), jcobayliss@tec.mx (J.C. Ortiz-Bayliss), terashima@tec.mx (H. Terashima-Marín), yshi@ucas.ac.cn (Y. Shi).

https://doi.org/10.1016/j.softx.2020.100628

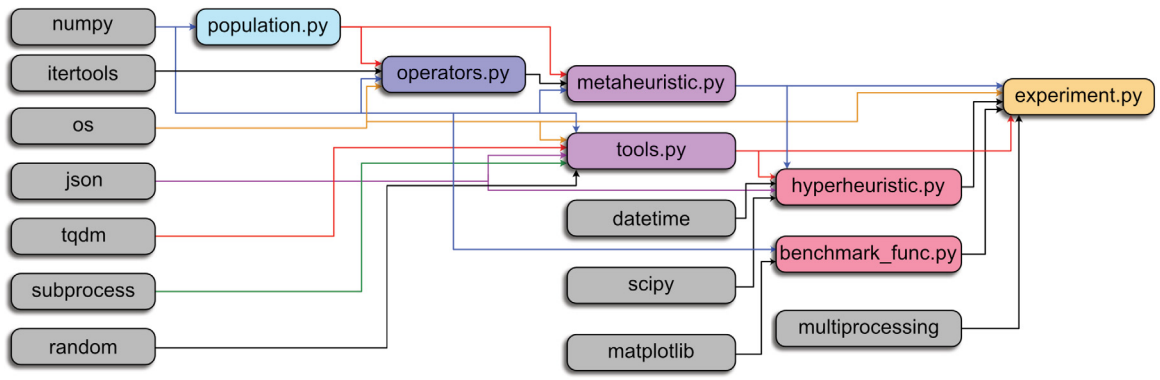


Fig. 1. Module dependency diagram of the CUSTOMHyS framework.

after analysing a bit the 'new' proposals, one can notice that they are not so different from conventional MHs such as Simulated Annealing (SA), Differential Evolution (DE), and Particle Swarm Optimisation (PSO), to mention a few. Roughly speaking, many of the existing metaheuristics are just combinations of search operators (SOs) from other MHs that can be slightly modified. These search operators, or heuristics, sit at the core of metaheuristics. Some examples include mutation [6], crossover [7], and Lévy flight [8]. Authors have taken advantage of this fact and have assembled algorithms by choosing two or more SOs, giving birth to MHs with astonishing performances in specific problems [9]. The idea of combining solvers is not entirely new, as it can be traced back to the 1960s. But, only recently, it has grown into an optimisation sub-area known as hyper-heuristics (HHs) [10]. Naturally, other similar methodologies have also appeared, *e.g.,* Algorithm Portfolios [11], but they are out of the scope of this work.

HHs provide a general approach, which has been defined as a high-level heuristic. Such a heuristic selects or modifies low-level heuristics to find better solutions for a problem domain [12]. It is interesting to see that many HHs have successfully addressed combinatorial problems [13], whilst only a few have dealt with continuous ones [12]. On the one hand, one of the most relevant strategies is the Hyper-Heuristics Flexible (HyFlex) framework, created by the ASAP research group for favouring the development and analysis of multiple HHs throughout several combinatorial problem domains [14]. HyFlex capitalises on domain-independence and automated search techniques by establishing a domain barrier, which separates HHs from the problem domain. Subsequently, several modifications and upgrades have been proposed to this framework [15–17]. Later on, Asta and Özcan presented a multi-stage hyperheuristic, called Tensor Based Hybrid Acceptance Hyper-heuristic (TeBHA-HH), which embeds tensor-based techniques for analysing the implemented heuristics to detect the latent correlations between those operators [18]. Majeed and Naz presented Deja Vu, a hyper-heuristic framework based on a 2R (Record and Recall) module [19]. In this strategy, when a new problem comes, Deja Vu performs a similarity measurement by comparing the problem against others already solved. Then, the procedure selects the algorithm recorded as the solver for the most similar problem. Deja Vu works with a varied collection of Machine Learning algorithms as well as with problems from different domains. Other frameworks worth mentioning in the combinatorial domain are Hyperion, hMod, SSHH, HH-SVM, HH-LS, and EMHH [20–25]. On the other hand, Miranda et al. proposed a Hybrid Hyper-Heuristic for Algorithm Design (H3AD) [26]. The authors used H3AD for optimising (or redesigning) the well-known PSO algorithm to 60 continuous benchmark problems. This strategy employed PSO grammar as a design schema, so it may prove somewhat tricky to extend this idea to other MHs. In a similar approach, Abell et al. generated an algorithm portfolio that included DE and PSO, which targeted the Black-Box Optimisation Benchmarking problems [27].

Despite the scarce research dealing with high-level solvers for continuous optimisation, the available ones lack a proper generalisation scheme. For example, many of the robust frameworks existing for combinatorial optimisation assume that a heuristic is the same as a metaheuristic, which is valid only under certain conditions. As a disclaimer, we want to stress that our intention is not to deepen into controversies, so we provide a summary of the different uses of techniques involving the word 'heuristic' in the following section.

In this work, we introduce CUSTOMHyS: Customising Optimisation Metaheuristics via Hyper-heuristic Search. Our proposed framework seeks to halt the frantic trend of publishing metaheuristics with similar operations but disguised with 'creative' metaphors. CUSTOMHyS provides a structured and innovative way to find a suitable method for solving a given continuous optimisation problem, only using mathematical operations over a population. Another reason behind the proposed framework is to facilitate the design flow of many practitioners that utilise MHs in diverse applications, where they have to choose or tune these methods. With this HH-based strategy, we contribute to fill the knowledge gap existing for metaheuristics, hyper-heuristics, and continuous optimisation.

CUSTOMHys is entirely coded in Python 3.7 with a full documented implementation and only using basic Python packages such as Numpy, SciPy, Matplotlib, and JSON. Our approach generates custom population-based MHs to solve continuous optimisation problems by cascading one or more search operators collected from well-known MHs. The number of search operators in the sequence is called 'cardinality', which can be adjusted to the needs of the implementation. For the current version, we analysed ten well-known metaheuristics by extracting their search operators (SOs) to generate heuristic spaces of different scales. In this framework, we also integrate a benchmark function module that comprises 107 continuous optimisation problems. Section 2 describes the overall structure of CUSTOMHyS and each of the modules that incorporate it. Besides, we provide several illustrative examples for using each module of the proposed framework and detail their features and capabilities.

2. Framework description

In this section, we describe the CUSTOMHyS architecture and their functionalities. We also present several illustrative examples. It is worth noting that all the modules are fully documented, and a brief description is available at the "Readme" file within the repository (https://github.com/jcrvz/customhys). This framework is composed by six main modules: `benchmark_func`, `experiment`, `hyperheuristic`, `metaheuristic`, `operators`,

Table 1
Set of standard functions implemented in the benchmark-functions module.

Id.	Function name	Ref.
1	Ackley 1	[28]
2	Ackley 4	[28]
3	Alpine 1	[28]
4	Alpine 2	[28]
5	Bohachevsky	[29]
6	Brent	[28]
7	Brown	[28]
8	Carrom Table	[29]
9	Chung Reynolds	[30]
10	Cigar	[29]
11	Cosine Mixture	[30]
12	Cross-in-Tray	[29]
13	Cross-Leg Table	[30]
14	Crowned Cross	[29]
15	Csendes	[28]
16	Deb 1	[28]
17	Deb 2	[30]
18	Deflected CorrugatedSpring	[30]
19	Dixon-Price	[28]
20	Drop Wave	[31]
21	Egg Holder	[28]
22	Ellipsoid	[32]
23	Expanded DecreasingMinima	[33]
24	Expanded Equal Minima	[33]
25	Expanded Five-Uneven-Peak Trap	[33]
26	Expanded Two-Peak Trap	[33]
27	Expanded Uneven Minima	[33]
28	Exponential	[28]
29	F2	[30]
30	Giunta	[28]
31	Griewank	[28]
32	Happy Cat	[31]
33	Hyper-Ellipsoid	[34]
34	Inverted Cosine-Wave	[30]
35	Jennrich-Sampson	[30]
36	K-Tablet	[35]
37	Katsuura	[29]
38	Levy	[36]
39	Lunacek N01	[30]
40	Lunacek N02	[30]
41	Michalewicz	[36]
42	Mishra 1	[28]
43	Mishra 2	[28]
44	Mishra 7	[28]
45	Mishra 11	[28]
46	Modified Vincent	[33]
47	Needle-Eye	[30]
48	Pathological	[28]
49	Periodic	[28]
50	Perm 01	[29]
51	Perm 02	[36]
52	Pinter	[28]
53	Powel Sum	[28]
54	Price 01	[28]
55	Qing	[28]
56	Quartic (noiseless)	[28]
57	Quintic	[28]
58	Rana	[28]
59	Rastrigin	[28]
60	Ridge	[28]
61	Rosenbrock	[28]
62	Rotated-Hyper-Ellipsoid	[36]
63	Salomon	[28]
64	Sargan	[28]
65	Schaffer N1	[31]
66	Schaffer N2	[31]
67	Schaffer N3	[31]
68	Schaffer N4	[31]
69	Schaffer N6	[28]
70	Schubert	[28]
71	Schubert 3	[28]
72	Schubert 4	[28]
73	Schumer Steiglitz	[30]
74	Schwefel	[28]
75	Schwefel 1.2	[28]
76	Schwefel 2.04	[28]
77	Schwefel 2.20	[28]
78	Schwefel 2.21	[28]
79	Schwefel 2.22	[28]
80	Schwefel 2.23	[28]
81	Schwefel 2.25	[28]
82	Schwefel 2.26	[28]
83	Sphere (De Jong)	[28]
84	Step	[28]
85	Step 2	[28]
86	Step 3	[28]
87	Step Int	[28]
88	Stochastic	[30]
89	Stretched V Sine Wave	[28]
90	Styblinski-Tank	[31]
91	Sum Squares	[28]
92	Trid	[36]
93	Trigonometric 1	[28]
94	Trigonometric 2	[28]
95	Type-I Simple Deceptive	[37]
96	Type-II Medium-ComplexDeceptive	[37]
97	Vincent	[29]
98	W-Wavy	[28]
99	Weierstrass	[28]
100	Whitley	[28]
101	Xin-She Yang 1	[28]
102	Xin-She Yang 2	[28]
103	Xin-She Yang 3	[28]
104	Xin-She Yang 4	[28]
105	Yao-Liu 09	[29]
106	Zakharov	[28]
107	Zero Sum	[29]

(*continued on next page*)

Table 1 (*continued*).

`population`, and `tools`. Fig. 1 shows the module dependency diagram of these components, where the required basic Python packages (grey boxes) are also presented.

In the following subsections, we present each one of the framework's main modules and explain their significant functionalities. They are organised in order of dependency with other modules within the framework. For the sake of clarity, modules with less dependency are described first.

