A C++ Application Programming Interface for Co-Evolutionary Biased Random-Key Genetic Algorithms for Solution and Scenario Generation

Beatriz Brito Oliveira[a], Maria Antónia Carravilla[a], José Fernando Oliveira[a], and Maurício G.C. Resende[b,c]

[a]INESC TEC, Faculty of Engineering, University of Porto;
[b]Amazon.com, Inc; [c]University of Washington

ABSTRACT

This paper presents a C++ application programming interface for a co-evolutionary algorithm for solution and scenario generation in stochastic problems. Based on a two-space biased random-key genetic algorithm, it involves two types of populations that are mutually impacted by the fitness calculations. In the solution population, high-quality solutions evolve, representing first-stage decisions evaluated by their performance in the face of the scenario population. The scenario population ultimately generates a diverse set of scenarios regarding their impact on the solutions.

This application allows the straightforward implementation of this algorithm, where the user needs only to define the problem-dependent decoding procedure and may adjust the risk profile of the decision-maker.

This paper presents the co-evolutionary algorithm and structures the interface. We also present some experiments that validate the impact of relevant features of the application.

KEYWORDS

Genetic algorithm; application programming interface; stochastic programming; scenario generation; co-evolutionary algorithm

1. Introduction

This paper proposes a C++ Application Programming Interface (API) for a co-evolutionary biased random-key genetic algorithm (BRKGA) that simultaneously generates good solutions and a diverse set of scenarios for stochastic problems.

Co-evolutionary genetic algorithms for scenario and solution generation are especially useful in the context of two-stage stochastic problems. In these problems, there are two distinct stages of decision-making: before and after uncertainty is revealed. Scenarios are often used to represent uncertainty as combinations of possible realizations of the uncertain parameters. The decisions taken before uncertainty is realized (first-stage decisions) cannot be changed later. However, in most of these problems, recourse actions (second-stage decisions) can be taken afterwards to mitigate the effect of the realized uncertainty. Therefore, we can model a two-stage stochastic problem with Equations 1, where x represents the first-stage decisions, and the term $\mathscr{Q}(x)$ in the objective function summarizes the value of the second-stage function or recourse function. $\mathscr{Q}(x) = \mathrm{E}_s Q(x, s)$ weights the expected value for all scenarios s of their

CONTACT B. B. Oliveira. Email: beatriz.oliveira@fe.up.pt

impact on the first-stage solutions x, considering (or not) adequate posterior recourse decisions. Stochastic programming has been successfully applied in several contexts, such as supply chain planning [8], shipping [2], or fleet size, mix and assignment [6, 13]. For a more detailed reading on stochastic programs, see [3].

$$\begin{aligned} \min F = c^T x + \mathscr{Q}(x) \\ \text{s.t.} Ax = b \\ x \geq 0 \end{aligned} \tag{1}$$

Co-evolutionary algorithms for solution and scenario generation proposed in [10] simultaneously evolve a population of solutions (first-stage decisions) and a population of scenarios linked by the fitness evaluation. On the one hand, the fitness of solutions will depend on how well they perform in the face of the scenarios in the scenario population. On the other hand, the fitness of scenarios will measure their contribution to the scenario population diversity, considering the impact they have on the solution performance.

Using two types of populations in genetic algorithms to co-evolve solutions and scenarios has been proposed by Herrmann [7] to solve minimax problems and find robust solutions and the worst-case scenario. The innovation of the approach proposed by Oliveira et al. [10] – implemented in this API – is that the goal is to obtain diverse and representative sets of scenarios, regarding the impact they have on the recourse value obtained by solutions. This approach towards uncertainty is more balanced and less conservative than a pure "robustness-oriented" approach (for an interesting paper comparing robust and stochastic models, in the case of dynamic pricing and inventory control, see [1]). As for the solutions, the goal is to obtain good performance. In this approach [10], we can select different criteria to define what "good performance" means. The proposed API also allows the user to define these criteria.

The co-evolutionary algorithm proposed in [10] uses as a base for the evolutionary procedures a biased random-key genetic algorithm (BRKGA) [5]. In BRKGAs the chromosomes, which are vectors of numbers between 0 and 1 (i.e., the genes), represent solutions to the problem and are "translated" and evaluated through a decoding procedure. This procedure also makes it easy to ensure feasibility. BRKGAs initiate with an entirely random initial population of chromosomes that evolve throughout a finite number of generations. The chromosomes are evaluated using a fitness function, and the evolution from the current generation to the following follows three main procedures:

- Copy the elite individuals (with the best fitness) to the following population;
- Add mutant individuals (randomly generated) to the new population;
- Generate as offspring the remaining individuals, by mating an elite and a non-elite parent from the current population. In this cross-over procedure, the offspring can receive each gene from either parent, yet there is a predefined probability higher than 0.5 of inheriting from the elite parent (thus "biasing" the algorithm).

Toso and Resende have proposed a C++ API for BRKGA [14], based on which we built the API proposed in this paper. The decisive difference between the "traditional" BRKGA and the co-evolutionary BRKGA, besides the duplication of the populations, resides on the decoding procedure, which involves both populations and different processes, detailed in this paper. Figure 1 represents the algorithmic flow of

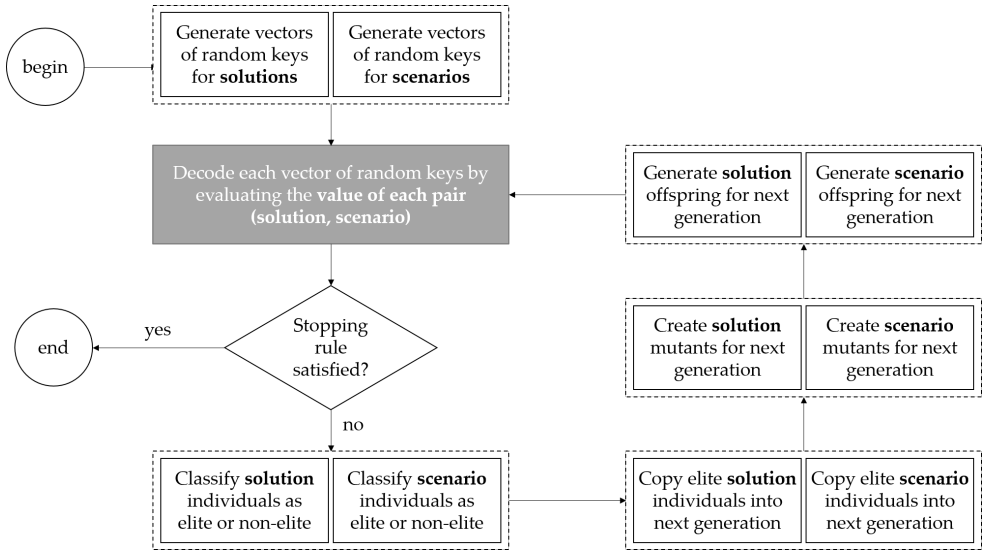


Figure 1. Algorithmic flow of a co-evolutionary BRKGA.

the co-evolutionary BRKGA and is based on a similar design for BRKGA in [14].

The most significant contribution of this paper is that it proposes an efficient and easy-to-use C++ API, as the users are only required to adapt the problem-dependent parts of the decoding procedure. This paper should work not only as a "proof of concept" of the implementation of the co-evolutionary BRKGA proposed in [10] but also as a self-contained guide or tutorial, even to users with limited experience with genetic algorithms and their implementation.

Therefore, this paper is structured as follows. Section 2 describes in detail the structure of the API, highlighting its innovations and different features. Then, in Section 3, we analyze the performance of the API and the impact of key parameters and user-defined processes. Finally, Section 4 presents some concluding remarks.

