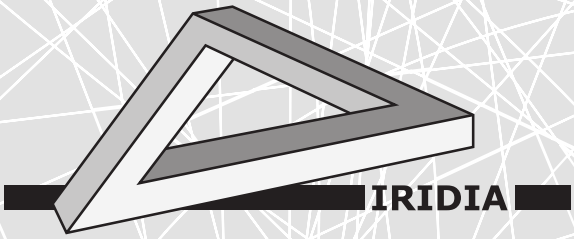


Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

A Unified Ant Colony Optimization Algorithm for Continuous Optimization

Tianjun Liao, Thomas Stützle, Marco A, and Marco Dorigo

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2013-002

February 2013

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2013-002

A Unified Ant Colony Optimization Algorithm for Continuous Optimization

Tianjun Liao[a], Thomas Stützle[a], Marco A. Montes de Oca[b], Marco Dorigo[a]

[a] *IRIDIA, Université Libre de Bruxelles, CP 194/6, Av. F. Roosevelt 50 – B-1050 Brussels, Belgium*
[b] *Dept. of Mathematical Sciences, University of Delaware, 15 Orchard Rd. Newark, DE 19716, USA*

Abstract

In this article, we propose UACOR, a unified ant colony optimization (ACO) algorithm for continuous optimization. UACOR includes algorithmic components from $ACO_{\mathbb{R}}$, $DACO_{\mathbb{R}}$ and $IACO_{\mathbb{R}}$-LS, three ACO algorithms for continuous optimization that have been proposed previously. Thus, UACOR can be seen as an algorithmic framework for continuous ACO algorithms from which earlier continuous ACO algorithms can be instantiated.

The design of UACOR allows the usage of automatic algorithm configuration techniques to automatically derive new, high-performing ACO algorithms for continuous optimization. To show the benefits of UACOR's design, we automatically configure two new ACO algorithms, UACOR-s and UACOR-c, and evaluate them on two sets of benchmark functions from a recent special issue of the Soft Computing (SOCO) journal and the IEEE 2005 Congress on Evolutionary Computation (CEC'05), respectively. UACOR-s's performance is competitive with the best of the 21 algorithms benchmarked on the SOCO benchmark set of 19 functions; UACOR-c is competitive with IPOP-CMA-ES and five other algorithms benchmarked on the CEC'05 set of 25 functions. These results show the high potential of ACO algorithms for continuous optimization and suggest that automatic algorithm configuration is a viable approach for designing state-of-the-art continuous optimizers.

Keywords:
Ant Colony Optimization, Continuous Optimization, Automatic Algorithm Configuration

1. Introduction

Metaheuristics are a family of optimization techniques that have seen increasingly rapid development and have been applied to numerous problems over the past few years. A prominent metaheuristic is ant colony optimization (ACO). ACO is inspired by the ants' foraging behavior and it was first applied to solve discrete optimization problems [1,

Email addresses: tliao@ulb.ac.be (Tianjun Liao), stuetzle@ulb.ac.be (Thomas Stützle), mmontes@math.udel.edu (Marco A. Montes de Oca), mdorigo@ulb.ac.be (Marco Dorigo)

2, 3]. Only much later, adaptations of ACO to continuous optimization problems were introduced. Socha and Dorigo [4] proposed one of the now most popular ACO algorithms for continuous domains, called $ACO_{\mathbb{R}}$. It uses a solution archive as a form of pheromone model for the derivation of a probability distribution over the search space. Leguizamón and Coello [5] proposed an extension of $ACO_{\mathbb{R}}$, called $DACO_{\mathbb{R}}$, that had the goal of better maintaining diversity during the search. Subsequently, Liao et al. [6] proposed $IACO_{\mathbb{R}}$-LS, an incremental ant colony algorithm with local search for continuous optimization. $IACO_{\mathbb{R}}$-LS uses a growing solution archive as an extra search diversification mechanism and a local search to intensify the search. $IACO_{\mathbb{R}}$-LS was benchmarked on two prominent sets of benchmark functions for continuous optimization, obtaining very good results. These benchmark function sets are the ones proposed for a recent special issue of the Soft Computing journal [7, 8] (we refer to this special issue as SOCO) and the special session on real parameter optimization of the 2005 IEEE Congress on Evolutionary Computation (CEC'05) [9].

In this article, we propose an ACO algorithm for continuous optimization which combines algorithmic components from $ACO_{\mathbb{R}}$, $DACO_{\mathbb{R}}$ and $IACO_{\mathbb{R}}$-LS. We call this algorithm UACOR. From UACOR, one can instantiate the original $ACO_{\mathbb{R}}$, $DACO_{\mathbb{R}}$ and $IACO_{\mathbb{R}}$-LS algorithms by using specific combinations of the available algorithmic components and parameter settings. However, one also can obtain combinations of the algorithmic components that are different from any of the already known ones, that is, one can instantiate new continuous ACO algorithms. UACOR's design makes the automatic generation of high performance continuous ACO algorithms possible through the use of automatic algorithm configuration tools. The combination of algoritmic components via automatic algorithm configuration tools has already shown its extraordinary potential for obtaining new state-of-the-art algorithms. For instance, KhudaBukhsh et al. [10] proposed SATenstein and instantiated a new state-of-the-art local search algorithm for the SAT problem. López-Ibáñez and Stützle [11] configured a multi-objective ACO algorithm that outperformed previously proposed multi-objective ACO algorithms for the bi-objective traveling salesman problem. Dubois-Lacoste et al. [12] configured new state-of-the-art algorithms for five variants of multi-objective flow-shop problems. More recently, the ideas behind the combination of algorithm frameworks and automatic algorithm configuration techniques have been extended to the programming by optimization paradigm [13].

UACOR is a highly configurable algorithm, and thus it can also be considered a framework from which new state-of-the-art ACO algorithms for continuous optimization can be derived. In this article, we show how this can be done through the use of an automatic configuration tool. In particular, we use Iterated F-race [14] as implemented in the irace package [15]. With Iterated F-race, algorithm parameters are tuned using a machine learning approach in which an algorithm is trained on a set of problem instances and later tested on another set. As the set of training benchmark functions we use low

dimensional versions of the functions used in the SOCO and CEC'05 benchmark sets. We configure two new ACO variants; UACOR-s is configured on the SOCO benchmark (the -s suffix stands for SOCO) set and UACOR-c on the CEC'05 benchmark set (the -c suffix stands for CEC). UACOR-s and UACOR-c are then tested on higher dimensional versions of the SOCO and CEC'05 benchmark functions. The results show that (i) UACOR-s is competitive or superior to all the 21 algorithms benchmarked on the SOCO function set and that (ii) UACOR-c is competitive to IPOP-CMA-ES [16] and other five recent state-of-the-art algorithms benchmarked on the CEC'05 function set. These experimental results illustrate the high potential of ACO algorithms for continuous optimization. To the best of our knowledge, this is also the first article where a continuous optimizer framework is automatically configured.

The article is organized as follows. Section 2 introduces ACO for continuous domains, reviews the three continuous ACO algorithms underlying UACOR, and identifies their algorithmic components in a component-wise view. Section 3 describes UACOR. In Section 4, we automatically configure UACOR to instantiate UACOR-s and UACOR-c and in Section 5, we evaluate their performance. We conclude and give directions for future work in Section 6.

2. ACO algorithms for continuous optimization

2.1. ACO Metaheuristic

The Ant Colony Optimization (ACO) metaheuristic [3] defines a class of optimization algorithms inspired by the foraging behavior of real ants. In ACO algorithms, artificial ants are stochastic candidate solution construction procedures that exploit a pheromone model and possibly available heuristic information on the problem being tackled. The pheromone model consists of a set of numerical values, called pheromones, that are modified at each iteration in order to bias ants toward the most promising regions of the search space; the heuristic information, if available, captures a priori knowledge on the particular problem instance being solved.

The main algorithmic components of the ACO metaheuristic are the ants' solution construction and the update of the pheromone information. Additional "daemon actions" are procedures that carry out tasks that cannot be performed by single ants. A common example is the activation of a local search procedure to improve an ant's solution or the application of additional pheromone modifications derived from globally available information about, for example, the best solutions constructed so far. Although daemon actions are optional, in practice they can greatly improve the performance of ACO algorithms.

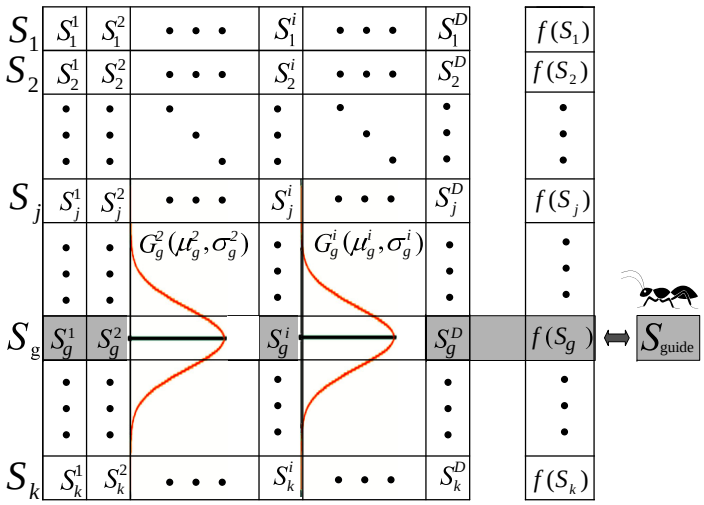


Figure 1: The structure of the solution archive and the Gaussian functions used to generate PDFs in $\text{ACO}_{\mathbb{R}}$

2.2. ACO for Continuous Domains

ACO algorithms were first proposed for tackling combinatorial optimization problems [1, 2, 3]. Many years later, the first ant-inspired algorithms for continuous optimization problems were proposed [17, 18, 19, 20, 21]. However, as explained in [4], most of these algorithms do not follow the principles of operation of the ACO metaheuristic. The first algorithm that can be truly classified as an ACO algorithm for continuous domains is $\text{ACO}_{\mathbb{R}}$ [4]. The reason for this is the following: In ACO algorithms pheromone values are associated with solution components (e.g. cities in a TSP problem). When a solution is constructed, the pheromone values are used to compute the probabilities of selecting the different components to assemble a new candidate solution. Thus, the pheromone model is in fact a set of discrete probability distributions over the values that each solution component can take. In $\text{ACO}_{\mathbb{R}}$, the discrete probability distributions are substituted by probability density functions (PDFs) (i.e., continuous probability distributions). $\text{ACO}_{\mathbb{R}}$ uses a solution archive [22] for the derivation of these PDFs over the search space. Additionally, $\text{ACO}_{\mathbb{R}}$ uses sums of weighted Gaussian functions to generate multimodal PDFs in order to fully capture the complexity of an objective function.