2.1. Benchmark functions module

The benchmark-functions module (`benchmark_func.py`) is designed for functions that can be scaled up in terms of the number of dimensions (*i.e.,* where D can be defined to be any $D > 1$). It implies that the module does not contain functions with a fixed number of dimensions. In the current version, this module accounts for a collection of 107 multidimensional benchmark functions commonly used in the literature [28–33, 36]. Table 1 shows these optimisation problems, which are contained within the special variable `__all__` of this module. An illustrative example of how to use this module is detailed in Appendix B.1. All attributes and methods of this module are detailed in Appendix C.1.

2.2. Tools module

The tools module contains several functions for supporting the execution of the remaining modules within the framework. These tools glue the interaction between modules and also bolster user interaction. In the current version, this module contains eight functions, which are listed in Appendix C.2. We remark one particular function, `printmsk`, that allows the user to know the structure of a dictionary with nested variables. This function

prints a tree-like structure with the type and length of inner variables. A very illustrative example of this function can be found in Appendix B.2.

In addition, the functions `revise_results` and `preprocess_files` are handy to process data from experiments using hyper-heuristics. The former checks if there is a repeated entry and fix it. Bear in mind that all results are stored in the folder "./raw/", and each solved instance (problem and dimension) is saved into a subfolder. The remaining function summarises data from all folders while neglecting certain information like historical values and population positions, among others.

2.3. Population module

This module contains the population class. So, an object corresponds to a group of agents within a domain given by the problem to deal with. Thus, a problem (function and domain) from `benchmark_func` must be provided to deploy the population adequately. Bear in mind that this module can be utilised standalone. However, consider that the population by itself does nothing—it is akin to a shepherdless herd. Detailed information about the attributes and methods of this class can be found in Appendix C.3.

Notwithstanding, for creating a population object, only the problem domain boundaries, population size, and boundary-constrained flag are required. Although, the last couple parameters could be ignored, and their default values would be assumed: `num_agents=30` and `is_constrained=True`. Once created, positions for each individual require initialisation. In the current version of this framework, such a process is carried out by default using a uniform distribution of random numbers; the other available option is to follow a grid-like initialisation. Then, these positions are evaluated by using the problem function and a selector is applied to find the best position of the entire population, of the particular individual, and of the historical process. The last two are the same fitness value since the population has not been modified. Appendix B.3 exemplifies the aforementioned steps using this module.

2.4. Operators module

This module, along with the population one, stands as one of the most important modules of the framework. Recall the herd analogy. In this case, operators serve as shepherds that guide the population of agents through a problem landscape. We collected the Search Operators (SOs) from the following ten well-known metaheuristics available in the literature: Random Search (RS) [38], Simulated Annealing (SA) [39], Genetic Algorithm (GA) [6], Cuckoo Search (CS) [8], Differential Evolution (DE) [40], Particle Swarm Optimisation (PSO) [41,42], Firefly Algorithm (FA) [43], Stochastic Spiral Optimisation Algorithm (SSOA) [44], Central Force Optimisation (CFO) [45], and Gravitational Search Algorithm (GSA) [46]. As with other modules, Appendix C.4 provides details about the operators of this one. Table 2 summarises the 12 SOs obtained, including the random sample as is the most straightforward manner of performing a search in an arbitrary domain. This table presents the operators, their control parameters, and default selector (Appendix A.4.1). We classified these parameters as variation and tuning. The first one concerns those parameters that can dramatically change the behaviour of the operator. The second one, in contrast, refines the search procedure. For further details about the parameters, we invite you to consult the code documentation or the related manuscripts [47,48]. It is nice to mention that each operator requires, at least, a population object (given as an argument) to work. Appendix B.4 shows a descriptive example for implementing a search operator with this module.

By applying recurrently the same or different search operators (or simple heuristics) to the population object, one can render a metaheuristic procedure. However, we do not recommend to execute `population` and `operators` directly. Instead, users can create their own optimisation methods just following the procedures described in the examples. Another option, and also the one we recommend, is to use the module `metaheuristic` which is described in the next section. To do so, a text file with the SOs and their parameter values and selectors must be provided as a heuristic collection. For that reason, along with the `operators` module, the folder "./collections/" contains three predefined collections. The first one, "default.txt", comprises a total of 205 SOs obtained by considering different variation parameters, predefined values for tuning parameters, and all the available selectors. The second file, "automatic.txt", has a total of 10877 SOs achieved by considering different variation parameters and all the available selectors, as well as five values for each tuning parameters. As its name indicates, this collection can be generated automatically through the `build_operators` method, also available in the `operators` module. Indeed, if the module is called as a script, such `build_operators` method is run automatically. Lastly, the third database consists of a list of 66 predefined metaheuristics (MHs) using the previously mentioned SOs. These MHs are instances of the 10 MHs selected for extracting their search operators.

It is essential to remark that each line of a collection file corresponds to a search operator (given by a tuple) or a list of search operators. Three elements compose each tuple: first, we find the name of the operator, then a dictionary with the parameters for such an operator, and finally, a selector to employ after applying the operator. Three selected cases of SOs from "default.txt" are presented to show the structure mentioned above:

```
('random_search', {'scale': 1.0, 'distribution': '
    uniform'}, 'greedy')
('spiral_dynamic', {'radius': 0.9, 'angle': 22.5,
    'sigma': 0.1}, 'all')
('gravitational_search', {'gravity': 1.0, 'alpha':
     0.02}, 'metropolis')
```

2.5. Metaheuristic module

This module contains the metaheuristic class to handle the search operators and the population for solving continuous optimisation problems. A metaheuristic object is then a procedure that follows the definition given in Appendix A.4.2. Two elements must be specified to create an MH object: the problem to solve and the simple heuristics to implement. The former should be a dictionary with keys: 'function', 'boundaries', and 'is_constrained'. Values for 'function' and 'boundaries' are the same as those that the population module uses, and 'is_constrained' is a Boolean to restrict individuals from exploring outside the domain. This dictionary can be obtained with the method `get_formatted_problem` from a problem object of the benchmark functions module (cf. Appendix C.1). Moreover, the simple heuristics can be a list of tuples or just a tuple as described in Section 2.4. Subsequently, once the metaheuristic is created, the search procedure is run through the `run` method. Pseudocode 1 describes the steps that this routine performs in terms of the nomenclature shown in Appendix A. Moreover, an example of how to implement a metaheuristic using this module is described in Appendix B.5.

2.6. Hyper-heuristic module

Similar to the metaheuristic module, this one has the hyper-heuristic (HH) class, according to Appendix A.4.3. A HH object searches within the heuristic space for building metaheuristics that best solves a given problem. To create a HH procedure, it

Table 2
Search operators from well-known metaheuristics in the literature. Values or ranges for variation and tuning parameters, as well as default selectors.

Operator name	Variation parameters	Tuning parameters	Selector
`central_force_dynamic`	–	`gravity, alpha, beta, dt`	`all`
`differential_crossover`	`version`[a]	`crossover_rate`	`greedy`
`differential_mutation`	`expression`[b]	`num_rands, factor`	`all`
`firefly_dynamic`	`distribution`[c]	`alpha, beta, gamma`	`all`
`genetic_crossover`	`pairing`[d], `crossover`[e]	`mating_pool_factor`	`all`
`genetic_mutation`	`distribution`[c]	`scale, elite_rate,mutation_rate`	`greedy`
`gravitational_search`	–	`gravity, alpha`	`all`
`local_random_walk`	`distribution`[c]	`probability, scale`	`greedy`
`random_flight`	`distribution`[c]	`scale, beta`	`greedy`
`random_sample`	–	–	`all`
`random_search`	`distribution`[c]	`scale`	`greedy`
`spiral_dynamic`	–	`radius, angle, sigma`	`all`
`swarm_dynamic`	`version`[f], `distribution`[c]	`factor, self_conf, swarm_conf`	`all`

[a] `version`: 'binomial' or 'exponential'.
[b] `expression`: 'rand', 'best', 'current', 'current-to-best', 'rand-to-best' or 'rand-to-best-and-current'.
[c] `distribution`: 'gaussian', 'uniform' or 'levy'.
[d] `pairing`: 'cost', 'rank', 'tournament' or 'random'.
[e] `crossover`: 'single', 'two', 'uniform', 'blend' or 'linear'.
[f] `version`: 'inertial' or 'constriction'.

Pseudocode 1. Metaheuristic scheme

Input: Problem domain $\mathfrak{X} \subseteq \mathbb{R}^D$, objective function $f(\vec{x})$, search operators $\boldsymbol{h}_p \in \mathfrak{H}_p^{\varpi}$, selector configuration $\boldsymbol{h}_s \in \mathfrak{H}_s^{\varpi}$, initialiser $h_i \in \mathfrak{H}_i$, finaliser $h_f \in \mathfrak{H}_f$, and population size N.
Output: Optimal solution $\vec{x}_*$

1: Initialise the population positions $X(0) \ni \vec{x}_n(0) \leftarrow h_i\{\mathfrak{X}\}, \forall\, n \in \{1, \dots, N\}$
2: Initialise additional features for each individual in the population
3: Evaluate the population $f_n(0) \leftarrow f(\vec{x}_n(0)),\ \forall\, n \in \{1, \dots, N\}$
4: Find $\vec{x}_*(0) \leftarrow \vec{x}_k(0)$ since $k = \text{argmin}\{f_1, \dots, f_N\}$, and set $t \leftarrow 0$
5: **repeat**
6: **for** $m = \{1, \dots, \varpi\}$ **do** ▷ $\varpi = \#\boldsymbol{h}_p = \#\boldsymbol{h}_s$
7: Apply the m^{th} search operator and selector, *i.e.,* $X(t) = (h_{s,m} \circ h_{p,m})\{X(t)\}$
8: Update $f_n(t),\ \forall\, n \in \{1, \dots, N\}, \vec{x}_*(t)$, and the additional population features
9: **end for**
10: $t \leftarrow t + 1$
11: **until** $h_f\{t, \mathfrak{X}, \vec{x}_*(t), f(\vec{x}_*(t)), \dots\} ==$ `True`

is mandatory to specify the problem to solve (`problem`), which follows the same structure mentioned in previous sections, and the heuristic collection (`heuristic_space`). This collection can be defined with its file name in "./collections/" or as a list of tuples, in the fashion specified in Section 2.4. An additional argument is `parameters`, which can be entered to generate a HH with customised settings. Further details about the methods and attributes of this class can be consulted in Appendix C.6.

The chief procedure that a HH object can carry out is based on the Simulated Annealing (SA) algorithm, which is summarised in Pseudocode 2. This algorithm starts by generating a candidate solution $\boldsymbol{h}$ from scratch, with $\varpi = \#\boldsymbol{h} = 1$ by randomly choosing a search operator from the heuristic space $\mathfrak{H}$. Random selection is weighted by an *a posteriori* probability distribution $P_{\mathfrak{H}}(h)$ if provided; otherwise, a uniform probability distribution is employed. Such $P_{\mathfrak{H}}(h)$ information can be also entered when creating the HH object as `weights_array`. In the current release, we equip the framework with a list of distributions called "operators_weights.json" for the "default.txt" collection from different categories of the problem functions with 2, 5, 10, 20, 30, 40, and 50 dimensions. These categories conform to the DSU binary-encoded triplet, where each letter stands for the Differentiable, Separable, and Unimodal feature, respectively.

