A Joint Python/C++ Library for Efficient yet Accessible Black-Box and Gray-Box Optimization with GOMEA

Anton Bouter
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Anton.Bouter@cwi.nl

Peter A.N. Bosman
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Peter.Bosman@cwi.nl

ABSTRACT

Exploiting knowledge about the structure of a problem can greatly benefit the efficiency and scalability of an Evolutionary Algorithm (EA). Model-Based EAs (MBEAs) are capable of doing this by explicitly modeling the problem structure. The Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) is among the state-of-the-art of MBEAs due to its use of a linkage model and the optimal mixing variation operator. Especially in a Gray-Box Optimization (GBO) setting that allows for partial evaluations, i.e., the relatively efficient evaluation of a partial modification of a solution, GOMEA is known to excel. Such GBO settings are known to exist in various real-world applications to which GOMEA has successfully been applied. In this work, we introduce the `GOMEA` library, making existing GOMEA code in C++ accessible through Python, which serves as a centralized way of maintaining and distributing code of GOMEA for various optimization domains. Moreover, it allows for the straightforward definition of BBO as well as GBO fitness functions within Python, which are called from the C++ optimization code for each required (partial) evaluation. We describe the structure of the `GOMEA` library and how it can be used, and we show its performance in both GBO and Black-Box Optimization (BBO).

CCS CONCEPTS

• **Mathematics of computing** → **Evolutionary algorithms**.

KEYWORDS

Keywords

ACM Reference Format:
Anton Bouter and Peter A.N. Bosman. 2023. A Joint Python/C++ Library for Efficient yet Accessible Black-Box and Gray-Box Optimization with GOMEA. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583133.3596361

1 INTRODUCTION

For many difficult optimization problems it is known that exploiting the problem structure is essential for an Evolutionary Algorithm (EA) to achieve good performance and scalability. A class of EAs that is known to be capable of capturing and exploiting problem structure is that of Model-Based Evolutionary Algorithms (MBEAs) [8, 10]. An algorithm that is among the state of the art among MBEAs, and in the field of Evolutionary Computation (EC) in general, is the Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) [4, 13, 22], which explicitly models the problem structure of a problem using a linkage model, and exploits this model when applying variation using the Gene-pool Optimal Mixing (GOM) variation operator.

In general, GOMEA is capable of learning the structure of the optimization problem during optimization. However, for many problems it is known *a priori* what the rough problem structure is. In such cases, it has been shown that this knowledge can greatly benefit the efficiency of EAs, both for benchmarks problems [4, 9, 23] as well as various real-world applications within the medical field [5], engineering [12], and vehicle routing [27]. Such a setting where a limited amount of domain knowledge is available and can be used by the optimization algorithm is called a Gray-Box Optimization (GBO) setting, in contrast to a Black-Box Optimization (BBO) setting where no domain knowledge is available. In particular, we consider the GBO setting where it is known how the fitness function is constructed from a number of subfunctions, and partial evaluations are possible. Such partial evaluations are relatively efficient function evaluations that are used to update the fitness of a solution after a (small) subset of its variables have been modified. Especially for the GBO setting that allows for partial evaluations, GOMEA has shown to achieve excellent performance and scalability [4].

In this work, we introduce the `GOMEA` library, which is a Python library that wraps optimization code of GOMEA written in C++. This C++ code is based on the original code provided by the authors of the most recent relevant publications of GOMEA within the respective domains [4, 13]. The `GOMEA` library serves the purpose of being a centralized way of distributing and maintaining the code of GOMEA, and it makes it easier to install and run GOMEA on user-specific problems. This is firstly the case because this library can be used within Python, one of the currently most commonly used programming languages, and secondly because it can be straightforwardly installed through the Python package installer `pip`, i.e., by running `pip install gomea`.

Furthermore, the `GOMEA` library supports the user to implement both BBO and GBO problems within Python, which are called from the C++ optimization code whenever a (partial) evaluation is required. The implementation of a GBO function within the `GOMEA` library is more straightforward than before, as it only requires the user to define the input and output of each subfunction, rather than knowing the inner workings of the GOMEA code. Further customization is possible for the implementation of more advanced

GECCO '23 Companion, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0120-7/23/07.
https://doi.org/10.1145/3583133.3596361

GBO functions, but can be omitted for relatively simple GBO problems. For the integration between Python and C++ code, Cython [1] is used, which is also used in many commonly used Python packages, including NumPy [16], SciPy [25], and Pandas [26].

Note that this paper describes the state of the GOMEA library version 1.0, and details are subject to change. The source code of the GOMEA library is publicly available on GitHub[1].

In the remainder of this paper, essential background work, including that required to understand and implement a GBO problem within the GOMEA library, is discussed in Section 2. In Section 3, the architecture and its implementation are described. The main features of the GOMEA library, and how they are used, are then described in Section 4. To show the general performance of the GOMEA library in both a BBO and a GBO setting, experimental results are shown in Section 5. Finally, future work is outlined and conclusions are drawn in Sections 6 and 7, respectively.

2 BACKGROUND

We consider optimization problems where the variables to optimize are denoted $\boldsymbol{X} = [X_0, X_1, \ldots, X_{\ell-1}]$. The problem variables are indexed through $\mathcal{I} = [0, 1, \ldots, \ell-1]$, and a realization of the problem variables is denoted $\boldsymbol{x} = \{x_0, x_1, \ldots, x_{\ell-1}\}$. Note that zero-based indexing is used here to be consistent with implementation details to be discussed later.

We consider both the domains of discrete optimization ($\boldsymbol{x} \in \mathbb{Z}^\ell$) and real-valued optimization ($\boldsymbol{x} \in \mathbb{R}^\ell$). The objective, or fitness, function f is either subject to minimization in real-valued optimization, or maximization in discrete optimization, corresponding to the respective conventions in these fields.

In this work, we only consider the GBO setting that allows for partial evaluations. Such a setting is formally defined in Section 2.1.

