UNIVERSITÄT AUGSBURG

A Framework for Modular Construction and Evaluation of Metaheuristics

H. Stegherr, L. Luley, J. Wurth, M. Heider, J. Hähner

Report 2023-01 April 2023

INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

A Framework for Modular Construction and Evaluation of Metaheuristics

HELENA STEGHERR*, LEOPOLD LULEY*, JONATHAN WURTH*, MICHAEL HEIDER, and JÖRG HÄHNER, University of Augsburg, Germany

This paper presents MAHF, a software framework for the highly flexible construction of metaheuristics from individual components and the subsequent evaluation of these algorithms. At that, MAHF is developed specifically for the experimental analysis of the algorithmic behaviour during the optimization process with a focus on the influences of the algorithm's components. Furthermore, uncommon and incompletely examined operators or frameworks of "novel" metaheuristics are included as well, so that their usefulness can be assessed. In the following, we will elaborate on MAHF's structure and its general goals and application possibilities. Concerning MAHF's component structure, we will provide examples of its usage and extension to ensure that it is reusable by others as well.

CCS Concepts: • **Computing methodologies** → *Genetic algorithms*; *Generative and developmental approaches.*

Additional Key Words and Phrases: optimization, metaheuristics, evolutionary algorithms, software framework

1 INTRODUCTION

Although performance benchmarking and controlled experimentation are common ways to gain new insights into metaheuristic algorithms and such empirical studies are regularly published, there is still a lot more to be experimentally learned about these algorithms. Especially in the light of explainability requirements [2], detailed analyses of the algorithms' behaviours during the search process are of increasing interest. Furthermore, reaching the ultimate goal of mapping algorithmic features to problem or fitness landscape characteristics to facilitate the configuration or selection of an appropriate algorithm or improve it according to experimental insights is still pending, though the number of studies in this area is increasing [e.g. 21, 27]. These research interests give rise to a plethora of different implementations of metaheuristics, both as larger frameworks where different methods are combined and as individual algorithms. However, especially for supposedly "novel" metaheuristics, which are usually strongly based on some kind of metaphor for a natural or social phenomenon [cf. 8], code is often not available, hard to understand or not readily comparable to implementations of other algorithms. Furthermore, published results often lack a rigorous and fair analysis [1, 34]. Especially the lack of consistent and transparent analyses led to the formulation of best practices for benchmarking and experimentation [3] and reproducibility [22].

MAHF[1] is a software framework, written in the Rust programming language, intended for the modular construction of metaheuristics and their experimental analysis. Its main goal is to enable the construction of any metaheuristic framework and algorithm, including known approaches, hybrids, and new ideas, utilizing a consistent basic set of components. In contrast to many other software frameworks, it aims at including operators from strongly metaphor-based metaheuristics as well, to enable their rigorous evaluation. While the merit of these is strongly disputed, it is not impossible that some of the researchers may have had potentially useful ideas, even if their overall scientific method or presentation is (deeply) flawed. However, we carefully consider whether an operator is clearly novel and will require a specific implementation, or if it is just a variation of an existing operator that can be unified in the original implementation.

*Authors contributed equally to this research.
[1]Not an acronym.

Authors' address: Helena Stegherr, helena.stegherr@uni-a.de; Leopold Luley, leopold.luley@uni-a.de; Jonathan Wurth, jonathan.wurth@uni-a.de; Michael Heider, michael.heider@uni-a.de; Jörg Hähner, jörg.hähner@uni-a.de, University of Augsburg, Universitätsstraße 2, Augsburg, Germany, 86159.

MAHF is explicitly designed for conducting experiments with metaheuristic algorithms. In terms of experimental evaluations, MAHF provides extensive logging options for a variety of parameters and measures, which can be adapted and expanded for the specific experimental design. It also presents different evaluation options and metrics for the gathered data. However, MAHF can be easily adapted to other purposes as well. Its algorithms can be used for classical performance benchmarking studies and its component structure makes it suitable for hyperheuristic approaches and automated algorithm design or configuration. Furthermore, applying algorithms constructed within the framework to real-world problems is possible, so it can be used for direct applications, although specialized operators or encodings might need to be added. Last but not least, MAHF aims at interoperability, for example by providing access to the COCO benchmark suite [17]—which is already possible—or, in the near future, enabling the evaluation of the experimental data with IOHanalyzer [38].

We are aware that the necessity of yet another framework is disputable, especially as there are calls for one overarching general-purpose framework [12, 32]. However, we believe that this, although a compelling idea, is not feasible and maybe not even sensible. While many frameworks are described as general-purpose, they often still focus on one specific aspect, e.g. application to real-world problems, benchmarking, easy manual construction of algorithms (often without many options for components), or automated algorithm configuration. These aspects overlap, but when wanting to investigate some specific research question, it is often necessary to expand or adapt the existing framework in that direction. This is not an easy task, e.g. due to the complexity of the existing code, sparse documentation, or simply insufficient proficiency in the programming language. Furthermore, there are cases where the performance of the code in terms of physical time is relevant, which restricts the use of certain programming languages.

In the following, a detailed introduction to the structure and functionalities of MAHF is given. Section 2 provides a short overview of other frameworks and fundamental concepts. In Section 3, the main design principles and concepts of MAHF are described in detail, highlighting the role and interactions of its different modules. This includes examples of the incorporated pre-defined algorithms and their application, the existing components and their implementation, and the usage of optimization problems. Finally, future concepts and extensions to be integrated into the framework are discussed in Section 4.

2 RELATED WORK

There exist numerous implementations of individual metaheuristics, as well as frameworks capable of providing different algorithms, in a multitude of programming languages. They usually differ in how they allow the configuration of metaheuristic algorithms, the operators and other components they provide, and the ways components can be combined. Most importantly, however, their main goals can differ, e.g. from focusing on only evolutionary or swarm-based algorithms, or being specialized in automated algorithm configuration.

While revisiting every individual metaheuristic implementation is out of scope, we want to give a short overview of existing metaheuristic frameworks to help the reader to categorize MAHF according to similarities in functionality and capabilities. An earlier and more extensive summary of metaheuristic frameworks was provided by [30]. However, some of the frameworks are not locatable any more or seem to be no longer maintained. Therefore, we compiled a (non-exhaustive) list of frameworks that don't suffer from these drawbacks (cf. Table 1).

While MAHF aims at experimental analyses, it will ultimately be similar to ParadisEO [12] and HeuristicLab [37]. However, it should—after incorporating further extensions (cf. Section 4)—be more versatile, with more components and operators available, while still providing excellent performance. Furthermore, extending MAHF by new components, operators or optimization problems should be much easier than adding those to ParadisEO or HeuristicLab, as the suitable structure for this is one of MAHF's main characteristics. There are, of course, also metaheuristic implementations written in Rust, though to our knowledge there are none that are comparable to the frameworks listed above in terms of variety and variability of the included algorithms. As MAHF utilizes a

Table 1. Some current metaheuristic frameworks

Framework	Language	URL	Sources
ParadisEO	C++	https://github.com/nojhan/paradiseo	[6, 12, 20]
FOM	Java	https://www.isa.us.es/3.0/tool/fom/	[29]
HeuristicLab	C#	https://dev.heuristiclab.com	[36, 37]
ECJ	Java	https://cs.gmu.edu/~eclab/projects/ecj/	[24, 32]
Opt4J	Java	https://sdarg.github.io/opt4j/index.html	[23]
jMetal	Java	https://github.com/jMetal/jMetal	[13, 26]
Cllib	Scala	https://github.com/ciren/cilib	[28]
JGEA	Java	https://github.com/ericmedvet/jgea	[25]
IOHexperimenter	C++	https://github.com/IOHprofiler	[10, 11]
PSO-X	C++	https://code.ulb.ac.be/lab/IRIDIA	[7]
DEAP	Python	https://github.com/deap/deap	[15]
Nevergrad	Python	https://github.com/facebookresearch/nevergrad	[31]
SPOT	R	https://cran.r-project.org/package=SPOT	[4]
MitLware	Scala	https://github.com/MitLware	[33]

unified representation of metaheuristics and easily interchangeable components, its structure is in parts similar to the metaheuristic design patterns as described by [19] and [39], though they were not used as a basis when implementing MAHF.

MAHF will come with its own evaluation tools. However, data formats for the use of existing tools will be provided in the future. This will include, for example, IOHanalyzer [38], DSCTool [14] and BIAS [35].