In the next step of the procedure, the heuristic sequence is used to build a metaheuristic MH^{ϖ} according to Appendix A.4.2, as Pseudocode 1 details. Subsequently, to assess the performance of each metaheuristic MH^{ϖ} when solving a given problem, each candidate sequence (or metaheuristic) is executed N_r times, and all the fitness values are registered to evaluate its performance. The performance metric implemented for this framework is given by

$$F(\text{MH}^{\varpi}|\mathfrak{X}) = (\text{med} + \text{iqr})\left(\bigcup_{r=1}^{N_r} f(\vec{x}_{r,*})\right), \tag{1}$$

where med and iqr are the median and interquartile range operators applied to the fitness values $f(\vec{x}_{r,*})$. This metric is computed by the static method `get_performance`, which can be easily replaced based on user needs. To do so, regard that the input argument is a dictionary with the statistics from fitness values, which is also obtained from the static method `get_statistics`. Further information about these methods can be found in Appendix C.6.

Afterwards, the HH approach generates neighbour heuristic sequences based on [49,50], as described in Pseudocode 3. Thus, a neighbour $\boldsymbol{h}_c$ of the current sequence $\boldsymbol{h}$ can be obtained by adding (`Add`), deleting (`Del`), or perturbing (`Per`) a heuristic at random. When `Add` or `Per` are selected, candidate heuristics are extracted from the collection of search operators. Later, the neighbour, or candidate metaheuristic MH_c^n, is built and evaluated, $F(\text{MH}_c^n|\mathfrak{X})$, as before. From this point onward, the remaining steps (lines 10–18 in Pseudocode 2) coincide with the ordinary Simulated Annealing algorithm. A practical example of how to use the functions that this module provides for finding a suitable metaheuristic that solves a given problem is detailed in Appendix B.6.

2.7. Experiment module

Last but not least, the experimental module integrates all the modules mentioned earlier to provide a crystal-clear tool for carrying out experiments. Nonetheless, note that each module can be used independently. In this framework, an experiment stands for performing a hyper-heuristic search in a specific optimisation problem from a given list of problems to solve. So, the creation of an experiment object takes but a couple of lines.

Pseudocode 2. Hyper-heuristic based on Simulated Annealing to tailor metaheuristics

Input: Initial temperature $\Theta_0 \in \mathbb{R}_+$, cooling rate $\delta \in]0, 1[$, maximum number of steps $t_{\max}$, problem domain $\mathfrak{X} \subseteq \mathbb{R}^D$, objective function $f(\vec{x})$, heuristic space $\mathfrak{H}$, initialiser h_i, finaliser h_f, and maximum cardinality $\varpi_{\max} \geq 1$
Output: Best metaheuristic MH_*^{ϖ} found

$\boldsymbol{h} \leftarrow$ ChooseRandomly($\mathfrak{H}$, $P_{\mathfrak{H}}(h)$) ▷ $P_{\mathfrak{H}}(h)$ is the probability distribution of $\mathfrak{H}$
$\mathrm{MH}^{\varpi} \leftarrow \{h_i, \boldsymbol{h}, h_f\}$ with $\varpi = \#\boldsymbol{h} = 1$
Evaluate MH^{ϖ} via $F(\mathrm{MH}^{\varpi}|\mathfrak{X})$ in (1) and using Pseudocode 1
Initialise $t \leftarrow 0$ and $\mathrm{MH}_*^{\varpi} \leftarrow \mathrm{MH}^{\varpi}$
while $(t \leq t_{\max})$ and $(s \leq s_{\max})$ **do**
    $\boldsymbol{h}_c \leftarrow$ GetNeighbour($\boldsymbol{h}$, $\mathfrak{H}$, $\varpi_{\max}$) and $u_r \sim \mathcal{U}(0, 1)$ ▷ Pseudocode 3
    $\mathrm{MH}_c^{\varpi} \leftarrow \{h_i, \boldsymbol{h}_c, h_f\}$ with $\varpi = \#\boldsymbol{h}_c$
    Evaluate MH_c^{ϖ} via $F(\mathrm{MH}_c^{\varpi}|\mathfrak{X})$ in (1) and using Pseudocode 1
    $\Delta E \leftarrow F(\mathrm{MH}_c^{\varpi}|\mathfrak{X}) - F(\mathrm{MH}^{\varpi}|\mathfrak{X})$ and $\Theta \leftarrow \Theta(1 - \delta)$
    if $u_r \leq \exp(-\Delta E/\Theta)$ **then**
        $\mathrm{MH}^{\varpi} \leftarrow \mathrm{MH}_c^{\varpi}$
    end if
    if $F(\mathrm{MH}_c^{\varpi}|\mathfrak{X}) < F(\mathrm{MH}_*^{\varpi}|\mathfrak{X})$ **then**
        $\mathrm{MH}_*^{\varpi} \leftarrow \mathrm{MH}^{\varpi}$ and $s \leftarrow 0$
    else
        $s \leftarrow s + 1$
    end if
    $t \leftarrow t + 1$
end while

Pseudocode 3. Procedure for generating a neighbour hyper-heuristic

procedure GetNeighbour($\boldsymbol{h}$, $\mathfrak{H}$, $\varpi_{\max}$)
    if $\varpi \geq \varpi_{\max}$, **then** `ActionSet` $\leftarrow$ {`Del`, `Per`} ▷ $\varpi = \#\boldsymbol{h}$
    else if $\varpi \leq 1$, **then** `ActionSet` $\leftarrow$ {`Add`, `Per`}
    else `ActionSet` $\leftarrow$ {`Add`, `Del`, `Per`}, **end if**
    `Action` $\leftarrow$ ChooseRandomly(`ActionSet`)
    if `Action` == `Add` **then**
        $h_a \leftarrow$ ChooseRandomly($\mathfrak{H} \backslash \boldsymbol{h}$)
        $k \sim \mathcal{U}\{1, \varpi + 1\}$
        $\boldsymbol{h}' \leftarrow \left(\cup_{i=1}^{k-1} h_i\right) \cup h_a \cup \left(\cup_{j=k}^{\varpi} h_j\right)$, $\forall\, h_i, h_j \in \boldsymbol{h}$
    else if `Action` == `Del` **then**
        $h_d \leftarrow$ ChooseRandomly($\mathfrak{H}$)
        $\boldsymbol{h}' \leftarrow \boldsymbol{h} \backslash h_d$
    else
        $h_p \leftarrow$ ChooseRandomly($\mathfrak{H} \backslash \boldsymbol{h}$)
        $k \sim \mathcal{U}\{1, \varpi\}$ and $\boldsymbol{h}' \leftarrow \boldsymbol{h}$
        $h'_k \leftarrow h_p : h'_k \in \boldsymbol{h}'$
    end if
    return $\boldsymbol{h}'$
end procedure

This object has the default configuration variables (`exp_config`, `hh_config`, and `prob_config`), which can be modified by entering them as arguments. Bear in mind that these arguments are dictionaries and they do not need to have all the keys or fields; only those different from default values must be provided. At this point, the experiment can be started in a fashion as simple as with other objects, that is using the `run` method. Results are saved in "./data_files/raw/', where each combination function and dimensionality corresponds to a subfolder containing the JSON files for each iteration. These files are generated internally by the Hyper-heuristic module (see Appendix B.6). Moreover, a detailed implementation of this module is exemplified in Appendix B.7. Further details about the experiment module can be consulted in Appendix C.7.

3. Impact

CUSTOMHyS is just a small step towards a general theory of heuristics. The main impact of this framework is to facilitate the exploration of new metaheuristic techniques for solving a wide variety of problems. In this first version, we provide several continuous optimisation problems, but as the literature has proven, heuristic methods are versatile enough to be implemented in other domains. Moreover, we present a standard and intuitive way of tailoring metaheuristics using search operators as building blocks. We believe this can stop the frenetic tendency of proposing 'novel' algorithms with bizarre metaphors. Through it, researchers can focus on what is essential: to polish the methods, to explore new operators, and to propose new ways for their interactions. The proposed framework is, take or leave words, a laboratory of metaheuristics. Thus it provides an environment for limitless research and the development of new contributions to the community. Our software offers enough tools for exploring a specific combination of search operators to build a metaheuristic and then to test it over a set of problems. For example, when a researcher states a hypothesis based on her/his experience about particular methods, she/he can easily combine their corresponding operators in a single algorithm (what is called hybrid method in the literature) by using CUSTOMHyS. Other research questions that can be pursued with this framework are, for instance, how the initialisation affects the overall performance of a method, and which is the best stopping criteria for practical implementations.

By using this building block scheme, we can think about multiple extensions, some of which are mentioned next. Using a circuit analogy, consider that the scheme proposed for metaheuristics exhibits a serial topology. Then, it is possible to explore MHs with different topologies, such as parallel, star, delta, among others. In the case of MHs with a parallel topology, we can enclose some well-known approaches, *e.g.,* those employing sub-populations or neighbourhoods, such as Unified Particle Swarm Optimisation [42]. However, here we are not limited to using the same search operators for each neighbourhood. Besides, the topology assumption considers that all agents perform the same number of operations. However, our strategy is flexible enough to accept different operations for each branch in the MH scheme.

Within the framework, we also provide a large number of benchmark functions from the literature. Different sources have tried to condense and document several functions. But, while we were collecting them, we noticed that some problems have multiple names, wrongly documented characteristics, and missing or erroneous theoretical optima. For that reason, we made an effort to revise 107 optimisation functions and code them in a standard form. Notwithstanding, improvements and suggestions are well received.

The software is public, and we encourage the scientific community to contribute to its continuous development. Hence, it is intended to reach each corner of the world where a practitioner or researcher needs using metaheuristics. For that reason, it was entirely coded in Python.

4. Conclusions

In this document, we introduced a hyper-heuristic (HH) framework based on Simulated Annealing (SA) for solving continuous optimisation problems by building customised population-based metaheuristics (MHs). We called it CUSTOMHyS after Customising Optimisation Metaheuristics via Hyper-heuristic Search. The chief motivation to develop such a framework is to halt the frenetic

trend of publishing metaheuristics with similar operations but disguised with 'creative' metaphors. So, we aim to provide a structured and innovative way to find a suitable method for solving a given problem, only using mathematical operations over a population. Another reason behind the proposed framework is to facilitate the design flow of many practitioners that utilise metaheuristics in diverse applications, where they have to choose among different methods or, at the very least, tune them. This may be a problem if one lacks moderate expertise of the metaheuristic optimisation algorithm to implement, as it may merely constitute a waste of time.

We believe this framework to be worthwhile, as it removes the need for handcrafting a metaheuristic, as well as the corresponding limitation of requiring solid foundations to do so. This is not an easy task, and a particular metaheuristic cannot fulfil the requirements for solving all the problems with guaranteed high performance, *i.e.,* the *No-Free-Lunch* theorem.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The research was supported by the Research Group in Intelligent Systems at the Tecnológico de Monterrey (México), the Project TEC-Chinese Academy of Sciences, and by the CONACyT, Mexico Basic Science Project with grant number 287479.