2. API structure

We propose an application programming interface (API) – `coEvolBrkgaAPI` – built on the `brkgaAPI` [14], which implements the co-evolutionary BRKGA proposed in [10] and described in Section 1. Since this API was built on an existing API for ("traditional") BRKGA algorithms, we will focus on describing the key alterations made. We will especially focus on the usability of the `coEvolBrkgaAPI`, describing the functions and files that users should adapt to implement a co-evolutionary BRKGA for solution and scenario generation, and other relevant and new parameters required.

This section starts by giving an overview of the adaptions required to build the `coEvolBrkgaAPI` from the existing brkgaAPI. Then, the main adaptations are detailed, dividing them into the changes to the existing classes and templates (part of the "black-box"), and the adaptions and new blocks of the interface files, where the user can adapt the problem-dependent part of the algorithm. Finally, we explain the problem exemplified in the `coEvolBrkgaAPI`.

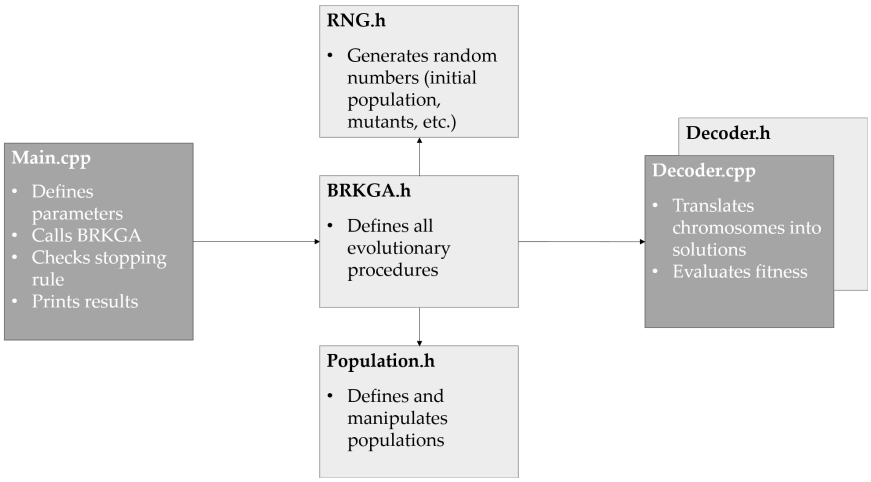


Figure 2. Structure of the `brkgaAPI` files, divided into header files (.h) and source files (.cpp).

2.1. From one to two types of population: overview of the required adaptations

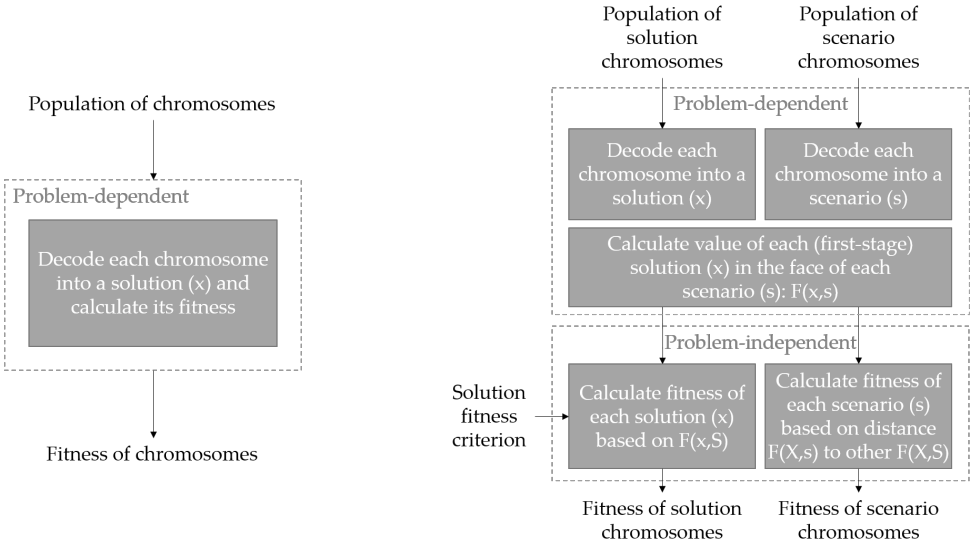
The `coEvolBrkgaAPI` is built on and follows the structure of `brkgaAPI` [14], which was designed to be efficient and straightforward.

Overview on the `brkgaAPI`

The `brkgaAPI` comprises two main classes: `Population`, with methods to define and manipulate populations of chromosomes, and `BRKGA`, where the problem-independent procedures of the genetic algorithm are coded. Besides these classes, there are two external blocks required: a random number generator, and a decoder that computes the fitness of chromosomes. In the `brkgaAPI`, both are implemented as interfaces (`RNG` and `Decoder`). However, as the former is problem-independent, users are not required to adapt this component to implement their algorithm, whereas the latter is the key interface module to implement the specific problem.

Figure 2 represents the file structure of the `brkgaAPI`, which contains six code files. Classes `BRKGA` and `Population` and interface `RNG` are implemented as header files and do not require the user to adapt or implement code. The main algorithmic flow is controlled by the main source file, where the user defines the parameters and the BRKGA procedure is called for each generation if the stopping criteria are not met. The `Decoder` interface is implemented in a source file that users must adapt to each problem, defining how a chromosome translates into a solution and how to evaluate its fitness. This interface also has a corresponding header file where the functions are defined, which the users do not need to adapt. The `brkgaAPI` also allows using multiple populations that evolve independently and that exchange their elite individuals in a predefined interval of generations.

Some parameters required concern the BRKGA procedures, such as the number of mutants to include at each generation, or the probability of inheriting a gene from the elite parent. Nevertheless, some parameters also relate to the algorithmic implementation, namely the number of independent populations, the number of elite elements to exchange among these populations and the interval to generations to perform this exchange, the maximum number of generations (stopping criterion), and the maximum number of threads to use in the parallel computation.



(a) Decoding procedure structure in the traditional BRKGA.

(b) Decoding procedure structure in the co-evolutionary BRKGA.

Figure 3. Differences between decoding procedures of BRKGA and the co-evolutionary BRKGA for solution and scenario generation.

Changes towards the `coEvolBrkgaAPI`

The `coEvolBrkgaAPI` presented in this paper is built on the `brkgaAPI` [14] described above. It uses and adapts its templates, file structure and design strategy since it is simple and efficient. Moreover, the `brkgaAPI` has been extensively used with success in a variety of applications such as routing [12], nesting [9], facility location [4], or minimum spanning trees [11]. As discussed before, the main innovation of the co-evolutionary method is that two types of population evolve in parallel, one where the chromosomes represent (first-stage) solutions, as in the traditional BRKGA, and other where the chromosomes represent scenarios. The evolution of both populations mutually influences each other through the fitness evaluation process. The key innovation in this co-evolutionary algorithm is, thus, in the decoding procedure, represented in Figure 3. In BRKGA, the procedure is linear, as we translate the chromosome into a solution, and then evaluate the fitness (Figure 3a). In the co-evolutionary BRKGA, after the chromosomes are decoded into solutions and scenarios, the resulting impact of each solution in the face of each scenario (problem-dependent) must be computed, and only then the fitness of the individuals of each type of population (problem-independent) is set. The fitness of solutions depends on the criterion selected, e.g., the average result in the face of all scenarios, and the fitness of the scenarios represents their contribution to population diversity, i.e., it is a metric of distance to the other individuals in the scenario population (Figure 3b).

