2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Effective Decomposition of Large-Scale Separable Continuous Functions for Cooperative Co-evolutionary Algorithms

Mohammad Nabi Omidvar
School of Computer Science and IT
RMIT University
Melbourne, Australia
Email: mohammad.omidvar@rmit.edu.au

Yi Mei
School of Computer Science and IT
RMIT University
Melbourne, Australia
Email: yi.mei@rmit.edu.au

Xiaodong Li
School of Computer Science and IT
RMIT University
Melbourne, Australia
Email: xiaodong.li@rmit.edu.au

***Abstract*—In this paper we investigate the performance of cooperative co-evolutionary (CC) algorithms on large-scale fully-separable continuous optimization problems. We have shown that decomposition can have significant impact on the performance of CC algorithms. The empirical results show that the subcomponent size should be chosen small enough so that the subcomponent size is within the capacity of the subcomponent optimizer. In practice, determining the optimal size is difficult. Therefore, adaptive techniques are desired by practitioners. Here we propose an adaptive method, MLSoft, that uses widely-used techniques in reinforcement learning such as the value function method and softmax selection rule to adapt the subcomponent size during the optimization process. The experimental results show that MLSoft is significantly better than an existing adaptive algorithm called MLCC on a set of large-scale fully-separable problems.**

I. Introduction

Cooperative co-evolutionary (CC) algorithms [1] have been widely used for function optimization [2]–[14]. A CC framework is appealing, especially for solving large-scale complex problems. Many such techniques have been used for large-scale global optimization [3], [5], [7]–[11], [15]. CC algorithms use a divide-and-conquer approach to decompose a large-scale problem into a set of lower dimensional problems which are easier to optimize.

The main challenge of using a CC framework is finding an optimal decomposition. It is clear that a problem of a certain size can be decomposed in many different ways. Among all possible decompositions, the one that results in the highest optimization performance is desirable. Such an optimal decomposition varies from problem to problem and depends on many factors. Variable interaction is an important factor that can significantly affect the performance of a particular decomposition. Variable interaction is loosely defined as the degree to which the fitness of one variable is influenced by the values taken by other variables. Variable interaction is also known as *epistasis* or *non-separability* [16], [17]. A class of problems known as partially-separable [18] problems consists of several groups of interacting (non-separable) variables with no interaction between groups. When a CC framework is used to optimize a partially-separable problem, it is important to use a decomposition that takes the variable interaction into account. The optimization performance degrades significantly if interacting variables are placed in different subcomponents.

In some problems the interaction structure is known, but in many problems the optimization algorithm has to automatically identify the underlying interaction structure of the decision variables. Several algorithms have been proposed for automatic identification of variable interaction [10], [12], [19]. Differential grouping [10] is the state-of-the-art decomposition algorithm that can identify non-separable and separable variables with high accuracy. Most of the studies that deal with variable interaction emphasize the importance of optimal decomposition of non-separable variables, and often neglect the importance of proper decomposition of separable variables and their impact on the performance. It should be noted that a partially-separable problem may contain a fully-separable subcomponent which is completely independent of other non-separable subcomponents. Consequently, proper subdivision of a fully-separable subcomponent of a partially-separable problem can directly lead to the improvement of the overall optimization performance. Therefore, focusing on fully-separable problems allows us to investigate the effect of subcomponent size that can be extended to partially-separable functions. In particular, the aim of this paper is to answer the following questions:

- How various decompositions of a fully-separable problem can affect the performance of a CC algorithm?
- What is an effective decomposition for a fully-separable problem?
- How an algorithm can automatically find a suitable decomposition in the course of optimization?

In a fully-separable problem there is no interaction between any pair of variables. Therefore, no particular decomposition is imposed due to variable interaction. It may appear that the best decomposition for such cases is to break the problem into a set of 1-dimensional problems. However, in a CC framework the more subcomponents there are, the more computational resources (fitness evaluations) are required in each co-evolutionary cycle. This means that the simplification of a problem by means of decomposition comes at the expense of using more computational resources. The other extreme decomposition is to group all variables in one subcomponent. This decomposition uses the least amount of resources in each co-evolutionary cycle, but makes the search space extremely large. In this paper, we show that a more effective decomposition of a fully-separable problem lies in between these two extreme cases. In particular, we show that the rule of thumb for selecting a subcomponent size is to choose it small enough so that it is within the capacity of the subcomponent optimizer, but it should not be made any smaller. In short, *the subcomponent should be made as small as possible, but not smaller*[1]. It should be noted that the capacity of an optimizer is not always known, and finding the optimal subcomponent size requires elaborate empirical studies. A more practical approach is to adapt the subcomponent size during the optimization. Multilevel cooperative co-evolution (MLCC) [5] is such a method that uses a probabilistic approach to adapt the subcomponent sizes during optimization. In this paper, we show that the problem of adapting the subcomponent size for a fully-separable problem can be seen as a reinforcement learning [20] problem. We also show that a simple reinforcement learning technique called the value function method and the softmax action selection rule [20] can be used to significantly improve the performance of MLCC.

The organization of the rest of this paper is as follows. Section II describes a general CC framework, and the MLCC algorithm. Section III describes the proposed method for adapting the subcomponent size based on action value method and softmax. Experimental results and their analysis are provided in Section IV. Finally, Section V concludes the paper and outlines possible future extensions.

