Fast, flexible, and fearless: a rust framework for the modular construction of metaheuristics

Jonathan Wurth, Helena Stegherr, Michael Heider, Leopold Luley, Jörg Hähner

Angaben zur Veröffentlichung / Publication details:

Wurth, Jonathan, Helena Stegherr, Michael Heider, Leopold Luley, and Jörg Hähner. 2023. "Fast, flexible, and fearless: a rust framework for the modular construction of metaheuristics." In *GECCO '23 Companion: proceedings of the Companion Conference on Genetic and Evolutionary Computation, Lisbon, Portugal, July 15-19, 2023*, edited by Sara Silva and Luís Paquete, 1900–1909. New York, NY: ACM.
https://doi.org/10.1145/3583133.3596335.

Fast, Flexible, and Fearless: A Rust Framework for the Modular Construction of Metaheuristics

Jonathan Wurth
jonathan.wurth@uni-a.de
Universität Augsburg
Augsburg, Germany

Helena Stegherr
helena.stegherr@uni-a.de
Universität Augsburg
Augsburg, Germany

Michael Heider
michael.heider@uni-a.de
Universität Augsburg
Augsburg, Germany

Leopold Luley
leopold.luley@uni-a.de
Universität Augsburg
Augsburg, Germany

Jörg Hähner
joerg.haehner@uni-a.de
Universität Augsburg
Augsburg, Germany

ABSTRACT

We present MAHF, a Rust framework for the modular construction and subsequent evaluation of evolutionary algorithms, but also any other metaheuristic framework, including non-population-based and constructive approaches. We achieve high modularity and flexibility by splitting algorithms into components with a uniform interface and allowing communication through a shared blackboard. Nevertheless, MAHF is aimed at being easy to use and adapt to the specific purposes of different practitioners. To this end, this paper focuses on providing a general description of the design of MAHF before illustrating its application with a variety of different use cases, ranging from simple extension of the set of implemented components and the subsequent construction of algorithms not present within the framework to hybridization approaches, which are often difficult to realize in specialized software frameworks. By providing these comprehensive examples, we aim to encourage others to utilize MAHF for their needs, evaluate its effectiveness, and improve upon its application.

CCS CONCEPTS

• **Software and its engineering → Object oriented frameworks**; • **Computing methodologies →** *Genetic algorithms*; *Generative and developmental approaches.*

KEYWORDS

Rust software framework, optimization, metaheuristics

ACM Reference Format:
Jonathan Wurth, Helena Stegherr, Michael Heider, Leopold Luley, and Jörg Hähner. 2023. Fast, Flexible, and Fearless: A Rust Framework for the Modular Construction of Metaheuristics. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3583133.3596335

ACM ISBN 979-8-4007-0120-7
https://doi.org/10.1145/3583133.3596335

1 INTRODUCTION

There is an excessive number of metaheuristic implementations available, ranging from "disposable" implementations for specific metaheuristics to generic frameworks, which try to prevent some of this "insanity" [35]. Many of these frameworks rely on component-based designs but ultimately offer only a limited amount of customization. This is not unsurprising, as evolutionary computation and metaheuristics in general cover a spectrum difficult to accommodate inside a single framework. It is therefore not a question of *if* specialization appears in a framework, but *where.*

Many frameworks additionally tend to focus only on the most established metaheuristics, putting so-called "novel" metaheuristics, highly relying on natural and social metaphors, aside for the wrong reasons. While the flaws found in the scientific methodology of those novel metaheuristics are often seen as a just cause to ignore them as a whole, the reasoning is only superficial. Indeed, there has been a growing movement in calling out novel metaheuristics that are simply specializations of established ones [4, 5, 37], and we believe with good reason. What remains an open question is if these novel metaheuristics and operators might still prove useful in some circumstances, even if they are specializations. The comparisons done by the authors of some of those metaheuristics, which claim to answer this question, are unfortunately highly biased, and the foundation of a fair and rigorous evaluation is necessarily the implementation in a framework that allows for such.

This paper introduces MAHF[1], a software framework written in Rust that aims to alleviate some of the aforementioned issues. One central goal of MAHF is to enable easy construction and experimental analysis of metaheuristics by breaking them down into their fundamental components. The framework thus makes it possible to design any metaheuristics, which means first-class support for single-solution- and population-based, local-search, perturbation, generative, and hybrid metaheuristics, by providing an extensive set of modular components and heuristic templates. These components also include such "novel" ones, merging them with their more generalized versions where appropriate. MAHF can be used for any type of experiment, including performance benchmarking studies, and possesses the ideal properties for (generative) hyper-heuristics and automated algorithm design. While it is primarily evolved as a tool for research, application to real-world problems is possible through extension. The framework is still in

[1] MAHF is not an acronym.

its initial development phase and maintained only by a small team, but this paper will demonstrate that the approach and concepts used in MAHF are highly promising. For this purpose, four case studies are presented in Section 4, after introducing and motivating the framework's most important design principles in Section 3 and giving a short outlook into other frameworks in Section 2. Finally, Section 5 presents future development work and a conclusion to the paper.

2 RELATED WORK

There are numerous generic frameworks with extensive functionalities available, each filling a slightly different niche. As evolutionary computation has grown and evolved over the years, different frameworks have necessarily put more attention on certain aspects than on others. ParadisEO [3, 15, 27] and HeuristicLab [45, 46] are some of the mature options available, with the former focusing on a modular design and a wide range of operators, and the latter providing experimental analysis and algorithm prototyping through a modern GUI. ECJ [28, 39], DEAP [11, 17] and jMetal [16, 31, 32] have similarly seen substantial progress over the years. Some more recent frameworks include Cllib [35], IOHexperimenter [10, 12, 47], SPOT [1], and JGEA [29]. In general, frameworks are available for a multitude of programming languages, the most popular being Java and C++. More comprehensive discussions of other frameworks can also be found in [15, 29, 36].

3 THE MAHF FRAMEWORK

MAHF [44] is a framework for the modular construction and experimental analysis of metaheuristics written in the Rust programming language[2]. Rust is well-known in fields associated with low-level computing for its compile-time guarantees of memory- and thread safety, which renders a garbage collector present in most other high-level languages unnecessary, and enables catching numerous errors related to concurrency at compile-time. This allows Rust code to reach comparable speed to C and C++ while avoiding the memory hazards often associated with manual memory management. Rust is much lesser known in the community of evolutionary computation and is still very much a newcomer in scientific computing in general, but its community is growing steadily and will continue to do so in the foreseeable future. Also, while a research-oriented project like MAHF might not benefit from features such as memory safety as much as production or safety-related code, we argue that these still improve development and code quality. Especially bugs involving parallelism are well-known to be notoriously hard to track down, which makes Rust's concept of "fearless concurrency" [3] a welcomed addition to the tool belt.