Converting the algorithm from one to two types of population required some adaptations, which we can divide into changes to the background API files (i.e., those that work as a "black-box", as the users do not need to consider them when implementing the algorithm), and the changes to the interface files, which the users will have to adapt. The following sections (Sections 2.2 and 2.3) describe these modules in detail, and Section 2.4 explains the example made available in `coEvolBrkgaAPI`.

2.2. Adaptations to the background API files

In this section, we will describe the changes made to the main "black-box" files in the API: the header files with the classes `Population` and `BRKGA`. Developing towards a co-evolutionary algorithm did not require any adaptation of the external random number generator; therefore, the `coEvolBrkgaAPI` uses the `RNG` class available in the `brkgaAPI`.

`Population` **class:** This class is defined as a comprehensive tool to manage populations in genetic algorithms. Its methods consist of defining a population as a set of vectors of double-precision floating points, with an associated fitness value. It contains some functions that are critical for the algorithm evolution, such as '`void sortFitness()`', which sorts the chromosomes in the population by the fitness value. Other functions allow accessing information about the population, such as '`double getBestFitness() const`', which returns the best fitness value in the population. The methods defined are general and can be applied to populations of solutions or scenarios, since both are composed of chromosomes represented by vectors of numbers between 0 and 1 with an associated fitness value. The only change required was the addition of a function to access all chromosomes in a population, required to compute the resulting value of the solutions in the face of the scenarios (see Section 2.3):

- `const std::vector< std::vector <double> >& getChromosomes() const`: returns a vector of all chromosomes, which are vectors of random-keys, in a read-only reference.

`BRKGA` **class:** This class, as defined in `brkgaAPI`, manages all evolutionary procedures required to implement a multi-population BRKGA, such as evolution (including copying elite individuals, cross-over and mutation) or exchange of top chromosomes between independent populations. This class requires the main user-defined parameters of the algorithm described in Section 2.1, as well as the interfaces `Decoder` and `RNG`. Due to the differences between the two types of population, some of these parameters must be defined for each population independently, and we must consider two types of decoder: a solution decoder (`SoluDecoder`) and a scenario decoder (`ScenDecoder`). Moreover, some new additional parameters are required, such the (optional) theoretical worst and best scenarios. This API also includes three alternative criteria for solution fitness based on the risk profile of the decision-maker: it may consider the worst case scenario (Pessimist criterion), the best case scenario (Optimist criterion), or the average value across all scenarios (Laplace criterion). This will be further explained in Section 2.3. Table 1 explains the parameters required for the `BRKGA` class in the `coEvolBrkgaAPI`.

Similarly, all functions defined in the class were adapted to consider the two types of population. For most, the adaptations were straightforward and consisted of "mimicking" the process for the two types of populations. These functions can be divided into:

- Functions that return copies of the internal parameters – e.g. `unsigned getN() const` was adapted to `unsigned getN_solu() const` and `unsigned getN_scen() const`;
- Functions that return information on the algorithm, such as current population, best fitness, best chromosome – e.g. `double getBestFitness() const` was adapted to `double getBestFitnessSolu() const` and `double getBest-`

Table 1. Parameters required for the adapted `BRKGA` class.

Parameters that were adapted to the two types of population:

`unsigned solu_n`	`unsigned scen_n`	Number of keys in the (solution/scenario) chromosome
`unsigned solu_p`	`unsigned scen_p`	Number of (solution/scenario) chromosomes in the population
`double solu_pe`	`double scen_pe`	Percent of the (solution/scenario) population that composes the elite
`double solu_pm`	`double scen_pm`	Percent of mutants in the (solution/scenario) population
`double solu_rhoe`	`double scen_rhoe`	Probability of inheriting a gene from the elite (solution/scenario) parent in cross-over
`const SoluDecoder& SoluDecoder`	`const ScenDecoder& ScenDecoder`	Reference to the interface class representing the (solution/scenario) decoder

New parameters for the co-evolutionary approach:

`const std::vector<double> extremeworst`	Vector of numbers representing the theoretical worst scenario
`const std::vector<double> extremebest`	Vector of numbers representing the theoretical best scenario
`const int solucriterion`	Criterion for evaluating solution fitness: =1 for Pessimist criterion, =2 for Laplace criterion, =3 for Optimist criterion

Parameters that did not change:

`RNG& RNG`	Reference to the interface class for random number generation
`unsigned K`	Number of independent pairs of (solution, scenario) populations
`unsigned MAX_THREADS`	Number of threads used (maximum) for parallel calculations

`FitnessScen() const`;

- Functions used in the genetic algorithm, such as `reset()` or `evolve()`, where some commands need to be called both for solution and scenario populations.

On the contrary, the adaptation of functions `initialize()` and `evolution()` required some algorithmic relevant changes due to the specificity of the co-evolutionary process. Both functions include fitness computation of populations, where the main changes are visible, as discussed before. Whereas in the original API, the fitness of each chromosome was computed in a single line using the `Decoder` class (see Figure 3a), here, additional steps are required (see Figure 3b). We create two auxiliary structures to support the fitness assignment for both solutions are scenarios:

- Matrix F of size `solu_p`×`scen_p` stores the resulting value of each solution in the face of each scenario. The function to calculate this matrix is problem-dependent and is defined by the user with an interface.
- Matrix D of size `solu_p`×2 stores the distance of each scenario to its nearest neighbours based on its range of impact calculated on F.

Afterwards, the fitness of each chromosome is computed using the corresponding decoder class and the previous structures as inputs. For the solution fitness calculation, the user needs to define the criterion as well. The calculations to compute F, D, and scenario and solution fitness are defined in an interface file. Therefore, we will further

describe and discuss them in the following Section 2.3.

The possibility to include theoretically extreme scenarios in the otherwise entirely random initial generation of scenarios is also an essential part of the co-evolutionary BRKGA. It required a relevant change in function `initialize()`, which generates the initial solution and scenario populations. This only impacts the generation of the initial scenario population, since the initial solution population remains entirely random. If the user defines theoretical worst and best scenarios, these are assigned to the two first chromosomes, and the remainder is randomly generated. In the following Section 2.3, we will describe how the user can define these extreme scenarios.

It is important to note that the `coEvolBrkgaAPI` is still able to use multiple independent populations of the same type. In this case, there can exist multiple pairs of solution and scenario populations that evolve independently. At a predefined interval of generations, exchanges of elite individuals can take place among these independent pairs of populations. Nevertheless, the exchange will always occur between the solution populations. In the case of scenarios, since the focus is on population performance rather than specific individuals, this is not extended.

2.3. Adaptations to the interface files

As described before, there are two main interface files: one where the main algorithmic loop is defined (herein named `Main.cpp`), and other where the (problem-dependent) decoding procedure is established (`Decoder.cpp`).

Main.cpp: The most significant adaptation of the `Main.cpp` file relates to the definition of parameters. Table 2 summarizes all parameters that the user must define in `coEvolBrkgaAPI`. The BRKGA-specific parameters are adaptations of the parameters defined in `brkgaAPI` related to the `BRKGA` class. The co-evolutionary parameters are new parameters added due to the co-evolutionary procedures in this API. Finally, the algorithm-related parameters are general and remain similar to `brkgaAPI`. Many of these are directly applied in the `BRKGA` class, and we have thoroughly discussed them in the previous section. Other changes are straightforward and also related to the two types of population.

`Decoder.cpp`**:** As described before, the main innovation of the `coEvolBrkgaAPI` resides in its decoding procedure. Figure 3 represents the alterations required and describe the main flow of this new decoding procedure.