Appendix A. Problem and background

A.1. Optimisation

Optimisation is somehow implicit in nature. Even so, it mainly concerns mathematical procedures for reaching the best result of a problem in practical engineering scenarios [51]. The current software has been developed based on the problem definition given by a feasible domain, $\mathfrak{X}$, and by an objective function, $f(\vec{x})$, to minimise. The former is established by

$$\mathfrak{X} = \{\vec{x} \in \mathbb{R}^D \ : \ (\exists \vec{x}_l, \vec{x}_u \in \mathbb{R}^D)[\vec{x}_l \preceq \vec{x} \preceq \vec{x}_u]\}, \tag{A.1}$$

where $\vec{x}_l$ and $\vec{x}_u$ are the lower and upper boundary vectors, and D represents the dimensionality of the problem. With that in mind, a minimisation problem is stated as

$$\vec{x}_* = \underset{\vec{x}\in\mathfrak{X}}{\operatorname{argmin}} \left\{f(\vec{x})\right\}, \tag{A.2}$$

since $f(\vec{x}) : \vec{x} \in \mathfrak{X} \subseteq \mathbb{R}^D \to \mathbb{R}$ is a real-valued function defined on a set $\mathfrak{X} \neq \emptyset$. Therefore, $\vec{x}_* \in \mathfrak{X}$ corresponds to the optimal vector (or solution) that minimises the objective function, *i.e.,* $f(\vec{x}_*) \leq f(\vec{x}),\ \forall \vec{x} \in \mathfrak{X}$.

A.2. Benchmark functions

An objective function represents the problem to tackle; a benchmark function is a commonly used and acknowledged objective function. Such functions mainly depend on model detail and modeller knowledge. For example, a heat sink for electronic cooling can be designed under different paradigms. The designer must choose whether to use a pure heat transfer model or one based on entropy generation. The latter naturally requires much more analysis, conditions, and knowledge [52]. Hence, our intention is not to deepen it into a particular problem with a given objective function, but to present a framework for any problem that can be described by expressions (A.1) and (A.2). For this reason, we use benchmark problems for continuous optimisation to test the methods that our framework generates. These problems are well-known in the literature, and most of them are fully characterised [28,33]. The characteristics of an optimisation problem offer a glance at how difficult it is to solve. They can be classified as *a priori* and *a posteriori*. Characteristics from the former category are obtained by analysing the mathematical formulation of the problem and by visually inspecting its landscape [28]. In turn, the latter ones are obtained via specialised studies such as Exploratory Landscape Analysis (ELA). These two categories are usually nominal and ordinal ones, respectively. In this work, we consider six representative binary features from the first class of characteristics, such as [28,33,53,54]: *Continuity*, *Differentiability*, *Separability*, *Unimodality*, *Scalability*, and *Convexity*.

A.3. Optimisation solvers

Once the optimisation problem is stated, it requires to be solved. For simple problem functions, *e.g.,* polynomial ones, the standard mathematical procedure based on the necessary and sufficient conditions is more than suitable [51]. Nonetheless, when the objective function and the constraints are somehow complex due to the problem nature, may classical methods are unable to find the optimal solution. That is the main reason for the existence of approximate methods that employ flexible procedures to assess the solution. A simple but adequate technique is the Random Walk that, as its name indicates, perform steps led by a random sampling method [55]. This method is tracked as the starting point for the so-called heuristic algorithms, which are vastly employed in most fields of knowledge [1,12].

A.4. Heuristics

A heuristic is a procedure that creates or modifies a candidate solution for a given problem instance. There are many classifications of heuristics in the literature. Most of them relate to combinatorial optimisation domains [10], whilst they are rather scarce for continuous ones [12]. Particularly, we categorise continuous heuristics in three groups, extending the ideas of [10,12]: *low-level*, *mid-level*, and *high-level*. These relate to *simple heuristics*, *metaheuristics*, and *hyper-heuristics*, respectively. Certainly, all of them are heuristics but operate under different conditions and domains.

A.4.1. Simple heuristics

A simple heuristic (SH) is the atomic unit in terms of search techniques that interact directly with problem domains. Let $h \in \mathfrak{H}$ be a SH from the heuristic space $\mathfrak{H}$ such that it either produces, modifies or evaluates a candidate solution $\vec{x} \in \mathfrak{X}$, using a fitness metric, *e.g.,* its objective function value $f(\vec{x})$. SHs are commonly categorised as *constructive* and *perturbative*. As their names suggest, a constructive heuristic renders new solutions from scratch while a perturbative heuristic modifies existing ones [56]. We consider an additional category called *sensitive*. Thus, we describe four representative SHs from these categories as follows.

- An *initialiser* (h_i) is a *constructive* SH that generates a solution within the search space $\vec{x} \in \mathfrak{X}$ from scratch. The most common one in the literature is to place the agents within the feasible search space randomly, *e.g.,* by using the uniform distribution.
- A *perturbator* (h_p) is a *perturbative* SH that yields a modified candidate position $\vec{y} \in \mathfrak{X}$ from the current position $\vec{x} \in \mathfrak{X}$. It can also be called *search operator* because it is the core of all heuristic-based searches.

- A *selector* (h_s) is a *perturbative* SH that updates the current solution $\vec{x} \in \mathfrak{X}$ by using a candidate solution $\vec{y} \in \mathfrak{X}$ and a selection criterion. A perturbator always precedes a selector, so we assume that each search operator implicitly implements a selector. There are several standard selection criteria such as, *e.g.,* direct, greedy, and Metropolis update.
- A *finaliser* (h_f) is a *sensitive* SH that evaluates the quality of a solution $\vec{x} \in \mathfrak{X}$, during an iterative procedure. It can raise a stop flag (convergence check) into an iterative procedure.

A.4.2. Metaheuristics (MHs)

They are defined as master strategies which control SHs. MHs are trendy in the literature because of their proven performance on different scenarios [2,5]. An iterative procedure is called metaheuristic (MH) when it renders an optimal solution $\vec{x}_*$ for a given optimisation problem with an objective function $f(\vec{x})$, using a finite sequence of simple heuristics (SHs). These SHs are applied iteratively until a stopping condition is met. Such a definition is exemplified by Fig. A.2.

Furthermore, an inherent property of metaheuristics is the cardinality ϖ, which is defined as the number of search operators implemented in it, *i.e.,* #MH $= \varpi$ while disregarding its initialiser and finaliser. By way of standardisation, we denote a metaheuristic and its cardinality such as MH^{ϖ}, where $\text{MH}^{1} =$ MH.

A.4.3. Hyper-heuristics (HHs)

Many researchers describe Hyper-heuristics as high-level heuristics controlling simple heuristics in the process of solving an optimisation problem [10]. Therefore, HHs move in the heuristic space to find a heuristic configuration that solves a given problem [12]. With that in mind, a HH also solves an optimisation problem, but which is provided by

$$(\boldsymbol{h}_*; \vec{x}_*) = \underset{\boldsymbol{h}\in\mathfrak{H}^{\varpi}, \vec{x}\in\mathfrak{X}}{\operatorname{argmax}} \{F(\boldsymbol{h}|\mathfrak{X})\} \tag{A.3}$$

where $\boldsymbol{h} \in \mathfrak{H}^{\varpi}$ is a heuristic configuration from the heuristic space $\mathfrak{H}$, and $F(\boldsymbol{h}|\mathfrak{X}) : \mathfrak{H}^{\varpi} \times \mathfrak{X} \to \mathbb{R}$ is a performance measure function. Therefore, the solution $\vec{x}_* \in \mathfrak{X}$ and its corresponding fitness value $f(\vec{x}_*)$ are found when $\boldsymbol{h}$ is applied on $\mathfrak{X}$, so its performance $F(\boldsymbol{h}|\mathfrak{X})$ can also be determined. In other words, a HH searches for the optimal heuristic configuration that produces the optimal solution with maximum performance. Notice that a heuristic configuration $\boldsymbol{h} \in \mathfrak{H}^{\varpi}$ is a way of referring to a metaheuristic MH.

A.5. Population

Many heuristic algorithms implement multi-search procedures to increase their exploration and exploitation capabilities. Such procedures are assigned to different agents or individuals that comprise a population. Some authors label them using nature-inspired Refs. [4,5], for example, genes, cells, bees, fish, hawks, whales, stars, galaxies, so forth. In this work, the population $X(t)$ is said to be a finite set of N candidate solutions (agents or individuals) for an optimisation problem given by $\mathfrak{X}$ and $f(\vec{x})$ at time t in an iterative procedure, *i.e.,* $X(t) = \{\vec{x}_1(t), \vec{x}_2(t), \ldots, \vec{x}_N(t)\}$. Then, $\vec{x}_n(t) \in \mathfrak{X}$ denotes the position of the nth search agent of, let us say, the population $X(t)$ of size N.

Appendix B. Illustrative examples

B.1. Benchmark functions module

All benchmark functions are coded as classes, so an object from a particular function corresponds to the problem of a specific dimensionality. For example, consider the Sphere or De Jong function in two dimensions (Function 83 from Table 1), which is called such as:

```
import benchmark_func as bf
fun = bf.Sphere(2)
```

This problem object, as well as the others, has methods and attributes that allow the manipulation of the problem. For example, we can evaluate a position, modify the problem domain, get the problem features with a specific encoding, and plot the function (in 2D) as shown:

```
fun.get_function_value([0, 0]) # Out: 0.0
fun.set_search_range([-100, -100], [100, 100])
fun.get_features(wrd='1', fts=['Separable', '
    Unimodal']) # Out: '11'
fun.plot()
```

Fig. B.3 shows the resulting plot for the last command and two additional plots from different problems.

B.2. Tools module

The `printmsk` function is included in the tools module. It is quite useful to know the structure of nested variables, for example, in large arrays or lists. Since these are the typical outputs of program modules, not restricted to our proposed framework, it can be used to support the development of other programs. To illustrate this function, we present the following example:

```
import tools as tl
variable = {"par1": [1, ['val1', 1.23]], "par2" :
    -4.5,
            "par2": {"subpar1": 7, "subpar2":
    (0.1, [10, 11])}}
tl.printmsk(variable)
# Out:
#  |-- {dict: 3}
#  |   |-- par1 = {list: 2}
#  |   |   |-- 0 = {int}
#  |   |   |-- 1 = {list: 2}
#  |   |   |   |-- 0 = {str}
#  |   |   |   |-- 1 = {float}
#  |   |-- par2 = {float}
#  |   |-- par3 = {dict: 2}
#  |   |   |-- subpar1 = {int}
#  |   |   |-- subpar2 = {tuple: 2}
#  |   |   |   |-- 0 = {float}
#  |   |   |   |-- 1 = {list: 2}
#  |   |   |   |   |-- 0 = {int}
#  :   :   :   :   :
```

B.3. Population module

In the following example, we show the steps for create and initialise a population, which corresponds to that implemented within the Metaheuristic module:

```
import benchmark_func as bf
import population as pp
fun = bf.Rastrigin(3)
pop = pp.Population(fun.get_search_range())
pop.initialise_positions('vertex')
pop.evaluate_fitness(lambda x: fun.
    get_function_value(x))
pop.update_positions(level='population', selector=
    'all')
```

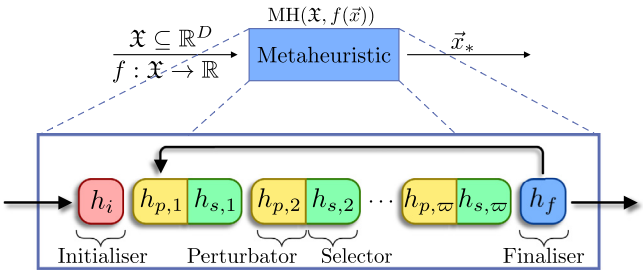


Fig. A.2. Scheme of a metaheuristic composed by an initialiser h_i, ϖ perturbators $h_{p,j}$ and selectors $h_{s,j}$, $\forall\, j \in \{1, \ldots, \varpi\}$, and a finaliser h_f. It is represented as a sequence of simple heuristics that iterates until a stopping flag (given by the finaliser) is raised.