3 THE MAHF FRAMEWORK

MAHF is a framework for the modular construction of metaheuristic algorithms and their application to optimization problems. It is implemented in the Rust programming language[2], which is known for its compile-time memory- and thread-safety guarantees while offering runtime speed comparable to C++ [18]. While language speed may not be relevant in (real-world) applications where the evaluation of the objective function takes up the majority of computing time anyway, and the algorithm has to provide a good enough solution only once, the same is not true for experiments on benchmark suits like COCO. Faster execution therefore directly aids practitioners in developing and evaluating algorithms, as more configurations can be examined in the same amount of time, enabling more comprehensive experiments. MAHF is open-source and can be found on GitHub[3], together with its related projects, e.g. its evaluation tools[4].

3.1 Goals and General Concept

MAHF primarily aims at

- providing an extensive collection of different components and operators,
- enabling the construction of any metaheuristic framework or algorithm from individual components, and
- facilitating fair experiments and comparisons of different metaheuristic algorithms.

[2]https://www.rust-lang.org/
[3]https://github.com/mahf-opt/mahf
[4]https://github.com/mahf-opt/mahf-evaluation

In the long run, however, MAHF should be as general-purpose as possible. That means, next to experimental studies and research, it should also be applicable in real-world optimization scenarios.

MAHF is implemented under consideration of the notion of a unified view of metaheuristics. That is, metaheuristics are modularized into components, some of which are the typical search operators, while others implement additional functionalities. These components can be used to construct any metaheuristic (framework and algorithm), additionally making all corresponding types of hybridizations between different metaheuristics possible, e.g. PSO and EAs or even ACO and EAs.

The following sections will provide a detailed overview of MAHF's modules, their functions, and their usage. The next section specifically will motivate and introduce the general architecture of the MAHF framework, after which the concepts of problems, heuristics, components, and conditions are described in detail. Other features like logging and evaluation metrics are also outlined. Generally, a focus will be on explaining how to use MAHF and how to extend and adapt it to one's requirements. To this end, we will provide sufficient examples of the individual functionalities. For detailed information on MAHF's implementation, there is documentation available within the repository, especially at `mahf::docs`.

3.2 Framework

The goal of MAHF is to provide a high-level view of metaheuristics, while still offering fine-grained control over low-level details. Its most important paradigms are:

- Divide static and dynamic information.
- Everything is a component.
- Components communicate through a shared blackboard.

Static and dynamic information. A metaheuristic execution is split into two parts:

- the algorithm description (e.g. pseudocode) and its initial parameters,
- and the assignment of all variables in the algorithm at a certain point in time.

The algorithm description, or *configuration*, is static and won't change over the execution of the algorithm. The assignment of variables, or algorithm *state*, on the other hand, is inherently temporary and may change in each step of the algorithm. In MAHF, a `Configuration` describes an algorithm, or specifically a metaheuristic, and is itself *stateless*. The `State` then contains all runtime information of an algorithm. Together, `Configuration` and `State` fully describe the execution of a metaheuristic, or algorithm in general.

Component-based architecture. A component is a building block that implements a (small) functionality. They may access and modify a private internal state, a state that is shared with other components, or the global metaheuristic state. MAHF treats any such functionality as a component, including any metaheuristic operators, logging, metrics, and even control flow itself. A reasonable approach to a metaheuristic framework is to specify a metaheuristic template and pre-implement relevant operators, which can then be used to construct a customized version of the heuristic. This works reasonably well in practice, as long as the structure of the metaheuristic itself is not part of the wanted customization. Such a case would then require its own implementation, which, depending on the framework, may not be time-consuming, but it certainly requires a deeper understanding of the inner workings of the framework and is a much higher hurdle in general. MAHF tries to avoid this by always operating on high-level building blocks, which simplifies development and makes it more elegant to implement large functionality. Low-level control can be gained by implementing custom components, which can be exactly as lightweight or powerful as needed.

Sharing information between components. Making everything a component is only useful if they share a common interface. Defining such an interface is not an easy task, as it should offer a level of abstraction high enough to gain an advantage over manual implementation, while still offering enough control to realize any reasonable project within the framework. MAHF employs a state container pattern for this, which itself utilizes Rust's type system. Components can insert, modify, read, and remove so-called *custom state* from the global `State` (blackboard). Custom state can be any type that implements the marker trait `CustomState`, which itself has no functionality. The `State` can hold an arbitrary amount of custom state, but a specific type only once. To define own custom state, a new type can be constructed, either with separate functionality or wrapping existing types. The visibility of the custom state type also controls who can access it, making it easy to encapsulate custom state. For example, if a component adapts its parameters at runtime, it may store a private parameter state type in the `State`, which can then only be modified by itself, because only it can access the type information. These visibility rules are necessarily enforced at compile-time.

MAHF defines some common state types, which are relevant for almost all metaheuristics. For example, `Populations` is the general state for storing `Individuals`, the basic unit which groups a solution to an optimization problem and its objective value(s). As its name suggests, it can not only store a single population of individuals but maintains a stack of populations that can be freely pushed and popped from by components. This makes it possible to generalize evolutionary operators like selection and replacement to operations on the population stack, creating a common interface. More details on this will be given in Section 3.5. Other common state types include (the number of) `Iterations` or `Evaluations`, the `BestIndividual` yet found in single-objective context, the currently known approximation of the `ParetoFront` for multi-objective problems, and a shared seeded random number generator `Random`. All of this is located in the `state` module, an overview of which is given in Figure 1.

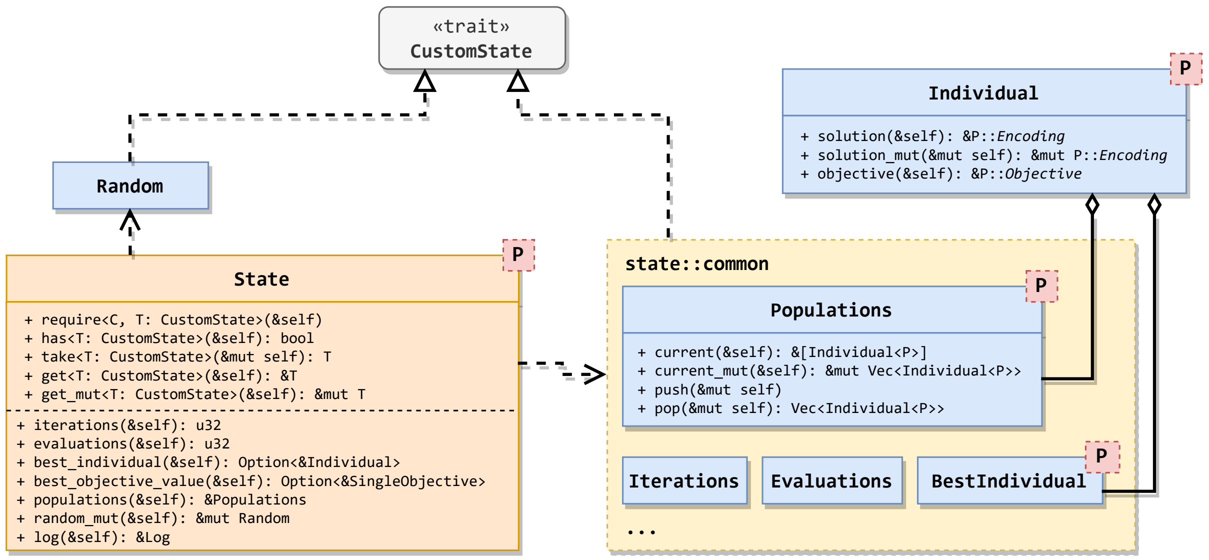
«trait»
CustomState
Random
Individual
P
+ solution(&self): &P::Encoding
+ solution_mut(&mut self): &mut P::Encoding
+ objective(&self): &P::Objective
State
P
+ require<C, T: CustomState>(&self)
+ has<T: CustomState>(&self): bool
+ take<T: CustomState>(&mut self): T
+ get<T: CustomState>(&self): &T
+ get_mut<T: CustomState>(&self): &mut T
+ iterations(&self): u32
+ evaluations(&self): u32
+ best_individual(&self): Option<&Individual>
+ best_objective_value(&self): Option<&SingleObjective>
+ populations(&self): &Populations
+ random_mut(&self): &mut Random
+ log(&self): &Log
state::common
Populations
P
+ current(&self): &[Individual<P>]
+ current_mut(&self): &mut Vec<Individual<P>>
+ push(&mut self)
+ pop(&mut self): Vec<Individual<P>>
Iterations
Evaluations
BestIndividual
P
...

Fig. 1. **State management in MAHF**: The `State` container stores arbitrary `CustomState` that can be read and written by components. Some common metaheuristic state like trackers for iterations and evaluations or storage for populations of `Individuals` are built into the framework. Some types are parametrized by the problem type P.