Figure 1 shows a sketch of a solution archive and the Gaussian functions that form the PDFs $\text{ACO}_{\mathbb{R}}$ samples values from to generate candidate solutions. The solution archive keeps track of a number of complete solutions for a problem, and, thus, it can be seen as an explicit memory of the search history.

$\text{DACO}_{\mathbb{R}}$ [5] and $\text{IACO}_{\mathbb{R}}$-LS [6] are two more recent ACO algorithms for continuous optimization. These two algorithms also use a solution archive and generate PDFs using sums of weighted Gaussian functions. Since the algorithmic components of UACOR are derived from the $\text{ACO}_{\mathbb{R}}$, $\text{DACO}_{\mathbb{R}}$ and $\text{IACO}_{\mathbb{R}}$-LS, the next sections describe in detail their operation.

2.2.1. $ACO_{\mathbb{R}}$

$\text{ACO}_{\mathbb{R}}$ initializes the solution archive with k solutions that are generated uniformly at random. Each solution is a D-dimensional vector with real-valued components $x_i \in [x_{\min}, x_{\max}]$, with $i = 1, \ldots, D$. In this paper, we assume that the optimization problems are unconstrained except possibly for bound constraints of the D real-valued variables x_i. The k solutions of the archive are kept sorted according to their quality (from best to worst) and each solution $\boldsymbol{S}_j$ has associated a weight ω_j. This weight is calculated using a Gaussian function as:

$$\omega_j = \frac{1}{qk\sqrt{2\pi}} e^{\frac{-(rank(j)-1)^2}{2q^2k^2}}, \tag{1}$$

where $rank(j)$ is the rank of solution $\boldsymbol{S}_j$ in the sorted archive, and q is a parameter of the algorithm. The result of the computing $rank(j) - 1$ is that the best solution receives the highest weight.

The weights are used to choose probabilistically a guiding solution around which a new candidate solution is generated. The probability of choosing solution $\boldsymbol{S}_j$ as a guiding solution is given by $\omega_j / \sum_{a=1}^{k} \omega_a$ so that the better the solution, the higher the chances of choosing that solution to guide the search. Once a guiding solution $\boldsymbol{S}_{\text{guide}}$ is chosen, the algorithm samples the neighborhood of the i-th real-valued component of the guiding solution s^i_{guide} using a Gaussian PDF with $\mu^i_{\text{guide}} = s^i_{\text{guide}}$, and σ^i_{guide} equal to

$$\sigma^i_{\text{guide}} = \xi \sum_{r=1}^{k} \frac{|s^i_r - s^i_{\text{guide}}|}{k-1}, \tag{2}$$

which is the average distance between the value of the i-th component of $\boldsymbol{S}_{\text{guide}}$ and the values of the i-th components of the other solutions in the archive, multiplied by a parameter ξ. This process is repeated for each dimension i a total of Na times (corresponding to the number of "ants") per iteration. Before the next iteration, the algorithm updates the solution archive keeping only the best k of the $k+Na$ solutions that are available after the solution construction process.

2.2.2. $DACO_{\mathbb{R}}$

Different from $\text{ACO}_{\mathbb{R}}$, $\text{DACO}_{\mathbb{R}}$ keeps the number of ants (Na) equal to the solution archive size k and each of the Na ants constructs at each algorithm iteration a new solution. A further difference of $\text{DACO}_{\mathbb{R}}$ with respect to $\text{ACO}_{\mathbb{R}}$ is the specific choice rule for the guiding solution $\boldsymbol{S}_{\text{guide}}$. With a probability $Q_{best} \in [0, 1]$, ant j chooses as $\boldsymbol{S}_{\text{guide}}$ the best solution, $\boldsymbol{S}_{\text{best}}$, in the archive; with a probability $1 - Q_{best}$, it chooses as $\boldsymbol{S}_{\text{guide}}$ the solution $\boldsymbol{S}_j$. A new solution is generated in the same way as in $\text{ACO}_{\mathbb{R}}$, and then compared to $\boldsymbol{S}_j$ (independently of whether $\boldsymbol{S}_{\text{best}}$ or $\boldsymbol{S}_j$ was chosen as guiding solution). If the newly generated solution is better than $\boldsymbol{S}_j$, it replaces $\boldsymbol{S}_j$ in the archive; otherwise it

is discarded. This replacement strategy is different from the one used in $\text{ACO}_{\mathbb{R}}$ in which all the solutions in the archive and all the newly generated ones compete.

2.2.3. IACO$_{\mathbb{R}}$-LS

$\text{IACO}_{\mathbb{R}}$-LS's main distinctive features are a solution archive whose size increases over time to enhance the algorithm's search diversification, and a local search procedure to enhance its search intensification. Additionally, $\text{IACO}_{\mathbb{R}}$-LS uses a different choice rule for the guiding solution than $\text{ACO}_{\mathbb{R}}$. At each algorithm iteration of $\text{IACO}_{\mathbb{R}}$-LS, the best solution in the archive $\boldsymbol{S}_{\text{best}}$ is chosen as the guiding solution $\boldsymbol{S}_{\text{guide}}$ with a probability equal to the value of a parameter $\mathit{EliteQ}_{best} \in [0, 1]$; with a probability of $1-\mathit{EliteQ}_{best}$, each solution in the archive is used as $\boldsymbol{S}_{\text{guide}}$ to generate a new solution. With this choice rule, either only one new solution is constructed by an "elite" guiding solution or k new solutions are constructed by k ants at each algorithm iteration. Each new solution is constructed in the same way as in $\text{ACO}_{\mathbb{R}}$. Finally, $\boldsymbol{S}_{\text{guide}}$ and the newly generated solution are compared. If the newly generated solution is better than $\boldsymbol{S}_{\text{guide}}$, it replaces it in the archive; otherwise it is discarded.

$\text{IACO}_{\mathbb{R}}$-LS initializes the archive with *InitAS* solutions. Every *GrowthIter* iterations a new solution is added to the archive until a maximum archive size is reached. The new solution is initialized as follows:

$$\boldsymbol{S}_{\text{new}} = \boldsymbol{S}_{\text{rand}} + \text{rand}(0, 1)(\boldsymbol{S}_{\text{best}} - \boldsymbol{S}_{\text{rand}}) \,, \tag{3}$$

where $\boldsymbol{S}_{\text{rand}}$ is a random solution and $\text{rand}(0, 1)$ is a random number uniformly distributed in $[0, 1)$.

In $\text{IACO}_{\mathbb{R}}$-LS the local search procedure is called at each iteration and for *LsIter* iterations. If the local search succeeds in improving the solution from which it is called, this improved solution replaces the original solution in the archive. The maximum number of times the local search procedure is called from the same initial solution is limited to *LsFailures* calls. The initial solution for the local search is chosen as follows. The best solution is deterministically chosen as the initial solution if it has been called less than *LsFailures* times. Otherwise a random solution from the archive is chosen as the initial solution, excluding all those solutions for which the number of times they have been chosen as initial solutions is equal to *LsFailures*.

The step size to be used in the local search procedure is set as follows. First, a solution different from the best one is chosen randomly in the archive. The step size is then set to the maximum norm ($||\cdot||_\infty$) of the vector that separates this random solution from the best solution. As a result, step sizes tend to decrease upon convergence of the algorithm and, in this sense, the step sizes are chosen adaptively to focus the local search around the best-so-far solution. In our previous experiments, Powell's conjugate directions set [23] and Lin-Yu Tseng's Mtsls1 [24] local search methods have shown very good performance.

IACO$_{\mathbb{R}}$-LS uses a default restart mechanism that restarts the algorithm and re-initializes the archive of size *InitAS* with the best-so-far solution $\boldsymbol{S}_{\text{best}}$ and *InitAS*-1 random solutions. The restart criterion is the number of consecutive iterations, *StagIter*, with a relative solution improvement lower than a threshold ϵ. IACO$_{\mathbb{R}}$-LS also integrates a second restart mechanism, which consists in restarting and initializing a new initial archive of size *RestartAS* (*RestartAS* is a parameter different from *InitAS*) with $\boldsymbol{S}_{\text{best}}$ in the current archive and *RestartAS*-1 solutions that are initialized at positions biased around $\boldsymbol{S}_{\text{best}}$; these positions are defined by $\boldsymbol{S}_{\text{best}} + 10^{Shakefactor} * (\boldsymbol{S}_{\text{best}} - \boldsymbol{S}_{\text{rand}})$. The restart criterion is the number of consecutive iterations, *StagIter*, with a relative solution improvement percentage lower than a certain threshold $10^{StagThresh}$.

2.3. Algorithmic Components

We define several algorithmic components for UACOR by abstracting the particular design alternatives taken in ACO$_{\mathbb{R}}$, DACO$_{\mathbb{R}}$ and IACO$_{\mathbb{R}}$-LS. This results in seven main groups of algorithmic components, which are described next, before detailing the outline of UACOR.

1. **Mode.** Two alternative UACOR modes, called *DefaultMode* and *EliteMode*, are identified. *DefaultMode* consists in deploying a number of ants in each algorithm iteration to construct solutions. *EliteMode* allows in each algorithm iteration to deploy only one "elite" ant with a probability of $EliteQ_{best} \in [0, 1]$. The "elite" ant selects $\boldsymbol{S}_{\text{best}}$ in the archive as $\boldsymbol{S}_{\text{guide}}$ to construct a new solution.
2. **Number of ants.** Two design choices for defining the number of ants deployed are identified. *Na* defines the number of ants as an independent parameter ($Na \leq k$) while *NaIsAS* defines the number of ants to be equal to k, the size of the solution archive.
3. **Choice of guiding solution.** This algorithmic component chooses how to select $\boldsymbol{S}_{\text{guide}}$ to sample new solutions. Three design choices are identified: (i) $\boldsymbol{S}_{\text{best}}$ is selected as $\boldsymbol{S}_{\text{guide}}$ with a probability $Q_{best} \in [0, 1]$; (ii) $\boldsymbol{S}_{\text{guide}}$ is probabilistically selected from the solutions in the archive depending on their weight; (iii) solution $\boldsymbol{S}_l$ is selected as $\boldsymbol{S}_{\text{guide}}$, where l is the index of the currently deployed ant.
4. **Update of solution archive.** The update of the solution archive concerns the replacement of solutions in the archive. We identified three design choices. A parameter *RmLocalWorse* defines whether UACOR globally removes the *Na* worst solutions among all k+*Na* solutions, or whether UACOR makes the decision about the acceptance of $\boldsymbol{S}_l$ locally. In the latter case, we use a parameter *SnewvsGsol* to decide whether the solution generated by ant l is compared with $\boldsymbol{S}_{\text{guide}}$ or with the previous l-th solution to remove the worse one.
5. **Local search.** We consider three options for the use of a local search procedure. If parameter *LsType* is set to F (for *false*), no local search procedure is used. Oth-

erwise, *LsType* invokes either Powell's conjugate directions set [23] or Mtsls1 [24]. Both local search procedures use a dynamic calling strategy and an adaptive step size, which follow the choices taken for IACO$_{\mathbb{R}}$-LS.