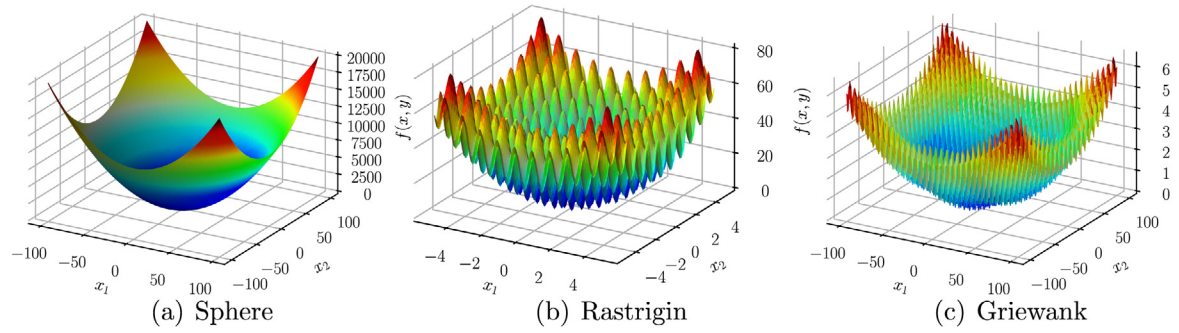


Fig. B.3. Illustrative example of two-dimensional benchmark functions.

```
pop.update_positions(level='particular', selector='all')
pop.update_positions(level='global', selector='greedy')
print(pop.get_state())
# Out: x_best = [-1.28 -1.28 -1.28], f_best = 40.536639
```

In this case, we picked the Rastrigin problem (function 59 from Table 1) in a three-dimensional domain. Notice that first and second arguments of `pop.update_positions` correspond to the level of selection and the selector employed, respectively. During a metaheuristic procedure (Appendix A.4.2), only the selector is applied to the 'population'; the greedy selector is utilised for the other levels. Plus, the variable `__selectors__` contains all the selectors available for updating the population positions:

```
pp.__selectors__ # Out: ['all', 'greedy', 'metropolis', 'probabilistic']
```

Furthermore, as mentioned above, the `initialise_positions` method can perform either a random or vertex distribution of the population initial positions. The latter refers to the assignment of agents at the vertices of nested hypercubes. Fig. B.4 illustrates these two options for a population of 32 individuals in two and three dimensions.

B.4. Operators module

Consider the population initialised in the previous example, *i.e.*, Appendix B.3. Therefore, to apply the SO from CFO (First row of Table 2) and update the population, the following instructions must be executed:

```
import operators as op
op.central_force_dynamic(pop, alpha=1.0) # gravity=1e-3, beta=1.5, dt=1
pop.evaluate_fitness(lambda x: fun.get_function_value(x))
pop.update_positions(level='population', selector='all')
pop.update_positions(level='global', selector='greedy')
print(pop.get_state())
# Out: x_best = [-2.188765 -2.188765  2.147112], f_best = 30.661195
```

It is noteworthy that the second instruction can be replaced with any search operator from Table 2.

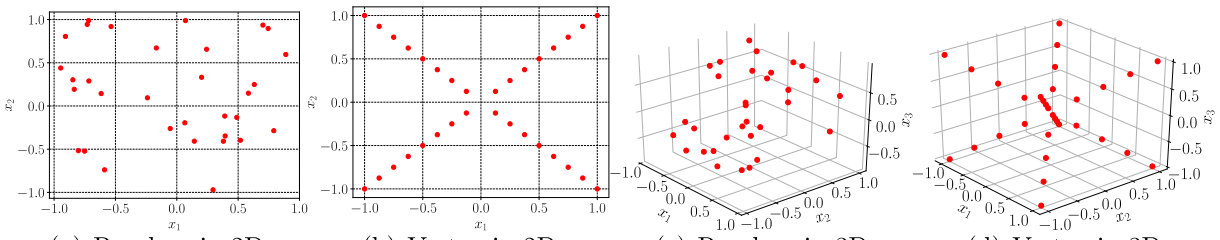
B.5. Metaheuristic module

An illustrative example of how to deploy a metaheuristic, using specific simple heuristics, for solving a given problem is shown below.

```
import benchmark_func as bf
import metaheuristic as mh
fun = bf.Rastrigin(3)
heur = [('differential_mutation', {'expression': 'current-to-best',
         'num_rands': 2, 'factor': 1.0}, 'greedy'),
        ('differential_crossover', {'crossover_rate': 0.2,
         'version': 'binomial'}, 'greedy')]
prob = fun.get_formatted_problem ()
met = mh.Metaheuristic(prob, heur, num_iterations=1000)
met.run()
print('x_best = {}, f_best = {}'.format(*met.get_solution()))
# Out: x_best = [-2.5357e-10  6.7133e-10 -1.2128e-10], f_best = 0.0
```

In this example, a metaheuristic based on the differential mutation and crossover operators is implemented to minimise the three-dimensional Rastrigin function. As the reader may have already noticed, this sequence of operators corresponds to the well-known Differential Evolution (DE) algorithm [40]. In the current version of this framework, the metaheuristic process halts when it reaches a maximum number of iterations (`num_iterations`). We plan to include more predefined stopping criteria in an upcoming release.

Furthermore, the most relevant attribute of a MH object after it has run is `historical`, which contains information of the best



(a) Random in 2D (b) Vertex in 2D (c) Random in 3D (d) Vertex in 3D

Fig. B.4. Illustrative example of random and vertex initialisation schemes.

position and its corresponding fitness, and the centroid and radius of the whole population for each iteration carried out. Fig. B.5 presents the evolution of the optimisation process in terms of fitness values, centroid positions, and radius values.

B.6. Hyper-heuristic module

To illustrate how this routine works, we present an example quite similar to that discussed in the previous Appendix B.5. However, this time, our approach is based on a hyper-heuristic instead of a metaheuristic. For this case, we selected the Griewank problem (function 31 from Table 1) in a five-dimensional domain.

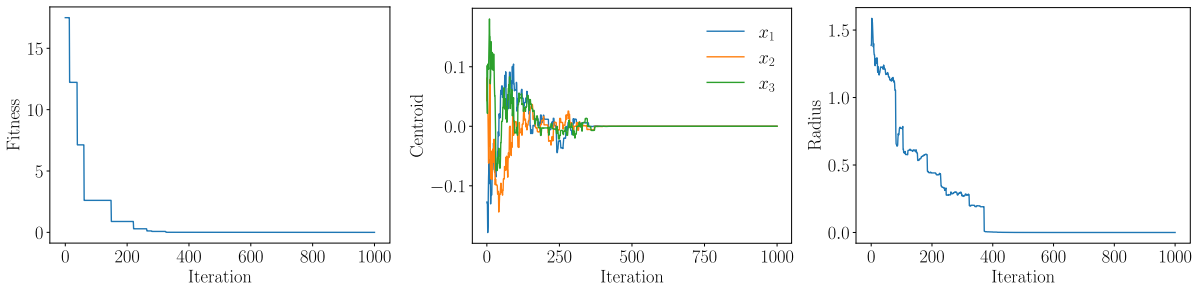
```
1  import benchmark_func as bf
2  import hyperheuristic as hh
3  import tools as tl
4  dime = 5
5  fun = bf.Griewank(dime)
6  cate = fun.get_features()
7  prob = fun.get_formatted_problem()
8  weig = tl.read_json('./collections/
       operators_weights.json')[
9      str(dime)][cate]
10 hyp = hh.Hyperheuristic('default.txt', problem=
       prob,
11     file_label='Griewank-5D-Exp1', weights_array=
       weig)
12 sol, perf, e_sol = hyp.run()
13 # Out:
14 # Exp1 :: Step: 0, Perf: 2.827145437019792, e-Sol:
       [141]
15 # Exp1 :: Step: 1, Perf: 1.821059930343987, e-Sol:
       [ 21 141]
16 # Exp1 :: Step: 2, Perf: 1.8171469534981388, e-Sol
       : [ 21  14 141]
17 # Exp1 :: Step: 3, Perf: 1.7022596777296206, e-Sol
       : [ 20  14 141]
18 # Exp1 :: Step: 5, Perf: 1.5934422384843325, e-Sol
       : [ 23  20 141]
19 # Exp1 :: Step: 7, Perf: 1.4207810829652476, e-Sol
       : [ 23 149  19]
20 # Exp1 :: Step: 10, Perf: 0.7424716708658518, e-
       Sol: [149 130]
21 # Exp1 :: Step: 15, Perf: 0.4774924758368956, e-
       Sol: [177]
22 # Exp1 :: Step: 30, Perf: 0.3627368911878881, e-
       Sol: [ 91 174]
23 # Exp1 :: Step: 32, Perf: 0.25850960815321566, e-
       Sol: [ 91 115 197]
24 # Exp1 :: Step: 49, Perf: 0.11767273680460102, e-
       Sol: [ 28 114   1]
25 print(sol)
26 # Out:
27 # [['differential_mutation'
28 #   {'expression': 'rand-to-best-and-current', '
       num_rands': 1,
29 #   'factor': 1.0}
30 #   'probabilistic']
31 #   ['genetic_crossover'
32 #   {'pairing': 'random', 'crossover': 'linear_0.5
       _0.5',
33 #   'mating_pool_factor': 0.4}
34 #   'greedy']
35 #   ['central_force_dynamic'
36 #   {'gravity': 0.001, 'alpha': 0.01, 'beta': 1.5,
       'dt': 1.0} 'all']]
```

As we did before, the function and its features are read to get the binary-encoded category (see line 6). With such information and the number of dimensions, we load the probability distribution for the search operators of the "default.txt" collection (line 8). Therefore, we create the hyper-heuristic object and execute the search (lines 10–12). In this example, the default configuration of parameters is used for the HH object, such as:

```
parameters = {
    'cardinality': 3, # Max. number of search
    operators
    'num_iterations': 100, # Max. number of
    iterations
    'num_agents': 30, # Number of agents (population
     size)
    'num_replicas': 50, # Number of replicas per
    metaheuristic
    'num_steps': 100, # Max. number of steps (hyper-
    heuristics)
    'stagnation_percentage': 0.3, # Allowed
    stagnation percentage
    'max_temperature': 200, # Initial temperature (
    Simulated Annealing)
    'cooling_rate': 0.05 # Cooling rate (Simulated
    Annealing)
    }
```

Certainly, a modified version of this variable can be used based on user requirements. Furthermore, note that `hyp.run` returns three arguments, `sol`, `perf`, and `e_sol`, corresponding to the solution (*i.e.,* a sequence of simple heuristics $\boldsymbol{h}$ with full details), the performance value of a metaheuristic represented by this sequence, and the encoded version of the solution (*i.e.,* the indices of search operators), respectively. Plus, whilst the HH search procedure is running, the solution state is printed on screen and saved as a JSON file on each improvement. Such outputs can be observed in lines 13–24 of the code above, where the step number, the current performance value, and the current encoded solution, are indicated. It is worth mentioning that for the first ten steps, the encoded solution somehow evolves through the addition, change, and removal of search operators, based on the process described in Pseudocode 3. In the next steps, no improvement occurs, and so nothing is printed. Now, take a look at the encoded solution of the last improving step: `[28 114 1]`. This corresponds to the solution printed by the command in line 25. The first simple heuristic h_{28} is the differential mutation operator (cf. Table 2) with a specific combination of parameters and a selector. Likewise, the second (h_{114}) and third (h_1) heuristics are the genetic crossover and the central force dynamic operators, respectively. With that in mind, we can say that the metaheuristic for this sequence has a cardinality of three ($\varpi = 3$). Such a MH and can be written as $\text{MH}^3 = \{h_{28}, h_{114}, h_1\}$.