Fitting it all together. Components in MAHF are *immutable* in the sense that they only contain parameters that are not changed during execution and are therefore static information. Note that any mutable state is inserted by components into the State at runtime and is therefore not strictly part of the component, but only managed by it. Specifically, a Configuration only contains a single Block component, which groups arbitrary other components. A Configuration can be executed by calling the optimize method with a given problem as argument, which returns the State after the execution has finished. While only the final state of the metaheuristic is relevant for applications where only the problem's optimum is sought after, experiments examining metaheuristic behaviour typically depend on information collected over the whole runtime. MAHF specifically addresses this issue with an extensive and customizable logging system (cf. Section 3.7).

Figure 2 gives an overview of the core functionality of MAHF, highlighting the interaction and relationships between Configurations, Components, and the State. Metaheuristics like GA and PSO are represented by Configurations, which are made up of Components and Conditions (components that evaluate to a boolean value). Components and conditions communicate using the State, which is instantiated by the Configuration. The Configuration, Components, and Conditions are parametrized by the problem type. It also shows a selection of common components extracted from metaheuristics and often-used conditions. The following sections will go into detail about these modules.

```rust
use mahf::prelude::*;
use problems::bmf::BenchmarkFunction;

// Specify the problem: Sphere function with 10 dimensions.
let problem: BenchmarkFunction = BenchmarkFunction::sphere(/*dim: */ 10);
// Specify the metaheuristic: Particle Swarm Optimization (pre-implemented in MAHF).
let config: Configuration<BenchmarkFunction> = pso::real_pso(
    /*params: */ pso::RealProblemParameters {
        num_particles: 20,
        weight: 1.0,
        c_one: 1.0,
        c_two: 1.0,
        v_max: 1.0,
    },
    /*termination: */ termination::FixedIterations::new(/*max_iterations: */ 500)
        & termination::DistanceToOpt::new(0.01),
);

// Execute the metaheuristic on the problem with a random seed.
let state: State<BenchmarkFunction> = config.optimize(&problem);

// Print the results.
println!("Found Individual: {:?}", state.best_individual().unwrap());
println!("Global Optimum: {}", problem.known_optimum());
```

Listing 1. **Applying PSO to the Sphere benchmark function**: The code constructs the benchmark function with 10 dimensions, configures a real-valued PSO, and executes it for 500 iterations, stopping early if within 0.01 of the known optimal objective value. Afterwards, the best found individual and the known global optimum are printed. Type annotations are given for clarity, but are not needed for execution.

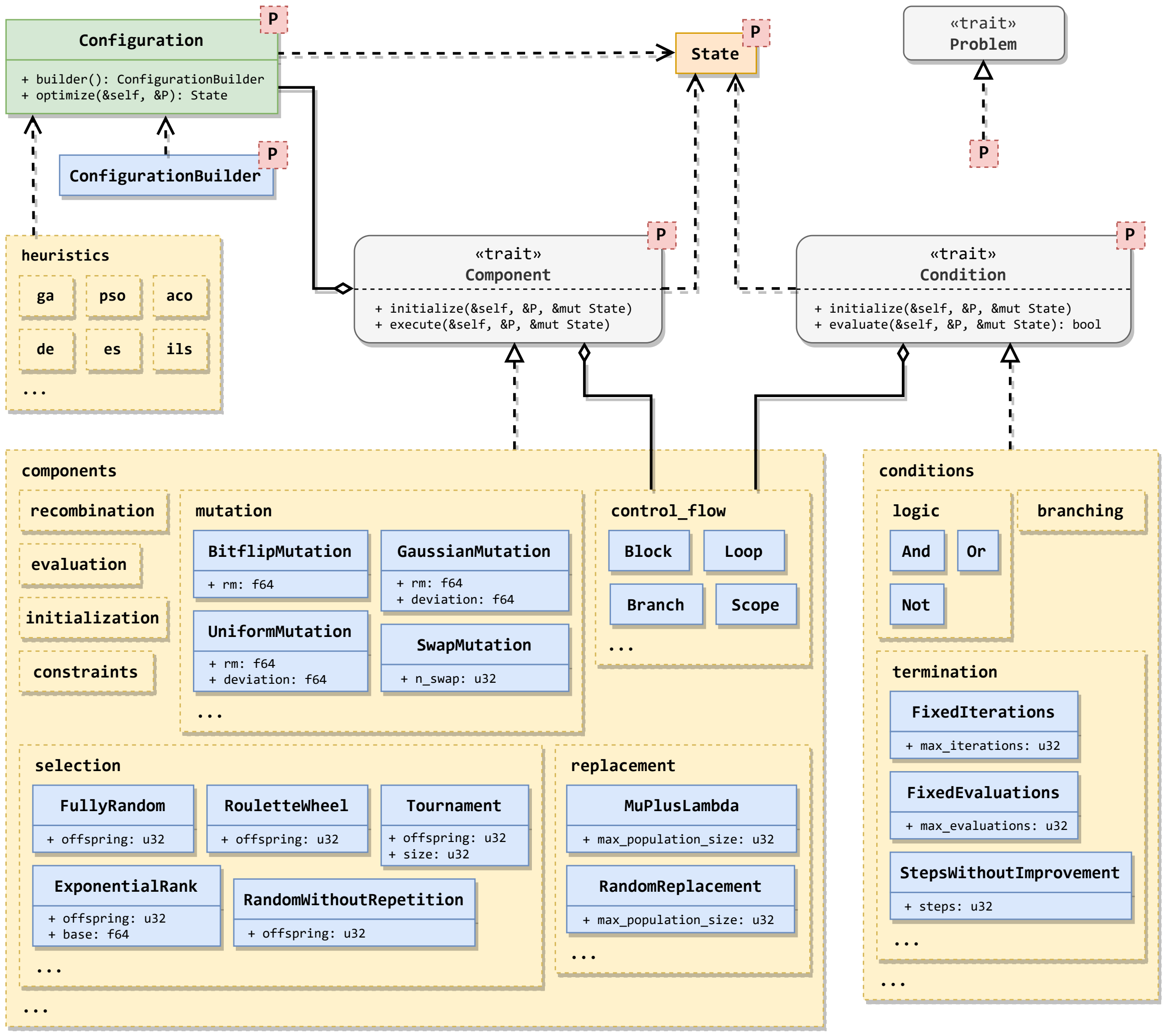

Fig. 2. **The MAHF framework:** Overview of the interaction between configurations, components, and the state.

Applying PSO to the sphere benchmark function might look like the example given in Listing 1. The example uses a pre-implemented PSO as optimizer. Listing 2 uses `Configuration`'s builder syntax to construct an Ant System and applies it to a TSPLIB instance. More information and examples of this will be given in Section 3.4.

```rust
use mahf::prelude::*;
use aco::ant_ops;
use problems::tsp::{self, SymmetricTsp};
use tracking::{files, functions, trigger};

// Specify the problem: TSPLIB instance Berlin52.
let problem: SymmetricTsp = tsp::Instances::BERLIN52.load();
// Specify the metaheuristic: Ant System (equivalent to `aco::ant_system`).
let config: Configuration<SymmetricTsp> = Configuration::builder()
    .do_(initialization::Empty::new())
    .while_(
        termination::FixedEvaluations::new(/*max_evaluations: */ 10_000),
        |builder| {
            builder
                .do_(ant_ops::AcoGeneration::new(
                    /*num_ants: */ 20, /*alpha: */ 2.0, /*beta: */ 1.0,
                    /*initial_pheromones: */ 0.0,
                ))
                .evaluate()
                .update_best_individual()
                .do_(ant_ops::AsPheromoneUpdate::new(
                    /*evaporation: */ 0.2, /*decay_coefficient: */ 1.0,
                ))
                .do_(tracking::Logger::new())
        },
    )
    .build();

// Execute the metaheuristic on the problem.
let state: State<SymmetricTsp> = config.optimize_with(&problem, |state| {
    // Set the seed to 42.
    state.insert(Random::seeded(42));
    // Log the best individual every 50 iterations.
    state.insert(
        tracking::LogSet::<SymmetricTsp>::new()
            .with(trigger::Iteration::new(50), functions::best_individual),
    );
});

// Save the log to file "aco_berlin52.log".
files::write_log_file("aco_berlin52.log", state.log()).unwrap();
```

Listing 2. **Applying Ant System to the TSPLIB Berlin52 instance**: The code constructs the TSPLIB instance, configures an Ant System, and executes it for 10000 objective function evaluations. Every 50 iterations, the best individual yet found is logged and written to a log file afterwards. Type annotations are given for clarity, but are not needed for execution.