The framework strongly relies on the concept of *unified metaheuristics* [e.g. 8, 9, 43] and as such adopts a component-based perspective, wrapping any functionality inside generic components. To our present knowledge, MAHF is the only framework written in Rust that offers such an extensive implementation of components, flexibility in the design of metaheuristics, and powerful tools for their evaluation. As it is frequently stated as an advantage for frameworks written in C(++), we believe that low runtimes of metaheuristics play a non-negligible role in the execution of experiments involving such. Especially earlier design phases are usually performed on benchmark problems where the objective function does not dominate runtime, leaving many opportunities for efficient metaheuristic implementations.

MAHF is open-source under the GPL-3.0 license[4] and can be found on GitHub[5]. It is additionally planned to publish MAHF on the crates.io crate registry, the official registry for Rust packages, in the short term.

This section is intended as a short introduction to the most relevant concepts and design principles of MAHF, leaving detailed descriptions of its architecture and modules to the technical report in [44]. An overview of its basic structure is given in Figure 1.

3.1 Problems

Optimization problems in MAHF are split into two traits[6]: `Problem` and `Evaluator`. The `Problem` trait specifies the fundamental properties of an optimization problem, starting with the solution encoding, nature of the objective (single- vs. multiobjective), and its name. Problem types can implement and define additional traits based on `Problem`, allowing access to (any) problem-specific information, for example, the problem having a vector-based solution encoding or the bounds of the search space for continuous problems. This means that components that rely on such information can be reused even across problem classes, provided the information can be represented in the same way. Consequently, it is only necessary to differentiate between single- and multi-objective problems where needed, which mostly boils down to components requiring the definition of a total order on the objective type. The `Evaluator` trait exposes means of evaluating solutions, or rather `Individuals` (basic unit of a solution and an optional objective value), possibly in a batched and parallelized manner.

MAHF currently comes with three problem types built-in:

- The widely recognized continuous optimization benchmark functions adopted from [21] and the "Virtual Library of Simulation Experiments"[7],
- bindings to the COCO benchmark suite [19], and
- a selection of TSPLIB instances[8].

3.2 State

MAHF adopts a notion of metaheuristic *state*, collecting all runtime information that may arise in the global `State`. The `State` acts as a shared variable storage (blackboard), which supports inserting, reading, writing, and removing custom state structs (or simply *custom state*). Components can communicate through this storage, enabling complex interactions. It is realized as a set of types, allowing for a degree of guarantees at compile-time. For example, encapsulation is possible by restricting the type visibility of a custom state, as components trying to access a type that is not visible to them leads to a compile-time error. Conversely, components can provide an interface to their mutable variables through

[2] https://www.rust-lang.org
[3] https://doc.rust-lang.org/book/ch16-00-concurrency.html
[4] https://opensource.org/license/gpl-3-0/
[5] https://github.com/mahf-opt/mahf
[6] Similar to interfaces, see https://doc.rust-lang.org/book/ch10-02-traits.html.
[7] https://www.sfu.ca/~ssurjano/optimization.html
[8] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

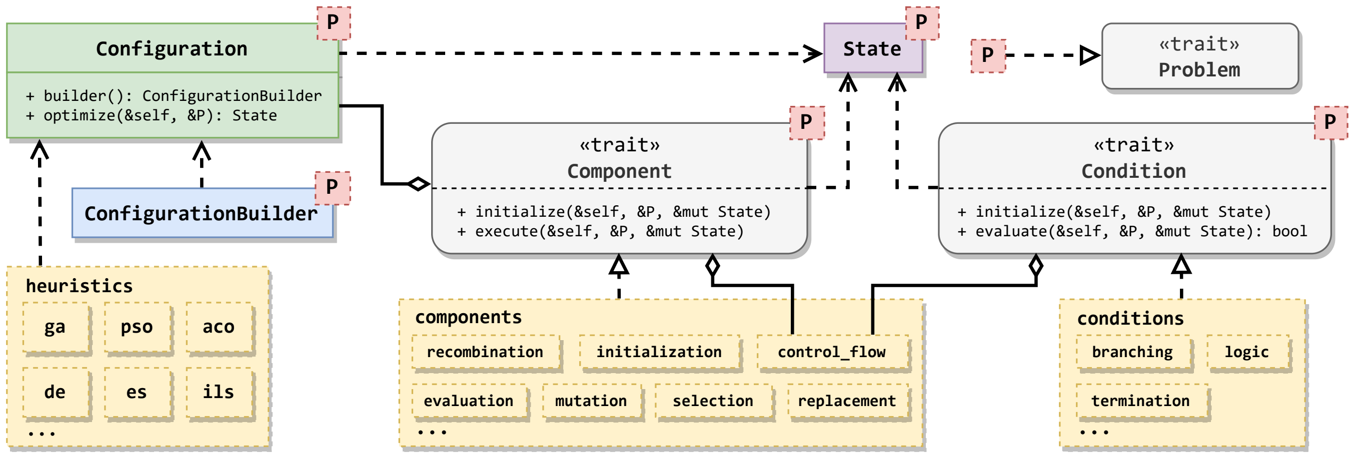

Figure 1: High-level overview of the MAHF framework: `Components` can be combined to metaheuristic `Configurations`, and communicate through the shared `State`. `Conditions` act as specialized components. Many structs are parametrized by the problem type `P`.

public types. Some state shared by almost all metaheuristics, so-called *common* state, is already implemented. This for example includes the number of `Iterations` and `Evaluations` performed, the `BestIndividual` yet found, and the current approximation of the `ParetoFront` for single- and multi-objective problems, respectively, or a seeded random number generator `Random`. The concept of a population of individuals is generalized to `Populations`, allowing components to freely push and pop populations on and off a stack. Classical evolutionary operators are then represented as operations on the so-called *population stack*:

- Initialization: Push a newly generated population.
- Selection: Select a subset from the population on top of the stack and push it.
- Generation: Modify only the top population.
- Replacement: Merge the two top-most populations.

This approach was observed to be flexible enough to allow for the realization of many different operations. The topic is taken up again in Section 4.2.