In a general genetic algorithm, decoding consists of two main sequential steps: "translating" the numeric keys of the chromosome to a solution of the original problem and calculating the fitness value associated with that solution. Both of these steps are dependent on the original problem. In the `coEvolBrkgaAPI`, the second step can be divided into subsequent steps: calculating the value of each solution in the face of each scenario, and calculating the fitness of each solution and scenario.

The co-evolutionary procedures proposed in [10], as represented in Figure 3b, have a problem-independent part, as well as a problem-dependent part. As explained in Section 2.2, the `BRKGA` class defines the following process:

(1) Calculating a matrix (F) that contains the value of each solution in the face of each scenario,
(2) Calculating D, the distance of each scenario to other individuals in the popula-

Table 2. Parameters defined by the user in `coEvolBrkgaAPI`

BRKGA-specific parameters

solu_n	Number of keys in the solution chromosome
solu_p	Number of solution chromosomes in the population
solu_pe	Percent of the solution population that composes the elite
solu_pm	Percent of mutants in the solution population
solu_rhoe	Probability of inheriting a gene from the elite solution parent in cross-over
scen_n	Number of keys in the scenario chromosome
scen_p	Number of scenario chromosomes in the population
scen_pe	Percent of the scenario population that composes the elite
scen_pm	Percent of mutants in the scenario population
scen_rhoe	Probability of inheriting a gene from the elite scenario parent in cross-over

Co-evolutionary parameters

extremeworst	Chromosome (vector of keys) of the theoretically worst scenario. If not defined, this must be an empty vector.
extremebest	Chromosome (vector of keys) of the theoretically best scenario. If not defined, this must be an empty vector.
solucriterion	Criterion for evaluating solution fitness: =1 for Pessimist criterion, =2 for Laplace criterion, =3 for Optimist criterion

Algorithm-related parameters

MAX_GENS	Maximum number of generations (stopping criterion)
X_INTVL	Interval of generations between exchanges among independent populations
X_NUMBER	Number of elite solutions to exchange among independent populations
K	Number of independent pairs of populations (solution, scenario)
MAXT	Maximum number of threads for parallel decoding

tion, based on its "impact range" (i.e., the bounds of values that are possible to obtain in the presence of each scenario),

(3) Calculating fitness of solutions based on F and the criterion selected,

(4) Calculating fitness of scenarios based on their contribution to population diversity (i.e., how "distant" they are from other scenarios in terms of impact range).

In `coEvolBrkgaAPI`, this is implemented by calling the required functions through the `BRKGA` class, yet defining these function in the `Decoder` interface. Therefore, this file includes the functions required to decode and compute the fitness of solutions and scenarios:

- `std::vector<std::vector<double>> matrixF(int MAX_T, std::vector< std::vector<double>> popx, std::vector<std::vector<double>> pops)`.

  The function **matrixF** is highly problem-dependent, as it requires translating both types of chromosomes into solutions and scenarios and calculate the second-stage value for each pair. It receives as inputs the solution and scenario populations, as well as the number of threads if parallel processing is enabled.
- `double SolutionDecoder::soludecode(const int id, const std:: vector<std::vector<double>> &F, const int solucriterion) const`

  The function **soludecode** is problem-independent, yet it depends on the solution fitness criterion defined by the user. It receives as input, besides the fitness criterion, the matrix F calculated before, and the identification of the solution within the solution population (id). It calculates the fitness of the solution ($x = id$) based on the value it obtains in the face of the scenarios (s) in the scenario population. If the pessimist criterion is selected, the fitness equals the solu-

tion performance in the face of the worst scenario ($\text{fitness}_x = \max_s F(x,s)$). On the contrary, if the user selects the optimist criteria, the solution is awarded as fitness the value obtained in the face of the best scenario ($\text{fitness}_x = \min_s F(x,s)$). With the Laplace criterion, the solution fitness equals the average of the impact of all scenarios ($\text{fitness}_x = \texttt{average}_s F(x,s)$).

- `std::vector<std::vector<double>> neighDist(int MAX_T, const std::vector<std::vector<double>> &F)`

  The function **neighDist** is also problem-independent and calculates the distance of each scenario to other scenarios in the population, based on their impact range. It receives as input matrix F (as well as the number of threads) and calculates the projection of each scenario in a two-axis system, representing the worst value obtained by a solution in the face of that scenario (how "bad" the scenario can be) and the best value obtained (how "good" it can be) – its impact range. Figure 4 represents this projection on a two-axis system of a population of scenarios. As the fitness of a scenario represents its contribution to population diversity, this projection supports the calculation of its distance to the nearest neighbours in the scenario population (see Figure 4). For each "axis", the distance measure is the multiplication of absolute difference towards the next scenario and the towards the previous scenario. To favour spreading the scenario space, the extreme scenarios on each axis receive the highest distance value. This function returns, for each scenario, the distance to the nearest neighbours in these two axes (here denoted as dW and dB). This calculation is mathematically represented by Equation 2 for dW, where, for clarity, we introduce the notation $pW_s = \min_x F(x,s)$ to represent the projection of scenario s on the worst axis. The calculation for dB follows a parallel approach considering the best value $pB_s = \max_x F(x,s)$.

$$dW_s = \begin{cases} \max_{s=1}^{\text{scen\_p}-1} dW_s + 1, & s = 0 \vee s = \text{scen\_p} - 1 \\ (pW_s - pW_{s-1}) \times (pW_{s+1} - pW_s), & \forall s \in \{1, \text{scen\_p} - 1\} \end{cases} \tag{2}$$

  Since this function is problem-independent, it applies to any scenario decoding the users might implement. Therefore, the user is not required to adapt it. Nevertheless, if the users desire to change the distance metric and adapt this calculation, this is possible since the full source code is available. Yet, it is essential to ensure the connection with the other functions.

- `double ScenarioDecoder::scendecode(const int id, const std::vector<double> &D) const`

  The function **scendecode** calculates the fitness of a scenario, identified with input *id*, based on the neighbour distances calculated with the previous function. The fitness value favours the most extensive distance (comparing the distances for the two axes).

The changes in file `Decoder.cpp` described above were translated into corresponding developments within the function definitions in file `Decoder.h`. However, due to the design selected for the implementation of this algorithm, the user does not need to adapt the header file, as the function definitions are problem-independent. From the four functions described for this interface, only one is problem-dependent – `matrixF` – as it requires the translation of the chromosome and calculating the value resulting for each pair (solution, scenario). The remaining functions are problem-independent,

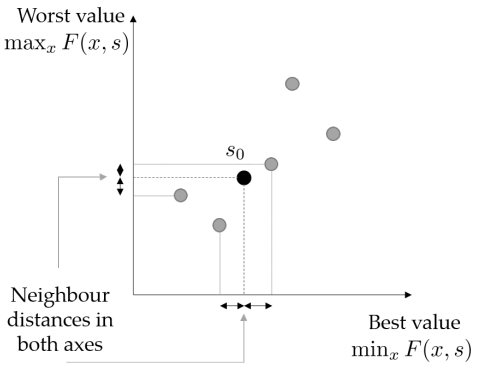


Figure 4. Representation of the two-axis system where scenarios are projected in the fitness calculation procedure. The distances of scenario s_0 to its nearest neighbours in each axes are also represented.

and the user is not required to adapt them. Nevertheless, we decided to maintain them in this interface so that users that want to build on this work and experiment, e.g. different solution fitness criteria, can easily do so.