3.3 Problems

Optimization problems in MAHF have three defining properties:

- a solution encoding E (genotype),
- an objective type O, usually single- ($O = \mathbb{R}$) or multi-objective ($O = \mathbb{R}^n$, $n > 1$), and
- an objective function in the form $f : E \rightarrow O$.

Those properties are grouped under the `Problem` and `Evaluator` traits, which form the minimal requirement for optimization problems. Several traits are built on top of `Problem`, which group problems by their characteristics (e.g. a vector-based solution encoding) or enable heuristics to access problem-specific information (e.g. the search space boundary in continuous optimization). The basic relationships of problems with single and multiple objectives are visualized in Figure 3.

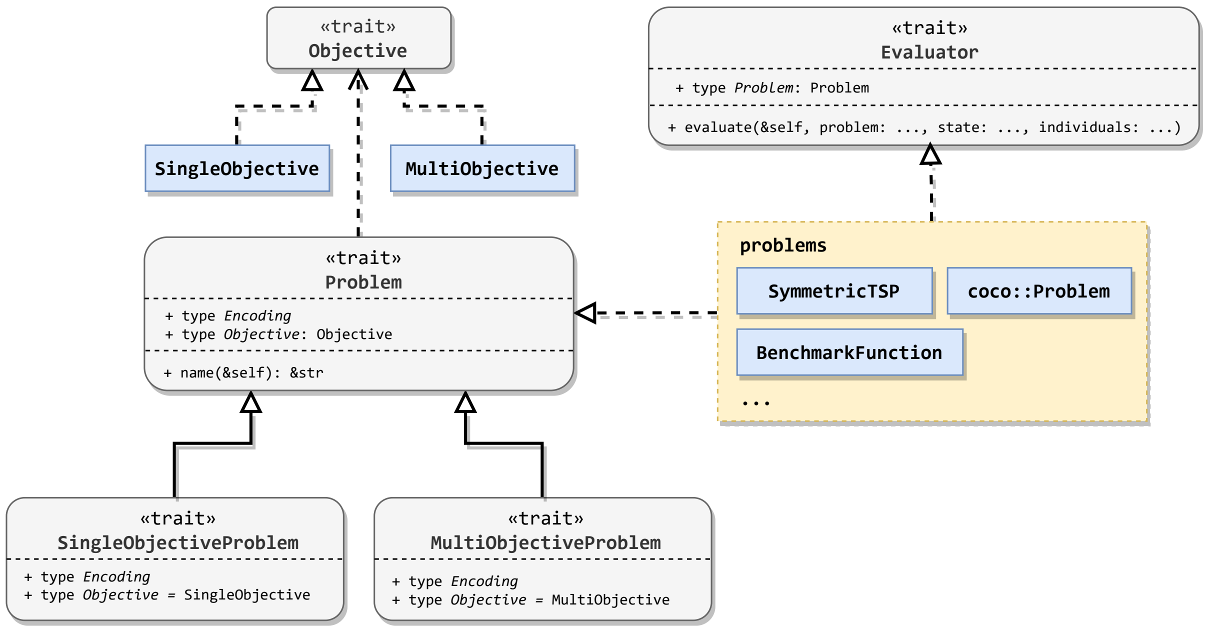

Fig. 3. **Problem representation in MAHF**: Every problem is associated with two types: the solution encoding (e.g. real-valued vector) and the objective type (e.g. single objective). It must then provide means of mapping from one to the other.

MAHF comes with three problems pre-implemented at the time of writing:

- a collection of popular continuous benchmark functions,
- bindings to the COCO benchmark suite, and
- a selection of TSPLIB instances.

Own problems can be added by implementing the `Problem` and `Evaluator` traits for the problem, providing both metadata and the possibility to evaluate solutions.

3.4 Heuristics

The `heuristics` module contains a submodule for each pre-implemented metaheuristic. A core goal of MAHF is to enable the investigation of specific research questions, which naturally entails wildly different requirements

```
Configuration::builder()
    .while_(termination, |builder| {
        builder
            .do_(selection)
            .do_(crossover)
            .if_(
                branching::RandomChance::new(/*mutation probability: */ pm),
                |builder| builder.do_(mutation),
            )
            .do_(constraints)
            .evaluate()
            .update_best_individual()
            .do_(replacement)
    })
    .build()
```

Listing 3. **An example GA defined using MAHF's builder syntax**: The `selection`, `crossover`, `mutation`, `constraints`, and `replacement` variables contain components performing the specific operations.

concerning metaheuristic implementations. This module, therefore, is in no way intended to be an exhaustive implementation of all (problem-specific) metaheuristics found in the literature, but an assembly of common or distinct heuristics for selected problem types. It has two main purposes: to show how certain components are used, and to serve as a starting point for custom heuristic definitions. The unmodified implementations in this module only fit basic needs by design, and practitioners are actively encouraged to rely on MAHF's powerful component system to adapt them to their purposes.

Algorithm 1: The example GA in Listing 3 as pseudocode.

```
Input: Objective function $f$ with domain $A$, pre-initialized Population $P$, mutation probability $p_m$
1  $i_{\text{best}} \leftarrow NULL$;
2  while termination criterion not met do
3      $C \leftarrow$ selection($P$);
4      crossover($C$);
5      Generate $p \in [0, 1]$;
6      if $p \leq p_m$ then
7          mutation($C$);
8      Constrain all individuals $\in C$ to $A$;
9      Evaluate all individuals $\in C$ using $f$;
10     Update the best individual $i_{\text{best}}$;
11     $P \leftarrow$ replacement($P$, $C$);
Output: Best individual $i_{\text{best}}$
```

```
Configuration::builder()
    .do_(initialization::RandomSpread::new_init(num_particles))
    .evaluate()
    .do_(state::PsoState::initializer(v_max))
    .while_(termination::FixedIterations::new(max_iterations), |builder| {
        builder
            .do_(generation::swarm::PsoGeneration::new(
                weight, c_one, c_two, v_max,
            ))
            .do_(constraints::Saturation::new())
            .evaluate()
            .do_(state::PsoState::updater())
    })
    .build()
```

Listing 4. PSO on a real-valued search space using MAHF's builder syntax.

Each metaheuristic submodule can be split into two parts:

- a generic metaheuristic template, and
- problem-specific metaheuristic configurations based on this template.

For example, the `heuristics::ga` module contains the generic `heuristics::ga::ga` template function, which constructs a component containing a genetic algorithm similar to Listing 3.

The problem-specific implementations, e.g. `heuristics::ga::real_ga` and `heuristics::ga::binary_ga`, define an initialization method for the population, insert appropriate components for selection, crossover, etc., into the template and convert it into a full `Configuration` ready for execution.

MAHF relies on Rust's builder pattern[5] to enable simple and idiomatic metaheuristic definitions. The syntax is kept as close as possible to pseudocode, so even users not familiar with Rust should be able to grasp the general structure. Listing 3 and Algorithm 1 show the definition of a simple GA using MAHF's builder syntax and pseudocode, respectively. The subroutines for selection, crossover, mutation, and replacement are left unspecified here.

Note how the Rust code translates into pseudocode almost line by line. A comparison of two definitions of PSO is shown in Listing 4 and Algorithm 2, this time with all components specified.

Currently, eleven heuristics are pre-implemented, which range from classic metaheuristics like the GA to more metaphor-based heuristics like Invasive Weed Optimization. Baseline heuristics like random searches are also included. Table 2 gives a short overview of those heuristics.

New heuristics can be added exactly like shown in Listings 3 to 4, i.e. using MAHF's `ConfigurationBuilder`, or, for convenience, `Configuration::builder`. It supports flow control operators like `do_`, `if_`, `if_else_`, or `while_`, but also shortcuts for often used components, e.g. `evaluate` instead of inserting the `Evaluator` component manually. This is often used in combination with `update_best_individual`, which saves the best individual yet found in a single-objective context. It is also possible to execute components independently of others by using the `scope_` operator, which creates a new child `State` for the block, leaving the parent state

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

Algorithm 2: The real-valued PSO in Listing 4 as pseudocode.

```
Input: Objective function $f$ with domain $A$, number of particles $n$, maximal velocity $v_{\max}$, inertia weight
       $\omega$, acceleration coefficients $\phi_1, \phi_2$, number of iterations $k$
1 Initialize $n$ particles with a random uniform position $\in A$ and velocity $\in [-v_{\max}, v_{\max}]$;
2 Evaluate all particles using $f$;
3 Initialize the local bests and the global best $g$;
4 for $i \leftarrow 0$ to $k$ do
5     Update position and velocity of all particles using $\omega, \phi_1, \phi_1, v_{\max}$;
6     Constrain all particle positions to $A$;
7     Evaluate all particle positions using $f$;
8     Update local bests and the global best $g$;
Output: Global best position $g$
```

Table 2. **All heuristics pre-implemented in MAHF (at the time of writing):** Note that many components (e.g. selection strategies) are not problem-specific, and therefore reusable for many different problem types. The "Problem types" column therefore only refers to the most specific requirements of the components used in a heuristic.