To conclude the explanation of this module, the files generated during the HH procedure are in the folder "./data_files/raw/".



(a) Fitness values (b) Centroid positions (c) Radius values

Fig. B.5. Example of historical values registered while performing an optimisation procedure.

Consider that each step is stored as a JSON file with plenty of information that is commonly ignored; but it could be used for an in-depth analysis. However, for practical purposes, we preprocess all the step files into a single JSON file. To do so, we utilise the `preprocess_files` method from the `Tools` module as exemplified below.

```
import tools as tl
tl.preprocess_files('./data_files/raw/',
    output_name='Exp1',
                    only_laststep=False)
data = tl.read_json('./data_files/Exp1.json')['
    results'][0]
```

Notice that the data variable is the first element of the summarised file because we only solve one problem. So, if various problems are solved, several positions must be considered. Moreover, by using the information of the data variable, we present a possible manner to illustrate how the hyper-heuristic search evolves (Fig. B.6). For the sake of brevity, plotting commands are not provided; also because they are somehow trivial. In Fig. B.6, one observes the fitness values of the HH search under different levels of detail. At the highest level, violin plots show the last fitness value found for each execution of the heuristic sequence working as a metaheuristic. Remember that each candidate metaheuristic is run several times to calculate its performance, as shown in (1). Then, we select the first and last candidate solutions, as well as an intermediate one, *i.e.,* steps 0, 7, and 49 of the HH procedure. At the detailed level, Fig. B.6 shows the convergence curves (fitness vs. iterations) per repetition, depicted in green lines, for each one of these candidates. Remark the difference between the words 'iteration' and 'step'. In this framework, we use the former only for the metaheuristic search, whereas the latter is restricted for hyper-heuristic evolution.

B.7. Experiment module

An experiment object requires three configuration variables, such as experiment (`exp_config`), hyper-heuristic (`hh_config`), and problem (`prob_config`). To create such an object is quite simple by using this module, as shown below:

```
import experiment as ex
expe = ex.Experiment()
expe.hh_config
# Out: {
# 'cardinality': 3, 'num_agents': 30, '
    num_iterations': 100,
# 'num_replicas': 10, 'num_steps': 100, '
    max_temperature': 200,
# 'stagnation_percentage': 0.3, 'cooling_rate': 0.05
# }
expe.exp_config
# Out: {
# 'experiment_name': 'demo_test', 'experiment_type':
    'default',
# 'heuristic_collection_file': 'default.txt',
# 'weights_dataset_file': 'operators_weights.json',
# 'use_parallel': True, 'parallel_pool_size': None,
# 'auto_collection_num_vals': 5
# }
expe.prob_config
# Out: {
# 'dimensions': [2, 5], 'functions': ['Csendes'], '
    is_constrained': True
# }
```

In this case, we showed the default parameters for `exp_config`, `hh_config`, and `prob_config`.

It is important to remark that the default function for the problem configuration variable, `expe.prob_config['functions']`, is chosen at random. Moreover, this module provides five additional predefined configurations for each variable, *i.e.,* `ex_configs`, `hh_configs`, and `pr_configs`. The default variables correspond to the first element of each list: `ex_configs[0]`, `hh_configs[0]`, and `pr_configs[0]`.

```
expe.run()
```

With these few commands and definitions, one can tweak the experiment object to run several optimisation procedures automatically. Notwithstanding, this is not a unique way to run experiments utilising the proposed framework. For example, in cases where the experiments comprise more than two problems to deal with, the steps mentioned above are somehow impractical. Assume that we need to find the best metaheuristic for each one of the functions in the benchmark functions module (currently they are 107 functions), for several dimensions, say 2, 5, 10, 20, 30, 40, and 50. So, our experiment must carry out 107×7 hyper-heuristic searches; which is the case of the non-default elements of the predefined problem configurations (`pr_configs[1:]`). For such kind of experiment, we may leave the machine working for some time or run the experiments remotely in another machine. Therefore, after setting up the experiment by adding new configuration variables or modifying the current ones, one can run the module as a script indicating the configuration index. For example, to run the same demonstrative experiment mentioned above, we can type the following command in the terminal:

```
python experiment.py 0
```

The argument `0` refers to the index of the default configuration. For both cases, *i.e.,* using the `run` method or running the module as a script, the experiment would print the current status of each hyper-heuristic search, as explained in Section 2.6. The following lines illustrate the prompt after running the experiment:

```
Csendes-2D-demo_test :: Step: 0, Perf: 0.000171, e-
    Sol: [67]
Csendes-5D-demo_test :: Step: 0, Perf: 1.431359, e-
    Sol: [61]
Csendes-2D-demo_test :: Step: 1, Perf: 6.88e-14, e-
    Sol: [199]
```

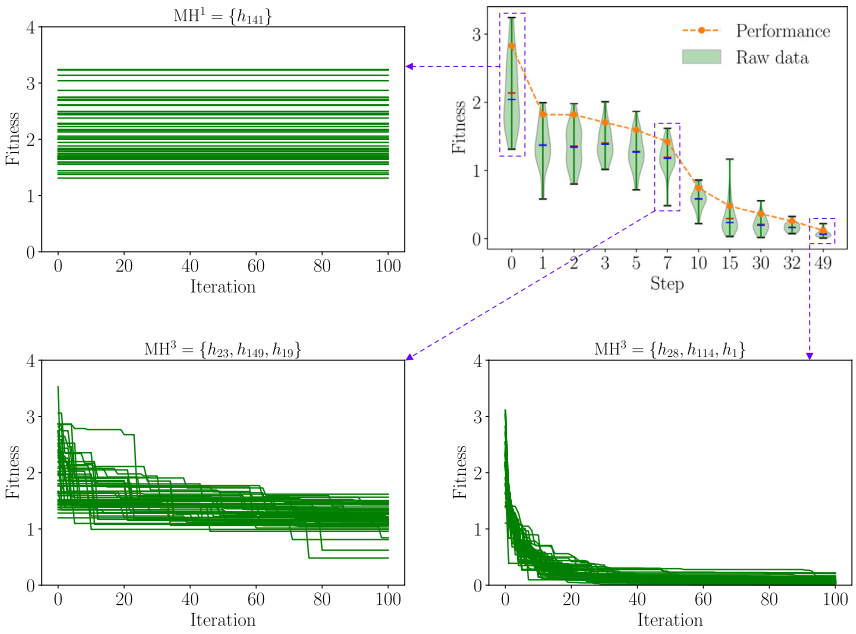


Fig. B.6. Evolution of the hyper-heuristic search where each step indicates the fitness values (violins) and the performance (orange dots) obtained by a candidate solution (metaheuristic). Three selected solutions (purple lines) detail their internal search within the problem domain for several repetitions (green lines). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
Csendes-5D-demo_test :: Step: 2, Perf: 0.654982, e-Sol: [74]
Csendes-2D-demo_test :: Step: 2, Perf: 8.88e-24, e-Sol: [ 41 199]
Csendes-5D-demo_test :: Step: 3, Perf: 0.006079, e-Sol: [74 98]
Csendes-2D-demo_test :: Step: 4, Perf: 1.10e-104, e-Sol: [196]
Csendes-5D-demo_test :: Step: 4, Perf: 1.37e-10, e-Sol: [121  74  98]
Csendes-2D-demo_test :: Step: 12, Perf: 5.47e-114, e-Sol: [ 15 103]
Csendes-5D-demo_test :: Step: 10, Perf: 2.20e-15, e-Sol: [173 117]
Csendes-2D-demo_test :: Step: 16, Perf: 3.18e-199, e-Sol: [103  20]
Csendes-5D-demo_test :: Step: 20, Perf: 8.54e-16, e-Sol: [ 35 117 139]
Csendes-2D-demo_test done!
Csendes-5D-demo_test :: Step: 37, Perf: 4.70e-66, e-Sol: [195 153]
Csendes-5D-demo_test :: Step: 56, Perf: 1.85e-123, e-Sol: [194  87 203]
Csendes-5D-demo_test :: Step: 63, Perf: 0.0, e-Sol: [ 39  29 106]
Csendes-5D-demo_test done!
```

Recall that for this experiment, we set the arbitrarily selected Csendes function (problem 15 from Table 1) in two- and five-dimensional domains. It is worth mentioning that each instance of the problem was executed in a different processor, which explains why the printed status lines are somehow unorganised. For the two-dimensional instance, the last performance value achieved was 3.18×10^{-199} at the 16th step. Whereas, for the five-dimensional case, it was 0.0 at the 63rd step. In practical terms, these two performances equal zero, which is an excellent performance value. Remember that this metric is determined as the sum of the median and the interquartile range of fitness values. So, a value of zero is the most desired case, especially for stochastic metaheuristic algorithms.

The resulting files from this experimental deployment are:

```
./data_files/
|-- raw
    |-- Csendes-2D-demo_test
    |   |-- 0-08_12_2020_12_24_43.json
    |   |-- 1-08_12_2020_12_24_49.json
    |   |-- 2-08_12_2020_12_24_59.json
    |   |-- 4-08_12_2020_12_25_13.json
    |   |-- 12-08_12_2020_12_26_16.json
    |   |-- 16-08_12_2020_12_26_51.json
    |-- Csendes-5D-demo_test
        |-- 0-08_12_2020_12_24_43.json
        |-- 2-08_12_2020_12_24_57.json
        |-- 3-08_12_2020_12_25_09.json
        |-- 4-08_12_2020_12_25_31.json
        |-- 10-08_12_2020_12_26_44.json
        |-- 20-08_12_2020_12_29_46.json
        |-- 37-08_12_2020_12_33_40.json
        |-- 56-08_12_2020_12_37_16.json
        |-- 63-08_12_2020_12_38_59.json
```

In the same way we did with the resulting file from running a unique hyper-heuristic search, we can also pre-process and visualise these data. From this point onward, our imagination is the only limit.

Appendix C. Attributes and methods for each module

C.1. Benchmark functions module

`BasicProblem`: Base class for all the problem classes in this module.

- `func_name`: Function name in readable form like those in Table 1.
- `get_features`: Return some categorical features with a defined format.
- `get_formatted_problem`: Return the problem in a dictionary to use in a solving procedure. This dictionary has three elements:
  - `function`: A lambda function to evaluate a position.
  - `boundaries`: A tuple with the lower and upper boundaries.

`is_constrained`: A flag to restrict individuals from exploring outside the domain.

`get_function_value`: Evaluate the problem function.

`get_optimal_fitness`: Return the theoretical optimum value (some functions do not have it).

`get_optimal_solution`: Return the theoretical solution (some functions do not have it).

`get_search_range`: Return the problem domain given by the lower and upper boundaries.

`plot`: Plot the current problem in 2D.

`plot_object`: The plot object for further manipulations.

`save_dir`: The folder where a plot will be saved.

`save_fig`: Save the 2D representation of the problem function.

`set_noise_level`: Specify the noise level.

`set_noise_type`: Specify the noise distribution to add.

`set_offset_domain`: Add an offset value for the problem domain.

`set_offset_function`: Add an offset value for the problem function.

`set_scale_domain`: Add a scale value for the problem domain.

`set_scale_function`: Add a scale value for the problem function.

`set_search_range`: Define the problem domain given by the lower and upper boundaries.

`list_functions`: List all available functions in screen.

`for_all`: Read a determined property or attribute for all the problems and return a list.

C.2. Tools module

`check_fields`: Return a dictionary with default keys and values updated with the dictionary entered.

`listfind`: Return all indices of a list corresponding to a value.

`preprocess_files`: Return data from results saved in the main folder.

`printmsk`: Print the meta-skeleton of a variable with nested variables.

`read_json`: Return data from a JSON file.

`read_subfolders`: Return a list of all subfolders contained in a folder.

`revise_results`: Revise a folder with subfolders and check if there are subfolder repeated.

`save_json`: Save a variable composed with diverse types of variables.

C.3. Population module

`Population`: Class of population objects.

`current_best_fitness`: The current best fitness value found in the population.

`current_best_position`: The current best position found in the population.

`current_worst_fitness`: The current worst fitness value found in the population.

`current_worst_position`: The current worst position found in the population.

`evaluate_fitness`: Evaluate the population positions in the problem function.

`fitness`: The current fitness values of all individuals in the population.

`get_positions`: Return the current population positions.

`get_state`: Return a string containing the current state of the population.

`global_best_fitness`: The best fitness value found since initialisation.

`global_best_position`: The best position found since initialisation.

`initialise_positions`: Initialise population by an initialisation scheme.

`is_constrained`: A flag to restrict individuals from exploring outside the problem domain.

`iteration`: The current value of the iteration counter.

`metropolis_boltzmann`: The Boltzmann constant for Metropolis selector.

`metropolis_rate`: The cooling rate for Metropolis selector.

`metropolis_temperature`: The initial temperature for Metropolis selector.

`num_agents`: The number of agents or population size (N).

`num_dimensions`: The number of dimensions or variables (D).

`particular_best_fitness`: The best fitness value of each individual found since initialisation.

`particular_best_positions`: The best position of each individual found since initialisation.

`positions`: The current positions ($N \times D$ array) of all individuals in the population.

`previous_fitness`: The previous fitness values of all individuals in the population.

`previous_positions`: The previous positions ($N \times D$ array) of all individuals in the population.

`previous_velocities`: The previous velocities ($N \times D$ array) of all individuals in the population.

`probability_selection`: The probability value for Probabilistic selector.

`set_positions`: Modify the current population positions.

`update_positions`: Update the population positions according to the level and selection scheme.

`velocities`: The current velocities ($N \times D$ array) of all individuals in the population.

C.4. Operators module

`build_operators`: Create a text file containing a list of all the available search operators.

`central_force_dynamic`: Apply the central force dynamic from Central Force Optimisation (CFO).

`differential_crossover`: Apply the differential crossover from Differential Evolution (DE).

`differential_mutation`: Apply the differential mutation from DE.

`firefly_dynamic`: Apply the firefly dynamic from Firefly algorithm (FA).

`genetic_crossover`: Apply the genetic crossover from Genetic Algorithm (GA).

`genetic_mutation`: Apply the genetic mutation from GA.

`get_rotation_matrix`: Determine the rotation matrix by multiplying all 2D rotation matrices.

`gravitational_search`: Apply the gravitational search from Gravitational Search Algorithm (GSA).

`local_random_walk`: Apply the local random walk from Cuckoo Search (CS).

`obtain_operators`: Generate a list of all the available search operators.

`process_operators`: Decode the list of search operator and deliver two lists, one with the ready-to-execute strings of these operators and another with strings of their associated selectors.

`random_flight`: Apply the random flight from Random Search (RS).

`random_sample`: Apply the random_sample to the population's positions.

`random_search`: Apply the random search from Random Search (RS).

`spiral_dynamic`: Apply the spiral dynamic from Stochastic Spiral Optimisation (SSO).

`swarm_dynamic`: Apply the swarm dynamic from Particle Swarm Optimisation (PSO).

C.5. Metaheuristic module

`Metaheuristic`: Class of metaheuristic objects.

- `get_solution`: Deliver the last position and fitness value obtained after `run`.
- `run`: Run the metaheuristic for solving the defined problem.
- `historical`: A dictionary with the historical values.
  - `fitness`: A list with the best fitness values obtained for each iteration.
  - `position`: A list with the best position obtained for each iteration.
  - `centroid`: A list with the population's centroid obtained for each iteration.
  - `radius`: A list with the radius values of the population for each iteration.
- `num_agents`: The number of agents or population size (N).
- `num_dimensions`: The number of dimensions or variables (D).
- `num_iterations`: The number of iterations to carry out.
- `operators`: The list of operators to apply for each iteration.
- `selectors`: The list of selectors to apply after each operator.
- `verbose`: A flag to enable printing the search status when each iteration is completed.

C.6. Hyper-heuristic module

`Hyperheuristic`: Class of hyperheuristic objects.

- `basic_metaheuristics`: Perform a brute force procedure solving the problem via all the predefined metaheuristics.
- `brute_force`: Perform a brute force procedure solving the problem via all the available search operators without integrating a high-level search method.
- `evaluate_metaheuristic`: Evaluate the current sequence of operators as a metaheuristic.
- `file_label`: A tag or label for saving files.
- `get_performance`: Return the performance from fitness values obtained from running a metaheuristic several times. This method uses information from `get_statistics`.
- `get_statistics`: Return statistics from all the fitness values found after running a metaheuristic several times. The statistics are in a dictionary with the following keys:
  - `nob`: Number of observations.
  - `Min`: Minimum of fitness values.
  - `Max`: Maximum of fitness values.
  - `Avg`: Average of fitness values.
  - `Std`: Standard deviation of fitness values.
  - `Skw`: Skewness of fitness values.
  - `Kur`: Kurtosis of fitness values.
  - `IQR`: Interquartile range of fitness values.
  - `Med`: Median of fitness values.
  - `MAD`: Median absolute deviation of fitness values.
- `heuristic_space`: The heuristic space or search space collection.
- `num_operators`: The number of search operators or size of the heuristic space.
- `parameters`: A dictionary of the parameters to implement the hyper-heuristic (HH) procedure.
  - `cardinality`: Maximum number of search operators allowed to build the metaheuristic.
  - `num_iterations`: Maximum number of iterations that a metaheuristic perform.

`num_agents`: Number of agents used for the population in the metaheuristic.

`num_replicas`: Times that a single process (a metaheuristic solving a problem) is repeated.

`num_steps`: Number of steps that the HH procedure carry out.

`stagnation_percentage`: Stagnation parameter for the HH search.

`max_temperature`: Initial temperature for the HH procedure based on Simulated Annealing.

`cooling_rate`: Cooling rate for the HH procedure based on Simulated Annealing.

`problem`: A dictionary containing the 'function', 'boundaries', and 'is_constrained', just like the output of `get_formatted_problem` in Appendix C.1.

`run`: Run the hyper-heuristic based on Simulated Annealing (SA) to find the best metaheuristic.

`weights_array`: The weights of the search operators, if there is *a priori* information about them.

C.7. Experiment module

`ex_configs`: A list of predefined experiments' configurations similar to `Experiment.exp_config`.

`Experiment`: Class of experiment objects.

`exp_config`: A dictionary for the experiment's configuration with the following keys:

`auto_collection_num_vals`: Number of parameter values for creating an automatic heuristic collection (if so).

`experiment_name`: Label for identifying the experiment.

`experiment_type` This type is employed to know how to save the data files. It can be 'brute_force', 'basic_metaheuristics', and 'hyper_heuristic' (or anything else).

`parallel_pool_size`: Number of processors available to use.

`heuristic_collection_file`: File name of the heuristic space located in './collections/'.