2.1 Gray-Box Optimization

As previously defined in [3], an objective function in a GBO setting can be written as:

$$f(\boldsymbol{x}) = g\left(f_0(\boldsymbol{x}_{\mathbb{I}_0}) \oplus f_1(\boldsymbol{x}_{\mathbb{I}_1}) \oplus \cdots \oplus f_{q-1}(\boldsymbol{x}_{\mathbb{I}_{q-1}})\right), \tag{1}$$

$$= g\left(\bigoplus_{i=0}^{q-1} \left(f_i(\boldsymbol{x}_{\mathbb{I}_i})\right)\right) \tag{2}$$

where each $f_i(\boldsymbol{x}_{\mathbb{I}_i})$ a subfunction that depends on a subset of variables of $\boldsymbol{x}$, namely those for which the index u is included in $\mathbb{I}_i \subseteq \mathcal{I}$. The set $\mathbb{I} = \{\mathbb{I}_0, \mathbb{I}_1, \ldots, \mathbb{I}_{q-1}\}$ defines the complete set of dependencies of each subfunction. Furthermore, $\oplus$ is a commutative binary operator with a known inverse $\ominus$ (e.g., addition or multiplication), and $g : \mathbb{R} \to \mathbb{R}$ is any (potentially non-linear) function.

A partial evaluation is a relatively efficient evaluation following the modification of a (small) number of variables of a solution. Consider a solution $\boldsymbol{x}$ in a certain state g, denoted $\boldsymbol{x}^g$, for which the fitness $f(\boldsymbol{x}^g)$ is known. After the modification of some variable x_u, we denote that this solution is in the state $\boldsymbol{x}^{g+1}$. Performing the partial evaluation following the modification of these variables first requires finding all $\mathbb{I}_j$ that contain u, as these indicate the subfunctions dependent on variables x_u or x_v. For each of these dependent subfunctions, the value given state $\boldsymbol{x}^g$ should be subtracted from the fitness, and the value given state $\boldsymbol{x}^{g+1}$ should be added. As such, the fitness of $\boldsymbol{x}^{g+1}$ is computed as follows:

$$f\left(\boldsymbol{x}^{g+1}\right) = g\left(\bigoplus_{i=0}^{q-1} f_i(\boldsymbol{x}^g_{\mathbb{I}_i}) \oplus \bigoplus_{\mathbb{I}_i \ni u} f_i(\boldsymbol{x}^{g+1}_{\mathbb{I}_i}) \ominus \bigoplus_{\mathbb{I}_i \ni u} f_i(\boldsymbol{x}^g_{\mathbb{I}_i})\right), \tag{3}$$

where $\mathbb{I}_i \ni u$ is shorthand for the set $\{\mathbb{I}_i \in \mathbb{I} \mid u \in \mathbb{I}_i\}$. However, because all subfunctions for state $\boldsymbol{x}^g$ have previously already been computed, this computation can be substantially accelerated by storing their sum in memory. Here, this sum is named a fitness buffer and denoted β. This fitness buffer is computed after initialization for each solution in the population, and continuously updated after any modification to the population.

Consider the fitness buffer β of solution $\boldsymbol{x}$. For the initial state of the solution, i.e., $\boldsymbol{x}^0$, the initial state of the buffer, i.e., β^0, is computed as the sum of all subfunctions as follows:

$$\beta^0 = \bigoplus_{i=0}^{q-1} f_i(\boldsymbol{x}^0_{\mathbb{I}_i}). \tag{4}$$

Then, following a modification to variable x_u, all current values of dependent subfunctions are subtracted from the fitness buffer, and new values are computed and added to the fitness buffer:

$$\beta^{g+1} = \beta^g \ominus \bigoplus_{\mathbb{I}_i \ni u} f_i(\boldsymbol{x}^g_{\mathbb{I}_i}) \oplus \bigoplus_{\mathbb{I}_i \ni u} f_i(\boldsymbol{x}^{g+1}_{\mathbb{I}_i}) \tag{5}$$

The update of the fitness buffer is similar when more than one variable is updated. In this case, the update considers the union of subfunctions dependent on any of the modified variables.

Note the similarities between Equations 3 and 5. Therefore, following an update to the fitness buffer β^{g+1}, the fitness of solution $\boldsymbol{x}^{g+1}$ can be computed in constant time (with respect to the problem size ℓ) as follows:

$$f(\boldsymbol{x}^{g+1}) = g(\beta^{g+1}) \tag{6}$$

A GBO function is not required to use only one fitness buffer, but can potentially use an arbitrary number of fitness buffers. As such, it is possible to define GBO functions similar to any of the following signatures:

$$f_{\text{Example1}}(\boldsymbol{x}) = g(\beta_0, \beta_1, \beta_2, \ldots) \tag{7}$$

$$f_{\text{Example2}}(\boldsymbol{x}) = g_0(\beta_0) + g_1(\beta_1) + g_2(\beta_2) + \ldots \tag{8}$$

$$f_{\text{Example3}}(\boldsymbol{x}) = h(g_0(\beta_0, \beta_1) + g_1(\beta_2)) \tag{9}$$

Note that none of these functions directly use problem variables, but only fitness buffers, because their complexity should not scale with the number of problem variables in order to maintain the relative efficiency of the GBO setting.

Due to saving the fitness buffer(s) in memory, the complexity of a partial evaluation scales with the number of dependent subfunctions. Therefore, a partial evaluation that requires the computation of k subfunction is considered to be the fraction k/q of an evaluation. It is assumed that all subfunctions have approximately the same computational complexity.

2.2 GOMEA

The Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) [22] is a Model-Based Evolutionary Algorithm (MBEA) that excels at using domain knowledge of the optimization problem to

[1] https://github.com/abouter/gomea/

improve the performance and scalability of the optimization. This domain knowledge can be learned based on the population during optimization, or, when the problem allows it, be supplied *a priori*.