3.3 Components and Conditions

Components are the heart of MAHF. Component-based design in itself is certainly not new, and frameworks like ParadisEO [15] and jMetal [32] commonly wrap their operator implementations within reusable interfaces. These operators usually include the classic evolutionary operators like selection, replacement, mutation, etc., and common functionality like termination criteria and evaluation of solutions. The main loop is then capsuled behind some generic optimizer interface and written specifically for some algorithm, taking its features and potential peculiarities into account. MAHF takes this concept of components to its logical end: encapsulating any functionality that may arise in the context of metaheuristics into components, which naturally includes the aforementioned (evolutionary) operators, any metaheuristic-specific operator, logging during the execution, calculation of metrics, and even control flow itself.

This is made possible by utilizing the flexibility of the interface offered by `State`, as components can freely write to and read from the blackboard. A component in MAHF implements the `Component` trait and must define its `execute` method, which performs some operations on the `State`, optionally using information offered by the `Problem` it is parametrized by. Optional setup can be done using the `initialize` method, and initial requirements regarding the `State` (i.e. presence of a certain custom state maintained by other components) can be issued in `require`. Components assume the `State` to have certain properties, some of which necessarily need to be checked at runtime, e.g. a mutation component needs at least one population on the stack to mutate. MAHF does not prevent stringing together components at compile-time that assume contradicting conditions or invalidate those of later ones, which is an inevitable byproduct of having such a generic interface. As such, if these conditions do not hold, execution is aborted with an error message detailing the violation, as this indicates an invalid metaheuristic definition.

An excellent tool for stating such conditions is the *Design by contract* approach, using preconditions, postconditions, and invariants. MAHF uses the `contracts`[9] crate to add checks to components with idiomatic syntax. The following snippet defines an invariant, precondition (requires), and postcondition (ensures) to reason about the population stack for a component performing a replacement strategy for single-solution-based metaheuristics.

```
#[invariant(state.populations().current().len() == 1, "top
↪  population before and after should contain a single
↪  individual")]
#[requires(state.populations().peek(1).len() == 1, "parent
↪  population should contain a single individual")]
#[ensures(state.populations().len() ==
↪  old(state.populations().len()) - 1, "the stack height should
↪  have decreased by one after this operation")]
fn execute(&self, problem: &P, state: &mut State<P>) { /*...*/ }
```

[9] https://docs.rs/contracts/latest/contracts/

Another important issue for the design of components is specialization vs. generality. The components in MAHF should be implemented as specialized as necessary, and as general as possible, which means that they should expose as much information as is sensible through the `State`, allowing for other components with additive functionality rather than reimplementations of existing components with new functionality. An example of this is parameter control: there is no need for an additional component with adaptive mutation strength if the existing component performing the mutation exposes its mutation strength in the first place, possibly enabling adaptation by other components.

While in theory components can express any behaviour, the well-established categories of metaheuristic components are still provided by MAHF in the form of specific modules, including components associated with selection, mutation, recombination, initialization, or replacement (cf. Figure 1). One special feature of MAHF is that control flow is also represented as components:

- `Block`: executes a group of components sequentially.
- `Loop`: executes a component (usually a `Block`) while a certain condition is true.
- `Branch`: executes either the "if" or "else" component depending on a condition.
- `Scope`: executes the component in a capsuled `State`, allowing shadowing of custom state for this duration while still enabling communication with the outer `State`. Usually used to nest metaheuristics transparently.

Conditions are represented through the `Condition` trait and are equivalent to components with the sole difference lying in their execution evaluating to a boolean value. They can be combined using the usual boolean operators.

3.4 Configurations

The previous section has introduced components and the concept of grouping them inside other components. This can be used to construct any metaheuristic algorithm or metaheuristic framework. Such a grouping of components is called a metaheuristic *configuration*, and the `Configuration` struct provides useful abstractions for dealing with them. Applying a `Configuration` to a problem is as simple as calling its `optimize` method with the problem as argument. The method returns the `State` after execution, enabling inspection of any custom state, including the final population, or the `BestIndividual`. It is also possible to pre-initialize the `State` before execution, e.g. to set the random number seed or customize what state should be logged at runtime. Because an object-oriented approach to specifying control flow is not very intuitive, `Configuration` also exposes a simple and idiomatic way to construct metaheuristics through Rust's builder pattern[10]: control flow methods such as `do_`, `while_`, `if_`, or `if_else_` can be used to define specifications very close to pseudo-code. Shortcuts to often-used components are available as well, e.g. the `evaluate` and `update_best_individual` methods instead of manually specifying the components with the respective functionality. Section 4 shows many examples of this and, specifically, Section 4.1 contrasts a metaheuristic template in pseudo-code and MAHF's builder syntax.

[10] See https://doc.rust-lang.org/1.0.0/style/ownership/builders.html for more information on the builder pattern.

MAHF currently defines templates for 12 metaheuristics using its many components, which include popular algorithms like the Genetic Algorithm (GA), Particle Swarm Optimization (PSO), or Ant Colony Optimization (ACO) [13, 14] along with lesser known, supposed "novel" metaheuristics like Invasive Weed Optimization (IWO) [30] and baseline heuristics like random searches. They allow customization of certain operators similar to other frameworks (e.g. a custom mutation operator) and are meant to provide a starting point for more specialized configurations.

4 CASE STUDIES

This section demonstrates the usefulness and flexibility of the concepts implemented in MAHF, which were introduced in earlier sections, through four case studies, which cover a broad spectrum of common metaheuristic applications present in research and industry. These include implementing new metaheuristics inside the framework, hybridizing existing ones, adding new or modifying existing operators and metrics, and comparing and evaluating them. For this purpose, most sections are accompanied by examples of the specific metaheuristic configurations.

4.1 Implementing Simulated Annealing

Simulated Annealing (SA) [25] is a well-studied single-solution-based metaheuristic capable of escaping from local optima [18, 34]. To add a new metaheuristic to MAHF, the first step lies in identifying the components that are metaheuristic-specific and those that can be reused from others, i.e. are already available. Of course, the entirety of components appearing in SA is already implemented in MAHF, so in reality we would be done here. For the sake of the example, assume that this is not the case, but metaheuristics like a GA are present, and so are their components.

The pseudo-code of SA is given in Algorithm 1. Initialization methods in MAHF accept an arbitrary population size, covering the `initalization` of the initial solution on line 1. Evaluating individuals (line 2,7) and tracking the best individual yet found (line 2,8) is similarly built in using the `evaluate` and `update_best_individual` methods (or their respective components). Also, iterations are automatically kept track of by the loop (line 3,14) and common termination criteria are assumed to be present from other algorithms. Neighborhood (line 5) and constraint (line 6) components are shared by many metaheuristics and can be chosen depending on the problem type. They similarly work for any population size. For continuous problems, this could be a mutation using normal-distributed values and clamping to the search space boundaries. The functionality unique to SA appears therefore to be its probabilistic candidate solution acceptance (line 9 to 12) and the cooling schedule (line 13), both related to the concept of *temperature*. A termination criterion related to temperature could also be imagined (e.g. terminate when the temperature $T < \epsilon$).