6. **Incremental archive size.** The possibility of incrementing the archive size is considered. If parameter *IsIncrement* is set to F, the incremental archive mechanism is not used. Otherwise, if *IsIncrement* is set to T (for *true*), UACOR invokes the incremental archive mechanism.
7. **Restart mechanism.** Three options for the restart mechanism are identified. If parameter *RestartType* is set to F, the restart mechanism is not used. Otherwise, *RestartType* invokes either of the two restart mechanisms, which are introduced in IACO$_{\mathbb{R}}$-LS. They are labeled as 1st and 2nd, respectively.

Table 1 summarizes the algorithmic components defined above and their options. Some algorithmic components are only significant for specific value of other components. We discuss the connection between these algorithmic components in Section 3.

Table 1: Algorithmic components of UACOR

Algorithm Components	Options	Description
Mode	{*DefaultMode*, *EliteMode*}	Definition of UACOR mode
AntsNumber	{*Na*, *NaIsAS*}	Definition of the number of ants deployed
SolutionConstructions	sample the neighborhood of the solution component of $\boldsymbol{S}_{\text{guide}}$	Using a Gaussian PDF
	select $\boldsymbol{S}_{\text{best}}$ in a proportion of $Q_{best} \in [0,1]$, select probabilistically by weights, select $\boldsymbol{S}_l$ for the ant l	How $\boldsymbol{S}_{\text{guide}}$ is selected from the solution archive
SolutionArchiveUpdate	remove by globally ranking, remove by comparing with $\boldsymbol{S}_{\text{guide}}$, remove by comparing with $\boldsymbol{S}_l$	How *Na* worse solutions are removed from the archive
LocalSearch	{F, Powell, Mtsls1}	Definition of a local search procedure
IncrementalArchive	{F, True}	Definition of an incremental archive mechanism
RestartMechanism	{F, 1st, 2st}	Definition of a restart mechanism

3. UACOR

The three ACO algorithms described in the previous section as well as many others that may result from the combination of their components are subsumed under the general algorithmic structure provided by UACOR. In this section, we describe the connections of the algorithmic components of UACOR by a flowchart and show how from UACOR we can instantiate the algorithms ACO$_{\mathbb{R}}$, DACO$_{\mathbb{R}}$ and IACO$_{\mathbb{R}}$-LS. The flowchart of UACOR is given in Fig. 2. The related parameters are given in Table 2. Some settings take effect in the context of certain values of other settings.

UACOR starts by randomly initializing and evaluating the solution archive of size *InitAS*. Next, UACOR selects a mode, which can be either the default mode or an elite mode.

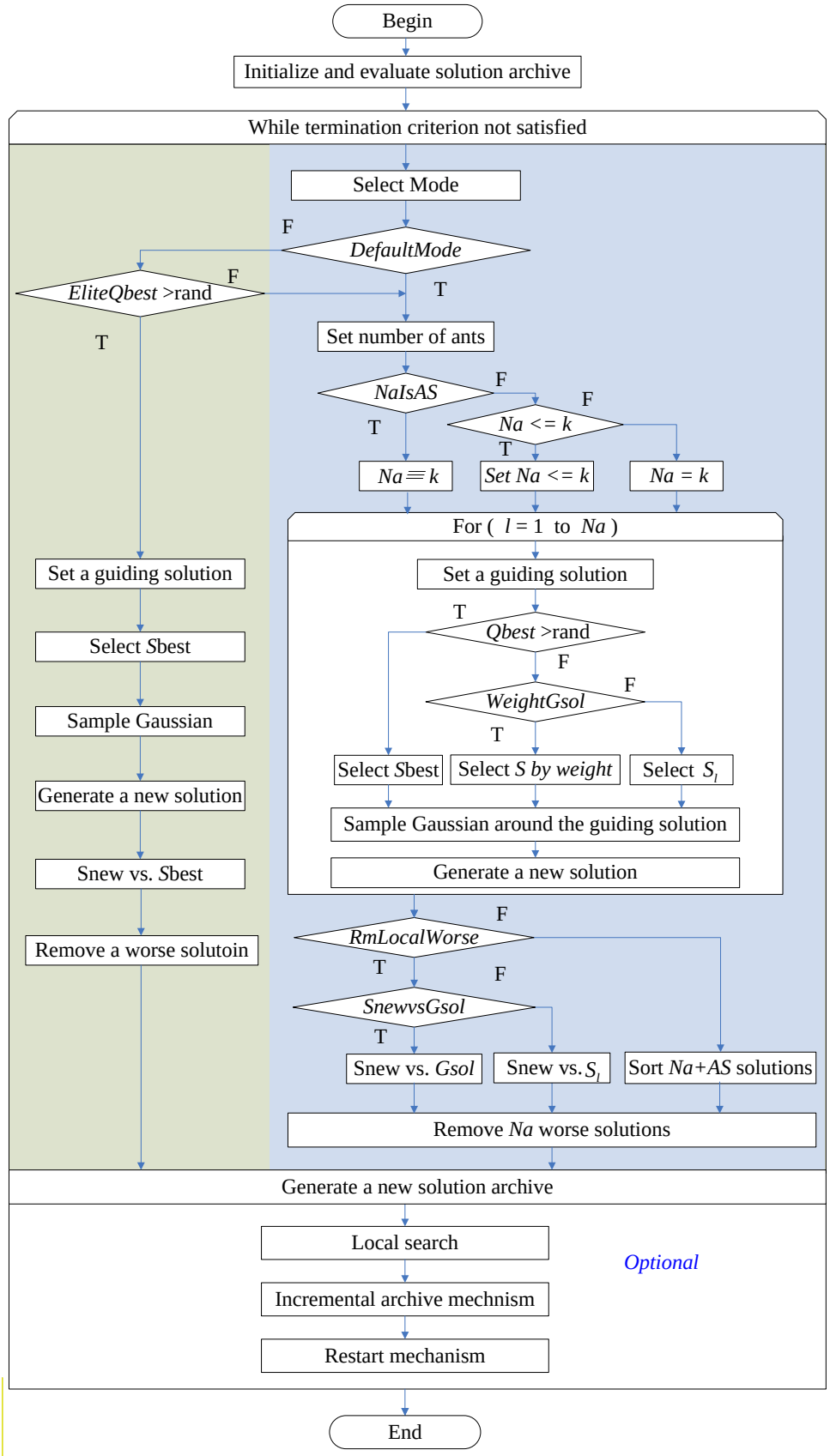


Figure 2: A flowchart for UACOR. For an explanation of the parameters we refer to the text.

We first describe the default mode, which is invoked if parameter *DefaultMode* is set to T (*true*). At each iteration, *Na* new solutions are probabilistically constructed by *Na* ants (recall that and ant in our case is the process through which a solution is generated). If the parameter *NaIsAS* is set to T, the number of ants is kept equal to the size of the solution archive. If the parameter *NaIsAS* is set to F (*false*), a parameter *Na*, $Na \leq k$, is activated. Each ant uses a choice rule for the guiding solution. The parameter $Q_{best} \in [0, 1]$ controls the probability of using $\boldsymbol{S}_{\text{best}}$ as $\boldsymbol{S}_{\text{guide}}$. With a probability $1 - Q_{best}$, $\boldsymbol{S}_{\text{guide}}$ is selected in one of two different ways. If parameter *WeightGsol* is T, $\boldsymbol{S}_{\text{guide}}$ is probabilistically selected from the solutions in the archive by their weights as defined by Equation 1. Otherwise, solution $\boldsymbol{S}_l$ (l is associated with the index of the current ant to be deployed) is chosen as $\boldsymbol{S}_{\text{guide}}$. Once $\boldsymbol{S}_{\text{guide}}$ is selected, a new solution is generated. This process is repeated for each of the *Na* ants. Next, UACOR updates the solution archive by removing *Na* solutions. If parameter *RmLocalWorse* is F, UACOR removes the *Na* worst solutions among all the $k + Na$ solutions as in $\text{ACO}_{\mathbb{R}}$. If parameter *RmLocalWorse* is T, one of two possibilities is considered. If parameter *SnewvsGsol* is T, each newly generated solution is compared to the corresponding $\boldsymbol{S}_{\text{guide}}$ to remove the worse one; otherwise, it is compared to the corresponding $\boldsymbol{S}_l$ to remove the worse one. Finally, a new solution archive is generated.

The elite mode is invoked if parameter *DefaultMode* is set to F. The elite mode at each algorithm iteration deploys only one "elite" ant. With a probability $EliteQ_{best}$, $0 \leq EliteQ_{best} \leq 1$, it selects $\boldsymbol{S}_{\text{best}}$ in the archive as $\boldsymbol{S}_{\text{guide}}$. If the newly generated solution is better than this $\boldsymbol{S}_{\text{best}}$, it replaces it in the solution archive; with a probability $1 - EliteQ_{best}$ the solution construction follows the default mode.

After updating the solution archive, UACOR sequentially considers three procedures. These are a local search procedure, a mechanism for increasing the archive size and a restart mechanism, respectively. Recall that the options of these procedures were described in Section 2.2.3.

We use a simple penalty mechanism to handle bound constraints for UACOR. We use

$$P(\boldsymbol{x}) = fes \cdot \sum_{i=1}^{D} Bound(x_i)\,, \tag{4}$$

where $Bound(x_i)$ is defined as

$$Bound(x_i) = \begin{cases} 0, & \text{if } x_{\min} \leq x_i \leq x_{\max} \\ (x_{\min} - x_i)^2, & \text{if } x_i < x_{\min} \\ (x_{\max} - x_i)^2, & \text{if } x_i > x_{\max} \end{cases} \tag{5}$$

and x_{min} and x_{max} are the minimum and maximum limits of the search range, respectively, and *fes* is the number of function evaluations that have been used so far. For avoiding that

the final solution is outside the bounds, the bound constraints are enforced by clamping the final solution $\boldsymbol{S}$ to the nearest solution on the bounds, resulting in solution $\boldsymbol{S}'$, if $\boldsymbol{S}$ violates some bound constraints. If $\boldsymbol{S}'$ is worse than the best feasible solution found in the optimization process, $\boldsymbol{S}'$ is replaced by it.