Heuristic	Problem types
Ant Colony Optimization (ACO)	TSP
Chemical Reaction Optimization (CRO)	real-valued
Differential Evolution (DE)	real-valued
Evolution Strategy (ES)	real-valued
Genetic Algorithm (GA)	real-valued, binary
Iterated Local Search (ILS)	real-valued, permutation-based
Invasive Weed Optimization (IWO)	real-valued
Local Search (LS)	real-valued, permutation-based
Particle Swarm Optimization (PSO)	real-valued
Random Search (RS)	real-valued, permutation-based
Random Walk (RW)	real-valued, permutation-based

unmodified by default. This is especially useful for nested local searches or hybrid approaches, which might perform multiple "inner" iterations with respect to the main loop.

3.5 Components

A major design principle of MAHF is to break a metaheuristic down into modular *components* that can easily be replaced or applied in other contexts. The `components` module contains many such modular building blocks, which can be used to construct arbitrary metaheuristics. The term "component" is used in a rather broad sense here, grouping both classical metaheuristic operators and additional functionality like evaluation, calculation of metrics, or even control flow.

In general, a component is a routine that receives the current state of the metaheuristic and performs some actions, modifying the metaheuristic state and its internal state in the process. Note that in MAHF, both of these states are represented by a single `State` struct. This structure allows the expression of any piece of reusable code as a component, using the `State` as a common interface to communicate. MAHF defines various categories of

Table 3. Component categories with an example component.

Category	Example component
Initialization	`RandomSpread`
Selection	`Tournament`
Recombination	`UniformCrossover`
Mutation	`FixedDeviationDelta`
Constraints	`Saturation`
Evaluation	`SequentialEvaluator`
Replacement	`MuPlusLambda`
Miscellaneous	`Debug`
State-bound	`ElitistArchive::update`

such components, which range from basic evolutionary operators like selection, recombination, mutation, and replacement, over heuristic-specific operators such as PSO's particle position update, to generic functionality like constraint-handling techniques. An overview of those categories is given in Table 3, each with an example component.

The following section will focus on the implementation of components in code, illustrating the main concepts needed for reading and understanding component definitions. Afterwards, the component categories implemented in MAHF at the time of writing are introduced and motivated using representative components. The component definitions may be shortened (constructor, asserts omitted, etc.) for the sake of brevity. For an overview and reference of all implemented components see the documentation of the `components` module on GitHub.

3.5.1 *Implementing components.* In code, defining a component can be split into three basic steps:

- Defining the component struct with its parameters,
- adding a constructor, which takes the arguments and returns a `Component` (optional, but recommended), and
- implementing the `Component` trait, i.e. its `execute` method, which capsules the functionality. Optional setup can be done in the `initialize` method.

Listing 5 shows an example `PrintPopulation` component without parameters, which simply prints the current population to the standard output. The `execute` method (line 16 to line 22) receives the `State` (and problem, which is not used in this case), extracts the current population, and iterates over all individuals, printing them.

It can be instantiated using its constructor, i.e. the `new` method: `PrintPopulation::new()`. Note that the preferred way to implement such minor test behaviour is to use the `Debug` component, or rather the short-hand `debug` method of the `ConfigurationBuilder`, together with a closure.

All components can be categorized by their generality: they can be specific to certain problem types (e.g. TSP), require the problem to have certain properties (e.g. the problem must have a single objective), or have no restrictions at all. Most components will impose at least some restrictions on the problems or solutions they can be applied to, which manifests in code as having restrictions on the generic problem type `P` (or not being parametrized at all). For example, the `PrintPopulation` component in Listing 5 requires the problem's solution encoding (`P::Encoding`) to implement a debug print (line 7 and line 14).

As similar types of components often operate on the state in similar ways, MAHF explicitly encourages the use of "translator" traits and structs to reduce code duplication. This is accomplished in code by defining a custom trait with a simpler interface and then implementing a wrapper struct of the trait, which implements the `Component` trait. This pattern is used in most of MAHF's component categories, usually to hide operations on the population

```rust
/// Prints the current population to stdout.
#[derive(Serialize, Clone)]
pub struct PrintPopulation; // No parameters
impl PrintPopulation {
    pub fn new<P: Problem>() -> Box<dyn Component<P>>
    where
        P::Encoding: std::fmt::Debug,
    {
        Box::new(Self)
    }
}
impl<P: Problem> Component<P> for PrintPopulation
where
    P::Encoding: std::fmt::Debug,
{
    fn execute(&self, _problem: &P, state: &mut State<P>) {
        let population: &[Individual<P>] = state.populations().current();
        for individual in population {
            println!("f({:?}) = {:?};", individual.solution(), individual.objective())
        }
    }
}
```

Listing 5. Definition of a simple example component which prints the current population to the standard output.

stack and to convert between `Individual` and the solution encoding. For example, the `GaussianMutation` only implements the `Generation` trait and wraps itself with the `Generator` struct in its constructor, which implements `Component`. The implementation of `GaussianMutation` is shown in Listing 7. Own components and translator traits and structs can be added in a similar fashion.

3.5.2 *Component categories.*

Control flow. Four basic primitives of control flow components are implemented in MAHF: `Block`, `Loop`, `Branch`, and `Scope`, which correspond to the `do_`, `while_`, `if_`, and `scope_` methods of `ConfigurationBuilder`.

Initialization. Initialization components initialize (a population of) individuals from scratch, using a specified distribution. They push a new population on the stack managed by the `Populations` state, ignoring other populations that may be present. For example, the `RandomSpread` component generates n uniform-distributed individuals in a real-valued search space. It requires the problem to define boundaries of the search space and a vector-based solution encoding. For binary problems, the `RandomBitstring` can be used to generate n bitstrings, generating a 1 at each position with a certain probability p.

Selection. Components implementing this evolutionary operator select a subset of the current population, copying them into a new population on the stack, which can then be modified by other components (e.g. mutation components). They do not modify the current population. An example is the `Tournament` component shown

```rust
/// Selects `offspring` using deterministic Tournament selection.
///
/// Solutions can be selected multiple times in a single iteration.
#[derive(Serialize, Deserialize, Clone)]
pub struct Tournament {
    /// Offspring per iteration.
    pub offspring: u32,
    /// Tournament size.
    pub size: u32,
}
impl<P: SingleObjectiveProblem> Selection<P> for Tournament {
    fn select_offspring<'p>(
        &self,
        population: &'p [Individual<P>],
        state: &mut State<P>,
    ) -> Vec<&'p Individual<P>> {
        let mut selection = Vec::new();
        for _ in 0..self.offspring {
            let winner = population
                .choose_multiple(state.random_mut(), self.size as usize)
                .min_by_key(|&i| i.objective())
                .unwrap();
            selection.push(winner);
        }
        selection
    }
}
```

Listing 6. **Definition of the `Tournament` selection component**: the winners of `offspring` tournaments with size `size` are selected.

in Listing 6, which performs the well-known tournament selection, selecting n individuals using a certain tournament size k.

Generation. This category groups all components which generate a new population from the current population, usually modifying the current population in-place. Special cases are initialization/generation hybrid components like `RandomSpread`, which can be configured to either be an initialization component (generating and pushing a new population on the stack) or as a generation component (replacing the current population). The generation category currently defines three subcategories, namely recombination, mutation, and specialized swarm operators.

Generation::Recombination. Recombination or crossover components combine multiple individuals into (multiple) new individuals. An exemplary component for this category is the `NPointCrossover` component, which splits two solutions at n points and interleaves them with a certain probability p, creating either one or two children depending on a parameter.

Generation::Mutation. Components of this generation subcategory perform a perturbation on single individuals. The `GaussianMutation` component (Listing 7), for example, executes a mutation of each solution dimension with a certain probability p using $\mathcal{N}(0, \sigma)$-distributed values, as often seen in real-valued evolutionary algorithms. For permutation-based problems, the `InversionMutation` component takes a random slice of the solution and inverts it.

Constraints. As generation components are not required to respect constraints and search space boundaries, components of this category convert infeasible solutions into feasible ones. The `Saturation` component, for example, implements a simple clamp strategy for keeping solutions inside the search space in real-valued optimization, while the `Mirror` component reflects the amount exceeding the boundary inwards at the same boundary.