The dependency structure of a problem is modeled with what is called a linkage model, described in more detail in Section 2.3. This linkage model is then used to guide the Gene-pool Optimal Mixing (GOM) variation operator, which applies variation to only a small number of variables of a parent solution. Furthermore, such a variation step is only accepted when it does not degrade the fitness of the parent. Due to the fact that variation is applied to a small number of variables, it is possible to exploit partial evaluations to greatly improve the efficiency of fitness evaluations.

GOMEA was first introduced for the domain of discrete optimization [22], but has since been extended to the domains of real-valued optimization [4] and genetic programming [24].

2.3 Linkage Models

Linkage models are used by GOMEA to model the dependency structure of the optimization problem. Such linkage models are described by a Family Of Subsets (FOS) $\mathcal{F} = \{\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{k-q}\}$, which is a subset of the powerset of $\mathcal{I}$. Therefore, it follows that $\mathcal{F}_i \subseteq \mathcal{I}$ for each $\mathcal{F}_i \in \mathcal{F}$. Each element $\mathcal{F}_i$, named a linkage set, is a set containing a number of indices of problem variables that are considered to be jointly dependent.

Each variation step with the GOM variation operator considers one parent individual and one linkage set. Variation is then applied to all variables for which the index is included in the respective linkage set. Depending on the domain of the optimization, this variation step can consist of either crossover with a donor solution, or the sampling from a probability distribution.

2.3.1 Marginal Products. A Marginal Product (MP) linkage model consists of any number of non-overlapping linkage sets that together cover all problem indices 1 to ℓ. The simplest such linkage model is the univariate model $\mathcal{F}^{\texttt{Uni}} = \{\{0\}, \{1\}, \ldots, \{\ell - 1\}\}$, which models a completely separable problem. The full linkage model $\mathcal{F}^{\texttt{Full}} = \{\{0, 1, \ldots, \ell - 1\}\}$ contains one linkage set with all problem indices, and models a problem with complete dependency between each pair of variables. Note that this linkage model can only be used in real-valued optimization, due to the nature of variation (i.e., crossover) in the discrete domain.

2.3.2 Linkage Tree. The Linkage Tree (LT) [21] is a hierarchical linkage model describing different levels of dependencies ranging from the single-variable level up to very high-level dependencies. A linkage tree is constructed using the Unweighted Pair Group Method with Arithmetic mean (UPGMA) clustering method [14]. This method is initialized with all single-variable linkage sets, and continuously merges the two (unmerged) linkage sets with the highest similarity until all linkage sets have been merged and it is left with only one linkage set containing all problem variables. Any such merged set is added to the linkage tree, alongside all initial (univariate) linkage sets. Therefore, the linkage tree contains all linkage sets encountered during the UPGMA process, ranging from univariate to the full linkage set. Note that the full linkage set is removed from the linkage model in discrete optimization.

An LT may be learned during optimization based on the population, or learned *a priori* based on a known similarity metric. When it is learned during optimization, possible similarity metrics include mutual information [22] or hamming distance [19]. When the LT is learned based on a known distance metric, it remains constant and is therefore named a static (or fixed) linkage tree.

2.3.3 Conditional. For real-valued optimization, conditional linkage models were previously introduced [7]. These linkage models allow for the application of variation to a certain subset of variables while conditioning on the values of remaining variables of the parent individual. This mainly benefits the optimization on problems with overlapping strong dependencies. The use of a conditional linkage model requires knowing the Variable Interaction Graph (VIG) of the optimization problem, which is only known in a GBO setting. Furthermore, the definition of a conditional linkage model may include a Bayesian factorization to specify which variables are sampled jointly dependent, in which case it is deemed a 'Multivariate Conditional' (MCond) model. Otherwise, the linkage model is deemed a 'Univariate Conditional' (UCond) model.

When variation is done with a specific linkage set, the variables to be sampled are conditioned on all variables connected in the VIG that are not in the respective linkage set. Also sampling from the full probability distribution is done using the conditional factorization modeled by the VIG, using forward sampling [20].

As with non-conditional linkage models, the variables to which variation is applied during GOM is determined by a linkage set. Therefore, when the FOS of the conditional linkage model has only one element consisting of all variables, GOM is only done once per generation. Hence, this model performs 'Generational GOM' (GG). When the FOS consists of each separate element as specified by the factorization, it is deemed to perform 'Factorized GOM' (FG). Finally, both can be combined to what is called 'Hybrid GOM' (HG), where GG and FG are both applied every generation. As such, a number of previously used conditional linkage models are the UCondFG, UCondGG, UCondHG, and MCondHG models [7].

2.4 Population-sizing scheme

By default, the `GOMEA` library uses an Interleaved Multi-start Scheme (IMS) [15] to avoid tuning the population size, as correctly setting this can have a large impact on the performance of an EA. Using the IMS, multiple populations of different sizes are independently subject to optimization, and run generations in an interleaved fashion. After each $c^{\texttt{IMS}}$ generations of a population with size n, one generation of a population with size $2n$ is performed. This applies to each of the interleaved populations, i.e., once the population with size $2n$ has performed $c^{\texttt{IMS}}$ generations, the population with size $4n$ will run one generation. The initial population in IMS is initialized with the base population size $n^{\texttt{base}}$, which is generally set to the smallest reasonable population size for the respective domain.

3 IMPLEMENTATION

The `GOMEA` library is mostly written in C++, with code required to interface with Python written in Cython. Cython allows for the relatively straightforward interfacing between Python and C++. In particular, for optimization in a GBO setting, this enables a

user to write a custom GBO function in Python without a deeper understanding of the C++ optimization code.

The root package name is `gomea`, and can simply be imported into Python through `import gomea`. This package has a number of subpackages that are structured as follows:

- `gomea`
  - `discrete`
  - `real_valued`
  - `fitness`
  - `linkage`
  - `output`

The `discrete` and `real_valued` subpackages contain code for optimization with the discrete and real-valued variants of GOMEA, respectively. These subpackages also each contain a `Config` class within which all input parameters for the optimization are stored, and which is passed to the C++ optimization code.