4. Automatic Algorithm Configuration

In this article, we automatically configure UACOR before evaluating its performance on benchmark functions. As the benchmark functions, we employ the 19 functions from the SOCO benchmark set [7] (f_{soco1}-f_{soco19}) and the 25 functions from the CEC05 benchmark set [9] (f_{cec1}-f_{cec25}). Note that in both benchmark sets, the functions allow for different dimensionalities. These two benchmark sets have been chosen as they have become standard benchmark sets for testing continuous optimizers. The SOCO benchmark set was used in a special issue of the journal Soft Computing and it extends the benchmark sets of earlier benchmarking studies on the scaling behavior of continuous optimizers such as the one held at the CEC'08 conference. The CEC'05 benchmark set was introduced in 2005 for a comparison of evolutionary optimizers; its central role is exemplified by the more than 480 citations in google scholar (as of July 2012) to the technical report describing this set of functions [9]. Classified by function characteristics, the SOCO benchmark set consists of seven unimodal and 12 multimodal functions, or, four separable and 15 non-separable functions. The CEC'05 benchmark set consists of five unimodal and 20 multimodal functions, or, two separable and 23 non-separable functions. For a detailed description of the benchmark functions, we refer the reader to [7, 9].

In our experiments, we follow the termination conditions suggested for the SOCO and CEC benchmarks [7, 9] to make our results comparable to those of other papers. In particular, we use a maximum of $5\,000 \times D$ function evaluations for the SOCO functions, and $10\,000 \times D$ for the CEC'05 functions, where D is the dimensionality of a function.

For automatically configuring UACOR, we employ Iterated F-Race [14], a method for automatic algorithm configuration that is included in the irace package [15]. Iterated F-Race repeatedly applies F-Race to a set of candidate configurations. F-Race is a racing method that at each iteration applies all surviving candidate configurations to an instance of a combinatorial problem or a function in the continuous optimization case. If a candidate configuration is found to perform statistically worse than others (as determined by the Friedman two-way analysis of variance by ranks and its associated post-tests), it is eliminated from the race. F-race finishes when only one candidate survive or the allocated computation budget to the race is used. Iterated F-Race then samples new candidate configurations around the best candidate configurations found so far. The whole process is repeated for a number of iterations (hence the name Iterated F-Race).

The automatic configuration tool handles all parameter types of UACOR: continuous (r), integer (i) and categorical (c). The performance measure used for tuning is the error

of the objective function value obtained by the tuned algorithm after a certain number of function evaluations. The error value is defined as $f(\boldsymbol{x}) - f(\boldsymbol{x}^*)$, where $\boldsymbol{x}$ is a candidate solution and $\boldsymbol{x}^*$ is the optimal solution. In the automatic tuning process, the maximum budget is set to 5 000 runs of UACOR. The settings of Iterated F-Race that we used in our experiments are the default [14, 15]. We conduct automatic configuration for UACOR in two stages. In the first stage, we tuned UACOR on the SOCO training instances to instantiate UACOR-s. 19 SOCO benchmark functions of dimension 10 were sampled as training instances in a random order. In the second stage, we tuned UACOR on the CEC'05 training instances to instantiate UACOR-c. 25 CEC'05 benchmark functions of dimension 10 were sampled as training instances in a random order.

The tuned parameter settings for both, UACOR-s and UACOR-c, are presented in the central and right part of Table 2. This table also gives the parameter settings for the UACOR's instantiations of $\text{ACO}_{\mathbb{R}}$, $\text{DACO}_{\mathbb{R}}$ and $\text{IACO}_{\mathbb{R}}$-Mtsls1. Their parameters were also automatically tuned as mentioned above for the SOCO and CEC'05 benchmark sets, respectively, and for these specific parameter configurations we again use the extensions '-s' and '-c' depending on the benchmark functions used for automatic configuration. Considering that UACOR-s does not use the restart mechanisms of UACOR after tuning and UACOR-c does, when tuning these three ACO algorithms on the SOCO training instances, we deploy them as proposed in the original literature; when tuning them on CEC'05 training instances, we extend them to use the restart mechanisms of UACOR to improve performance.

As a further illustration of the respective algorithm structures, we highlight UACOR-s and UACOR-c in the flowchart of UACOR in Fig. 3. Both use *DefaultMode*, select $\boldsymbol{S}_{\text{best}}$ as $\boldsymbol{S}_{\text{guide}}$ with a probability $Q_{best} \in [0, 1]$, use Mtsls1 local search and the incremental archive mechanism. The parameter settings in which they differ, imply a more explorative search behavior of UACOR-c than that of UACOR-s. In fact, (i) UACOR-c sets the number of ants equal to the size of the solution archive while UACOR-s defines it as an independent parameter ($Na \leq k$); (ii) UACOR-c frequently chooses all solutions of the archive are chosen as $\boldsymbol{S}_{\text{guide}}$ (as in $\text{DACO}_{\mathbb{R}}$), while UACOR-s probabilistically selects $\boldsymbol{S}_{\text{guide}}$ based on its weight; (iii) UACOR-c makes a local acceptance decision comparing $\boldsymbol{S}_l$ to $\boldsymbol{S}_{\text{guide}}$, while UACOR-s globally removes the Na worst solutions among all $k+Na$ solutions; (iv) UACOR-c uses a restart mechanism for diversifying the search while UACOR-s does not. Considering parameter values, UACOR-c has larger initial archive size and less iterations of local search exploitation, which is consistent with the idea of a strong search exploration than UACOR-s; the larger values of Q_{best} and *GrowthIter* would imply UACOR-c and UACOR-s differ. Similar remarks hold also for the settings of the '-c' and '-s' variants of $\text{ACO}_{\mathbb{R}}$, $\text{DACO}_{\mathbb{R}}$ and $\text{IACO}_{\mathbb{R}}$-Mtsls1. Note that the more explorative settings on the CEC'05 benchmark set are somehow in accordance with the perceived higher difficulty of this benchmark set than the SOCO set. In fact, in the CEC'05 benchmark set the best

available algorithms fail to find quasi-optimal solutions much more frequently than in the SOCO benchmark function set.

Table 2: The left part of the table gives the list of parameter settings and their domains. Some settings are only significant for certain values of other settings. The parameter settings of the automatically configured algorithms are given in the central part and the right part, depending on which training set of benchmark functions was used for tuning.

Module	Para Name	Type	Domain	Tuning on SOCO				Tuning on CEC'05			
				$ACO_{\mathbb{R}}$-s	$DACO_{\mathbb{R}}$-s	$IACO_{\mathbb{R}}$-Mtsls1-s	UACOR-s	$ACO_{\mathbb{R}}$-c	$DACO_{\mathbb{R}}$-c	$IACO_{\mathbb{R}}$-Mtsls1-c	UACOR-c
Mode	*DefaultMode*	c	{T, F}	T	T	F	**T**	T	T	F	**T**
	$EliteQ_{best}$	r	[0, 1]	*	*	**0.0508**	*	*	*	**0.7974**	*
DefNants	*InitAS*	i	[20, 100]	**87**	**40**	**6**	**48**	**92**	**81**	**54**	**66**
	NaIsAS	c	{T, F}	F	T	T	**F**	F	T	T	**T**
	Na	i	[2, 20]	**2**	*	*	**16**	**14**	*	*	*
SolConstr	Q_{best}	r	[0, 1]	0	**0.1193**	0	**0.1895**	0	**0.1287**	0	**0.5351**
	WeightGsol	c	{T, F}	T	F	F	**T**	T	F	F	**F**
	q	r	(0, 1)	**0.2869**	*	*	**0.2591**	**0.09401**	*	*	*
	ξ	r	(0, 1)	**0.7187**	**0.6705**	**0.8782**	**0.6511**	**0.6998**	**0.7357**	**0.9164**	**0.6945**
SAUpdate	*RmLocalWorse*	c	{T, F}	F	T	T	**F**	F	T	T	**T**
	SnewvsGsol	c	{T, F}	*	F	T	*	*	F	T	**T**
LS	*LsType*	c	{F, Powell, Mtsls1}	F	F	Mtsls1	**Mtsls1**	F	F	Mtsls1	**Mtsls1**
	LsIter	i	[1, 100]	*	*	**85**	**84**	*	*	**39**	**28**
	LsFailures	i	[1, 20]	*	*	**1**	**8**	*	*	**3**	**7**
IncArch	*IsIncrement*	c	{T, F}	F	F	T	**T**	F	F	T	**T**
	GrowthIter	i	[1, 30]	*	*	**4**	**4**	*	*	**10**	**13**
RestartMech	*RestartType*	c	{F, 1st, 2nd}	F	F	1st	**F**	**2nd**	**2nd**	**2nd**	**2nd**
	StagIter	r	[1, 1000]	*	*	**18**	*	**939**	**313**	**6**	**11**
	StagThresh	r	[-15, 0]	*	*	*	*	**-3.386**	**-2.302**	**-3.041**	**-2.539**
	Shakefactor	r	[-15, 0]	*	*	*	*	**-4.993**	**-4.163**	**-0.04979**	**-0.02061**
	RestartAS	i	[2, 100]	*	*	*	*	**66**	**71**	**3**	**10**

* denotes the value of the parameter is not relevant for the corresponding algorithm.