2.4. A simple example

In the `coEvolBrkgaAPI` made available, we use a simple example to implement a co-evolutionary BRKGA and to test and validate the API. This problem is based on the one used as sample code in the `brkgaAPI`, yet extends it to consider uncertainty, in terms of the impact of different scenarios.

The `brkgaAPI`, as well as the proposed `coEvolBrkgaAPI`, are prepared to tackle minimization problems, i.e., the algorithm favours individuals with the lowest fitness value. In the `brkgaAPI` example, the genes of each (solution) chromosome were sorted, and the fitness of the chromosome was equal to the first gene, i.e. the smallest. The goal is to minimize this value. In `coEvolBrkgaAPI`, similarly, we sort the genes of the solution chromosome, and the goal is to minimize the sum of the first N genes, where the scenario defines N. N represents a fraction of the solution genes, determined by the average gene value of the scenario chromosome. Equations 3 and 4 represent this calculation, where *soluChrom* and *scenChrom* represent the vectors of genes that compose the (sorted) solution chromosome and the scenario chromosome, respectively.

$$N = \left\lfloor \frac{\sum_{i=1}^{\text{scen\_n}} \text{scenChrom}[i]}{\text{scen\_n}} \right\rfloor \tag{3}$$

$$F(x,s) = \sum_{i=1}^{N} \text{soluChrom}[i] \tag{4}$$

Moreover, this simple problem allows defining theoretically extreme values. As we aim to minimize the sum that composes F, the worst possible scenario is the one that maximizes N, i.e., a chromosome such that the gene values g are all equal to one: $\{g_0, g_1, ..., g_{\text{scen\_n}}\} = \{1, 1, ..., 1\}$. Following the same logic, we can define the best scenario as $\{g_0, g_1, ..., g_{\text{scen\_n}}\} = \{0, 0, ..., 0\}$.

3. Experiments

In this section, we will test and validate the impact of the key `coEvolBrkgaAPI` parameters, as well as the potential and efficiency of parallel decoding. For the `brkgaAPI`, the authors [14] analyse the impact of the population and chromosome sizes, the maximum number of threads available and the decoder complexity. Here, we focus on the innovative aspects of `coEvolBrkgaAPI` and propose experiments that allow to investigate the impact of the size of the two different types of populations and corresponding chromosomes, of the different criteria for solution fitness, of the addition (or not) of theoretical extremes to the initial population of scenarios, of the definition of multiple independent pairs of populations, and of the potential of parallel computing. We focus not only on the run time but also on the solution quality and the diversity of the generated sets of scenarios.

For these tests, we implemented the problem described in Section 2.4 and made available in the `coEvolBrkgaAPI`. For the parameters not analyzed in detail in this section, we use the default values proposed in [14]. We set the stopping criterion to 3000 generations, and run all configurations with ten different seed numbers. All results presented refer to the average values obtained. All experiments were run in a Windows virtual machine, with a processor Intel Xeon Gold 6148 CPU @ 2.40 GHz with 96 GB of installed RAM.

A complex real-world application of the method made available in the `coEvolBrkgaAPI`, as well as its main results and managerial insights, is described in [10].

3.1. Size of populations and chromosomes

For these experiments, we analyzed the the parameters that control problem size, as described before, i.e., the solution and scenario chromosome size in terms of number of genes (`solu_n` and `scen_n`). At the same time, we analyzed the impact of the population sizes, i.e., the number of individuals in each population (`solu_p` and `scen_p`). While the former parameters define intrinsically different instances, the latter impact solely the structures of the solution method. Therefore, although both types of parameters significantly influence the time required to run, they impact the method in distinct ways.

For this specific experiment, we selected the Laplace solution fitness criteria, added theoretically extreme scenarios to the initial population, and did not use multiple independent solution populations. We run these experiments in a single-thread environment. The values tested for the size of the chromosomes `solu_n` and `scen_n` were 10, 100, 500, and 1000. As for the size of the solution population `solu_p`, we tested 50, 100, and 1000, since we expect it to have a direct impact on the quality of the final solution *ceteris paribus*. As for the size of the scenario population `scen_p`, since the goal is to obtain a diverse solution, its size should not be too large. Therefore, we tested the values 30, 50 and 100, ensuring it was not more extensive than the solution population.

Figure 5 presents the total run time for the different configurations run. As expected, the time increases exponentially with the problem size and is also impacted by the population size. The impact of increasing the populations is magnified for larger problems in a relevant scale. When considering the number of pairs (solution, scenario) that the decoding procedure evaluates (Figure 6), the time per evaluation is approximately constant, depending solely on the size of the problem, or the instance,

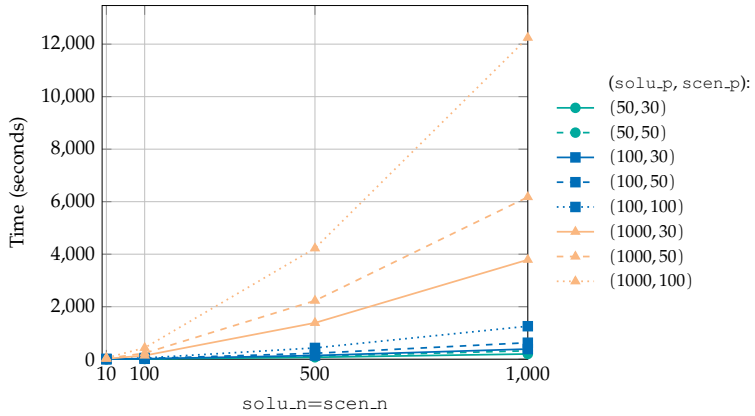


Figure 5. Average total time in single-thread runs for different configurations.

Table 3. Distance between scenarios: average for the sets of the last 100 generations.

Size of populations (`solu_p`, `scen_p`)	**Size of problem** (`solu_n=scen_n`) 10	100	500	1000
(50,30)	0.03	3.68	82.73	319.65
(100,30)	0.04	3.86	84.41	319.69
(1000,30)	0.05	4.24	89.20	332.09
(50,50)	0.01	1.27	31.01	119.51
(100,50)	0.01	1.33	31.15	119.13
(1000,50)	0.01	1.49	32.87	125.06
(100,100)	0.00	0.31	7.89	31.42
(1000,100)	0.00	0.34	8.28	32.40

considered.

It is also noteworthy to analyze the quality of the solution fitness and its evolution throughout generations. Figure 7 shows, for each instance (i.e., chromosome size), the evolution of the best solution fitness throughout generations. For larger problems, the convergence is less steep, as expected. Interestingly, within each instance, there is a distinct behaviour difference where the size of the solution population is the driver. On the contrary, the size of the scenario population does not impact the rate of convergence. It is important to remember that, especially under the Laplace solution fitness criterion, the scenarios truly impact the value obtained by each solution. Nevertheless, the size of this diverse population of scenarios does not impact the convergence to the final solution.

On the contrary, when the diversity of the scenario populations is analyzed (Table 3), the size of the solution population presents no significant impact, whereas the smaller the scenario populations, the larger the average distance between scenarios. This is expected since the theoretically extreme scenarios define the same bounds for all configurations, and if there are more scenarios present, the average distance between them will tend to decrease. Nevertheless, the difference observed is perceptible and underlines the importance of keeping a limited number of scenarios in this type of population due to its evolutionary goal.