The `fitness` subpackage contains Cython classes that can be extended by a user-defined Python class for the implementation of a custom fitness function, with the following hierarchy:

- `FitnessFunction`
  - GBOFitnessFunction
    - GBOFitnessFunctionDiscrete
    - GBOFitnessFunctionRealValued
  - BBOFitnessFunction
    - BBOFitnessFunctionDiscrete
    - BBOFitnessFunctionRealValued

Each of these Cython classes has a member variable that is a pointer to a C++ class which mirrors the Cython class. This C++ class is instantiated during the initialization of the Cython class. A pointer to the Cython class is passed to the constructor of the C++ class (with type `PyObject*`), and stored as a member variable. When a (partial) evaluation is required during the optimization, a public Cython function (implemented in `EmbeddedFitness.pxi`) is called from the C++ class, and the pointer to the Cython class is passed as an argument. Other arguments may include, e.g., a vector of problem variables and the index of a subfunction that is to be evaluated. Within the public Cython function, the `PyObject*` pointer is typecast to one of the aforementioned Cython classes, such that its user-defined methods (overloading the default of the Cython class), e.g., the evaluation of a subfunction, can be called.

To pass the problem variables, they must be converted from a C++-type vector to a type that is interpretable by Python, for which we use the `ndarray` type included in NumPy. This `ndarray` is initialized by creating an array wrapper around the given pointer without copying the data pointed to, which is essential to maintain the performance and scalability of GOMEA in a GBO setting, as the complexity of a partial evaluation would otherwise no longer be constant, but scale in the same way as a full evaluation.

The `linkage` subpackage includes Cython classes that can be instantiated by the user (in Python), and passed as a parameter to a `GOMEA` optimization class, indicating what type of linkage model is to be used. Each Cython class in the `linkage` subpackage wraps a pointer to an instance of the C++ class `linkage_config_t`, to which all necessary parameters are passed during its instantiation. During the construction of a `linkage_config_t` instance, no linkage model is yet built, but all necessary parameters are contained in this class such that a linkage model can be constructed when required by the optimization.

Finally, the `output` subpackage contains a Python class responsible for wrapping output statistics, named `OutputStatistics`, and a Cython wrapper for this class. This Cython class wraps a pointer to an instance of a C++ class that is used to store all output statistics, and is returned by the C++ code at the end of an optimization run. All data within this instance is copied to a member variable of the Python class `OutputStatistics`, as there is no guarantee of the lifetime of the pointer within the C++ class. In fact, all this data is erased when the instance of the EA is used to perform another run, in which case it is undesired to lose previous output.

4 FEATURES

At the time of publication, the `GOMEA` library supports optimization with the single-objective versions of the discrete GOMEA [13, 22], and the Real-Valued GOMEA (RV-GOMEA) [4]. The (C++) source code used for these algorithms was supplied by the original authors and adapted for unification purposes and integration with the Python API.

One of the primary features of the `GOMEA` library is its compatibility with user-defined GBO functions written in Python. This feature is elaborated on in Section 4.1. This section includes guidelines on how to implement such functions, and gives examples of how to implement well-known optimization functions as a GBO function.

Most linkage models, as discussed in Section 2.3, previously used in literature are available in the `GOMEA` library. This includes the filtered linkage tree [2] for discrete optimization, and conditional linkage models for real-valued optimization [7]. The use of different linkage models in the `GOMEA` library is discussed in Section 4.3. Remaining input parameters of both versions of GOMEA are specified in Section 4.4, and the output is discussed in Section 4.5.

4.1 Custom Gray-Box Optimization Function

The implementation of a custom GBO function according to the definition in Equation 1 requires the user to define a Python class that extends one of the following classes included in the `gomea.fitness` subpackage:

- `GBOFitnessFunctionDiscrete`
- `GBOFitnessFunctionRealValued`

depending on whether the domain of the optimization problem is discrete or real-valued, respectively. Each of these classes extends base class `FitnessFunction`, which is also present in the `gomea.fitness` subpackage. Note that discrete optimization functions are subject to maximization and real-valued optimization functions are subject to minimization, corresponding to the conventions within these respective fields.

Such a Python class requires the user to override at least the following methods:

```
def number_of_subfunctions(self) -> int
def inputs_to_subfunction(self, subfunction_index) -> np.ndarray
def subfunction(self, subfunction_index, variables) -> float
```

The method `number_of_subfunctions` returns the total number of subfunctions, corresponding to q in Equation 1. The method `inputs_to_subfunction` returns an array indicating which variables are input for the subfunction with index `subfunction_index`.

This corresponds to $\mathbb{I}_i$ in Equation 1 when `subfunction_index` is equal to i. Finally, the method `subfunction` returns the output of the subfunction with index `subfunction_index`, corresponding to $f_i(\boldsymbol{x}_{\mathbb{I}_i})$ in Equation 1 when `subfunction_index` is equal to i. Note that `variables` is an array containing all problem variables x_1 through x_ℓ and should be indexed as such. However, only variables contained in $\mathbb{I}_i$ should be actively used for the calculation of f_i. If any variable x_u is used for the calculation of f_i when x_u is not contained in $\mathbb{I}_i$, any modification of x_u will not trigger the calculation of f_i, leading to inconsistency in the objective value.

An example of the implementation of the concatenated trap function is shown in Code Block 1. Here, the `__new__` method is overridden to assign the trap size `k` as a member variable of the class, and to assert that the number of variables is a multiple of the trap size. Furthermore, the method `inputs_to_subfunction` is defined such that it returns the range $[ki, \ldots, k+ki]$ given the subfunction index i as input, as this range defines the indices of the problem variables used by subfunction f_i, i.e., the i^{th} trap function. Within the method `subfunction`, these variables are then retrieved and stored into `trap_vars`, after which they are summed using numpy. Finally, the fitness contribution of the subfunction f_i is then calculated given the calculated `unitation`, and returned.