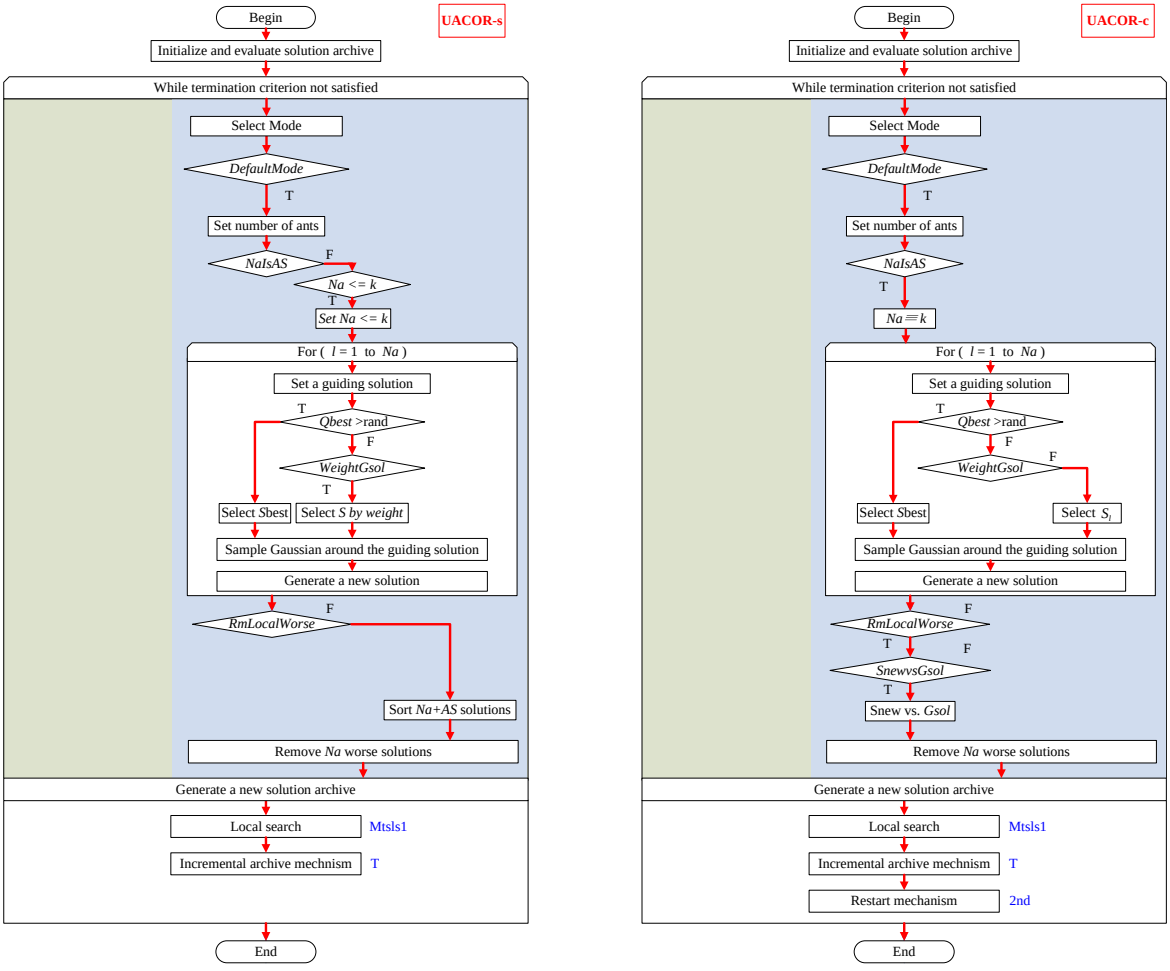


Figure 3: UACOR-s (left side) and UACOR-c (right side) are highlighted in the flowchart of UACOR.

5. Experimental Study

In this section, we evaluate UACOR-s and UACOR-c on the 19 SOCO benchmark functions of dimension 100 and 25 CEC'05 benchmark functions of dimensions 30 and 50. Each algorithm was independently run 25 times on each function. Whenever a run obtains a new best error value, we record the number of function evaluations used, and the new best error value. Following the rules of the SOCO algorithm comparison, error values lower than 10^{-14} are approximated to 10^{-14} (10^{-14} is referred as optimum threshold for SOCO functions). For CEC'05 functions, error values lower than 10^{-8} are approximated to 10^{-8} (10^{-8} is the optimum threshold for CEC'05 functions). On each benchmark function of each dimensionality we compute the average error obtained by an algorithm. These average errors on all test functions in each benchmark set (SOCO or CEC'05) are then used to compare the algorithms' performance. To analyze the results we first use a Friedman test at the 0.05 α-level to determine whether there are significant differences among the algorithms compared [25]. In fact, in all cases the null hypothesis of equal performance is rejected and we then determine the significance of the difference between the algorithms of interest based on the computed minimum difference between the sum of the ranks that is statistically significant.

5.1. Experiments on SOCO benchmark set

First, we compare UACOR-s with the three ACO algorithms, $\text{ACO}_{\mathbb{R}}$-s, $\text{DACO}_{\mathbb{R}}$-s and $\text{IACO}_{\mathbb{R}}$-Mtsls1-s. The left plot of Figure 4 shows that UACOR-s improves upon these ACO algorithms on the distribution of average errors across the 19 SOCO benchmark functions. In particular, UACOR-s performs statistically significantly better than $\text{ACO}_{\mathbb{R}}$-s and $\text{DACO}_{\mathbb{R}}$-s. This test is based on the average error values that are reported in Table 3. In fact, on 14 of the 19 functions the average error obtained by UACOR-s is below the optimum threshold, while for $\text{ACO}_{\mathbb{R}}$-s, $\text{DACO}_{\mathbb{R}}$-s and $\text{IACO}_{\mathbb{R}}$-Mtsls1-s such low average error values are only obtained 0, 1, and 8 times, respectively. (The main responsible for the large differences between the performance of $\text{ACO}_{\mathbb{R}}$-s and $\text{DACO}_{\mathbb{R}}$-s on one side and UACOR-s and $\text{IACO}_{\mathbb{R}}$-Mtsls1-s on the other side is due to the usage or not of a local search procedure to improve candidate solutions.) The larger number of optimum thresholds reached also is the reason why UACOR-s has a lower 75th percentile than $\text{ACO}_{\mathbb{R}}$-Mtsls1-s. Only on three functions, on which UACOR-s does not reach the optimum threshold, it obtains slightly worse average errors than $\text{IACO}_{\mathbb{R}}$-Mtsls1-s.

As a next step, we investigate the benefit of the incremental archive mechanism used by UACOR-s when compared to a fixed archive size. The right boxplot of Figure 4 shows that UACOR-s performs more effective than with archive sizes fixed to 1, 50 and 100, respectively. (Note that for an archive size one, the resulting algorithm is actually an iterated Mtsls1 local search algorithm [24].) The differences are statistically significant

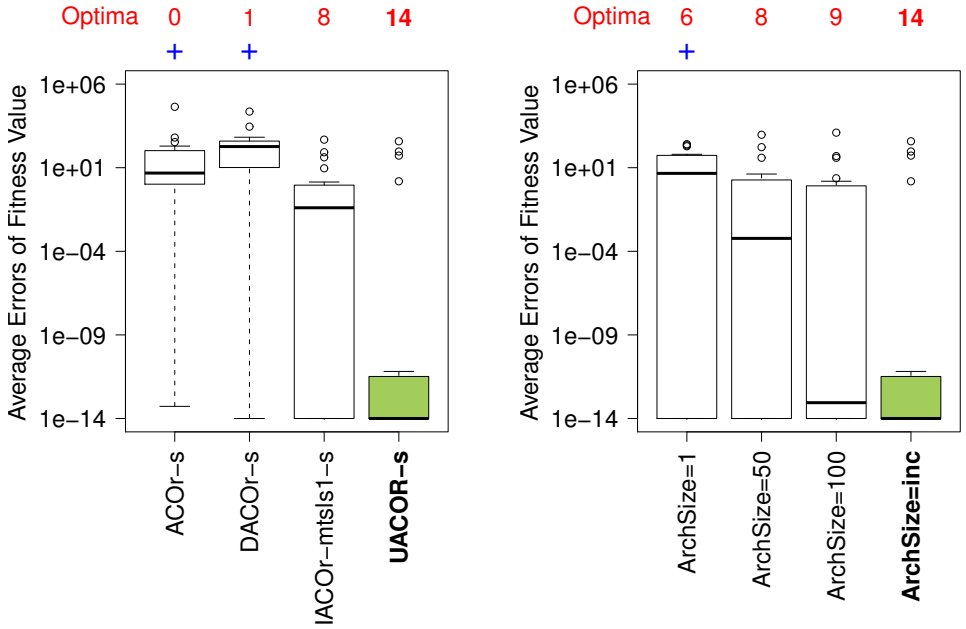
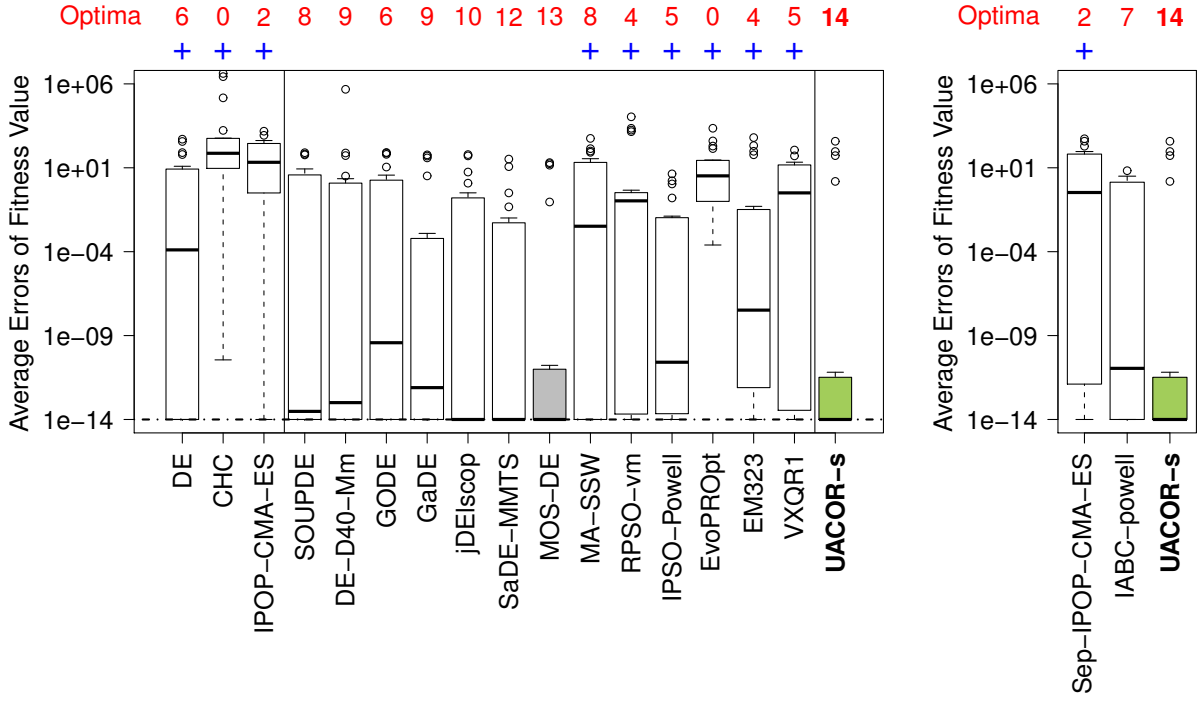


Figure 4: The box-plots show the distribution of the average errors obtained on the 19 SOCO benchmark functions of dimension 100. The left plot compares the performance of UACOR-s with $ACO_{\mathbb{R}}$-s, $DACO_{\mathbb{R}}$-s and $IACO_{\mathbb{R}}$-Mtsls1-s. The right plot shows the benefit of the incremental archive size used in UACOR-s. A + symbol on top of each box-plot denotes a statistically significant difference at the 0.05 α-level between the results obtained by the indicated algorithm and those obtained with UACOR-s. The absence of a symbol means that the difference is not statistically significant. The numbers on top of a box-plot denote the number of the averages below the optimum threshold 10^{-14} found by the indicated algorithms.

for the archive sizes 1 and the average errors of UACOR-s obtain the largest number of times the optimum threshold number (14 versus 6, 8 and 9, respectively).