Evaluation. This category contains components related to objective value(s). For example, the `Evaluator` computes objective value(s) for the current population, while the `UpdateBestIndividual` component tracks the best individual yet found for problems with a single objective.

Replacement. Replacement components perform the evolutionary operator of replacing individuals from the parent generation with individuals from the child generation, i.e. mix the current and previous population, reducing the size of the population stack by one. Example components for this are the `MuPlusLambda` and `Generational` components, which use a $\mu + \lambda$ strategy (keep n fittest individuals from both generations; cf. Listing 8) or completely replace the parent population with the child population, respectively.

Miscellaneous. This category groups special components meant as helpers and for debugging. A main example of this is the `Debug` component, which allows for quickly inserting custom functionality into a configuration. It is most efficiently used through the `debug()` and `assert()` methods on `ConfigurationBuilder` (Listing 9), which accept closures operating on the `State` or problem. Note that they are meant for debugging only, real functionality should be implemented by implementing new components.

State-bound. It is not unusual for metaheuristics to have metadata beyond the current population of individuals. PSO, for example, additionally tracks the velocities and history of individuals, defined in its `PsoState`. Components that only revolve around updating a particular state are called state-bound and are usually constructed through a method on the respective state struct. The component updating an archive of the n best individuals evaluated (`ElitistArchive`) using the current population is, for example, constructed using the `ElitistArchive::update` method.

3.6 Conditions

Conditions are similar to components in the sense that they group reusable functionality behind a common interface. The main difference lies in their application: while components interact only through the state, conditions are evaluated directly by components. The boolean return value of the `evaluate` method (condition equivalent of components `execute` method) is then used to decide how to proceed: for example, the `Branch` control flow component evaluates a `Condition` to decide on the execution of either the “if” or “else” branch.

3.6.1 *Implementing conditions.*
Conditions are defined identically to components (cf. Section 3.5.1), only implementing the `Condition` trait instead of `Component`.

3.6.2 *Condition categories.*

Logic. Logic conditions allow combining other conditions using boolean operators like AND, OR, and NOT. They implement the bit operators in Rust, allowing concatenating conditions using the `&`, `|`, and `!` operators.

```rust
/// Applies a gaussian mutation to each position depending on mutation rate.
/// The distribution is centered around 0 and the resulting value is added to the value of the solution.
///
/// Uses a Gaussian distribution.
///
/// If rm = 1, all positions of `solution` are mutated.
#[derive(Serialize, Deserialize, Clone)]
pub struct GaussianMutation {
    /// Probability of mutating one position.
    pub rm: f64,
    /// Standard Deviation for the mutation.
    pub deviation: f64,
}
impl GaussianMutation {
    pub fn new<P>(rm: f64, deviation: f64) -> Box<dyn Component<P>>
    where
        P: Problem<Encoding = Vec<f64>>,
    {
        Box::new(Generator(Self { rm, deviation }))
    }
}
impl<P> Generation<P> for GaussianMutation
where
    P: Problem<Encoding = Vec<f64>>,
{
    fn generate_population(
        &self,
        population: &mut Vec<P::Encoding>,
        _problem: &P,
        state: &mut State<P>,
    ) {
        let distribution = rand_distr::Normal::new(0.0, self.deviation).unwrap();
        let rng = state.random_mut();

        for solution in population {
            for x in solution {
                if rng.gen_bool(self.rm) {
                    *x += distribution.sample(rng);
                }
            }
        }
    }
}
```

Listing 7. **Definition of the `GaussianMutation` component:** it mutates each solution dimension with probability `rm` using a $\mathcal{N}(0, \sigma)$ distribution. Because it only implements the `Generation` trait, it has to wrap itself in a `Generator` in its constructor to be a full `Component`.

```
/// Always keeps the fittest individuals.
#[derive(Serialize, Deserialize, Clone)]
pub struct MuPlusLambda {
    /// Limits the population growth.
    pub max_population_size: u32,
}
impl<P: SingleObjectiveProblem> Replacement<P> for MuPlusLambda {
    fn replace_population(
        &self,
        parents: &mut Vec<Individual<P>>,
        offspring: &mut Vec<Individual<P>>,
        _state: &mut State<P>,
    ) {
        parents.append(offspring);
        parents.sort_unstable_by_key(|i| *i.objective());
        parents.truncate(self.max_population_size as usize);
    }
}
```

Listing 8. **Definition of the `MuPlusLambda` replacement component:** as the name suggests, it performs a $\mu+\lambda$ replacement strategy to replace the previous generation.

Termination. Most metaheuristics are not specific to certain termination criteria or can be easily extended to custom ones. Conditions of this type are primarily intended to control the termination of the main loop, for example, performing a certain amount of iterations (`FixedIterations`, cf. Listing 10) or stopping when no better solution was found in the last n steps (`StepsWithoutImprovement`).

Branching. Conditions for branching currently include `RandomChance`, which evaluates to a random outcome with a certain probability, and metaheuristic-specific criteria for, e.g. CRO.

3.7 Logging

MAHF has an extensive built-in logging system based on triggers and extractors. Triggers describe when something should be logged by specifying a logging condition, while extractors define what should be logged. MAHF similarly provides common triggers that allow logging every x iteration or when some state is changed by a certain amount, but custom triggers can be defined as well. The most commonly used extractor is the `auto` extractor, which takes care of logging any primitive state such as integer or floating point values. For more complex cases, such as logging the population, a custom extractor can be defined as well. Extractors can log whatever state they want in whatever form they want, so they are a very powerful logging tool. These triggers and extractors are then combined into a `LogSet`, which can be inserted into the `State` when calling `Configuration::optimize_with`. A simple example of this is given in Listing 2, while Listing 11 shows how more detailed logging might look. When reaching a `Logger` component in the evaluation, the `LogSet` will run every extractor where the trigger's criteria are met. This simple system allows highly flexible and powerful logging. When defining a heuristic configuration, a `Logger` can be inserted in places where logging should

```
1  Configuration::builder()
2      // Initialize the single solution.
3      .do_(initialization::RandomPermutation::new_init(/*initial_population_size: */1))
4      .evaluate()
5      .update_best_individual()
6      // Main loop of the iterated local search: 10 outer iterations.
7      .while_(termination::FixedEvaluations::new(/*max_evaluations: */10), |builder| {
8          builder
9              // Perturb the solution for restart.
10             .do_(generation::RandomPermutation::new_gen())
11             // Do we still have only a single solution in the population?
12             .assert(|state| state.populations().current().len() == 1)
13             .evaluate()
14             // Inner local search loop wrapped in a new scope: 100 inner iterations.
15             .do_(selection::All::new())
16             .scope_(|builder| {
17                 builder.while_(termination::FixedIterations::new(/*max_evaluations: */100), |builder| {
18                     builder
19                         // Generate and evaluate 50 neighbors and replace the solution if better.
20                         .do_(selection::DuplicateSingle::new(/*offspring: */50))
21                         .do_(generation::mutation::SwapMutation::new(/*n_swap: */5))
22                         .evaluate()
23                         .do_(replacement::MuPlusLambda::new(/*max_population_size: */1))
24                 })
25             })
26             .update_best_individual()
27             // Let's see what the solution found by the local search looks like.
28             .debug(|_problem, state| println!("{:?}", state.populations().current()))
29             .do_(replacement::MuPlusLambda::new(1))
30     })
31     .build()
```

Listing 9. **Example of the debug and assert capabilities of `ConfigurationBuilder` on debugging an ILS:** line 12 and line 28 show the usage of `assert` and `debug`, respectively.

be performed, usually after an iteration. Figure 4 provides a general outline of the logging module and the relationships within.