Code block 1 Concatenated trap function

```
import gomea
import numpy as np
class ConcatTrapGBO(gomea.fitness.GBOFitnessFunctionDiscrete):
    def __new__(self, number_of_variables, k):
        assert( number_of_variables % k == 0 )
        self.k = k  # Trap size
        return super().__new__(self,number_of_variables)

    def number_of_subfunctions(self) -> int:
        return self.number_of_variables // self.k

    def inputs_to_subfunction(self, subf_index) -> np.ndarray:
        return range(self.k*subf_index,self.k*subf_index+self.k)

    def subfunction(self, subf_index, variables) -> float:
        trap_vars =
        ↪  variables[self.inputs_to_subfunction(subf_index)]
        unitation = np.sum(trap_vars)
        if unitation == self.k:
            return unitation
        else:
            return self.k - unitation - 1
```

In the concatenated trap function, the function g as defined in Equation 1 is simply the identity function. To implement an optimization function for which g is not simply the identity function, this must be implemented by overriding the method:

```
def objective_function(self, obj_index, fitness_buffers) -> float
```

This is the method corresponding to g in Equation 1 which computes the fitness of an individual given an array of fitness buffer values. By default, `fitness_buffers` is an array containing the sum of all subfunctions at index 0. Note that the parameter `obj_index` is present for future compatibility with multi-objective optimization, but can remain unused in single-objective optimization.

By overriding the method `objective_function`, it can be defined as any function of the fitness buffers of an individual. This function does not have access to the variables of the respective solution, because this is required to be done within the implementation of `subfunction`. Excessive access of the variables within the `objective_function` can negate all benefits of a GBO setting and have a substantial negative impact on performance.

In order to use multiple fitness buffers, similar to the functions shown in Equation 7, it is necessary also override the following methods, in addition to those listed at the start of Section 4.1:

```
def number_of_fitness_buffers(self) -> int
def fitness_buffer_index_for_subfunction(self, subf_index) -> int
```

The method `number_of_fitness_buffers` specifies the total number of fitness buffers. The index of the fitness buffer to which the result of a subfunction (with index `subf_index`) needs to be added is specified by `fitness_buffer_index_for_subfunction`.

The definition of a constraint function is possible in a similar way to that of an objective function, by overloading the method:

```
def constraint_function(self, fitness_buffers) -> float
```

When this method is not overloaded, it returns 0 by default, meaning that every possible solution is feasible.

Though the most straightforward way of implementing a GBO function is in Python, it is possible to implement it in C++ for better performance. For this, it is currently recommended to implement the custom GBO function into the class `YourFitnessFunction` (the `Discrete` or `RealValued` variant) and compiling from source.

4.2 Custom Black-Box Optimization Function

Similar to the definition of a GBO fitness function as discussed in Section 4.1, also for the definition of a BBO function a class needs to be defined by the user, which in this case is required to extend one of the following classes in the `gomea.fitness` subpackage:

- `BBOFitnessFunctionDiscrete`
- `BBOFitnessFunctionRealValued`

The choice among these classes depends on whether the domain of the optimization problem is discrete or real-valued, respectively. Each of these classes extends base class `FitnessFunction`, which is also present in the `gomea.fitness` subpackage.

This user defined class is then only required to implement the following method, which returns the objective value of the solution defined by the input `variables`:

```
def objective_function(self, objective_index, variables) -> float
```

Note that `objective_index` is unused for single-objective optimization, but is required for future compatibility with multi-objective optimization.

4.3 Linkage Models

All linkage models are implemented in the `gomea.linkage` subpackage. The linkage models available for both real-valued and discrete optimization are the following:

- `Univariate()`
- `BlockMarginalProduct(block_size)`
- `LinkageTree(sim_measure,filtered,max_set_size)`
- `StaticLinkageTree(max_set_size)`
- `Custom(file)`
- `Custom(fos)`

Additionally, the following linkage models are only available for real-valued optimization:

- `Full()`
- `Conditional(max_clique_size,inc_cliques,inc_full)`
  - `UCondGG()`
  - `UCondFG()`
  - `UCondHG()`
  - `MCondHG(max_clique_size)`

Each of these linkage models extend the base class `LinkageModel`. The above linkage models and their parameters are discussed in this section. The details of these linkage models are discussed in the following sections.

4.3.1 Univariate. This linkage model uses a FOS with ℓ univariate elements, and requires no input parameters.

4.3.2 Block Marginal Product. This linkage model uses a FOS where each element consists of `block_size` consecutive variables starting from 0 up to $\ell - 1$. To use a marginal product FOS that does not adhere to this structure, it is advised to use the `Custom` linkage model.

4.3.3 Linkage Tree. The linkage tree model expects the parameters `sim_measure` `filtered`, and `max_set_size`. The parameter `sim_measure` is a string from one of the possible options `'MI'` or `'NMI'`, indicating whether Mutual Information (MI) or Normalized Mutual Information (NMI) should be used as similarity metric. If `filtered` is set to true, superfluous linkage sets are filtered [2]. Finally, if `max_set_size` is set to any number larger than 0, no linkage sets larger than this number are formed during the linkage tree construction.

The `StaticLinkageTree(max_set_size)` is a linkage tree that is constant throughout optimization, and accepts only the parameter `max_set_size`, identical to the non-static linkage tree. By default, this linkage tree uses connectivity within the VIG as a similarity measure. Therefore, it cannot be used in a BBO setting. With this linkage model, variables between which no path (of any length) exists in the VIG will never occur in the same linkage set.

A custom similarity measure can be used instead by overriding the following method of the custom fitness function that overrides a `GBOFitnessFunction` class, as discussed in Section 4.1:

```
def similarity_measure(self,var_a,var_b) -> float
```

This method requires as input the indices of two variables, `var_a` and `var_b`, and returns a similarity measure of these variables, with higher values indicating that these variables will be merged sooner in the construction process of the linkage tree. Note that this method is expected to be symmetric, i.e., the same output is expected when the values for `var_a` and `var_b` are swapped.