Finally, we compare UACOR-s with all 13 candidate algorithms published in the SOCO special issue and to the three algorithms that were chosen as reference algorithms.[1] In addition, we compare it to two other recent algorithms [26, 27] that were benchmarked on the SOCO benchmark set. Figure 5 shows that UACOR-s performs statistically significantly better than 10 other algorithms. Recall that IPOP-CMA-ES [16] is considered to be a representative of the state-of-the-art for continuous optimization and MA-SSW [28, 29] was the best performing algorithm at the CEC'2010 competition on high-dimensional numerical optimization. UACOR-s performs statistically significantly better than these two algorithms. The best performing algorithm from the SOCO competition is MOS-DE [30], an algorithm that combines differential evolution and the Mtsls1 local search algorithm. It is noteworthy that UACOR-s performs competitive to MOS-DE. Although UACOR-s does not obtain on more functions lower average errors than MOS-DE than vice versa, UACOR-s gives on more functions the zero threshold (14 versus 13).

[1]Information about these 16 algorithms is available at `http://sci2s.ugr.es/eamhco/CFP.php`



(a) 16 algorithms in SOCO (b) Other algorithms

Figure 5: The box-plots show the distribution of the average errors obtained on the 19 SOCO benchmark functions of dimension 100. The results obtained by the three reference algorithms (left), 13 algorithms (middle) published in SOCO and UACOR-s (right) are shown on the left plot. The line at the bottom of the boxplot represents the optimum threshold (10^{-14}). The results obtained with two other recent algorithms benchmarked on SOCO functions and UACOR-s are shown on the right boxplot. A + symbol on top of the two box-plot denotes a statistically significant difference at the 0.05 α-level between the results obtained with the indicated algorithm and those obtained with UACOR-s detected with a Friedman test and its associated post test on the 19 algorithms. The absence of a symbol means that the difference is not significant. The numbers on top of a box-plot denote the number of averages below the optimum threshold 10^{-14} found by the indicated algorithms.

5.2. Experiments on the CEC'05 benchmark set

We next evaluate UACOR-c on the CEC'05 benchmark set of dimension 30 and 50. Tables 4 and 5 show the average error values across the 25 CEC'05 benchmark functions obtained by UACOR-c, $\text{ACO}_{\mathbb{R}}$-c, $\text{DACO}_{\mathbb{R}}$-c, $\text{IACO}_{\mathbb{R}}$-Mtsls1-c, IPOP-CMA-ES [16] and other five recent state-of-the-art algorithms.

Table 4 shows that UACOR-c gives across the 30 and 50 dimensional problems, on more functions lower average errors than $\text{ACO}_{\mathbb{R}}$-c, $\text{DACO}_{\mathbb{R}}$-c and $\text{IACO}_{\mathbb{R}}$-Mtsls1-c than vice versa. Considering the average error values across all these CEC'05 benchmark functions, UACOR-c performs statistically significantly better than $\text{DACO}_{\mathbb{R}}$-c and $\text{IACO}_{\mathbb{R}}$-Mtsls1-c. It is important to highlight the following two observations. First, UACOR-c significantly outperforms $\text{IACO}_{\mathbb{R}}$-Mtsls1-c on the CEC'05 benchmarks while UACOR-s was superior to $\text{IACO}_{\mathbb{R}}$-Mtsls1-s but without statistical significance. Second, the opposite happens with $\text{ACO}_{\mathbb{R}}$: $\text{ACO}_{\mathbb{R}}$-c performs roughly on par with UACOR-c but $\text{ACO}_{\mathbb{R}}$-s is significantly outperformed by UACOR-s on the SOCO benchmark set. (Recall that

also $ACO_{\mathbb{R}}$ and $IACO_{\mathbb{R}}$-Mtsls1 were tuned for each of the benchmark sets.) In fact, the flexibility of UACOR makes it adaptable to each of the benchmark sets and allows it to outperform other available ACO algorithms.

Of particular interest is the comparison between UACOR-c and IPOP-CMA-ES, the data of which are taken from the literature [16]. The latter is an acknowledged state-of-the-art algorithm on the CEC'05 benchmark set. UACOR-c shows competitive performance to IPOP-CMA-ES and it gives on slightly more functions lower average errors than IPOP-CMA-ES than vice versa. The average error values that correspond to a better result between UACOR-c and IPOP-CMA-ES are highlighted in Table 4.

As a final step, we compare UACOR-c with five recent state-of-the-art continuous optimization algorithms published since 2011. These reference algorithms include HDDE [31], Pro-JADE [32], Pro-SaDE [32], Pro-DEGL [32] and ABC-MR [33]. In the original literature, these algorithms were tested on the CEC'05 benchmark set for which the parameter values of the algorithms were either set by experience or they were manually tuned. We directly obtain the data of the five algorithms on the CEC'05 benchmark set from the original papers. Table 5 shows that UACOR-c gives on the 30 and 50 dimensional problems on more functions lower average errors than each of these five state-of-the-art algorithms. For each algorithm, Table 6 summarizes the average ranking, the number of times the optimum thresholds is reached and the number of lowest average error values obtained across all six algorithms that are compared. Although with the exception of ABC-MR the differences to these algorithms are not found to be statistically significant (neither are the differences among these algorithms statistically significant), it is a noteworthy result that UACOR-c obtains the best average ranking, the highest number of optimum thresholds and that it is the best performing algorithm for most functions.

6. Conclusions

In this article, we proposed UACOR, a unified ant colony framework, that integrates algorithmic components from three previous ACO algorithms for continuous optimization problems, $ACO_{\mathbb{R}}$ [4], $DACO_{\mathbb{R}}$ [5] and $IACO_{\mathbb{R}}$-LS [6]. The UACOR framework is flexible and it allows the instantiation of new ACO algorithms for continuous optimization through the exploitation of automatic algorithm configuration techniques. In fact, in this way we can obtained from the available algorithmic components new ACO algorithms that have not been considered or tested before.

In the experimental part of this article, we have shown that the combination of a flexible, unified algorithm for continuous optimization and automatic algorithm configuration tools can be instrumental for generating new, very high performing algorithms for continuous optimization. We have configured UACOR using Iterated F-Race [14], an automatic algorithm configuration technique implemented in the irace package [15],

on small dimensional training functions taken from two well-known benchmark sets, the SOCO and the CEC'05 benchmark sets, for continuous optimization. The computational results showed that the tuned UACOR algorithms obtain better performance on each of the benchmark sets than the tuned variants of the three ACO algorithms that underly the UACOR framework, namely $ACO_{\mathbb{R}}$, $DACO_{\mathbb{R}}$ and $IACO_{\mathbb{R}}$-LS . Moreover, when UACOR is automatically configured for the SOCO benchmark set, it performs better or competitive to all the recent 21 algorithms benchmarked on this benchmark set; when configured for the CEC'05 benchmark set, it performs competitive to IPOP-CMA-ES, the acknowledged state-of-the-art algorithm on this benchmark set and also competitive or superior to other five recent high-performance continuous optimizers that were evaluated on this benchmark set.

In a nutshell, in this paper we have proven the high potential of ACO algorithms have for continuous optimization and that automatic algorithm configuration has a high potential also for the development of continuous optimizers out of algorithm components. This should encourage also other researchers to apply such automatic algorithm configuration techniques in the design of continuous optimizers. The work presented here can be extended along several directions. A first one is to extend UACOR to synthesize other probability density functions for the generation of candidate solutions, alternative ways of handling the archive, constraint handling techniques for tacklind constrained continuous optimization problems, and other local search algorithms. Another promising direction is to design a more general algorithm framework from which different types of continuous optimizers other than ACO algorithms can be automatically configured.

7. Acknowledgments

The research leading to the results presented in this paper has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nº 246939, and funding from the Scientific Research Directorate of the French Community of Belgium. Thomas Stützle and Marco Dorigo acknowledge support from the Belgian F.R.S.-FNRS, of which they are a Research Associate and a Research Director, respectively. Tianjun Liao acknowledges a fellowship from the China Scholarship Council.

Table 3: The average errors obtained by $ACO_{\mathbb{R}}$-s, $DACO_{\mathbb{R}}$-s, $IACO_{\mathbb{R}}$-Mtsls1-s, MOS-DE and UACOR-s for each SOCO function. The numbers in parenthesis at the bottom of the table represent the number of times an algorithm is better, equal or worse, respectively, than UACOR-s. Error values lower than 10^{-14} are approximated to 10^{-14}. The average errors that correspond to a better result between MOS-DE and UACOR-c are highlighted.

Dim	f_{soco}	$ACO_{\mathbb{R}}$-s	$DACO_{\mathbb{R}}$-s	$IACO_{\mathbb{R}}$-Mtsls1-s	MOS-DE	UACOR-s
100	f_{soco1}	5.32E−14	1.00E−14	1.00E−14	1.00E−14	1.00E−14
	f_{soco2}	2.77E+01	3.82E+01	5.27E−12	2.94E−12	6.53E−12
	f_{soco3}	1.96E+02	2.86E+03	4.77E+02	2.03E+01	3.81E+02
	f_{soco4}	6.34E+02	3.89E+02	1.00E−14	1.00E−14	1.00E−14
	f_{soco5}	2.96E−04	4.96E−01	1.00E−14	1.00E−14	1.00E−14
	f_{soco6}	2.04E−08	3.49E+00	1.00E−14	1.00E−14	1.00E−14
	f_{soco7}	9.94E−10	5.01E−01	1.00E−14	1.00E−14	1.00E−14
	f_{soco8}	4.39E+04	2.28E+04	1.39E+00	9.17E−02	1.54E+00
	f_{soco9}	4.78E+00	1.84E+02	1.62E−01	1.00E−14	1.00E−14
	f_{soco10}	2.75E+00	2.09E+01	1.00E−14	1.00E−14	1.00E−14
	f_{soco11}	3.96E+00	1.90E+02	1.67E−01	1.00E−14	1.00E−14
	f_{soco12}	1.13E+01	2.39E+02	4.01E−02	1.00E−14	1.00E−14
	f_{soco13}	1.47E+02	3.92E+02	4.19E+01	1.75E+01	9.31E+01
	f_{soco14}	3.40E+02	2.40E+02	9.23E+00	1.68E−11	1.00E−14
	f_{soco15}	5.89E−01	4.41E+00	1.00E−14	1.00E−14	1.00E−14
	f_{soco16}	1.61E+00	4.18E+02	4.24E−01	1.00E−14	1.00E−14
	f_{soco17}	6.27E+01	6.65E+02	8.40E+01	1.43E+01	5.30E+01
	f_{soco18}	1.57E+01	1.17E+02	1.07E−01	1.00E−14	1.00E−14
	f_{soco19}	1.48E+00	1.61E+01	1.00E−14	1.00E−14	1.00E−14
By f_{soco}		(1, 0, 18)†	(0, 1, 18)†	(3, 8, 8)	(5, 13, 1)	

† denotes a significant difference between the corresponding algorithm and UACOR-s by a Friedman test at the 0,05 α-level over the distribution of average errors of $ACO_{\mathbb{R}}$-s, $DACO_{\mathbb{R}}$-s, $IACO_{\mathbb{R}}$-Mtsls1-s and UACOR-s.