As the temperature seems to be shared by at least two components, it should be represented as a public custom state (e.g. a `Temperature` struct wrapping a float value). Initializing this state in the solution acceptance component seems to be a natural fit, so it could be used with a (low) static temperature value by itself without a cooling schedule. The current (S) and candidate solution (S') can be represented as two populations on the stack, using the usual concepts of selection and replacement. Selection for SA means copying S to a new population, which is equivalent to selecting all solutions from the population (`selection::All`), and pushing it on the stack. Replacement is done through the previously identified candidate solution acceptance component (e.g. called `ExponentialAnnealingAcceptance`), which either discards the top population (S') or the one below (S) depending on their objective values and the `Temperature`. For the population stack logic, it could define a state contract similar to one listed in Section 3.3. The cooling schedule might either read the number of `Iterations` (k) from the `State` and calculate the new `Temperature` value (e.g. using $T_0 \cdot \alpha^k$) or adapt it solely based on the old value. With this, all functionality of SA is covered, and we can define its configuration in MAHF (cf. Listing 1).

Algorithm 1: Simulated Annealing

Input: Objective f with domain A, initial temperature T_0

$S \leftarrow initialization()$;
Evaluate S; $i_{\text{best}} \leftarrow S$;
$k \leftarrow 0$;
while *termination criterion not met* **do**
    $S' \leftarrow$ neighbor(S);
    Constrain S' to A;
    Evaluate S';
    Update the best individual i_{best};
    if $f(S') < f(S)$ **then**
        $S \leftarrow S'$;
    else
        $S \leftarrow S'$ with probability $\exp((f(S) - f(S'))/T_k)$;
    $T_{k+1} \leftarrow$ cooling_schedule(k);
    $k \leftarrow k + 1$;

Output: Best individual i_{best}

```
Configuration::builder()
  .do_(initialization)
  .evaluate()
  .update_best_individual()
  .while_(!termination, |builder| {
      builder
          .do_(selection::All::new())
          .do_(neighbor)
          .do_(constraints)
          .evaluate()
          .update_best_individual()
          .do_(ExponentialAnnealingAcceptance::new(t_0))
          .do_(cooling_schedule)
  })
  .build();
```

Listing 1: The SA in Algorithm 1 as `Configuration`: the `initialization`, `termination`, `neighbor`, `constraints`, and `cooling_schedule` components are left unspecified.

4.2 Hybridizing Particle Swarm Optimization

The high degree of component reusability made it easy to implement SA in the previous section. While the component-based design certainly showed its strength in implementing a new metaheuristic, hybridizing metaheuristics is another well-supported use case. This is where the high level of generality of components can be exploited in its entirety, almost seamlessly combining components that are by their definition usually not used together.

Particle Swarm Optimization (PSO) [24, 38] is one of the most widely recognized population-based metaheuristics introduced in recent decades, and has been hybridized with a wealth of different metaheuristics [see 40, 41]. This section means to illustrate how MAHF's component design generalizes well beyond the common applications of metaheuristic frameworks through implementing three PSO hybrids from the literature, namely PSO with GA, Differential Evolution (DE), and SA, respectively.

PSO is split into the following components, with the `ParticleSwarm` custom state holding PSO-specific information. The position of particles is represented as a population of individuals.

- `ParticleSwarm::initializer`: initializes velocity v, personal best *pbest* for each particle and the global best *gbest*.
- `ParticleSwarmGeneration`: performs the classical velocity and position update.
- `ParticleSwarm::updater`: updates *pbest* and *gbest*.

PSO with GA. GA-PSO [23] tries to leverage the advantages of both PSO and GA by applying them to the worse and better half of the population, respectively. The GA applied to the better half uses an arithmetic crossover and Gaussian mutation. PSO's update of *pbest* and *gbest* are done on the whole population. While Kao and Zahara [23] technically define a custom termination criterion based on the standard deviation of the best $N/4 + 1$ individuals, it is omitted here as it is not connected specifically to hybridization. Listing 2 shows the configuration using MAHF's builder syntax. The hybrid behaviour is translated into components for splitting the population into two by their objective values, pushing both on the stack (`SplitFittest`), swapping the two top-most populations (`SwapPopulations`), and merging the two top-most populations back into one (`Merge`). These operations on the population stack are visualized in Figure 2a, denoting the worse half as $P^W_{N/2}$ and the better as $P^B_{N/2}$ of a population P_N of size N. Note how the selection-replacement-cycle of the GA temporarily pushes a population itself, staying oblivious to the presence of other populations.

PSO with DE. DEPSO [20] is based on the idea of utilizing both the differential information generated by DE and the memory capabilities of PSO. The population update of DE is probabilistically replaced with the particle update of PSO, keeping the better individual at each index, respectively. As Figure 2b shows, PSO operates on a copy of the population to enable the replacement on a per-individual basis. Execution of either DE or PSO is decided by chance, strongly biasing the result towards DE (cf. Listing 3). To be precise, DEPSO is rather an integration of PSO into DE than the other way around, as the DE update is performed significantly more often.