II. Background

A. Cooperative Co-evolution

Cooperative Co-evolution (CC) [1] has been proposed as a means of breaking a complex problem down into a set of smaller subproblems. Once the problem is decomposed, subcomponents are co-adapted in a round-robin fashion. There are many different ways of decomposing a problem. An optimal decomposition is governed by the underlying interaction structure of the decision variables. Algorithms such as differential grouping [10] automatically identify the interaction structure of variables and returns a decomposition that can be used in a CC framework. The subcomponents returned by such algorithms are usually of different sizes, but for simplicity in this section we assume equal subcomponent sizes. Algorithm 1 shows the general framework of a CC algorithm. The function `grouping` can be any grouping procedure such as differential grouping [10]. Lines 4-11 show the main loop of the CC framework. The inner loop (lines 5-10) iterates over all subcomponents and optimizes them using the `optimizer` function for a predetermined number of iterations. The `optimizer` can be any evolutionary algorithm. In this paper we use SaNSDE [21] which is a variant of differential evolution (DE) [22] with neighborhood search that adapts the crossover rate and the scaling fact of DE. In this paper the `grouping` function subdivides a n-dimensional problem into s d-dimensional subcomponents and the order of

Algorithm 1: CC($func, lbounds, ubounds, n$)

```
1.  $groups \leftarrow$ grouping($f, lbounds, ubounds, n$)    ▷ grouping stage.
2.  $pop \leftarrow$ rand($popsize, n$)    ▷ optimization stage.
3.  $(best, best\_val) \leftarrow$ min($f(pop)$)
4.  for $i \leftarrow 1$ to $cycles$ do
5.      for $j \leftarrow 1$ to size($groups$) do
6.          $indicies \leftarrow groups[j]$
7.          $subpop \leftarrow pop[:, indicies]$
8.          $(subpop, best, best\_val) \leftarrow$ optimizer($f, best, subpop, FE$)
9.          $pop[:, indicies] \leftarrow subpop$
10.     end for
11. end for
```

the decision variables is fixed during a run. Here we abbreviate a CC framework with SaNSDE as its optimizer to DECC. It should be noted in Algorithm 1 the groups do not change during the optimization and they are determined only once prior to the optimization phase.

B. Multi-level Cooperative Co-evolution

Multilevel cooperative co-evolution (MLCC) [5] uniformly decomposes a problem into s d-dimensional problems. MLCC maintains a list of several common subcomponent sizes (also known as decomposers) and assigns a performance score to each decomposer based on their performance in each co-evolutionary cycle. Finally, it uses a probabilistic approach to select a decomposer to subdivide a large-scale problem in each cycle. In order to increase the probability of placing two interacting variables in a common subcomponent, MLCC randomly rearranges the decision variables at the beginning of each cycle. MLCC has been initially designed to deal with non-separable problems. However, it is now superseded by algorithms such as differential grouping [10] that can identify the interaction structure of variables with high accuracy. Although MLCC is not the most efficient method for dealing with non-separable variable, its way of adapting the subcomponent size can be used for decomposing fully-separable problems.

Major steps of the MLCC algorithm are as follows:

1) The initial random population is created and evaluated.
2) The vector of subcomponent sizes is initialized to the values provided by the user: $S = (s_1, \dots, s_k)$, where s_i is the ith decomposer, and k is the number of available decomposers.
3) A vector is created for maintaining the performance of each subcomponent: $R = (r_1, \dots, r_k)$. The performance records r_i is initialized to 1.
4) A decomposer is randomly chosen from S with a probability drawn from the following distribution:

$$p_i = \frac{e^{7r_i}}{\sum_{j=1}^{k} e^{7r_j}}, \qquad (1)$$

where the constant 7 is suggested based on empirical results.
5) Once a decomposer s_i is chosen, the decision variables are first randomly reordered and then are divided into groups of size s_i.
6) Each subcomponent is optimized for a predetermined number of iterations.

[1]"*everything should be made as simple as possible, but not simpler*". Source: of unknown origin, but often attributed to Albert Einstein.

7) The performance record of the ith decomposer is updated as follows:

$$r_i = \frac{(f - f')}{|f|}, \tag{2}$$

where f is the objective value of the best individual in the current cycle and f' is the objective value of the best individual in the previous cycle.

8) Go back to step 4 until a stopping criterion is met.

On CEC'2008 [23] benchmark problems, MLCC has shown significant improvement over random and static decomposition techniques [5]. However, the experimental results reported in [5] suggest that the performance of MLCC is close to other methods on fully-separable functions. We speculate that the improvement in performance is largely attributed to the decomposition of non-separable variables and MLCC performs poorly on fully-separable functions. In this paper, we compare the performance of MLCC with 10 different static approaches and show that MLCC is not effective when dealing with large-scale fully-separable functions. In the next section we show how value action method and softmax action selection rule can be used to improve the performance of MLCC on separable problems.

III. Proposed Algorithm

In this section we propose a simple modification to the MLCC algorithm that can significantly improve its performance. We apply some simple methods taken from reinforcement learning – called action value method and softmax action selection rule [20] – to improve the performance of MLCC.

Broadly speaking, every reinforcement learning problem has the following major components [20]:

- *policy*: determines how an agent behaves in an environment;
- *reward function*: assigns a reward to a given action. In other words, it measures the desirability of a given action;
- *value function*: unlike a reward function that measures the immediate utility of an action, a value function measures the long-term utility of an action;
- *model*: a model of the environment helps an agent determining the next state and its reward given a state and action. Models are used for planning and predicting the future.

The main goal of reinforcement learning is to find a policy so that an agent, in an uncertain environment, can maximize its long-term reward. In the context of optimizing fully-separable functions, this translates into finding a strategy for selecting a decomposition over the course of evolution in order to maximize the overall optimization performance.