MAHF logs are stored as dense tables in the CBOR [5] format. It uses the CBOR format to save memory compared to JSON, but the logs can easily be viewed as such using the tool available on https://cbor.me/. When viewed as JSON, logs have a structure as shown in Listing 12. MAHF also provides a Python library to conveniently load stored log files. This allows experiments to be conducted in Rust while doing the evaluation and performance analysis in Python. Other languages can be used easily as well, as long as they have a CBOR implementation.

```rust
/// Terminates after a fixed number of iterations.
#[derive(Serialize, Deserialize, Clone)]
pub struct FixedIterations {
    /// Maximum number of iterations.
    pub max_iterations: u32,
}
impl<P> Condition<P> for FixedIterations
where
    P: Problem,
{
    fn initialize(&self, _problem: &P, state: &mut State<P>) {
        state.require::<Self, Iterations>();
        state.insert(Progress(0.));
    }

    fn evaluate(&self, _problem: &P, state: &mut State<P>) -> bool {
        let iterations = state.iterations();
        state.set_value::<Progress>(iterations as f64 / self.max_iterations as f64);

        iterations < self.max_iterations
    }
}
```

Listing 10. Definition of the `FixedIterations` condition.

```rust
let state = config.optimize_with(&problem, |state| {
    state.insert(
        LogSet::new()
            .with_common_extractors(trigger::Iteration::new(50))
            .with(
                trigger::Iteration::new(50),
                functions::normalized_diversity::<_, TrueDiversity>,
            )
            .with(trigger::Iteration::new(100), functions::best_individual)
            .with(trigger::FinalIter::new(), functions::best_individual),
    )
});
```

Listing 11. **Logging in MAHF:** the number of evaluations, the termination progress (`common_extractors`), and a diversity measure are logged every 50 iterations, while the best individual found is logged every 100, and in the final iteration.

```
{
  "names": ["Iteration", "BestFitness"],
  "entries": [
    [
      {"key": 0, "value": 1},
      {"key": 1, "value": 100.0}
    ],
    [
      {"key": 0, "value": 2}
    ]
  ]
}
```

Listing 12. **An example log file as JSON**

While MAHF uses a custom format that is incompatible with other existing formats (such as the IOHprofiler format [11]), when logging at the required steps, it contains all the data needed to generate files in the respective formats. In the future, such functionality will likely be provided by MAHF directly.

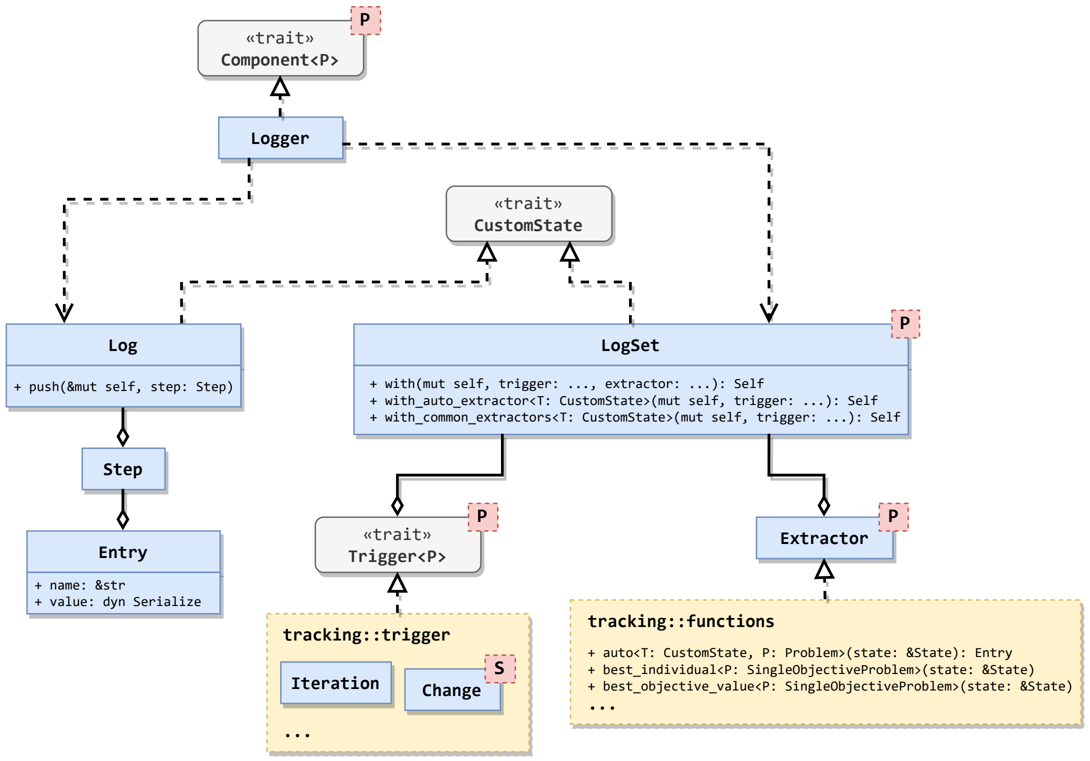
«trait» Component<P>
P
Logger
«trait» CustomState
Log
+ push(&mut self, step: Step)
LogSet
+ with(mut self, trigger: ..., extractor: ...): Self
+ with_auto_extractor<T: CustomState>(mut self, trigger: ...): Self
+ with_common_extractors<T: CustomState>(mut self, trigger: ...): Self
Step
Entry
+ name: &str
+ value: dyn Serialize
«trait» Trigger<P>
Extractor
tracking::trigger
Iteration
Change
S
...
tracking::functions
+ auto<T: CustomState, P: Problem>(state: &State): Entry
+ best_individual<P: SingleObjectiveProblem>(state: &State)
+ best_objective_value<P: SingleObjectiveProblem>(state: &State)
...

Fig. 4. **Logging in MAHF:** The LogSet defines when and what to log using Triggers and Extractors, while the Logger component performs the actual logging and stores it in the Log for later retrieval.

3.8 Experiments and Evaluation with MAHF

Tools for the evaluation and analysis of the data can be found in the MAHF-evaluation repository on GitHub[6]. While many functionalities of the evaluation procedures are still work in progress, basic tools for examining the experimental data are already available.

Evaluation and data analysis procedures intended for MAHF's generated data include data formatting and preprocessing functions, summary statistics, statistical analysis methods, including Bayesian analysis, for performance and behavioural analysis, and the output of results as suitable plots and tables. While evaluation scripts will be available for common experiment designs, adapting those for own experiments is, of course, possible. Additionally, common analysis approaches are provided in Jupyter notebooks to exemplify the individual steps and intermediate results.

Next to extending available analysis procedures, there will also be an option to get the data generated with MAHF in other data formats, most notably the COCO data format [16]. This will allow, for example, the usage of IOHanalyzer [38] for analysing and evaluating the results.

Metrics and Measures. MAHF comes with an assortment of metrics and measures to directly include additional information on the algorithmic runs in the logs. This way, unnecessarily complex and data heavy approaches such as logging entire populations to calculate the respective values of interest in the subsequent analysis can be avoided. Furthermore, the resulting values can be used within the algorithm as criteria for adaptation strategies.

Currently, MAHF incorporates different diversity measures as described in [9]. These are applicable to all population-based algorithms for real-valued problem types. The following measures can be found in the `state::diversity` module:

- `DimensionWiseDiversity`
- `PairwiseDistanceDiversity`
- `TrueDiversity`
- `DistanceToAveragePointDiversity`

All of those measures are normalized as described in [9].

The usage of additional measures and metrics is connected to the logging capabilities of MAHF and thus quite simple. Other measures, e.g. ones related to convergence, will be added in future releases.

4 FUTURE WORK

MAHF is still in its early stages, but while its main focus currently is on behaviour analysis and experimental comparisons of metaheuristic algorithms, its modular structure is also promising for other extensions and approaches. Therefore, in terms of its primary purpose, MAHF will be extended by adding different operators to recreate more metaheuristics for analysis and comparison. This will especially entail the incorporation of state-of-the-art algorithms. Furthermore, the assortment of benchmark problems and also real-world problems will be extended. While MAHF is currently focused on numerical optimization, these extensions will include a variety of combinatorial problems and algorithmic operators as well.

Additionally, extending MAHF entails expanding its capabilities to multi-objective problems and respective algorithmic optimization approaches. While the framework already extends to multiple objectives, no specific problems and components are implemented yet. Presumably, the COCO bbob-biobj benchmark functions will be usable as bi-objective benchmark problems.

Finally, with respect to MAHF's focus on analysis and experimentation, the evaluation procedures, statistical analysis variations, and plotting options will be extended as well, to provide basic templates for adapting to the current experimental question.

[6]https://github.com/mahf-opt/mahf-evaluation

In terms of other extensions, future work on MAHF will most probably include the development and testing of hyperheuristic approaches. MAHF's components are most suitable to be utilized by constructive hyperheuristics and automated algorithm configuration techniques. This can even be extended to not defining the metaheuristic framework beforehand, but including conditions and internal heuristics in the procedure. Additionally, MAHF facilitates exchanging components and operators during the search procedure, which will provide additional options to automatically adapt an algorithm to a given problem.

While special care was taken in the design of MAHF to be understandable by practitioners who are not proficient in Rust, it is still an obstacle to its more widespread adoption. Exposing some of MAHF's functionality to Python will alleviate this issue while enabling integration with the wealth of frameworks for, e.g. machine learning currently available in the Python ecosystem.