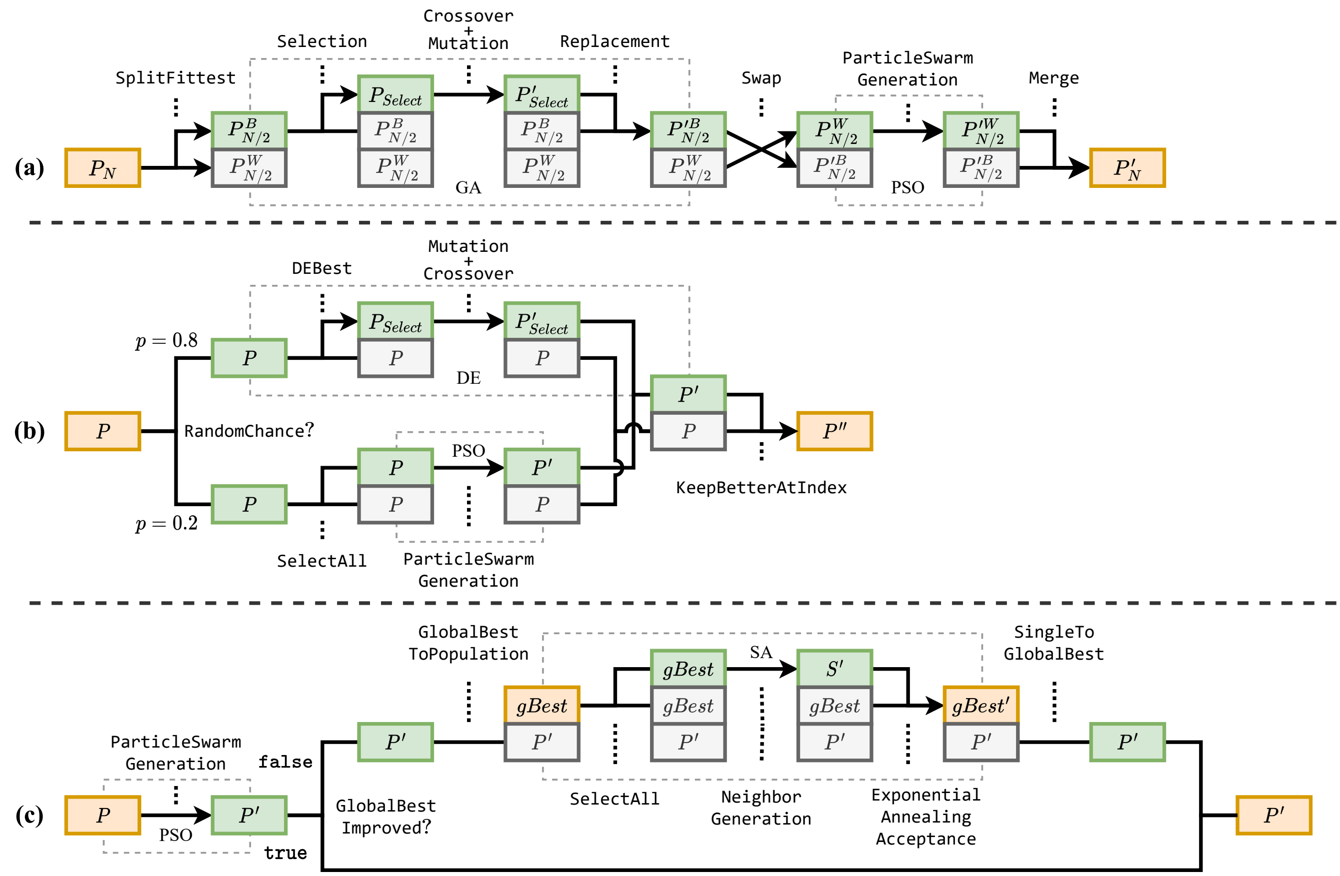

Figure 2: Layout of the population stack for (a) GA-PSO [23], (b) DEPSO [20], and (c) PSO-SA [22] during an iteration: only the population on top of the stack (green) is modified by generation components like mutation and recombination. The start and end populations in a loop are marked in orange. Arrows symbolize the populations affected by an operation. Note that PSO-SA (c) may perform multiple inner SA loops depending on if a better solution was found.

PSO with SA. PSO-SA [22] aims to improve the social behaviour of PSO by updating *gbest* through a nested SA if no improvement was found in the current cycle. The inner SA is run until no improvement is made, and the current solution S is assigned as new *gbest*. To realize this functionality, a condition to check if *gbest* has improved since the last iteration is needed (`GlobalBestImproved`), along with means to insert *gbest* into a new population on the stack (`GlobalBestToPopulation`), and update it using an existing single solution (`SingleToGlobalBest`). The condition needs to access the objective value of *gbest* from the last iteration, which can be implemented through a private custom state, which is updated after reading the old value. The inner SA loop is realized through the pre-defined `sa` template (cf. Listing 4), which internally uses the very components designed in Section 4.1. As cooling and constraint checking is done in both the outer PSO and inner SA loop, the components are defined beforehand, and inserted at the respective locations. The implementation of SA as a selection-replacement-cycle is illustrated in Figure 2c.

4.3 Implementing PSO operator variants

Hybridizing metaheuristics through the incorporation of operators from other metaheuristics is often a promising approach to improve performance, but adapting the metaheuristic through new variants of its components is even more common. MAHF makes the addition of new components very easy from the start, and it becomes even easier if functionality from other components can be reused. This section will showcase how PSO might be enhanced by controlling one of its primary parameters: the inertia weight w used to scale the old velocity in its update formula. Three techniques found in the literature will be shortly introduced and the possible implementation in MAHF will be discussed. For this example, it will be assumed that the `ParticleSwarmGeneration` component exposes w as public custom state, i.e. a `InertiaWeight` wrapper around a float value. This enables additive components rather than a reimplementation with additional features (see Section 3.3). The component also implicitly implements a static inertia weight and accepts custom (initial) values as argument to its constructor.

```
Configuration::builder()
  .do_(RandomSpread::new_init(/*population_size: */30))
  .evaluate()
  .update_best_individual()
  .do_(ParticleSwarm::initializer(/*v_max: */1.))
  .while_(LessThanN::<Iterations>::new(/*n: */ 500), |builder| {
      builder
        .do_(SplitFittest::new())
        .do_(ArithmeticCrossover::new_both(/*pc: */1.))
        .do_(GaussianMutation::new(/*pm: */0.2, /*sigma: */0.1))
        .do_(SwapPopulations::new())
        .do_(ParticleSwarmGeneration::new(
          /*weight: */ 0.8, /*c_one: */ 1.7,
          /*c_two: */ 1.7, /*v_max: */ 1.0,
        ))
        .do_(replacement::Merge::new())
        .do_(constraints::Saturation::new())
        .evaluate()
        .update_best_individual()
        .do_(ParticleSwarm::updater())
  })
  .build()
```

Listing 2: `Configuration` of the GA-PSO [23] hybrid.

```
Configuration::builder()
  .do_(RandomSpread::new_init(/*population_size: */30))
  .evaluate()
  .update_best_individual()
  .do_(ParticleSwarm::initializer(/*v_max: */1.))
  .while_(LessThanN::<Iterations>::new(/*n: */ 500), |builder| {
      builder
        .if_else_(RandomChance::new(/*p: */0.8),
            |builder| {
                builder
                  .do_(DEBest::new(y))
                  .do_(DEMutation::new(y, f))
                  .do_(DEBinomialCrossover::new(/*pc: */0.8))
            },
            |builder| {
                builder
                  .do_(selection::All::new())
                  .do_(ParticleSwarmGeneration::new(
                    /*weight: */ 0.8, /*c_one: */ 1.7,
                    /*c_two: */ 1.7, /*v_max: */ 1.0,
                  ))
            }
        )
        .do_(constraints::Saturation::new())
        .evaluate()
        .update_best_individual()
        .do_(replacement::KeepBetterAtIndex::new())
        .do_(ParticleSwarm::updater())
  })
  .build()
```