In MLCC the selection policy is determined probabilistically by Equation (1). In the context of reinforcement learning the softmax action selection method uses a Gibbs distribution which is very similar to Equation (1) and has the following general form:

$$p_i = \frac{e^{V_t(i)/\tau}}{\sum_{j=1}^{k} e^{V_t(j)/\tau}}, \tag{3}$$

where $V_t(i)$ is the value function that determines the long-term value of ith action, and the parameter τ determines the balance between exploration and exploitation. A larger value of τ makes all actions almost equiprobable, whereas when $\tau \to 0$ the algorithm greedily chooses the action that it perceives to be the best. A typical way of estimating the value of an action is to take the arithmetic mean of all rewards received when a particular action was taken.

$$V_t(i) = \frac{r_{1i} + \cdots + r_{qi}}{q_i}, \tag{4}$$

where r_{ji} is the reward received when ith action was taken at jth time step for $j \in \{1, \ldots, q\}$.

Equation (2) can be regarded as a reward function that measures the immediate utility of the ith decomposer at an arbitrary iteration (time step). It is clear that MLCC uses the reward function in place of the value function. This makes MLCC very greedy as it only uses the immediate rewards for action selection (selecting a decomposer). MLSoft on the other hand uses action values as determined by Equation (4) for assigning probabilities to decomposers. In order to avoid keeping an archive of rewards for all decomposers, the value function can be incrementally updated as follows:

$$V_{t+1}(i) = \frac{r_{1i} + \cdots + r_{(q_i+1)}}{(q_i + 1)} = \frac{q_i V_t(i) + r_{(q_i+1)}}{(q_i + 1)}. \tag{5}$$

High-level steps of the MLSoft algorithm can be summarized as follows:

1) The initial random population is created and evaluated.
2) The vector of subcomponent sizes is initialized to the values provided by the user: $S = (s_1, \ldots, s_k)$, where s_i is the ith decomposer, and k is the number of available decomposers.
3) A vector is created for maintaining the performance of each subcomponent: $R = (r_1, \ldots, r_k)$. The performance records r_i are initialized to 0. This results in a uniform distribution for the first iteration.
4) A decomposer is randomly chosen from S with a probability drawn from a Gibbs distribution defined in Equation (3).
5) Once a decomposer s_i is chosen, the decision variables are divided into groups of size s_i.
6) Each subcomponent is optimized for a predetermined number of iterations.
7) The performance record of the ith decomposer is updated according to Equation (2), and the value function is updated according to Equation (5).
8) Go back to step 4 until a stopping criterion is met.

IV. Experimental Results and Analysis

In this section we compare the performance of MLCC against DECC with different subcomponents sizes. We show empirically how an effective subcomponent size for a CC framework can be determined. Finally, we compare the performance of MLCC with MLSoft which is proposed in this paper.

TABLE I. A LIST OF FULLY-SEPARABLE AND SCALABLE BENCHMARK PROBLEMS.

Function	Equation	Domain	Optimum
Sphere Function	$f_1(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$	$[-100, 100]^n$	$\mathbf{x}^* = \mathbf{0}, f_1(\mathbf{x}^*) = 0$
Quadratic Funciton	$f_2(\mathbf{x}) = \sum_{i=1}^{n} i x_i^4 + \mathcal{N}(0,1)$	$[-100, 100]^n$	$\mathbf{x}^* = \mathbf{0}, f_2(\mathbf{x}^*) = 0$
Elliptic Function	$f_3(\mathbf{x}) = \sum_{i=1}^{n} 10^{6\frac{i-1}{n-1}} x_i^2$	$[-100, 100]^n$	$\mathbf{x}^* = \mathbf{0}, f_3(\mathbf{x}^*) = 0$
Rastrigin's Function	$f_4(\mathbf{x}) = \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right]$	$[-5, 5]^n$	$\mathbf{x}^* = \mathbf{0}, f_4(\mathbf{x}^*) = 0$
Ackley's Function	$f_5(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e$	$[-32, 32]^n$	$\mathbf{x}^* = \mathbf{0}, f_5(\mathbf{x}^*) = 0$
Schwefel's Function	$f_6(\mathbf{x}) = 418.9829n - \sum_{i=1}^{n} x_i \sin(\sqrt{\lvert x_i \rvert})$	$[-512, 512]^n$	$\mathbf{x}^* = \mathbf{1}, f_6(\mathbf{x}^*) = 0$
Sum of Different Powers	$f_7(\mathbf{x}) = \sum_{i=1}^{n} \lvert x_i \rvert^{i+1}$	$[-1, 1]^n$	$\mathbf{x}^* = \mathbf{0}, f_7(\mathbf{x}^*) = 0$
Styblinski-Tang Function	$f_8(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n} (x_i^4 - 16x_i^2 + 5x_i) + 38.16599n$	$[-5, 5]^n$	$\mathbf{x}^* = -2.903534 \times \mathbf{1}$ $f_8(\mathbf{x}^*) = 0$

TABLE II. MEDIAN OF 25 INDEPENDENT RUNS. BEST RESULTS ARE HIGHLIGHTED (PAIR-WISE MWW WITH HOLM CORRECTION USING $\alpha = 0.05$).