5 SUMMARY

This paper presented MAHF, a framework for the modular construction of metaheuristics utilizing a large set of components. MAHF's primary goal is the conduction of large experiments for fair and reproducible comparisons of different algorithmic strategies, with a focus on analysing their search behaviour next to their general performance. To this end, it provides extensive algorithm configuration possibilities, measures related to the algorithmic behaviour, and a wide range of logging options. However, it can also be used for other purposes, for example real-world applications, hyperheuristic approaches or automated algorithm configuration.

Next to detailing MAHF's goals, this paper also described the general implementation of its modules to make MAHF generally easier to use, extend, and adapt to one's own purposes. This included the choice or implementation of optimization problems, the use of pre-configured algorithms in the `heuristics` module, and the combination and addition of components. While MAHF is still actively developed and expanded, we hope that the detailed explanation in this paper nevertheless encourages other researchers to try it and see if it fits their needs.

ACKNOWLEDGMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] Claus Aranha, Christian L. Camacho Villalón, Felipe Campelo, Marco Dorigo, Rubén Ruiz, Marc Sevaux, Kenneth Sörensen, and Thomas Stützle. 2021. Metaphor-based metaheuristics, a call for action: the elephant in the room. *Swarm Intelligence* 16, 1 (nov 2021), 1–6. https://doi.org/10.1007/s11721-021-00202-9

[2] Jaume Bacardit, Alexander E. I. Brownlee, Stefano Cagnoni, Giovanni Iacca, John McCall, and David Walker. 2022. The intersection of evolutionary computation and explainable AI. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM. https://doi.org/10.1145/3520304.3533974

[3] Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel Lopez-Ibanez, Katherine M. Malan, Jason H. Moore, Boris Naujoks, Patryk Orzechowski, Vanessa Volz, Markus Wagner, and Thomas Weise. 2020. Benchmarking in Optimization: Best Practice and Open Issues. (July 2020). arXiv:2007.03488 [cs.NE]

[4] Thomas Bartz-Beielstein, Martin Zaefferer, and Frederik Rehbach. 2017. In a Nutshell – The Sequential Parameter Optimization Toolbox. (Dec. 2017). arXiv:1712.04076 [cs.MS]

[5] Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). RFC 8949. https://doi.org/10.17487/RFC8949

[6] Sébastien Cahon, El-Ghazali Talbi, and Nordine Melab. 2003. ParadisEO: a framework for parallel and distributed biologically inspired heuristics. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE Comput. Soc. https://doi.org/10.1109/ipdps.2003.1213274

[7] Christian L. Camacho-Villalon, Marco Dorigo, and Thomas Stutzle. 2022. PSO-X: A Component-Based Framework for the Automatic Design of Particle Swarm Optimization Algorithms. *IEEE Transactions on Evolutionary Computation* 26, 3 (jun 2022), 402–416. https://doi.org/10.1109/tevc.2021.3102863

[8] Felipe Campelo and Claus Aranha. 2022. The Evolutionary Computation Bestiary. https://github.com/fcampelo/EC-Bestiary. https://github.com/fcampelo/EC-Bestiary

[9] 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/10.1109/tevc.2011.2170075

[10] Jacob de Nobel, Furong Ye, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. 2021. IOHexperimenter: Benchmarking Platform for Iterative Optimization Heuristics. (Nov. 2021). arXiv:2111.04077 [cs.NE]

[11] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. 2018. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. (Oct. 2018). arXiv:1810.05281 [cs.NE]

[12] 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. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2021). ACM. https://doi.org/10.1145/3449726.3463276

[13] 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*. IEEE. https://doi.org/10.1109/cec.2010.5586354

[14] Tome Eftimov, Gašper Petelin, and Peter Korošec. 2020. DSCTool: A web-service-based framework for statistical comparison of stochastic optimization algorithms. *Applied Soft Computing* 87 (feb 2020), 105977. https://doi.org/10.1016/j.asoc.2019.105977

[15] 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 (jul 2012), 2171–2175.

[16] Nikolaus Hansen, Anne Auger, Steffen Finck, and Raymond Ros. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Experimental Setup*. Technical Report. INRIA.

[17] 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*. https://doi.org/10.5281/zenodo.2594848

[18] Nikolay Ivanov. 2022. Is Rust C++-Fast? Benchmarking System Languages on Everyday Routines. https://doi.org/10.48550/arXiv.2209.09127 arXiv:arXiv:2209.09127

[19] Krzysztof Krawiec, Christopher Simons, Jerry Swan, and John Woodward. 2018. Metaheuristic Design Patterns. In *Handbook of Research on Emergent Applications of Optimization Algorithms*. IGI Global, 1–36. https://doi.org/10.4018/978-1-5225-2990-3.ch001

[20] 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*. Springer Berlin Heidelberg, 87–117. https://doi.org/10.1007/978-3-642-11218-8_5

[21] Fu Xing Long, Diederick Vermetten, Bas van Stein, and Anna V. Kononova. 2022. BBOB Instance Analysis: Landscape Properties and Algorithm Performance across Problem Instances. (Nov. 2022). https://doi.org/10.48550/ARXIV.2211.16318 arXiv:2211.16318 [cs.NE]

[22] Manuel López-Ibáñez, Jürgen Branke, and Luís Paquete. 2021. Reproducibility in Evolutionary Computation. *ACM Transactions on Evolutionary Learning and Optimization* 1, 4 (dec 2021), 1–21. https://doi.org/10.1145/3466624

[23] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. 2011. Opt4J. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*. ACM Press. https://doi.org/10.1145/2001576.2001808

[24] Sean Luke. 2017. ECJ then and now. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '17*. ACM Press. https://doi.org/10.1145/3067695.3082467

[25] Eric Medvet, Giorgia Nadizar, and Luca Manzoni. 2022. JGEA. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM. https://doi.org/10.1145/3520304.3533960

[26] 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*. ACM. https://doi.org/10.1145/2739482.2768462

[27] Ana Nikolikj, Risto Trajanov, Gjorgjina Cenikj, Peter Korosec, and Tome Eftimov. 2022. Identifying minimal set of Exploratory Landscape Analysis features for reliable algorithm performance prediction. In *2022 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. https://doi.org/10.1109/cec55065.2022.9870439

[28] Gary Pampara and A. P. Engelbrecht. 2015. Towards A Generic Computational Intelligence Library: Preventing Insanity. In *2015 IEEE Symposium Series on Computational Intelligence*. IEEE. https://doi.org/10.1109/ssci.2015.207

[29] José Antonio Parejo, Jesús Racero, Fernando Guerrero, Terence Kwok, and K. A. Smith. 2003. FOM: A Framework for Metaheuristic Optimization. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 886–895. https://doi.org/10.1007/3-540-44864-0_91

[30] José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. 2011. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing* 16, 3 (aug 2011), 527–561. https://doi.org/10.1007/s00500-011-0754-8

[31] Jérémy Rapin and Olivier Teytaud. 2018. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad.

[32] Eric O. Scott and Sean Luke. 2019. ECJ at 20. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM. https://doi.org/10.1145/3319619.3326865

[33] Jerry Swan, Steven Adriaensen, Alexander E.I. Brownlee, Kevin Hammond, Colin G. Johnson, Ahmed Kheiri, Faustyna Krawiec, J.J. Merelo, Leandro L. Minku, Ender Özcan, Gisele L. Pappa, Pablo García-Sánchez, Kenneth Sörensen, Stefan Voß, Markus Wagner, and David R. White. 2022. Metaheuristics "In the Large". *European Journal of Operational Research* 297, 2 (2022), 393–406. https://doi.org/10.1016/j.ejor.2021.05.042

[34] Kenneth Sörensen. 2013. Metaheuristics-the metaphor exposed. *International Transactions in Operational Research* 22, 1 (feb 2013), 3–18. https://doi.org/10.1111/itor.12001

[35] Diederick Vermetten, Bas van Stein, Fabio Caraffini, Leandro Minku, and Anna V. Kononova. 2021. BIAS: A Toolbox for Benchmarking Structural Bias in the Continuous Domain. (sep 2021). https://doi.org/10.36227/techrxiv.16594880.v1 Preprint.

[36] 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

[37] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller. 2014. Architecture and Design of the HeuristicLab Optimization Environment. In *Topics in Intelligent Engineering and Informatics*. Springer International Publishing, 197–261. https://doi.org/10.1007/978-3-319-01436-4_10

[38] 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 (mar 2022), 1–29. https://doi.org/10.1145/3510426

[39] John R. Woodward and Jerry Swan. 2014. Template method hyper-heuristics. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM. https://doi.org/10.1145/2598394.2609843