Listing 3: `Configuration` of the DEPSO [20] hybrid.

```
let constraints = constraints::Saturation::new();
let cs = GeometricCooling::new::<_, Temperature>(/*alpha: */0.95);
Configuration::builder()
  .do_(RandomSpread::new_init(/*population_size: */30))
  .evaluate()
  .update_best_individual()
  .do_(ParticleSwarm::initializer(/*v_max: */1.))
  .while_(LessThanN::<Iterations>::new(/*n: */ 500), |builder| {
    builder
      .do_(ParticleSwarmGeneration::new(
        /*weight: */ 0.8, /*c_one: */ 1.7,
        /*c_two: */ 1.7, /*v_max: */ 1.0,
      ))
      .do_(constraints.clone())
      .evaluate()
      .update_best_individual()
      .do_(state::ParticleSwarm::updater())
      .if_else_(!GlobalBestImproved::new(),
        |builder| {
          builder
            .do_(GlobalBestToPopulation::new())
            .do_(sa::sa(
                sa::Parameters {
                    t_0: 1.0,
                    generation: GaussianMutation::new(
                        /*pm: */1., /*sigma: */0.1,
                    ),
                    cooling_schedule: cs.clone(),
                    constraints,
                },
                /*termination: */
                StepsWithoutImprovement::new(1),
            ))
            .do_(SingleToGlobalBest::new())
        },
        |builder| builder.do_(cs.clone()),
      )
  })
  .build()
```

Listing 4: `Configuration` of the PSO-SA [22] hybrid.

Random inertia weight. A random inertia weight uniformly distributed in the range $[0.5, 1]$ is used in [26] to improve the performance of PSO in a dynamic environment, where adaptations developed for static fitness landscapes are unsuitable. A component performing this operation can retrieve the seeded random number generator `Random` from the `State` and overwrite the old value of `InertiaWeight` with the generated one.

Time-varying inertia weight. A time-varying inertia weight can be used to guide the exploration capabilities of PSO. For example, a common extension is to decrease w throughout iterations, starting with a high degree of exploration and exploiting good regions of the search space towards the end [41]. A linearly-varying inertia weight can be expressed as follows [42]:

$$w^k = (w_{\max} - w_{\min}) \cdot (1 - \rho_{\text{progress}}) + w_{\min} \tag{1}$$

where $w_{\max}$ and $w_{\min}$ are initial and final values of the inertia weight, and ρ_{progress} is the progress of the algorithm from 0 to 1, which might, for example, be the ratio of the current iteration and the maximum number of iterations. A component implementing this adaptation defines $w_{\max}$, $w_{\min}$ as parameters, reads the current progress from the `State` (e.g. the `LessThanN<T>` condition exposes its progress using the `Progress<T>` custom state), and writes the value of w using Equation (1).

Adaptive inertia weight. The inertia weight might also be modified depending on one or more feedback parameters. In [33], the authors propose using the percentage of particles that improved their *pbest* in an iteration as a replacement for $(1 - \rho_{\text{progress}})$ in Equation (1). Realizing this as a component thus needs a way to keep track of the *pbest*s of the last iteration, to calculate this percentage. This can be implemented similarly to the `GlobalBestImproved` condition (cf. Section 4.2), updating the custom state with new *pbest*s after reading the old values. Calculating and writing the new value for w is then identically to the time-varying inertia weight introduced previously.

4.4 Comparing diversity measures

Components in MAHF are not required to contribute to the optimization process, but can also compute and track additional information, which includes (any) measures and metrics. This section is meant to give a short introduction to how logging of values calculated at runtime is possible, through the example of population diversity. Rather than logging whole populations and computing measures afterwards, MAHF encourages calculating the values on the fly and logging only relevant information. This firstly serves to decrease the sheer amount of information that needs to be post-processed, and secondly makes incorporation of the very same information into the search process trivial. For performing experiments, MAHF provides an extensive and flexible logging system, enabling logging of any value at possibly any time using the concepts of *triggers* and *extractors*, which can be customized. Four normalized measures for the diversity of the population were taken from [6] and [7], namely dimension-wise and "true" diversity, the distance to the average point, and the pairwise distance between individuals. Each measure was realized as a component `C`, exposing the diversity in the `NormalizedDiversity<C>` state, and logging was then configured to extract these values in every iteration. MAHF stores its logs as dense tables in the CBOR format [2], which can easily be converted into JSON or read by the implementations available for a variety of platforms. To generate Figure 3, the `cbor2`[11] Python library was used to read the logs, after which `pandas`[12] and `seaborn`[13] were used to aggregate and plot the data. Note that multiple runs and entire experiments are trivial to parallelize in MAHF. For this case study, the 50 runs performed were executed in parallel using the `rayon` library[14].

5 FUTURE WORK AND CONCLUSION

MAHF is in its early stages, and while the use cases in Section 4 have demonstrated that it possesses the flexibility and extensibility to be used for metaheuristic experiments, several points require

[11] https://pypi.org/project/cbor2/
[12] https://pandas.pydata.org
[13] https://seaborn.pydata.org
[14] https://docs.rs/rayon/latest/rayon/

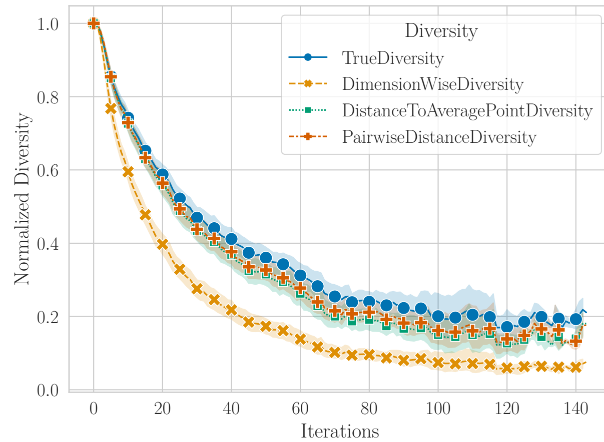

Figure 3: The means and 95% confidence interval of the four diversity measures plotted against iterations. A total of 50 runs were performed on the Rastrigin benchmark function with 30 dimensions, using GA-PSO [23] as an exemplary optimizer.

attention. A wide range of components and operators is already implemented, but especially the extension with components from state-of-the-art metaheuristics is a primary goal going forward. This also includes multi-objective algorithms, as MAHF has the support for both single- and multi-objective problems and components built-in, but no specific operators and problems are available yet. The framework is currently focused on numerical optimization, so more options for combinatorial problems and operators will be added. MAHF's component-based design provides an excellent basis for hyper-heuristic approaches and the automated design of algorithms, which will be a focus of future research. While the framework was designed with users in mind that are not proficient in Rust, it still presents an obstacle to its broader acceptance and usage. Similar to other frameworks, MAHF also aims to increase interoperability by exposing some of its functionality to Python. There are also many challenges and opportunities to improve the API of MAHF further: for example, it currently requires a moderate amount of boilerplate code to define components and custom state, which could be alleviated by providing macros for common use cases.