4.3.4 Custom. A custom FOS requires exactly one named parameter: either `file` as a string, or `fos` as a vector of integer vectors. The input file will have one linkage set per line, with each of its elements separated by commas or spaces. Each element is required to be within the range $[0, \ell - 1]$.

4.3.5 Full. The full linkage model contains one linkage set containing all problem variables, and requires no parameters. This linkage model can only be used for real-valued optimization.

4.3.6 Conditional. Conditional linkage models are implemented in `Conditional(max_clique_size,inc_cliques,inc_full)`, where `max_clique_size` determines the maximum size of factors within the Bayesian factorization, as specified in Section 2.3. These factors are constructed by finding all maximal cliques up to the specified `max_clique_size`. A `max_clique_size` equal to 1 is used by each UCond linkage model. The boolean parameter `inc_cliques` specifies whether each of these cliques should be included in the FOS, and the boolean parameter `inc_full` specifies whether the full FOS element should be included. As such, setting only the former to true means using FG, while only setting the latter to true means using GG. Setting both to true means using HG.

4.4 Optimization

The subpackages for discrete and real-valued optimization with GOMEA are named `discrete` and `real_valued`, respectively. An instance of either one of these algorithms can be instantiated by calling `gomea.DiscreteGOMEA` or `gomea.RealValuedGOMEA`. These algorithms have some domain-specific input parameters that are discussed in Sections 4.4.1 and 4.4.2. The input parameters they have in common, with potentially different default values for the Discrete (D) and Real-Valued (RV) domains, are as follows:

- `fitness` (required)
  - Any class with base class `FitnessFunction` from subpackage `gomea.fitness`.
- `linkage_model` (default: `StaticLinkageTree()`)
  - Any class with base class `LinkageModel` from subpackage `gomea.linkage`.
- `max_number_of_populations` (default: `25`)
  - Maximum number of interleaved populations within IMS. Set to 1 to disable IMS.
- `base_population_size` (default: `2` (D) / `10` (RV))
  - Population size of the initial population in IMS. Acts as the population size when IMS is disabled.
- `IMS_subgeneration_factor` (default: `4` (D) / `8` (RV))
  - Number of generations that each interleaved population performs per generation of the next largest population (with double the population size).
- `max_number_of_generations` (default: `-1` (No limit))
  - Maximum number of generations that each interleaved population will perform before terminating.
- `max_number_of_evaluations` (default: `-1` (No limit))
- `max_number_of_seconds` (default: `-1` (No limit))
- `random_seed` (default: `-1` (Randomly generated))

After instantiation of any such algorithm class (i.e., either the `DiscreteGOMEA` or `RealValuedGOMEA` class), optimization can be started by calling the `run` method, which requires no input parameters, and returns an instance of the `OutputStatistics` class. The details of this class, containing the results of the optimization throughout each generation, are discussed in Section 4.5.

4.4.1 Discrete. The discrete GOMEA currently has no domain-specific input parameters.

4.4.2 Real-Valued. The real-valued GOMEA has the following two domain-specific input parameters:

- `lower_init_range` (default: `0`)
- `upper_init_range` (default: `1`)

These two input parameters define range between which each variable is initialized uniformly at random.

4.5 Output

The output of any run with an instance of GOMEA returns an instance of the `OutputStatistics` class, which is included in the `gomea.output` subpackage. This class has a dictionary as member variable, named `metrics_dict`, which is accessible using the `[]` operator, and contains lists of statistics for various metrics, where the key of the dictionary is the name of the metric, and the value is the list of data points for the metric. When IMS is enabled, for each metric, a data point is appended at the end of every 10 generations of each (sub)population, as to not give an abundance of output. When IMS is not enabled, for each metric, a data point is appended at the end of each generation of the EA. As such, data points with the same index (of different metrics) correspond to the same state/point in time of the EA By default, the following metrics are recorded, including the key used to retrieve the data from the dictionary:

- Number of generations (key: `generation`)
- Number of evaluations (key: `evaluations`)
- Elapsed time (seconds) (key: `time`)
- Elapsed evaluation time (seconds) (key: `eval_time`)
- Population index (key: `population_index`)
- Population size (key: `population_size`)
- Best objective value (key: `best_obj_val`)
- Best constraint value (key: `best_cons_val`)

The full list of metrics is accessible as the property `metrics` of the `OutputStatistics` class.

In Code block 2 example Python code is shown of how an arbitrary convergence plot can be made given the output of a run with `RealValuedGOMEA`.

Code block 2 Example of plotting the output of a run.

```
import gomea
import matplotlib.pyplot as plt
frv = gomea.fitness.RosenbrockFunction(20,value_to_reach=1e-10)
lm = gomea.linkage.Univariate()
rvgom = gomea.RealValuedGOMEA(fitness=frv, linkage_model=lm)
result = rvgom.run()
plt.plot(result['evaluations'],result['best_obj_val'])
```

5 EXPERIMENTS

This section describes the performance and scalability of the GOMEA library on a number of typical benchmark problems. Furthermore, it is shown what the benefit is of implementing an objective function in a GBO setting compared to a BBO setting. For reproducability, code to repeat all experiments described in this section is provided in the repository of the GOMEA library.

5.1 Set-up

All experiments are performed on a server running Fedora 36 with 20 Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz and 126 GB RAM. Each run of an EA used only a single core. Default parameters of the GOMEA library are used unless specified otherwise. All plotted data points show the median and interdecile range of 30 independent successful runs. A run is considered successful if, for discrete problems, the optimum was found, or, for real-valued problems, the value to reach of 10^{-10} was found. A time limit of one hour was used, and the budget of function evaluations was set to 10^7 for discrete problems and 10^8 for the real-valued problem. No experiments were performed beyond the displayed range of dimensionality.

5.2 Benchmark functions

We use a number of well-known benchmark functions from the domains of discrete and real-valued optimization. This includes the concatenated deceptive trap function [11], the MaxCut problem [17], and the Rosenbrock function.