$s \times d$	1000×1	100×10	10×100	1×1000	200×5	20×50	2×500	500×2	50×20	5×200	MLCC
f_1	4.25e-02	3.98e-05	**4.52e-13**	5.17e+00	2.16e-04	5.12e-11	1.92e-04	3.79e-05	7.73e-07	3.86e-11	1.76e-02
f_2	1.07e+02	2.78e+01	6.35e+00	1.15e+05	4.34e+01	1.08e+01	4.90e+01	6.71e+01	1.97e+01	**4.16e+00**	9.33e+01
f_3	3.14e+03	3.05e+00	**2.24e-08**	9.58e+03	1.54e+01	3.10e-06	1.48e+00	2.56e+00	5.58e-02	2.33e-06	1.55e+03
f_4	1.24e+00	6.95e+02	1.44e+03	1.37e+03	7.04e+01	1.66e+03	1.09e+03	**5.06e-02**	1.35e+03	1.08e+03	7.46e-01
f_5	8.93e-03	2.59e-04	**4.89e-08**	1.17e+01	5.93e-04	3.14e-07	7.63e+00	2.57e-04	3.83e-05	2.52e+00	5.51e-03
f_6	1.93e+00	9.80e+00	5.94e+04	1.20e+05	1.87e-01	4.61e+04	8.76e+04	**1.57e-02**	3.31e+03	4.93e+04	1.05e+00
f_7	1.72e-05	1.64e-15	**1.50e-30**	6.42e-24	9.94e-11	**4.19e-30**	1.40e-25	3.41e-08	6.80e-23	**2.77e-28**	1.09e-03
f_8	3.45e-03	2.52e-05	7.21e+02	4.75e+03	3.41e-05	8.48e+01	3.82e+03	**3.24e-06**	6.45e-06	1.99e+03	1.59e-03

A. Benchmark Problems and Parameter Settings

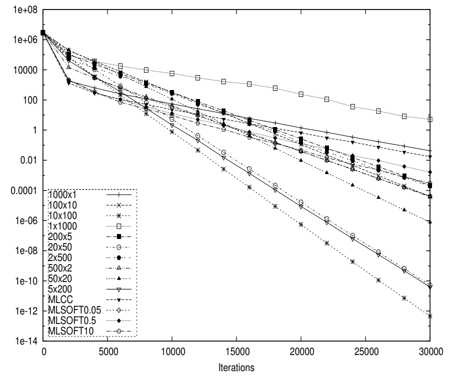
The experimental results in this paper are based on eight fully-separable functions which are listed in Table I. Functions f_1 and f_2 are from De Jong suite [24], and the remaining are commonly used functions for benchmarking continuous optimization algorithms [18], [25], [26]. Functions f_1, f_3 and f_7 are unimodal and the remaining five functions are multi-modal. The scalability of these functions make them ideal for the purposes of this study. The total number of fitness evolutions FE used in this study is set to $3e + 6$. Various decomposers that is used in this study are as follows $S = \{1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$. These values that represent low, medium and high dimensional subcomponent sizes allow us to *approximately* determine the optimal subcomponent size.

B. Results and Analysis

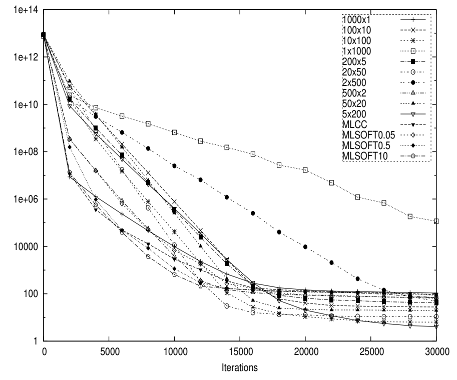
Table II shows the median of the final results obtained by 25 independent runs of DECC with different subcomponent sizes and MLCC. For testing the statistical significance of the results, first the Kruskal-Wallis one-way ANOVA [27] is used to find out if there is any significant difference between the algorithms. If a significant difference is detected, then a series of pair-wise Mann-Whitney-Wilcoxon (MWW) tests are conducted with Holm correction [27] in order to find the best performing algorithm. Holm correction is a simple technique for controlling the family-wise error rate. Family-wise error rate is the accumulation of type I errors when more than one pair-wise comparison is used to rank a set of results. All statistical tests are based on 5% significance level. For each function, the statistically best results are marked in bold. From the table, one can see that 100 is the best subcomponent size [2], which led to statistically best results on 4 from a total 8 functions. On function f_2 subcomponent size of 200 is the best performing subcomponent size, followed by the subcomponent size of 100 (4.16e+00 versus 6.35e+00). The subcomponent size of 2 comes next, showing statistically best performance on 3 functions. On f_7 three different subcomponent sizes (50, 100 and 200) resulted in statistically similar results. However, the subcomponent size of 100 still has the best median value. Overall, on the eight benchmark functions, one can see that the best subcomponent size is either 100 or 2. Figure 1 shows the convergence curves of the median of the compared algorithms on all the benchmark functions. From the figure, one can see that the convergence curve of the best subcomponent size is not always the lowest throughout the entire search process. For example, on f_1, the convergence curve of the best DECC with 10×100 is above a number of other compared algorithms (even including 1000×1) before $5000th$ iteration. This indicates a potential to adaptively change the subcomponent size (e.g., 1 for the first 5000 iterations and 100 afterwards for f_1) to further enhance the performance of the algorithm.