`use_parallel`: Flag to run the experiment utilising a pool of processors.

`weights_dataset_file`: Path where is the file of weights or probability distribution of heuristic space.

`hh_config`: A dictionary for the hyper-heuristic's configuration with the following keys:

`cardinality`: Maximum cardinality (number of operators) allowed to build metaheuristics.

`cooling_rate`: Cooling rate for the Simulated Annealing operations within the HH.

`max_temperature`: Initial temperature for Simulated Annealing operations within the HH.

`num_agents`: Number of agents (population size) employed by the metaheuristic.

`num_iterations`: Maximum number of iterations used by each metaheuristic.

`num_replicas`: Number of replicas per metaheuristic.

`num_steps`: Maximum number of steps that the hyper-heuristic performs.

`stagnation_percentage`: Percentage of stagnation used by the HH.

`prob_config`: A dictionary for the problems' configuration with the following keys:

`dimensions`: List of dimensions for the problem domains.

`functions`: List of function names of the optimisation problems.

`is_constrained`: True if the problem domain is hard constrained.

`run`: Run the experiment according to the configuration variables entered.

`weights_data`: Weights of the search operators.

`hh_configs`: The list of predefined hyper-heuristics' configurations similar to `Experiment.hh_config`.

`pr_configs`: The list of predefined problems' configurations similar to `Experiment.prob_config`.

References

[1] Sörensen K, Sevaux M, Glover F. A history of metaheuristics. Handb Heuristics 2018;2(2):791–808. http://dx.doi.org/10.1007/978-3-319-07124-4_4.

[2] Hussain K, Salleh MNM, Cheng S, Shi Y. Metaheuristic research: a comprehensive survey. Artif Intell Rev 2019;52(4):2191–233.

[3] Adam SP, Alexandropoulos S-AN, Pardalos PM, Vrahatis MN. No free lunch theorem : A review. In: Demetriou I, Pardalos P, editors. Approximation and optimization. Cham: Springer; 2019, p. 57–82. http://dx.doi.org/10.1007/978-3-030-12767-1_5.

[4] Sörensen K. Metaheuristics—the metaphor exposed. Int Trans Oper Res 2015;22(1):3–18. http://dx.doi.org/10.1111/itor.12001.

[5] Dokeroglu T, Sevinc E, Kucukyilmaz T, Cosar A. A survey on new generation metaheuristic algorithms. Comput Ind Eng 2019;137:106040.

[6] Ahn CW. Practical genetic algorithms, vol. 18. Wiley-Interscience; 2006, http://dx.doi.org/10.1007/11543138_2.

[7] Storn R, Price K. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. J Global Optim 1997;11(4):341–59.

[8] Yang X-S, Deb S. Cuckoo search via Lévy flights. In: 2009 world congress on nature & biologically inspired computing (NaBIC). IEEE; 2009, p. 210–4.

[9] Del Ser J, Osaba E, Molina D, Yang X-S, Salcedo-Sanz S, Camacho D, et al. Bio-inspired computation: Where we stand and what's next. Swarm Evol Comput 2019;48:220–50.

[10] Burke EK, Hyde MR, Kendall G, Ochoa G, Özcan E, Woodward JR. A classification of hyper-heuristic approaches: revisited. In: Handbook of metaheuristics. Springer; 2019, p. 453–77.

[11] Gomes CP, Selman B. Algorithm portfolios. Artificial Intelligence 2001;126(1–2):43–62. http://dx.doi.org/10.1016/S0004-3702(00)00081-3.

[12] Pillay N, Qu R. Hyper-heuristics: Theory and applications. Springer; 2018.

[13] Sanchez M, Cruz-Duarte JM, Ortiz-Bayliss JC, Ceballos H, Terashima-Marin H, Amaya I. A systematic review of hyper-heuristics on combinatorial optimization problems. IEEE Access 2020;8:128068–95. http://dx.doi.org/10.1109/ACCESS.2020.3009318, https://ieeexplore.ieee.org/document/9139914/.

[14] Ochoa G, Hyde M, Curtois T, Vazquez-Rodriguez JA, Walker J, Gendreau M, et al. Hyflex: A benchmark framework for cross-domain heuristic search. In: European conference on evolutionary computation in combinatorial optimization; 2012. p. 136–147.

[15] Asta S, Özcan E, Parkes AJ. Batched mode hyper-heuristics. In: Nicosia P, Pardalos G, editors. In: Lecture notes in computer science, Berlin Heidelberg: Springer; 2013, p. 404–9. http://dx.doi.org/10.1007/978-3-642-44973-4_43.

[16] Van Onsem W, Demoen B. ParHyFlex: A framework for parallel hyper-heuristics. In: Belgian/Netherlands artificial intelligence conference; 2013. p. 231–8.

[17] Parkes AJ, Özcan E, Karapetyan D. A software interface for supporting the application of data science to optimisation. In: Lecture notes in computer science, Springer International Publishing; 2015, p. 306–11. http://dx.doi.org/10.1007/978-3-319-19084-6_31.

[18] Asta S, Özcan E. A tensor-based selection hyper-heuristic for cross-domain heuristic search. Inform Sci 2015;299:412–32. http://dx.doi.org/10.1016/j.ins.2014.12.020.
[19] Majeed H, Naz S. Deja Vu: a hyper heuristic framework with Record and Recall (2R) modules. Cluster Comput 2019;22(s3):7165–79. http://dx.doi.org/10.1007/s10586-017-1095-x.
[20] Swan J, Özcan E, Kendall G. Hyperion - A recursive hyper-heuristic framework. Lecture Notes in Comput Sci 2011;6683:616–30. http://dx.doi.org/10.1007/978-3-642-25566-3_48.
[21] Ryser-welch P, Miller JF. A review of hyper-heuristic frameworks. In: Proceedings of the Evo20 workshop. AISB; 2014, p. 7.
[22] Kheiri A, Keedwell E. A sequence-based selection hyper-heuristic utilising a hidden markov model. In: GECCO 2015 - Proceedings of the 2015 genetic and evolutionary computation conference. 2015, p. 417–24. http://dx.doi.org/10.1145/2739480.2754766.
[23] Sabar NR, Yi X, Song A. A bi-objective hyper-heuristic support vector machines for big data cyber-security. IEEE Access 2018;6:10421–31. http://dx.doi.org/10.1109/ACCESS.2018.2801792.
[24] Turky A, Sabar NR, Dunstall S, Song A. Hyper-heuristic local search for combinatorial optimisation problems. Knowl-Based Syst 2020;205:106264. http://dx.doi.org/10.1016/j.knosys.2020.106264.
[25] Hao X, Qu R, Liu J. A unified framework of graph-based evolutionary multitasking hyper-heuristic. IEEE Trans Evol Comput 2020;14(8). http://dx.doi.org/10.1109/TEVC.2020.2991717.
[26] Miranda PB, Prudêncio RB, Pappa GL. H3ad: A hybrid hyper-heuristic for algorithm design. Inform Sci 2017;414:340–54.
[27] Abell T, Malitsky Y, Tierney K. Fitness landscape based features for exploiting black-box optimization problem structure. Tech. rep. December, Copenhagen: IT University; 2012.
[28] Jamil M, Yang XS. A literature survey of benchmark functions for global optimisation problems. Int J Math Model Numer Optim 2013;4(2):150. http://dx.doi.org/10.1504/IJMMNO.2013.055204.
[29] Gavana A. Global optimization benchmarks and AMPGO. 2013, URL http://infinity77.net/global_optimization.
[30] Al-Roomi AR. Unconstrained single-objective benchmark functions repository. 2015, URL https://www.al-roomi.org/benchmarks/unconstrained.
[31] Ardeh MA. Benchmark function toolbox. 2016, URL http://benchmarkfcns.xyz/about/.
[32] Hansen N, Finck S, Ros R, Auger A. Real-parameter black-box optimization benchmarking 2009: noiseless functions definitions. Tech. rep. INRIA Saclay, 2009.
[33] Qu BY, Liang JJ, Wang ZY, Chen Q, Suganthan PN. Novel benchmark functions for continuous multimodal optimization with comparative results. Swarm Evol Comput 2016;26:23–34. http://dx.doi.org/10.1016/j.swevo.2015.07.003.
[34] Pohlheim H. Examples of objective functions. Retrieved 2007;4(10):2012.
[35] Sakuma J, Kobayashi S. Real-coded ga for high-dimensional k-tablet structures. Trans Jpn Soc Artif Intell 2004;19:28–37.
[36] Molga M, Smutnicki C. Test functions for optimization needs, Test functions for optimization needs 101. 2005.
[37] Suzuki H, Sawai H. Chemical genetic algorithms-coevolution between codes and code translation. In: Proceedings of the eighth international conference on artificial life (artificial life VIII); 2002. p. 164–72.
[38] Bergstra J, Bengio Y. Random search for hyper-parameter optimization. J Mach Learn Res 2012;13(Feb):281–305.
[39] Kirkpatrick S, Gelatt Jr CD, Vecchi MP. Optimization by simulated annealing optimization by simulated annealing. Science 1983;220(4598):671–80.
[40] Das S, Mullick SS, Suganthan PN. Recent advances in differential evolution-an updated survey. Swarm Evol Comput 2016;27:1–30. http://dx.doi.org/10.1016/j.swevo.2016.01.004.
[41] Kennedy J, Eberhart R. Particle swarm optimization (PSO). In: Proc. IEEE international conference on neural networks, Perth, Australia; 1995. p. 1942–8.
[42] Clerc M, Kennedy J. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. IEEE Trans Evol Comput 2002;6(1):58–73.
[43] Yang X-S, et al. Firefly algorithm. In: Nature-inspired metaheuristic algorithms, vol. 20. 2008, p. 79–90.
[44] Cruz-Duarte JM, Martin-Diaz I, Munoz-Minjares JU, Sanchez-Galindo LA, Avina-Cervantes JG, Garcia-Perez A, et al. Primary study on the stochastic spiral optimization algorithm. In: 2017 IEEE international autumn meeting on power, electronics and computing (ROPEC), vol. 1. IEEE; 2017, p. 1–6.
[45] Formato RA. Central force optimization: A new deterministic gradient-like optimization metaheuristic. Opsearch 2009;46(1):25–51. http://dx.doi.org/10.1007/s12597-009-0003-4.
[46] Biswas A, Mishra KK, Tiwari S, Misra AK. Physics-inspired optimization algorithms: A survey. J. Optim. 2013;2013:1–16. http://dx.doi.org/10.1155/2013/438152.
[47] Cruz-Duarte JM, Ivan A, Ortiz-Bayliss JC, Conant-Pablos SE, Terashima-Marín H. A primary study on hyper-heuristics to customise metaheuristics for continuous optimisation. In: 2020 IEEE congress on evolutionary computation (CEC); 2020. p. 1–8.
[48] Cruz-Duarte JM, Ivan A, Ortiz-Bayliss JC, Conant-Pablos SE, Terashima-Marín H, Shi Y. Hyper-heuristics to customise metaheuristics for continuous optimisation. Swarm Evol Comput 2020;33. [Under review].
[49] Garza-Santisteban F, Sanchez-Pamanes R, Puente-Rodriguez LA, Amaya I, Ortiz-Bayliss JC, Conant-Pablos S, et al. A simulated annealing hyper-heuristic for job shop scheduling problems. In: 2019 IEEE congress on evolutionary computation (CEC). IEEE; 2019, p. 57–64. http://dx.doi.org/10.1109/CEC.2019.8790296.
[50] Garza-Santisteban F, Cruz-Duarte JM, Amaya I, Ortiz-Bayliss C, Conant-Pablos SE, Terashima-Marin H. Influence of instance size on selection hyper-heuristics for job shop scheduling problems. In: 2019 IEEE symposium series on computational intelligence (SSCI). Xiamen, China: IEEE; 2019, p. 8.
[51] Rao SS. Engineering optimization: Theory and practice. John Wiley & Sons; 2009.
[52] Cruz-Duarte JM, Garcia-Perez A, Amaya-Contreras IM, Correa-Cely CR, Romero-Troncoso RJ, Avina-Cervantes JG. Design of microelectronic cooling systems using a thermodynamic optimization strategy based on cuckoo search. IEEE Trans Compon Packag Manuf Technol 2017;7(11):1804–12. http://dx.doi.org/10.1109/TCPMT.2017.2706305.
[53] Garden RW, Engelbrecht AP. Analysis and classification of optimisation benchmark functions and benchmark suites. In: Proceedings of the 2014 IEEE congress on evolutionary computation, CEC 2014 vol. 1. 2014, p. 1641–9. http://dx.doi.org/10.1109/CEC.2014.6900240.
[54] Dieterich JM, Hartke B. Empirical review of standard benchmark functions using evolutionary global optimization. Appl Math 2012;03(10):1552–64. http://dx.doi.org/10.4236/am.2012.330215, arXiv:1207.4318.
[55] Schumer MA, Steiglitz K. Adaptive step size random search. IEEE Trans Automat Control 1968;13(3):270–6. http://dx.doi.org/10.1109/TAC.1968.1098903.
[56] Woumans G, De Boeck L, Beliën J, Creemers S. A column generation approach for solving the examination-timetabling problem. European J Oper Res 2016;253(1):178–94.