In this paper, MAHF was presented as a Rust framework for the modular construction and experimental analysis of metaheuristics, introducing and motivating its fundamental concepts. Its flexibility and capabilities were demonstrated in four common use cases that arise in applications and experiments, highlighting its first-class support for hybrids and unusual approaches. MAHF is in active development and will build further on its strong foundations.

REFERENCES

[1] Thomas Bartz-Beielstein, Martin Zaefferer, and Frederik Rehbach. 2021. In a Nutshell – The Sequential Parameter Optimization Toolbox. https://doi.org/10.48550/arXiv.1712.04076 arXiv:arXiv:1712.04076
[2] Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). https://doi.org/10.17487/RFC8949

[3] Sébastien Cahon, El-Ghaza Talbi, and Nordine Melab. 2003. ParadisEO: A Framework for Parallel and Distributed Biologically Inspired Heuristics. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 9 pp.–. https://doi.org/10/drp4d5

[4] Christian L. Camacho-Villalón, Marco Dorigo, and Thomas Stützle. 2022. An Analysis of Why Cuckoo Search Does Not Bring Any Novel Ideas to Optimization. *Computers & Operations Research* 142 (June 2022), 105747. https://doi.org/10/grv9ff

[5] Christian L. Camacho-Villalón, Marco Dorigo, and Thomas Stützle. 2022. Exposing the Grey Wolf, Moth-Flame, Whale, Firefly, Bat, and Antlion Algorithms: Six Misleading Optimization Techniques Inspired by Bestial Metaphors. *International Transactions in Operational Research* (July 2022), 24. https://doi.org/10/gr4sqk

[6] Shi Cheng, Yuhui Shi, Quande Qin, Qingyu Zhang, and Ruibin Bai. 2014. Population Diversity Maintenance In Brain Storm Optimization Algorithm. *Journal of Artificial Intelligence and Soft Computing Research* 4, 2 (April 2014), 83–97. https://doi.org/10/ggrd47

[7] Guillaume Corriveau, Raynald Guilbault, Antoine Tahan, and Robert Sabourin. 2012. Review and Study of Genotypic Diversity Measures for Real-Coded Representations. *IEEE Transactions on Evolutionary Computation* 16, 5 (Oct. 2012), 695–710. https://doi.org/10/f4ct44

[8] Jorge M. Cruz-Duarte, José C. Ortiz-Bayliss, Iván Amaya, Yong Shi, Hugo Terashima-Marín, and Nelishia Pillay. 2020. Towards a Generalised Metaheuristic Model for Continuous Optimisation Problems. *Mathematics* 8, 11 (Nov. 2020), 2046. https://doi.org/10/gr4hqv

[9] Jesica de Armas, Eduardo Lalla-Ruiz, Surafel Luleseged Tilahun, and Stefan Voß. 2022. Similarity in Metaheuristics: A Gentle Step towards a Comparison Methodology. *Natural Computing* 21, 2 (June 2022), 265–287. https://doi.org/10/grsff5

[10] Jacob de Nobel, Furong Ye, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. 2022. IOHexperimenter: Benchmarking Platform for Iterative Optimization Heuristics. https://doi.org/10.48550/arXiv.2111.04077 arXiv:arXiv:2111.04077

[11] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: A Python Framework for Evolutionary Algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12)*. Association for Computing Machinery, New York, NY, USA, 85–92. https://doi.org/10/ghmcsq

[12] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. 2018. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. https://doi.org/10.48550/arXiv.1810.05281 arXiv:arXiv:1810.05281

[13] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant Colony Optimization. *IEEE Computational Intelligence Magazine* 1, 4 (Nov. 2006), 28–39. https://doi.org/10/dq339r

[14] Marco Dorigo and Gianni Di Caro. 1999. Ant Colony Optimization: A New Meta-Heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Vol. 2. 1470–1477 Vol. 2. https://doi.org/10/b5h9xz

[15] Johann Dreo, Arnaud Liefooghe, Sébastien Verel, Marc Schoenauer, Juan J. Merelo, Alexandre Quemy, Benjamin Bouvier, and Jan Gmys. 2021. Paradiseo: From a Modular Framework for Evolutionary Computation to the Automated Design of Metaheuristics: 22 Years of Paradiseo. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 1522–1530. https://doi.org/10/grwfh7

[16] Juan J. Durillo, Antonio J. Nebro, and Enrique Alba. 2010. The jMetal Framework for Multi-Objective Optimization: Design and Architecture. In *IEEE Congress on Evolutionary Computation*. 1–8. https://doi.org/10/ckz4f3

[17] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13, 70 (2012), 2171–2175.

[18] Michel Gendreau and Jean-Yves Potvin. 2018. *Handbook of Metaheuristics* (3rd ed. 2019 edition ed.). Springer, New York, NY.

[19] Nikolaus Hansen, Dimo Brockhoff, Olaf Mersmann, Tea Tusar, Dejan Tusar, Ouassim Ait ElHara, Phillipe R. Sampaio, Asma Atamna, Konstantinos Varelas, Umut Batu, Duc Manh Nguyen, Filip Matzner, and Anne Auger. 2019. COmparing Continuous Optimizers: Numbbo/COCO on Github. Zenodo. https://doi.org/10.5281/zenodo.2594848

[20] Zhi-Feng Hao, Guang-Han Guo, and Han Huang. 2007. A Particle Swarm Optimization Algorithm with Differential Evolution. In *2007 International Conference on Machine Learning and Cybernetics*, Vol. 2. 1031–1035. https://doi.org/10/c8bbsj

[21] Momin Jamil and Xin-She Yang. 2013. A Literature Survey of Benchmark Functions For Global Optimization Problems. *International Journal of Mathematical Modelling and Numerical Optimisation* 4, 2 (2013), 150. https://doi.org/10/gh48qq arXiv:1308.4008 [cs, math]

[22] Farhad Javidrad and Mohammad Nazari. 2017. A New Hybrid Particle Swarm and Simulated Annealing Stochastic Optimization Method. *Applied Soft Computing* 60 (Nov. 2017), 634–654. https://doi.org/10/gckzdx