Table II shows that the best subcomponent size is not consistent over all the benchmark functions. Thus, it is necessary to investigate the factors that may affect the best subcomponent size. One intuitive factor is the behavior of the subcomponent optimizer, which is problem dependent and largely determines the performance of the overall CC framework. Since SaNSDE was selected as the subcomponent optimizer for this study, we investigated its behavior on the benchmark functions and derive its relationship with the best subcomponent size shown

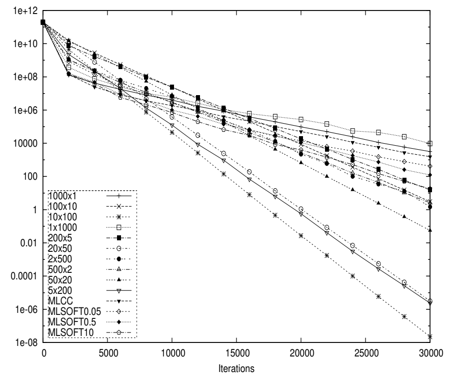
[2]best among the available subcomponent sizes defined in S.



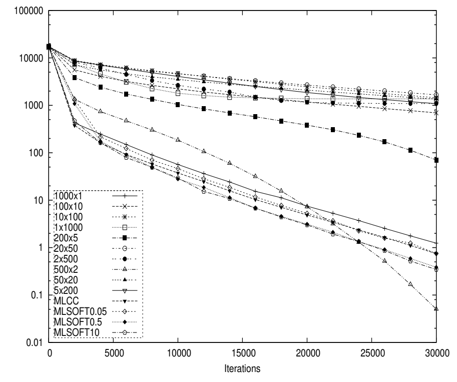
(a) f_1



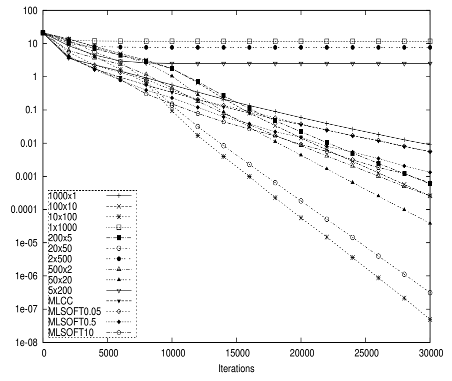
(b) f_2



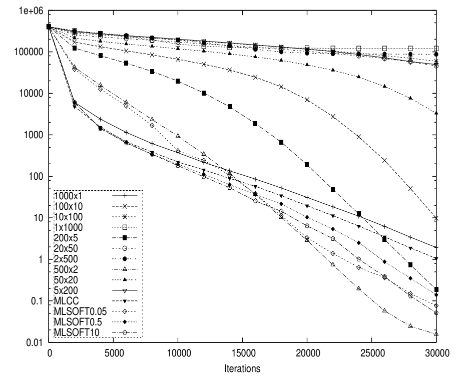
(c) f_3



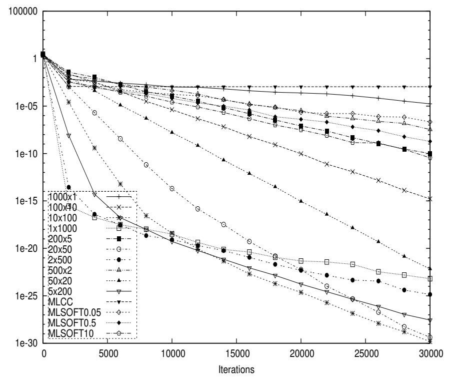
(d) f_4



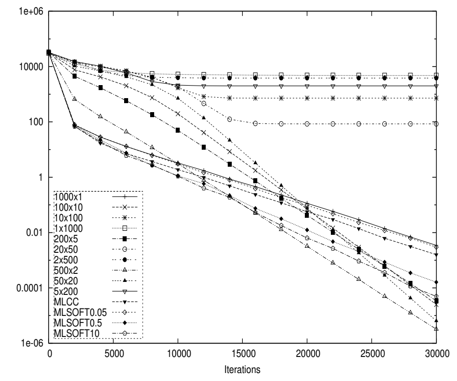
(e) f_5



(f) f_6



(g) f_7



(h) f_8

Fig. 1. Convergence plots of different algorithms on fully-separable functions using $3e6$ fitness evolutions. Each iteration is equal to 100 fitness evolutions. Each point is the median of 25 independent runs.

in Table II. To this end, a performance metric normalized by the dimension of problem is required for a fair comparison of SaNSDE's performance on various dimensions.

Normalized performance ϕ: Assuming that the number of fitness evaluations required for an algorithm to solve a given 1-dimensional optimization problem to an acceptance level (e.g., an average deviation of 10^{-10} from the global optimum) is c, then the normalized performance ϕ of the algorithm on the corresponding n-dimensional fully-separable optimization problem is defined as the performance (e.g., mean and median) given $c \times n$ fitness evaluations. This can be considered as optimizing n independent 1-dimensional problems. Therefore, $c \times n$ is an approximation to the number of fitness evaluations required. It should be noted that this linear approximation is accurate for small n where the problem size does not significantly affect the search efficiency of the algorithm and thus more fitness evaluations may lead to better results through more comprehensive exploitation. However, it is known that the relationship is nonlinear due to the issue of scalability when n becomes large. In other words, for large n the assumption that the number of required fitness evolutions for solving a problem grows linearly with respect to n becomes invalid, and we expect to see a significant increase in the value of ϕ for large values of n. Therefore, when we plot the value of ϕ with respect to n we expect to see an initial decrease in the value of ϕ as n is increased followed by a relatively sharp increase in ϕ for larger values of n. The plots shown in Figure 2 are constructed using this method and the behavior mentioned above can be clearly seen. This figure can be used to approximately find the optimal dimension size that an optimizer can solve most efficiently. More specifically, the size (n) for which ϕ is minimum are the best subcomponent sizes for a give problem.