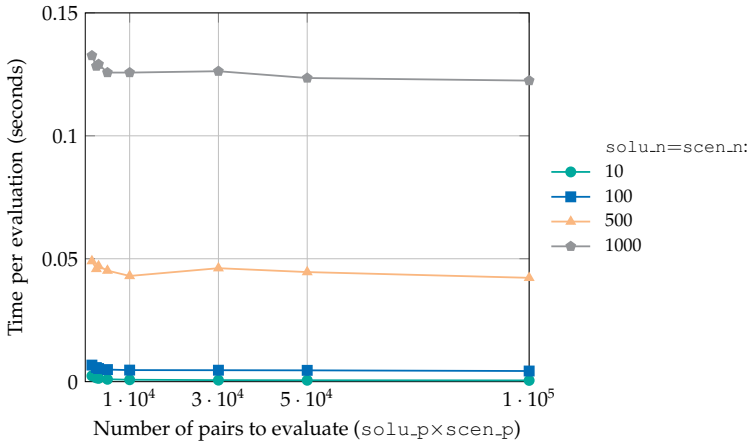


Figure 6. Average time per evaluation of pairs (solution, scenario) for different configurations.

Table 4. Best solution fitness obtained for different criterion and considering the addition (Yes) or not (No) of the theoretically extreme scenarios to the otherwise random initial scenario population.

Theor. extreme scenarios added?	**Generation**	**Solution fitness criterion**		
		Pessimist	Laplace	Optimist
Yes	Initial	10.57	42.10	0.00
	Last	0.36	0.96	0.00
No	Initial	8.39	11.72	6.04
	Last	0.08	0.20	0.00

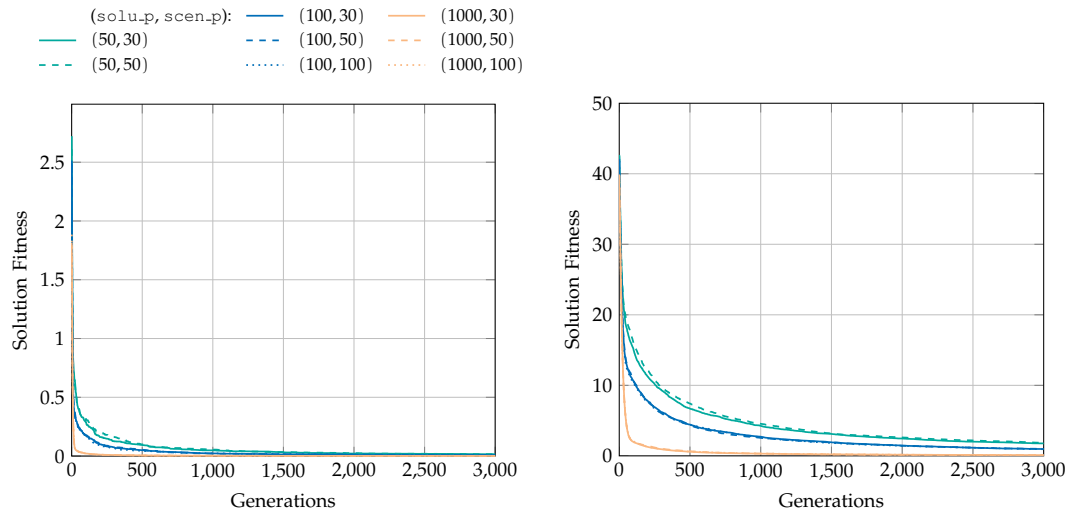
For the remainder of the experiments, we used an instance of size `solu_n=scen_n`= 100, as it represents a large problem which the `coEvolBrkgaAPI` can solve quickly in the number of generations defined. Based on the previous analyses, we set the size of the populations to `solu_p`= 100 and `scen_p`= 30.

3.2. Solution fitness criteria and theoretically extreme scenarios

The objective of this analysis is to study the impact on solution fitness of using the three different criteria, as well as the possibility to add theoretically extreme scenarios to the initial population. In this minimization context, Figure 8 presents the evolution of the best fitness throughout generations for the different combinations of these, whereas Table 4 details the initial and final solution fitness value for these approaches.

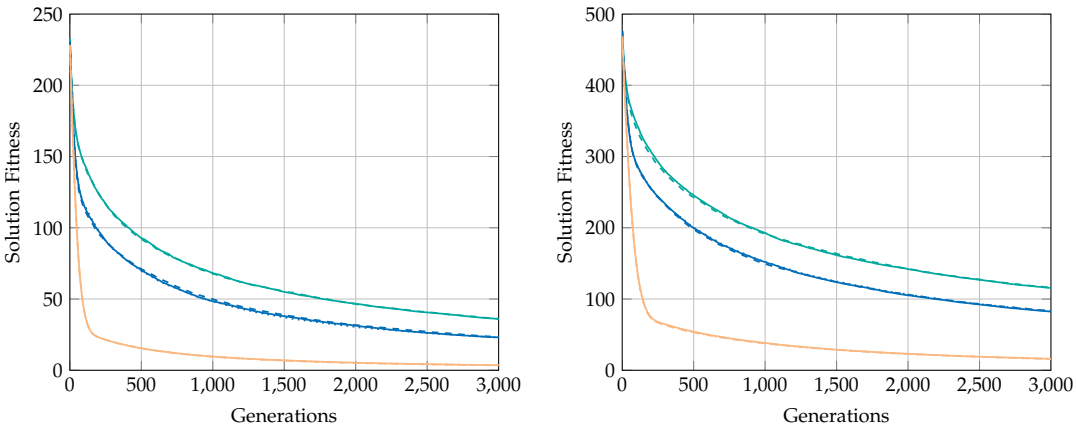
When the user selects the Optimist criterion and adds theoretically extreme scenarios, the fitness of solutions shows a distinct behaviour since it is always null throughout the generations. This happens due to the structure of this specific problem. Since the goal is to minimize the sum of the N first genes of the solution chromosome, with N given by the scenario, the best scenario possible is the one where $N = 0$. With this scenario, the solution fitness is always zero, independently of the solution genes. The best solution fitness, evaluated with this criterion as the performance in the face of the best scenario, is thus always zero.

When considering the other two criteria, a difference of magnitude of the values obtained at the start of the algorithm is perceptible when theoretically extreme sce-



(a) `solu_n=scen_n`= 10

(b) `solu_n=scen_n`= 100



(c) `solu_n=scen_n`= 500

(d) `solu_n=scen_n`= 1000

Figure 7. Evolution of solution best fitness throughout generations for different instances (i.e., sizes of chromosomes `solu_n=scen_n`).

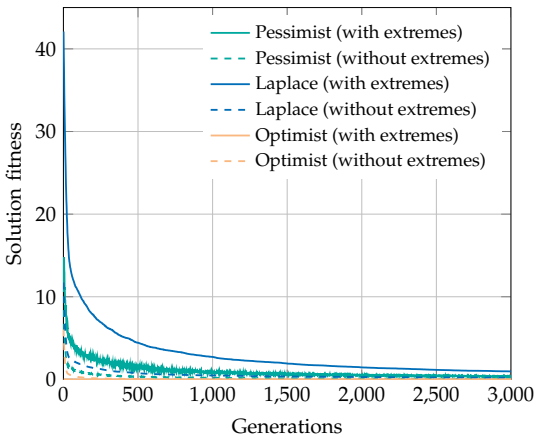


Figure 8. Evolution of best solution fitness throughout generations for different criterion and considering the addition or not of the theoretically extreme scenarios to the random initial scenario population).

narios are added, especially in the Laplace criterion. This results from the meaningful difference between the impact of theoretical extremes and the remaining initial population. On the one hand, the extreme scenarios cause the best and worst value for the solutions in the initial population. On the other hand, the remaining entirely random scenario chromosomes have similar – average – impact values.