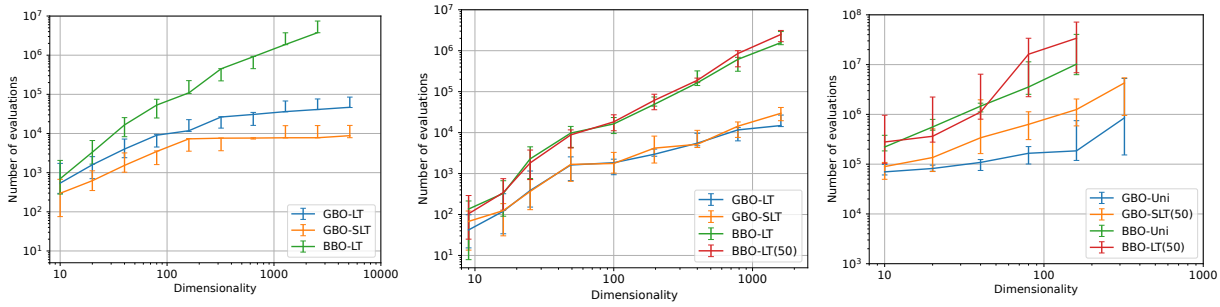
These benchmark problems are selected as they are from different optimization domains and/or exhibit different dependency structures. The concatenated deceptive trap function is a well-known discrete optimization problem with strong dependencies within small disjoint subsets of variables. We use a trap size of 5 for all experiments. The MaxCut problem is also a discrete optimization problem, where each vertex of a given graph is assigned to a set or its complement, and the weight of edges between vertices in opposing sets is to be maximized. We specifically use unweighted graphs with the structure of a square grid with wrap-around, i.e., a torus, leading to a non-separable optimization problem and each variable having a constant number of dependent variables. Finally, the Rosenbrock function is a real-valued non-separable problem with overlapping dependencies.

5.3 Results

Figure 1 shows the scalability of the GOMEA library on the respective benchmark problems in terms of the number of evaluations. Though the scalability of GOMEA on benchmark functions was previously already shown to be excellent in a GBO setting [4], we here confirm these findings for the GOMEA library. These results are mainly relevant for optimization problems where the overwhelming majority of computation time is spent on function evaluations. For these kinds of problems, the reduction of the number of function evaluations by an order of magnitude would also reduce the total computation time by approximately one order of magnitude.

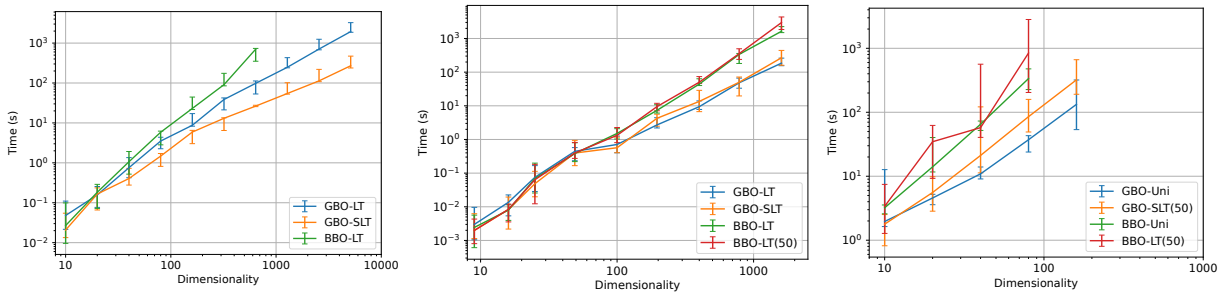
In Figure 2, we show the scalability of the GOMEA library in terms of computation time, for different linkage models in both a BBO and a GBO setting. All problems shown are implemented in Python. These results clearly show the benefit of using a GBO setting, mainly for large-scale optimization problems. Moreover, the use of a GBO settings enables the use of an SLT rather than a (dynamic) LT, possibly providing additional benefit. Finally, bounding the maximum size of linkage sets can offer a benefit, but shows no benefit on the MaxCut problem.

Though the most straightforward way of implementing an optimization function for the GOMEA library is using Python, it is also possible to implement such a function within the C++ code of the GOMEA library, as this leads to an increase in performance. In Figure 3 we show the difference in evaluation time between programming languages for BBO and GBO settings, for one linkage model per problem, to give an indication of the impact of these different implementations. Only the time spent within the evaluation function is shown, as this is the only part of the executed code that is different.



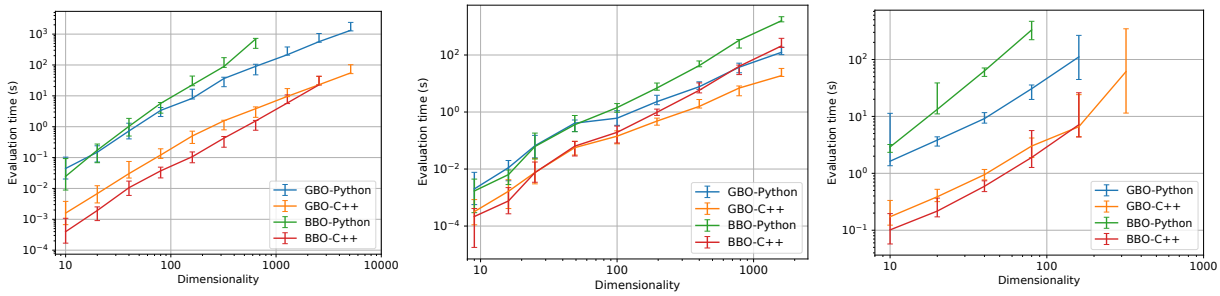
(a) Concatenated Deceptive Trap (b) MaxCut (c) Rosenbrock

Figure 1: Scalability plots for the number of function evaluations required for different linkage models in a BBO or GBO setting. Numbers within parentheses indicate upper bound for the linkage set size.