In the DECC framework on fully-separable functions, the overall fitness evaluations are uniformly allocated to each of the equally-sized subcomponents. The number of fitness evaluations assigned to each subcomponent is thus dependent on the number of subcomponents, or the size of each subcomponent. Supposing that the original n-dimensional decision vector are decomposed into s d-dimensional subcomponents ($n = s \times d$), then the number of fitness evaluations assigned to each subcomponent is $\frac{FE}{s} = \frac{FE \times d}{n}$, where FE is the overall fitness evaluations. It is clear that the performance of DECC on fully-separable functions mainly depends on the performance of the subcomponent optimizer given the assigned fitness evaluations of $\frac{FE \times d}{n}$. Such performance is consistent with the defined normalized performance ϕ by setting $c = \frac{FE}{n}$.

To make ϕ consistent with the performance of DECC, it is specifically defined as follows:

$$\phi_{\text{median}} = s \times \texttt{median}\left(\frac{FE \times d}{n}, 25\right) \tag{6}$$

where $\texttt{median}(\frac{FE \times d}{n}, 25)$ is the median of the final results over 25 independent runs given $\frac{FE \times d}{n}$ fitness evaluations. This is because the final result obtained by DECC is the sum of the results of all the subcomponents, which can be approximated by s times the result of each subcomponent.

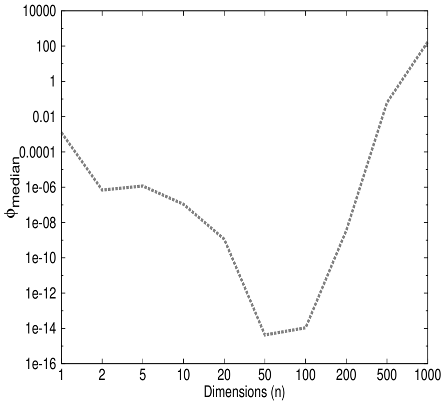
Figure 2 shows ϕ_{median} of SaNSDE for different dimensions of all the benchmark functions. It can be seen that for all of the benchmark functions, there is an obvious trend that ϕ_{median} first decreases and then increases with the increase of dimension of the function. The only exception is f_2, for which ϕ_{median} keeps constant up to 200 dimensions, and then increases. More importantly, the best subcomponent size shown in Table II is highly consistent with the bottom of the curve. For almost all functions, the best or the second best performing dimensions in Figure 2 match the best performing subcomponent size in Table II (100 for f_1, f_3, f_5 and f_7; 2 for f_4 and f_6). The only exception is f_8 for which the best subcomponent size of 2 is far away from the bottom of the curve which occurs at 50. It is noteworthy that according to Table II, the subcomponent size of 50 is the second best performing subcomponent size. Such inconsistency might be caused by large variance of the final results for problems with higher number of dimensions.

TABLE III. MEDIAN OF 25 INDEPENDENT RUNS. COMPARISON WITH MLCC AS THE CONTROL GROUP USING PAIR-WISE MWW ($\alpha = 0.05$).

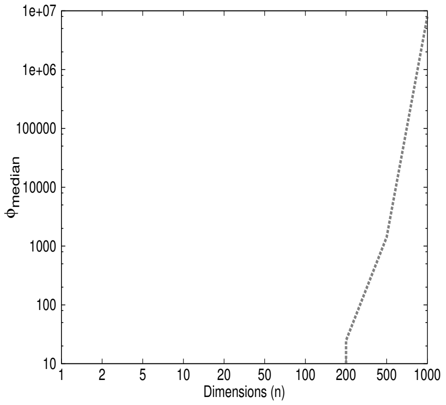
	MLCC	MLSoft ($\tau = 0.05$)	MLSoft ($\tau = 0.5$)	MLSoft ($\tau = 10$)	DECC's Best Performing Decomposer
f_1	1.76e-02	**4.16e-05**	**1.65e-03**	**3.02e-04**	4.52e-13 (10×100)
f_2	9.33e+01	6.51e+01	8.74e+01	**6.97e+01**	4.16e+00 (5×200)
f_3	1.55e+03	4.19e+02	**1.23e+02**	**1.82e+01**	2.24e-08 (10×100)
f_4	7.46e-01	7.49e-01	**3.85e-01**	**3.45e-01**	5.06e-02 (500×2)
f_5	5.51e-03	5.56e-03	**1.33e-03**	**6.30e-04**	4.89e-08 (10×100)
f_6	1.05e+00	7.62e-02	**1.39e-01**	**5.14e-02**	1.57e-02 (500×2)
f_7	1.09e-03	**2.18e-07**	**1.91e-09**	**3.53e-11**	1.50e-30 (10×100)
f_8	1.59e-03	2.96e-03	**1.61e-04**	**4.92e-05**	3.24e-06 (500×2)

In summary, the above analysis shows that when applying DECC to solve large-scale fully-separable optimization problems, an effective subcomponent size is neither too small nor too large, and is dependent on the performance of the subcomponent optimizer SaNSDE normalized by the problem dimension, ϕ_{median}, as defined in Equation (6). However, in practice, such performance of subcomponent optimizer is not often known in advance. In this case, adaptively identifying an effective subcomponent size during the optimization process becomes desirable, thereby the MLSoft approach is proposed.