Consequently, without theoretically extreme scenarios, the algorithm starts from different positions in the initial generation and a faster convergence is visible (Figure 8). Nevertheless, the final values obtained show also some interesting differences. Both for Laplace and Pessimist criteria, adding theoretically extreme scenarios results in a higher (worse) final solution fitness. However, this does not mean that the solutions obtained are necessarily worse. They are evaluated against different sets of scenarios, and the scenario sets that result from the initial addition of extreme scenarios tend to be more diverse and, thus, create a broader range of impact on the solutions. On the contrary, one could argue that solutions obtained in this case are more "robust", as their performance is evaluated against a broader range of scenarios.

Figure 9 allows to further understand this, at it represents the initial and final scenario populations for each configuration run. Here, scatter points depict the scenarios, representing their range of impact since the x-coordinate marks the best value $F(x, s)$ obtained in the presence of scenario s and the y-coordinate marks the worst value. For all criteria, the initial populations follow the same behaviour. If entirely random, all scenarios have similar, average impacts on solutions. When the two theoretical extreme scenarios are added, they visibly mark the "scenario space" that random generation overlooks.

For the Pessimist and Laplace criteria (Figures 9a and 9b), the impact of evolution is similar. Since the solutions evolve towards better performance, the best value obtained improves significantly, and the diversity of scenario impact is more visible in the worst value obtained, especially, as expected, with the Pessimist criterion. It is also visible that adding the theoretically extreme scenarios causes a more spread population of scenarios, thus supporting the claim that solutions obtained with this configuration are not necessarily weaker and may even show a stronger ability to perform under

Table 5. Distance between scenarios: average for the sets of the last 100 generations.

Solution fitness criterion	K= 1	K= 3
Pessimist	3.81	4.65
Laplace	3.86	4.64
Optimist	4.13	5.09

severe scenarios.

As discussed before, the Optimist criterion (Figure 9c), in this specific problem, leads to the solution characteristics not affecting the evaluation of $F(x, s)$ if the theoretical best scenario is present in the population. As solutions do not change significantly throughout the evolutionary process (see Figure 8), the scenarios spread over a similar space throughout generations. Even when the theoretically best scenario is not added, as the algorithm evolves to spread the range of impact of scenarios, a similar behaviour is observed.

3.3. Multiple independent populations

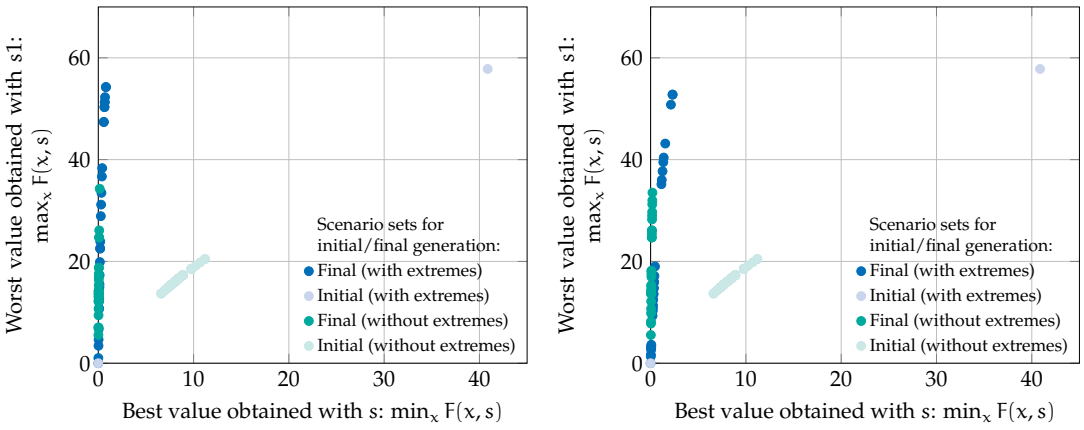
The `coEvolBrkgaAPI` allows the user to define multiple independent pairs of populations, as well as an interval of generations to exchange the top solutions between them. In this section, we analyze the impact of using multiple independent populations on the solution fitness and time to run.

Figure 10 presents the evolution of solution fitness throughout generations using the Pessimist and Laplace criteria, both for the case where a single pair of populations is considered ($K = 1$) and for three parallel pairs ($K = 3$). We considered the addition of theoretically extreme scenarios. Since the Optimist criterion resulted in a constant fitness evaluation (see Section 3.2), it was not represented in this figure. For both criteria, using multiple independent pairs of populations allows convergence in a smaller number of generations. Nevertheless, the time per generation is also significantly higher. In these cases, it was around 2.8 times slower.

As for the impact on the diversity of the final scenario population, Table 5 shows that using multiple independent pairs of populations slightly increases the distance between scenarios for all solution fitness criteria. Concluding, the use of this feature will depend on the problem characteristics, and computation power and run time available.

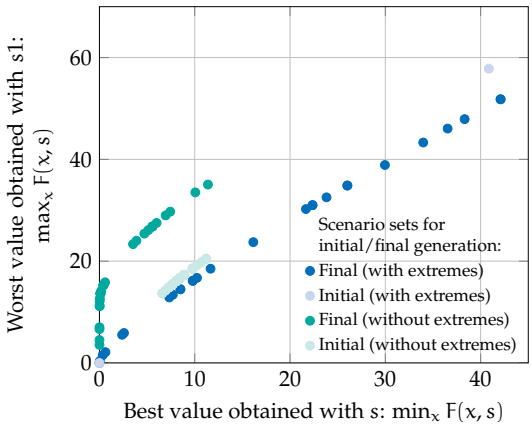
3.4. Efficiency of parallel decoding

To analyze the throughput and efficiency of parallel decoding, we varied the size of the chromosomes, representing problems that are different in size, as well as the number of chromosomes in the solution population. We selected as the baseline, as before, the configuration with `solu_n`=`scen_n`= 100, `solu_p`= 100 and `scen_p`= 30, and we analyzed the impact of increasing the problem size and the solution population size. Due to the discussion in Section 3.1, the size of scenario chromosome was not changed. We run each configuration varying the number of maximum threads used (`MAX_T`) and measured the wall-clock execution time. From this, we computed two metrics: throughput, measured as the number of pairs (solution, scenario) evaluated per second, and efficiency, calculated with Equation 5 [14], where i represents the number of processors. Since these experiments were run in a virtual machine and due to the ensuing access rules to different processors, these analyses represent proxies for the values of



(a) Pessimist criterion

(b) Laplace criterion



(c) Optimist criterion

Figure 9. Representation of the initial and last scenario populations for three runs with the same seed value and three different solution fitness criteria. Each scenario $s$is represented as a point, where the x-coordinate represents the best value $F(x, S)$ obtained with it and the y-coordinate represents the worst value.

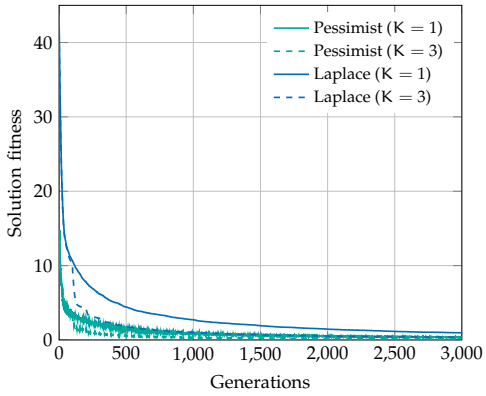


Figure 10. Evolution of solution fitness for Pessimist and Laplace criterion, using one or three independent pairs of population (`K`).