[23] Yi-Tung Kao and Erwie Zahara. 2008. A Hybrid Genetic Algorithm and Particle Swarm Optimization for Multimodal Functions. *Applied Soft Computing* 8, 2 (March 2008), 849–857. https://doi.org/10/bwckjw

[24] James Kennedy and Russell Eberhart. 1995. Particle Swarm Optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. 1942–1948 vol.4. https://doi.org/10/bdc3t3

[25] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (May 1983), 671–680. https://doi.org/10/cn7jh2

[26] Mi Li, Huan Chen, Xin Shi, Sa Liu, Ming Zhang, and Shengfu Lu. 2019. A Multi-Information Fusion "Triple Variables with Iteration" Inertia Weight PSO Algorithm and Its Application. *Applied Soft Computing* 84 (Nov. 2019), 105677. https://doi.org/10/gr3r3c

[27] Arnaud Liefooghe, Laetitia Jourdan, Thomas Legrand, Jérémie Humeau, and El-Ghazali Talbi. 2010. ParadisEO-MOEO: A Software Framework for Evolutionary Multi-Objective Optimization. In *Advances in Multi-Objective Nature Inspired Computing*, Carlos A. Coello Coello, Clarisse Dhaenens, and Laetitia Jourdan (Eds.). Springer, Berlin, Heidelberg, 87–117. https://doi.org/10.1007/978-3-642-11218-8_5

[28] Sean Luke. 2017. ECJ Then and Now. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 1223–1230. https://doi.org/10/gr3mnd

[29] Eric Medvet, Giorgia Nadizar, and Luca Manzoni. 2022. JGEA: A Modular Java Framework for Experimenting with Evolutionary Computation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 2009–2018. https://doi.org/10/grtdmw

[30] Ali Reza Mehrabian and Caro Lucas. 2006. A Novel Numerical Optimization Algorithm Inspired from Weed Colonization. *Ecological Informatics* 1, 4 (Dec. 2006), 355–366. https://doi.org/10/dc4njs

[31] Antonio J. Nebro, Juan J. Durillo, and Matthieu Vergne. 2015. Redesigning the jMetal Multi-Objective Optimization Framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 1093–1100. https://doi.org/10/ghmcsn

[32] Antonio J. Nebro, Javier Pérez-Abad, José F. Aldana-Martin, and José García-Nieto. 2021. Evolving a Multi-objective Optimization Framework. In *Applied Optimization and Swarm Intelligence*, Eneko Osaba and Xin-She Yang (Eds.). Springer, Singapore, 175–198. https://doi.org/10.1007/978-981-16-0662-5_9

[33] Ahmad Nickabadi, Mohammad Mehdi Ebadzadeh, and Reza Safabakhsh. 2011. A Novel Particle Swarm Optimization Algorithm with Adaptive Inertia Weight. *Applied Soft Computing* 11, 4 (June 2011), 3658–3670. https://doi.org/10/d3f47w

[34] Alexander G. Nikolaev and Sheldon H. Jacobson. 2010. Simulated Annealing. In *Handbook of Metaheuristics*, Michel Gendreau and Jean-Yves Potvin (Eds.). Springer US, Boston, MA, 1–39. https://doi.org/10.1007/978-1-4419-1665-5_1

[35] Gary Pamparà and Andries Engelbrecht. 2015. Towards A Generic Computational Intelligence Library: Preventing Insanity. In *2015 IEEE Symposium Series on Computational Intelligence*. 1460–1467. https://doi.org/10/gr3mm6

[36] José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. 2012. Metaheuristic Optimization Frameworks: A Survey and Benchmarking. *Soft Computing* 16, 3 (March 2012), 527–561. https://doi.org/10/c3c43h

[37] Adam P. Piotrowski, Jaroslaw J. Napiorkowski, and Pawel M. Rowinski. 2014. How Novel Is the "Novel" Black Hole Optimization Approach? *Information Sciences* 267 (May 2014), 191–200. https://doi.org/10/f5xttc

[38] Riccardo Poli, James Kennedy, and Tim Blackwell. 2007. Particle Swarm Optimization. *Swarm Intelligence* 1, 1 (June 2007), 33–57. https://doi.org/10/dhnq29

[39] Eric O. Scott and Sean Luke. 2019. ECJ at 20: Toward a General Metaheuristics Toolkit. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1391–1398. https://doi.org/10/grwfh2

[40] Saptarshi Sengupta, Sanchita Basak, and Richard Alan Peters. 2019. Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives. *Machine Learning and Knowledge Extraction* 1, 1 (March 2019), 157–191. https://doi.org/10/gh6wwh

[41] Tareq M. Shami, Ayman A. El-Saleh, Mohammed Alswaitti, Qasem Al-Tashi, Mhd Amen Summakieh, and Seyedali Mirjalili. 2022. Particle Swarm Optimization: A Comprehensive Survey. *IEEE Access* 10 (2022), 10031–10061. https://doi.org/10/grsff6

[42] Yuhui Shi and Russell .C. Eberhart. 1999. Empirical Study of Particle Swarm Optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Vol. 3. 1945–1950 Vol. 3. https://doi.org/10/bd4nj9

[43] Helena Stegherr, Michael Heider, Leopold Luley, and Jörg Hähner. 2021. Design of Large-Scale Metaheuristic Component Studies. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 1217–1226. https://doi.org/10/grsffx

[44] Helena Stegherr, Leopold Luley, Jonathan Wurth, Michael Heider, and Jörg Hähner. 2023. *A framework for modular construction and evaluation of metaheuristics.* Technical Report. Fakultät für Angewandte Informatik. 25 pages.

https://opus.bibliothek.uni-augsburg.de/opus4/103452
[45] Stefan Wagner and Michael Affenzeller. 2005. HeuristicLab: A Generic and Extensible Optimization Environment. In *Adaptive and Natural Computing Algorithms*. Springer-Verlag, 538–541. https://doi.org/10.1007/3-211-27389-1_130
[46] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. 2014. Architecture and Design of the HeuristicLab Optimization Environment. In *Advanced Methods and Applications in Computational Intelligence*, Ryszard Klempous, Jan Nikodem, Witold Jacak, and Zenon Chaczko (Eds.). Springer International Publishing, Heidelberg, 197–261. https://doi.org/10.1007/978-3-319-01436-4_10
[47] Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, and Thomas Bäck. 2022. IOHanalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics. *ACM Transactions on Evolutionary Learning and Optimization* 2, 1 (April 2022), 3:1–3:29. https://doi.org/10/gr3mm7