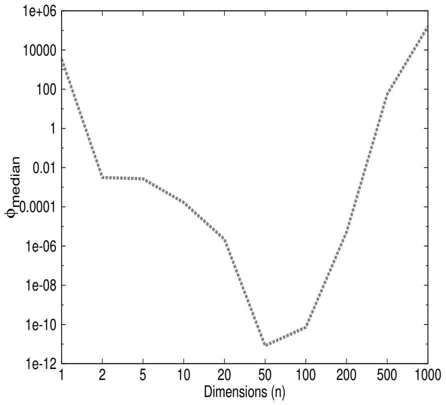
Table III shows the median of the results of MLCC and MLSoft with different τ's over 25 independent runs, along with the state-of-the-art results obtained by the best subcomponent size shown in Table II. For each version of MLSoft, the results that are statistically better than the results of MLCC are marked in bold. Here the Holm correction is not required because all algorithms are compared with MLCC as a control group. The last column is included for reference and was not included in the statistical tests. The table clearly shows that all versions of MLSoft outperform MLCC. MLSoft with $\tau \in \{0.5, 10\}$ performed the best, obtaining significantly better results on almost all benchmark functions. Although MLSoft with $\tau = 0.05$ performed significantly better than MLCC on only two functions, most of its median values are lower than MLCC. The reason for this behavior can be attributed to higher variance of the results when $\tau = 0.05$. This is intuitive because a lower value of τ results in a more exploitative behavior which can potentially increase the standard deviation of the final results. It was mentioned previously that MLCC uses the reward function in place of the value function. This means that MLCC behaves greedily and tries to maximize the immediate reward that may not result in a high long-term reward. According to Equation (2) the parameter τ for MLCC is $\frac{1}{7} \approx 0.15$. From the results of Table III we see that



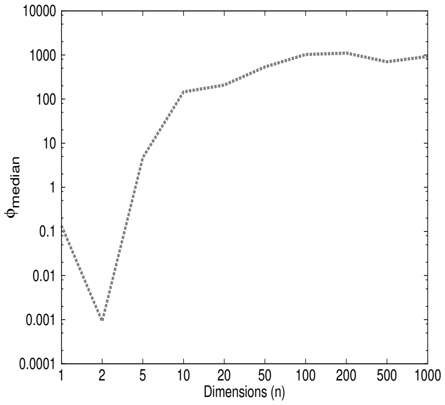
(a) f_1



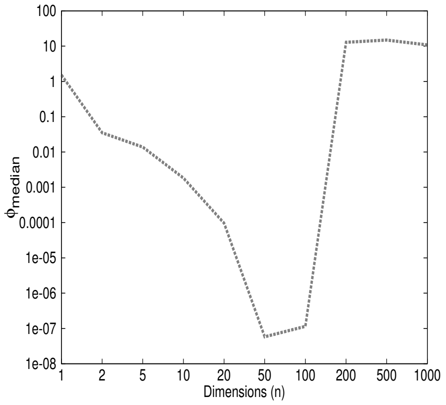
(b) f_2



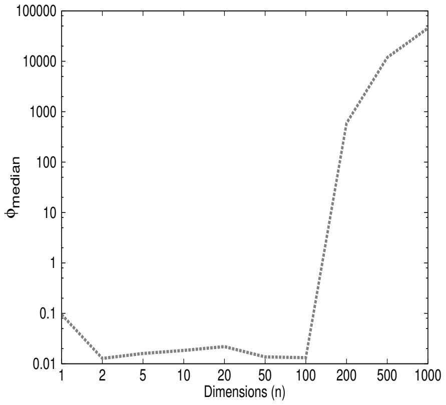
(c) f_3



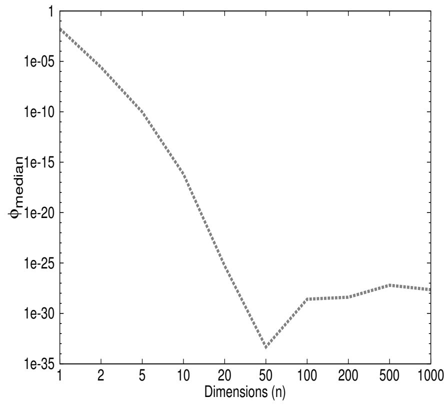
(d) f_4



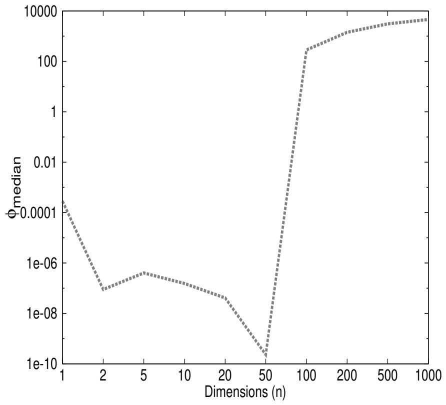
(e) f_5



(f) f_6



(g) f_7



(h) f_8

Fig. 2. Scalability plots of SaNSDE on different fully-separable functions. Each point on the plots is calculated based on the normalized performance ϕ as defined in Equation (6) where $FE = 3e6$ using 25 independent runs.

higher values of τ which results in higher exploratory power generally perform better. However, MLCC with $\tau \approx 0.15$ performs worse than MLSoft with $\tau = 0.05$. This shows that using a proper value function in order to estimate the long-term rewards is essential. Finally, the table shows that although three versions of MLSoft outperformed MLCC, they still perform worse than DECC with best obtained subcomponent size. The reason for this behavior might be related to the non-stationary nature of the problem. In general, a non-stationary problem is the one in which the true values of its actions change over time. In this context non-stationary means that the performance of different decomposers changes over time. This is evident from the convergence plots shown in Figure 1. The technique of averaging (Equation (4)) is suitable for stationary problems. For a non-stationary problem it is more intuitive to assign a higher weight to more recent rewards. Finding better methods for dealing with non-stationary nature of this problem requires a separate study. Nevertheless, we have shown in this paper that reinforcement learning techniques have great potential for adapting the subcomponent size for fully-separable problems.