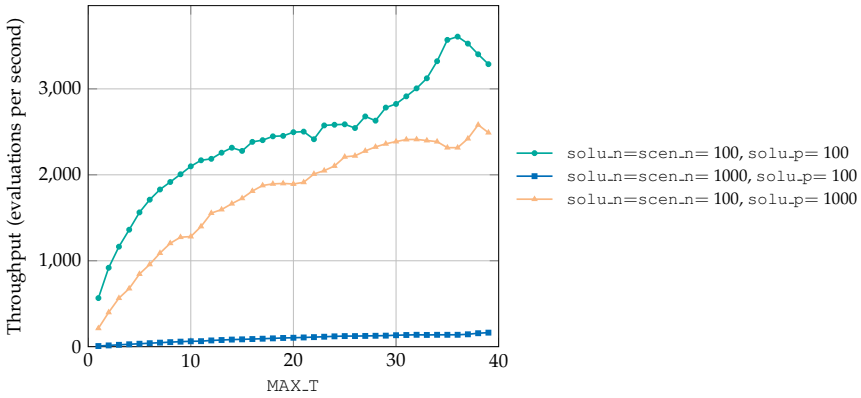


Figure 11. Throughput, measured as number of pairs (solution, scenario) evaluated per second, for different values of parameter `MAX_T`.

throughput and efficiency. Nevertheless, they can bring relevant insights for the users of `coEvolBrkgaAPI`.

$$\text{efficiency}(i) = \frac{t_1}{i \cdot t_i} \tag{5}$$

Figure 11 shows that increasing the number of available threads (`MAX_T`) impacts the throughput in a prominently different scale depending on the size of the problem (given by `solu_n=scen_n`). For smaller sized problems, increasing the available threads increases the throughput significantly up to around `MAX_T`= 35, where it slightly decreases. Increasing the size of the solution population has some effect in decreasing throughout, but the general trend and scale are similar. On the contrary, when the size of the problem increases, this growing trend is still present but in a significantly reduced scale.

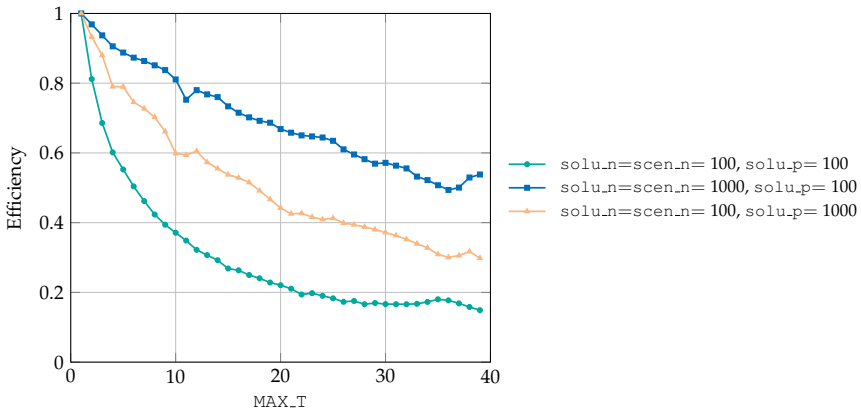


Figure 12. Efficiency for different values of parameter `MAX_T`.

As for efficiency (Figure 12), it decreases, as expected, with the increasing value of `MAX_T`. However, as the solution population or problem sizes increase, the efficiency rate of decay seems to decelerate. For larger problems, the efficiency is above 50% up to `MAX_T`= 39. For the baseline and the increased solution population size, this value decreases to `MAX_T`= 6 and `MAX_T`= 17, respectively. Nevertheless, parallel decoding allows users to accelerate the time to run this algorithm, keeping a significant level of efficiency for the more frequent number of processors available.

4. Concluding remarks

This work proposes an API for the co-evolutionary BRKGA for scenario and solution generation – `coEvolBrkgaAPI`. Based on `brkgaAPI`, this API allows users to easily implement this algorithm, providing an interface for decoding procedures for both solution and scenario chromosomes. Some experiments allow to validate its potential use and to analyze the impact of the main "algorithm-related" parameters, including the addition of the theoretically extreme scenarios to the initial population, the possibility to select different solution fitness criteria, as well as other "computation-related" parameters such as parallel decoding or using multiple independent pairs of populations. The source code for this API is available for download at: `https://drive.google.com/file/d/1YtwZi-sn5PPaTrOvlv_rIyJvdzP2S-CI/view?usp=sharing`[1]. As future work, scenario generation using genetic algorithms can be explored, built on the critical idea of scenario diversity regarding impact on solutions. The resulting modular algorithm would provide representative sets of scenarios that could be applied and coordinated with other optimization settings.

[1]Repository to be updated after acceptance

Funding

This work is partially financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project "POCI-01-0145-FEDER-029279".

References

[1] E. Adida and G. Perakis, *Dynamic pricing and inventory control: Robust vs. stochastic uncertainty models-a computational study*, Annals of Operations Research 181 (2010), pp. 125–157.

[2] R. Bakkehaug, E.S. Eidem, K. Fagerholt, and L.M. Hvattum, *A stochastic programming formulation for strategic fleet renewal in shipping*, Transportation Research Part E: Logistics and Transportation Review 72 (2014), pp. 60–76.

[3] J.R. Birge and F. Louveaux, *Introduction to Stochastic Programming*, 2nd ed., Springer, New York, 2010.

[4] M. Caserta and S. Voβ, *A general corridor method-based approach for capacitated facility location*, International Journal of Production Research [In Press] (2019).

[5] J.F. Gonçalves and M.G.C. Resende, *Biased random-key genetic algorithms for combinatorial optimization*, Journal of Heuristics 17 (2011), pp. 487–525.

[6] C. Gundegjerde, I.B. Halvorsen, E.E. Halvorsen-Weare, L.M. Hvattum, and L.M. Nonås, *A stochastic fleet size and mix model for maintenance operations at offshore wind farms*, Transportation Research Part C: Emerging Technologies 52 (2015), pp. 74–92.

[7] J.W. Herrmann, *A genetic algorithm for minimax optimization problems*, Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999 2 (1999), pp. 1099–1103.

[8] C. Lima, S. Relvas, and A. Barbosa-Póvoa, *Stochastic programming approach for the optimal tactical planning of the downstream oil supply chain*, Computers and Chemical Engineering 108 (2018), pp. 314–336.

[9] L.R. Mundim, M. Andretta, and T.A. de Queiroz, *A biased random key genetic algorithm for open dimension nesting problems using no-fit raster*, Expert Systems with Applications 81 (2017), pp. 358–371.

[10] B.B. Oliveira, M.A. Carravilla, J.F. Oliveira, and A.M. Costa, *A co-evolutionary matheuristic for the car rental capacity-pricing stochastic problem*, European Journal of Operational Research 276 (2019), pp. 637–655.

[11] E. Ruiz, M. Albareda-Sambola, E. Fernández, and M.G. Resende, *A biased random-key genetic algorithm for the capacitated minimum spanning tree problem*, Computers and Operations Research 57 (2015), pp. 95–108.

[12] E. Ruiz, V. Soto-Mendoza, A.E. Ruiz Barbosa, and R. Reyes, *Solving the open vehicle routing problem with capacity and distance constraints with a biased random key genetic algorithm*, Computers and Industrial Engineering 133 (2019), pp. 207–219.

[13] H.D. Sherali and X. Zhu, *Two-Stage Fleet Assignment Model Considering Stochastic Passenger Demands*, Operations Research 56 (2008), pp. 383–399.

[14] R.F. Toso and M.G.C. Resende, *A c++ application programming interface for biased random-key genetic algorithms*, Optimization Methods and Software 30 (2015), pp. 1–16.