Table 4: The average errors obtained by $ACO_{\mathbb{R}}$-c, $DACO_{\mathbb{R}}$-c, $IACO_{\mathbb{R}}$-Mtsls1-c, IPOP-CMA-ES and UACOR-c for each CEC'05 function. The numbers in parenthesis at the bottom of the table represent the number of times an algorithm is better, equal or worse, respectively, compared to UACOR-c. Error values lower than 10^{-8} are approximated to 10^{-8}. The average errors that correspond to a better result between IPOP-CMA-ES and UACOR-c are highlighted.

Dim	f_{cec}	$ACO_{\mathbb{R}}$-c	$DACO_{\mathbb{R}}$-c	$IACO_{\mathbb{R}}$-Mtsls1-c	IPOP-CMA-ES	UACOR-c
30	f_{cec1}	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08
	f_{cec2}	1.00E−08	4.74E+00	1.00E−08	1.00E−08	1.00E−08
	f_{cec3}	3.88E+05	4.21E+06	2.19E+05	1.00E−08	2.92E+05
	f_{cec4}	2.75E−04	2.62E+02	9.45E+03	1.11E+04	4.11E+03
	f_{cec5}	1.00E−08	1.00E−08	6.79E−08	1.00E−08	1.00E−08
	f_{cec6}	8.99E+00	2.50E+01	1.64E+02	1.00E−08	2.92E+01
	f_{cec7}	1.80E−02	1.54E−02	1.00E−02	1.00E−08	8.96E−03
	f_{cec8}	2.00E+01	2.02E+01	2.00E+01	2.01E+01	2.00E+01
	f_{cec9}	2.50E+01	6.35E+01	1.00E−08	9.38E−01	1.00E−08
	f_{cec10}	4.51E+01	6.58E+01	1.19E+02	1.65E+00	1.06E+02
	f_{cec11}	3.75E+01	3.19E+01	2.36E+01	5.48E+00	2.15E+01
	f_{cec12}	3.59E+03	1.92E+04	7.89E+03	4.43E+04	1.16E+04
	f_{cec13}	3.67E+00	4.57E+00	1.28E+00	2.49E+00	1.46E+00
	f_{cec14}	1.25E+01	1.34E+01	1.32E+01	1.29E+01	1.29E+01
	f_{cec15}	3.40E+02	3.28E+02	2.48E+02	2.08E+02	2.26E+02
	f_{cec16}	1.33E+02	1.93E+02	2.49E+02	3.50E+01	2.29E+02
	f_{cec17}	1.49E+02	1.81E+02	3.35E+02	2.91E+02	2.62E+02
	f_{cec18}	9.12E+02	9.07E+02	9.02E+02	9.04E+02	8.77E+02
	f_{cec19}	9.11E+02	9.07E+02	8.92E+02	9.04E+02	8.82E+02
	f_{cec20}	9.12E+02	9.07E+02	8.97E+02	9.04E+02	8.78E+02
	f_{cec21}	5.38E+02	5.00E+02	5.12E+02	5.00E+02	5.00E+02
	f_{cec22}	9.08E+02	8.70E+02	9.90E+02	8.03E+02	9.80E+02
	f_{cec23}	5.75E+02	5.35E+02	5.66E+02	5.34E+02	5.34E+02
	f_{cec24}	2.27E+02	7.85E+02	1.26E+03	9.10E+02	8.30E+02
	f_{cec25}	2.19E+02	2.31E+02	5.60E+02	2.11E+02	4.74E+02
50	f_{cec1}	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08
	f_{cec2}	1.80E−04	2.61E+03	4.08E−07	1.00E−08	1.00E−08
	f_{cec3}	6.68E+05	6.72E+06	5.60E+05	1.00E−08	6.06E+05
	f_{cec4}	8.36E+03	4.69E+04	5.33E+04	4.68E+05	4.15E+04
	f_{cec5}	2.23E−05	2.02E−01	7.95E−07	2.85E+00	1.00E−08
	f_{cec6}	2.92E+01	5.18E+01	1.73E+02	1.00E−08	5.00E+01
	f_{cec7}	9.93E−03	1.08E−02	4.53E−03	1.00E−08	7.68E−03
	f_{cec8}	2.00E+01	2.02E+01	2.00E+01	2.01E+01	2.00E+01
	f_{cec9}	4.82E+01	1.15E+02	1.00E−08	1.39E+00	1.00E−08
	f_{cec10}	9.77E+01	1.42E+02	2.83E+02	1.72E+00	2.63E+02
	f_{cec11}	7.30E+01	5.81E+01	4.63E+01	1.17E+01	4.57E+01
	f_{cec12}	2.74E+04	1.07E+05	1.47E+04	2.27E+05	5.26E+04
	f_{cec13}	7.24E+00	1.06E+01	2.13E+00	4.59E+00	2.38E+00
	f_{cec14}	2.24E+01	2.29E+01	2.26E+01	2.29E+01	2.24E+01
	f_{cec15}	3.26E+02	3.61E+02	2.64E+02	2.04E+02	3.00E+02
	f_{cec16}	1.10E+02	1.64E+02	2.89E+02	3.09E+01	2.74E+02
	f_{cec17}	1.62E+02	2.45E+02	5.65E+02	2.34E+02	4.64E+02
	f_{cec18}	9.33E+02	9.26E+02	9.31E+02	9.13E+02	8.83E+02
	f_{cec19}	9.34E+02	9.27E+02	9.18E+02	9.12E+02	8.83E+02
	f_{cec20}	9.36E+02	9.27E+02	9.19E+02	9.12E+02	8.95E+02
	f_{cec21}	5.39E+02	9.82E+02	5.12E+02	1.00E+03	5.00E+02
	f_{cec22}	9.48E+02	9.07E+02	1.06E+03	8.05E+02	1.06E+03
	f_{cec23}	5.56E+02	1.02E+03	5.53E+02	1.01E+03	5.39E+02
	f_{cec24}	2.94E+02	9.06E+02	1.40E+03	9.55E+02	1.30E+03
	f_{cec25}	2.63E+02	3.39E+02	9.52E+02	2.15E+02	7.59E+02
By f_{cec}		(19, 7, 24)	(14, 4, 32)†	(8, 8, 34)†	**(20, 8, 22)**	

† denotes a significant difference between the corresponding algorithm and UACOR-s by a Friedman test at the 0,05 α-level over the distribution of average errors of $ACO_{\mathbb{R}}$-c, $DACO_{\mathbb{R}}$-c, $IACO_{\mathbb{R}}$-Mtsls1-c and UACOR-c.

Table 5: The average errors obtained by HDDE, Pro-JADE, Pro-SaDE, Pro-DEGL, ABC-MR and UACOR-c for for each CEC'05 function. The numbers in parenthesis at the bottom of the table represent the number of times an algorithm is better, equal or worse, respectively, compared to UACOR-c. Error values lower than 10^{-8} are approximated to 10^{-8}. The lowest average errors values are highlighted.

Dim	f_{cec}	HDDE	Pro-JADE	Pro-SaDE	Pro-DEGL	ABC-MR	UACOR-c
30	f_{cec1}	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08
	f_{cec2}	8.13E+00	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08
	f_{cec3}	2.31E+06	1.85E+04	2.28E+06	4.20E+04	2.20E+05	2.92E+05
	f_{cec4}	1.33E+02	1.00E−08	2.00E−05	1.00E−08	1.00E−08	4.11E+03
	f_{cec5}	7.66E+02	5.90E+01	5.51E+01	1.91E+01	6.02E+03	1.00E−08
	f_{cec6}	3.19E+01	1.89E+01	1.36E+00	1.20E+00	1.38E+02	2.92E+01
	f_{cec7}	4.70E+03	4.70E+03	4.70E+03	4.69E+03	1.49E−02	8.96E−03
	f_{cec8}	2.09E+01	2.09E+01	2.10E+01	2.09E+01	2.09E+01	2.00E+01
	f_{cec9}	3.91E+00	1.00E−08	1.00E−08	3.58E+01	6.60E+01	1.00E−08
	f_{cec10}	6.01E+01	8.18E+01	1.01E+02	5.18E+01	2.01E+02	1.06E+02
	f_{cec11}	2.57E+01	3.01E+01	3.37E+01	2.01E+01	3.56E+01	2.15E+01
	f_{cec12}	7.86E+03	2.63E+04	1.48E+03	2.35E+04	9.55E+04	1.16E+04
	f_{cec13}	2.04E+00	3.32E+00	2.95E+00	3.40E+00	1.07E+01	1.46E+00
	f_{cec14}	1.27E+01	1.29E+01	1.31E+01	1.24E+01	1.88E−01	1.29E+01
	f_{cec15}	3.17E+02	3.71E+02	3.86E+02	3.53E+02	2.88E+02	2.26E+02
	f_{cec16}	8.58E+01	1.14E+02	6.97E+01	1.76E+02	3.06E+02	2.29E+02
	f_{cec17}	1.01E+02	1.45E+02	7.20E+01	1.60E+02	3.01E+02	2.62E+02
	f_{cec18}	9.03E+02	8.60E+02	8.56E+02	9.09E+02	8.12E+02	8.77E+02
	f_{cec19}	9.04E+02	8.90E+02	8.67E+02	9.10E+02	8.17E+02	8.82E+02
	f_{cec20}	9.04E+02	8.96E+02	8.52E+02	9.10E+02	8.23E+02	8.78E+02
	f_{cec21}	5.00E+02	5.06E+02	5.00E+02	6.79E+02	6.42E+02	5.00E+02
	f_{cec22}	8.76E+02	8.95E+02	9.09E+02	8.94E+02	9.04E+02	9.80E+02
	f_{cec23}	5.34E+02	5.00E+02	5.00E+02	6.77E+02	8.20E+02	5.34E+02
	f_{cec24}	2.00E+02	2.00E+02	2.00E+02	7.77E+02	2.01E+02	8.30E+02
	f_{cec25}	1.28E+03	1.67E+03	1.63E+03	1.64E+03	2.00E+02	4.74E+02
50	f_{cec1}	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08	1.00E−08
	f_{cec2}	3.30E+02	1.00E−08	7.40E−04	1.00E−08	1.00E−08	1.00E−08
	f_{cec3}	4.44E+06	4.30E+04	7.82E+05	1.93E+05	1.13E+06	6.06E+05
	f_{cec4}	2.94E+03	3.19E−01	6.64E+01	1.08E−01	3.82E+02	4.15E+04
	f_{cec5}	3.22E+03	1.83E+03	1.95E+03	2.23E+03	1.03E+04	1.00E−08
	f_{cec6}	5.74E+01	1.04E+01	1.15E+01	8.77E−01	2.47E+03	5.00E+01
	f_{cec7}	6.20E+03	6.20E+03	6.20E+03	6.20E+03	8.10E−01	7.68E−03
	f_{cec8}	2.11E+01	2.10E+01	2.11E+01	2.11E+01	2.11E+01	2.00E+01
	f_{cec9}	1.87E+01	2.77E+01	6.61E−01	7.94E+01	2.59E+02	1.00E−08
	f_{cec10}	1.13E+02	1.99E+02	6.23E+01	9.24E+01	4.58E+02	2.63E+02
	f_{cec11}	5.19E+01	6.03E+01	6.61E+01	6.14E+01	7.03E+01	4.57E+01
	f_{cec12}	3.84E+04	9.45E+04	7.34E+03	6.32E+04	8.88E+05	5.26E+04
	f_{cec13}	4.36E+00	9.14E+00	6.90E+00	5.41E+00	3.50E+01	2.38E+00
	f_{cec14}	2.23E+01	2.26E+01	2.28E+01	2.26E+01	2.32E+01	2.24E+01
	f_{cec15}	2.62E+02	3.80E+02	3.96E+02	3.44E+02	2.52E+02	3.00E+02
	f_{cec16}	1.05E+02	1.44E+02	4.85E+01	1.63E+02	3.39E+02	2.74E+02
	f_{cec17}	1.15E+02	1.92E+02	9.36E+01	1.94E+02	3.08E+02	4.64E+02
	f_{cec18}	9.17E+02	9.26E+02	9.05E+02	9.28E+02	9.85E+02	8.83E+02
	f_{cec19}	9.16E+02	9.32E+02	8.97E+02	9.28E+02	9.70E+02	8.83E+02
	f_{cec20}	9.16E+02	9.33E+02	9.11E+02	9.29E+02	9.70E+02	8.95E+02
	f_{cec21}	7.37E+02	5.00E+02	5.00E+02	9.50E+02	8.34E+02	5.00E+02
	f_{cec22}	9.01E+02	9.49E+02	9.60E+02	9.28E+02	8.75E+02	1.06E+03
	f_{cec23}	7.85E+02	5.00E+02	5.06E+02	9.35E+02	5.71E+02	5.39E+02
	f_{cec24}	2.00E+02	2.00E+02	2.00E+02	6.81E+02	2.01E+02	1.30E+03
	f_{cec25}	1.37E+03	1.71E+03	1.69E+03	1.67E+03	2.01E+02	7.59E+02
By f_{cec}		(17, 4, 29)	(19, 7, 24)	(21, 6, 23)	(18, 4, 28)	(15, 4, 31)†	