(a) Concatenated Deceptive Trap (b) MaxCut (c) Rosenbrock

Figure 2: Scalability plots for the computation time required for benchmark functions implemented in Python for different linkage models in a BBO or GBO setting. Numbers within parentheses indicate upper bound for the linkage set size.



(a) Concatenated Deceptive Trap (LT) (b) MaxCut (LT) (c) Rosenbrock (Uni)

Figure 3: Computation time spent within the evaluation function for implementations in different languages for BBO or GBO settings. Used linkage models are indicated in parentheses.

6 FUTURE WORK

One of the major points of future work includes the inclusion of multi-objective variants of GOMEA, for both discrete [18] and real-valued [6] optimization, as well as the inclusing of GOMEA for the domain of genetic programming (GP) [24]. Furthermore, future work could include further customization of input parameters and output statistics of the `GOMEA` library.

7 CONCLUSION

In this paper, we introduced the `GOMEA` library, a Python library around C++ optimization code of the state-of-the-art MBEA GOMEA. This library makes it easier for users to run GOMEA on their own user-specific problems, as the `GOMEA` library can be easily installed, and optimization functions for both BBO and GBO can be implemented in Python. In this paper, the initial state of the `GOMEA` library was described, its structure, and how it can be used for optimization. In our experimental results, we have shown the performance of the `GOMEA` library on various BBO and GBO benchmark problems using different linkage models. Furthermore, the difference in performance is shown between optimization functions implemented in either Python or C++.

With the introduction of the `GOMEA` library, a large hurdle for optimization in a GBO setting has been lifted. As such, this opens the door for users to apply GOMEA to real-world problems of their interest in a GBO setting with a much smaller time commitment.

REFERENCES

[1] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39.

[2] Peter A. N. Bosman and Dirk Thierens. 2013. More concise and robust linkage learning by filtering and combining linkage hierarchies. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 359–366.

[3] Anton Bouter, Tanja Alderliesten, Arjan Bel, Cees Witteveen, and Peter A N Bosman. 2018. Large-scale parallelization of partial evaluations in evolutionary algorithms for real-world problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 1199–1206.

[4] Anton Bouter, Tanja Alderliesten, and Peter A. N. Bosman. 2021. Achieving highly scalable evolutionary real-valued optimization by exploiting partial evaluations. *Evolutionary Computation* 29, 1 (2021), 129–155.

[5] Anton Bouter, Tanja Alderliesten, Bradley R Pieters, Arjan Bel, Yury Niatsetski, and Peter A N Bosman. 2019. GPU-accelerated bi-objective treatment planning for prostate high-dose-rate brachytherapy. *Medical Physics* 46, 9 (2019), 3776–3787.

[6] Anton Bouter, Ngoc Hoang Luong, Cees Witteveen, Tanja Alderliesten, and Peter A. N. Bosman. 2017. The multi-objective real-valued gene-pool optimal mixing evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 537–544. https://doi.org/10.1145/3071178.3071274

[7] Anton Bouter, Stefanus C Maree, Tanja Alderliesten, and Peter A N Bosman. 2020. Leveraging conditional linkage models in gray-box optimization with the real-valued gene-pool optimal mixing evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 603–611.

[8] Larry Bull. 1999. On model-based evolutionary computation. *Soft Computing* 3 (1999), 76–82.

[9] Wenxiang Chen, Darrell Whitley, Renato Tinós, and Francisco Chicano. 2018. Tunneling between plateaus: Improving on a state-of-the-art MAXSAT solver using partition crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 921–928.

[10] Ran Cheng, Cheng He, Yaochu Jin, and Xin Yao. 2018. Model-based evolutionary algorithms: A short survey. *Complex & Intelligent Systems* 4, 4 (2018), 283–292.

[11] Kalyanmoy Deb and David E Goldberg. 1993. Analyzing deception in trap functions. In *Foundations of genetic algorithms*. Vol. 2. Elsevier, 93–108.

[12] Kalyanmoy Deb and Christie Myburgh. 2016. Breaking the billion-variable barrier in real-world optimization using a customized evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 653–660.

[13] Arkadiy Dushatskiy, Marco Virgolin, Anton Bouter, Dirk Thierens, and Peter A N Bosman. 2021. Parameterless Gene-pool Optimal Mixing Evolutionary Algorithms. *arXiv preprint arXiv:2109.05259* (2021).

[14] I. Gronau and S. Moran. 2007. Optimal implementations of UPGMA and other common clustering algorithms. *Information Processing Letters* 104, 6 (2007), 205–210.

[15] G.R. Harik and F.G. Lobo. 1999. A parameter-less genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers Inc., 258–265.

[16] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[17] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Springer, 85–103.

[18] Ngoc Hoang Luong, Han La Poutré, and Peter A. N. Bosman. 2014. Multi-objective gene-pool optimal mixing evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 357–364.

[19] Mitchell Olsthoorn and Annibale Panichella. 2021. Multi-objective test case selection through linkage learning-based crossover. In *Search-Based Software Engineering: 13th International Symposium, SSBSE 2021, Bari, Italy, October 11–12, 2021, Proceedings 13*. Springer, 87–102.

[20] Siddhartha Shakya and Roberto Santana. 2012. *Markov networks in evolutionary computation*. Springer.

[21] Dirk Thierens. 2010. The linkage tree genetic algorithm. In *International Conference on Parallel Problem Solving from Nature*. Springer, 264–273.

[22] Dirk Thierens and Peter A. N. Bosman. 2011. Optimal mixing evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 617–624.

[23] R. Tintos, D. Whitley, and F. Chicano. 2015. Partition crossover for pseudo-boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*. ACM, 137–149.

[24] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter A. N. Bosman. 2017. Scalable genetic programming by gene-pool optimal mixing and input-space entropy-based building-block learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 1041–1048.

[25] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[26] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

[27] Darrell Whitley, Doug Hains, and Adele Howe. 2009. Tunneling between optima: Partition crossover for the traveling salesman problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 915–922.