† denotes a significant difference between the corresponding algorithm and UACOR-c by a Friedman test at the 0,05 α-level over the distribution of average errors of HDDE, Pro-JADE, Pro-SaDE, Pro-DEGL, ABC-MR and UACOR-c.

Table 6: Given are the average rank, the number of optimum thresholds reached, and the number of times the lowest average errors reached by each algorithm presented in Table 5. In addition, we give the publication source for each reference algorithm.

Algorithms	Average Ranking	Num of Optima	Num of lowest average error values	Publication Sources
UACOR-c	**3.05**	**8**	**21**	
Pro-SaDE	3.19	4	16	IEEE TEC, 2011
Pro-JADE	3.40	6	13	IEEE TEC, 2011
HDDE	3.44	2	7	IEEE TEC, 2011
Pro-DEGL	3.69	5	10	IEEE TEC, 2011
ABC-MR	4.23	5	13	Information Sciences, 2012

References

[1] M. Dorigo, V. Maniezzo, A. Colorni, The Ant System: An autocatalytic optimizing process, Tech. Rep. 91-016 Revised, Dipartimento di Elettronica, Politecnico di Milano, Italy (1991).

[2] M. Dorigo, V. Maniezzo, A. Colorni, Ant System: Optimization by a colony of cooperating agents, IEEE Transactions on Systems, Man, and Cybernetics – Part B 26 (1) (1996) 29–41.

[3] M. Dorigo, T. Stützle, Ant Colony Optimization, MIT Press, Cambridge, London, England, 2004.

[4] K. Socha, M. Dorigo, Ant colony optimization for continuous domains, European Journal of Operational Research 185 (3) (2008) 1155–1173.

[5] G. Leguizamón, C. Coello, An alternative ACOR algorithm for continuous optimization problems, in: M. Dorigo, et al. (Eds.), Proceedings of the Seventh International Conference on Swarm Intelligence, ANTS'10, Vol. 6234 of LNCS, Springer, Berlin, Germany, 2010, pp. 48–59.

[6] T. Liao, M. A. Montes de Oca, D. Aydın, T. Stützle, M. Dorigo, An incremental ant colony algorithm with local search for continuous optimization, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'11, ACM, New York, NY, USA, 2011, pp. 125–132.

[7] F. Herrera, M. Lozano, D. Molina, Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems, URL: `http://sci2s.ugr.es/eamhco/` (2010).

[8] M. Lozano, D. Molina, F. Herrera, Editorial scalability of evolutionary algorithms and other metaheuristics for large-scale continuous optimization problems, Soft Computing - A Fusion of Foundations, Methodologies and Applications 15 (2011) 2085–2087.

[9] P. Suganthan, N. Hansen, J. Liang, K. Deb, Y. Chen, A. Auger, S. Tiwari, Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization, Tech. Rep. 2005005, Nanyang Technological University (2005).

[10] A. KhudaBukhsh, L. Xu, H. Hoos, K. Leyton-Brown, Satenstein: Automatically building local search SAT solvers from components, in: Proceedings of International Joint Conference on Artificial Intelligence, IJCAI'09, 2009, pp. 517–524.

[11] M. López-Ibáñez, T. Stützle, Automatic configuration of multi-objective aco algorithms, in: M. Dorigo, et al. (Eds.), Proceedings of the Seventh International Conference on Swarm Intelligence, ANTS'10, Vol. 6234 of LNCS, Springer, Berlin, Germany, 2010, pp. 95–106.

[12] J. Dubois-Lacoste, M. López-Ibáñez, T. Stützle, Automatic configuration of state-of-the-art multi-objective optimizers using the TP+PLS framework, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'11, ACM, New York, NY, USA, 2011, pp. 2019–2026.

[13] H. H. Hoos, Programming by optimization, Communications of the ACM 55 (2) (2012) 70–80.

[14] M. Birattari, Z. Yuan, P. Balaprakash, T. Stützle, F-Race and iterated F-Race: An overview, Experimental Methods for the Analysis of Optimization Algorithms (2010) 311–336Springer, Berlin, Germany.

[15] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, M. Birattari, The `irace` package, iterated race for automatic algorithm configuration, Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011).

[16] A. Auger, N. Hansen, A restart CMA evolution strategy with increasing population size, in: Proceeding of IEEE Congress on Evolutionary Computation, CEC'05, IEEE Press, Piscataway, NJ, 2005, pp. 1769–1776.

[17] G. Bilchev, I. Parmee, The Ant Colony Metaphor for Searching Continuous Design Spaces, in: T. Fogarty (Ed.), AISB Workshop on Evolutionary Computing, Springer-Verlag, 1995, pp. 25–39.

[18] N. Monmarché, G. Venturini, M. Slimane, On how Pachycondyla apicalis ants suggest a new search algorithm, Future Generation Computer Systems 16 (9) (2000) 937–946.

[19] J. Dréo, P. Siarry, Continuous interacting ant colony algorithm based on dense heterarchy, Future Generation Computer Systems 20 (5) (2004) 841–856.

[20] X.-M. Hu, J. Zhang, Y. Li, Orthogonal methods based ant colony search for solving continuous optimization problems, Journal of Computer Science and Technology 23 (2008) 2–18.

[21] X.-M. Hu, J. Zhang, H. S.-H. Chung, Y. Li, O. Liu, SamACO: Variable sampling ant colony optimization algorithm for continuous optimization, IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics 40 (2010) 1555–1566.

[22] M. Guntsch, M. Middendorf, A population based approach for ACO, in: S. Cagnoni, et al. (Eds.), Proceedings of EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTim, Vol. 2279 of LNCS, Springer, Berlin, Germany, 2002, pp. 71–80.

[23] M. Powell, An efficient method for finding the minimum of a function of several variables without calculating derivatives, The Computer Journal 7 (2) (1964) 155.

[24] L. Tseng, C. Chen, Multiple trajectory search for large scale global optimization, in: Proceeding of IEEE Congress on Evolutionary Computation, CEC'08, IEEE Press, Piscataway, NJ, 2008, pp. 3052–3059.

[25] W. J. Conover, Paractial Nonparametric Statistics, 3rd Edition, John Wiley & Sons, New York, NY, USA, 1999.

[26] T. Liao, M. A. Montes de Oca, T. Stützle, Tuning parameters across mixed dimensional instances: A performance scalability study of Sep-G-CMA-ES, in: Proceedings of the Workshop on Scaling Behaviours of Landscapes, Parameters and Algorithms of the Genetic and Evolutionary Computation Conference, GECCO'11, ACM, New York, NY, USA, 2011, pp. 703–706.

[27] D. Aydın, T. Liao, M. A. Montes de Oca, T. Stützle, Improving performance via population growth and local search: The case of the artificial bee colony algorithm, Tech. Rep. TR/IRIDIA/2011-015, IRIDIA, Université Libre de Bruxelles, Belgium (2011).

[28] D. Molina, M. Lozano, F. Herrera, MA-SW-Chains: Memetic algorithm based on local search chains for large scale continuous global optimization, in: Proceeding of IEEE Congress on Evolutionary Computation, CEC'10, IEEE Press, 2010, pp. 1–8.

[29] D. Molina, M. Lozano, A. Snchez, F. Herrera, Memetic algorithms based on local search chains for large scale continuous optimisation problems: MA-SSW-Chains, Soft Computing - A Fusion of Foundations, Methodologies and Applications 15 (2011) 2201–2220.

[30] A. LaTorre, S. Muelas, J.-M. Pea, A MOS-based dynamic memetic differential evolution algorithm for continuous optimization: a scalability test, Soft Computing - A Fusion of Foundations, Methodologies and Applications 15 (2011) 2187–2199.

[31] B. Dorronsoro, P. Bouvry, Improving classical and decentralized differential evolution with new mutation operator and population topologies, IEEE Transactions on Evolutionary Computation 15 (1) (2011) 67–98.

[32] M. Epitropakis, D. Tasoulis, N. Pavlidis, V. Plagianakos, M. Vrahatis, Enhancing differential evolution utilizing proximity-based mutation operators, IEEE Transactions on Evolutionary Computation 15 (1) (2011) 99–119.

[33] B. Akay, D. Karaboga, A modified artificial bee colony algorithm for real-parameter optimization, Information Sciences 192 (2012) 120–142.