V. Conclusion

In this paper we have shown that different subcomponent sizes can have significant impact on the performance of a CC framework on large-scale fully-separable continuous optimization problems. We have shown empirically that the two extreme decompositions i.e. $n \times 1$ and $1 \times n$ are often the worst performing decompositions and should be avoided in a CC framework. We have seen that, in general, the subcomponents should be neither too small nor too large. The empirical results suggest that the rule of thumb for selecting a subcomponent size is to choose it small enough so that it is within the capacity of the subcomponent optimizer, but it should not be made any smaller. It should be noted that in practice the capacity of the subcomponent optimizer on various problems is often unknown. Therefore, adaptively identifying the an effective subcomponent size during the optimization process is desirable, thereby the MLSoft approach has been proposed. The comparative study with MLCC has shown that algorithms with higher exploratory power generally perform better. We have also seen that the use of a value function for estimating long-term rewards also significantly improves the performance of MLSoft. However, due to non-stationary nature of the problem there is a large gap between the performance of MLCC and the best performing decomposer. Finding better value functions for dealing with the non-stationary aspect of the problem is the subject of our future investigations.

Acknowledgement

This work was partially supported by an ARC Discovery grant (No. DP120102205).

References

[1] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *Proc. of International Conference on Parallel Problem Solving from Nature*, vol. 2, 1994, pp. 249–257.

[2] F. van den Bergh and A. P. Engelbrecht, "A cooperative approach to particle swarm optimization," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 3, pp. 225–239, 2004.

[3] Z. Yang, K. Tang, and X. Yao, "Large scale evolutionary optimization using cooperative coevolution," *Information Sciences*, vol. 178, pp. 2986–2999, August 2008.

[4] Y. Liu, X. Yao, Q. Zhao, and T. Higuchi, "Scaling up fast evolutionary programming with cooperative coevolution," in *Proc. of IEEE Congress on Evolutionary Computation*, 2001, pp. 1101–1108.

[5] Z. Yang, K. Tang, and X. Yao, "Multilevel cooperative coevolution for large scale optimization," in *Proc. of IEEE Congress on Evolutionary Computation*, June 2008, pp. 1663–1670.

[6] D. Sofge, K. D. Jong, and A. Schultz, "A blended population approach to cooperative coevolution fordecomposition of complex problems," in *Proc. of IEEE Congress on Evolutionary Computation*, 2002, pp. 413–418.

[7] M. N. Omidvar, X. Li, and X. Yao, "Smart use of computational resources based on contribution for cooperative co-evolutionary algorithms," in *Proc. of Genetic and Evolutionary Computation Conference*. ACM, 2011, pp. 1115–1122.

[8] ——, "Cooperative co-evolution with delta grouping for large scale non-separable function optimization," in *Proc. of IEEE Congress on Evolutionary Computation*, 2010, pp. 1762–1769.

[9] M. N. Omidvar, X. Li, Z. Yang, and X. Yao, "Cooperative co-evolution for large scale optimization through more frequent random grouping," in *Proc. of IEEE Congress on Evolutionary Computation*, 2010, pp. 1754–1761.

[10] M. N. Omidvar, X. Li, Y. Mei, and X. Yao, "Cooperative co-evolution with differential grouping for large scale optimization," *Evolutionary Computation, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013.

[11] X. Li and X. Yao, "Cooperatively coevolving particle swarms for large scale optimization," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 2, pp. 210–224, April 2012.

[12] W. Chen, T. Weise, Z. Yang, and K. Tang, "Large-scale global optimization using cooperative coevolution with variable interaction learning," in *Proc. of International Conference on Parallel Problem Solving from Nature*, ser. Lecture Notes in Computer Science, vol. 6239. Springer Berlin / Heidelberg, 2011, pp. 300–309.

[13] X. Li and X. Yao, "Tackling high dimensional nonseparable optimization problems by cooperatively coevolving particle swarms," in *Proc. of IEEE Congress on Evolutionary Computation*, 2009, pp. 1546–1553.

[14] Y. Shi, H. Teng, , and Z. Li, "Cooperative co-evolutionary differential evolution for function optimization," in *Proc. of International Conference on Natural Computation*, 2005, pp. 1080–1088.

[15] A. Zamuda, J. Brest, B. Boskovic, and V. Zumer, "Large scale global optimization using differential evolution with self-adaptation and cooperative co-evolution," in *Evolutionary Computation (CEC), 2008 IEEE Congress on*, 2008, pp. 3718–3725.

[16] Y. Davidor, "Epistasis Variance: Suitability of a Representation to Genetic Algorithms," *Complex Systems*, vol. 4, no. 4, pp. 369–383, 1990.

[17] T. Weise, R. Chiong, and K. Tang, "Evolutionary Optimization: Pitfalls and Booby Traps," *Journal of Computer Science and Technology (JCST)*, vol. 27, no. 5, pp. 907–936, 2012, special Issue on Evolutionary Computation.

[18] X. Li, K. Tang, M. N. Omidvar, Z. Yang, and K. Qin, "Benchmark functions for the CEC'2013 special session and competition on large-scale global optimization," RMIT University, Melbourne, Australia, Tech. Rep., 2013, http://goanna.cs.rmit.edu.au/ xiaodong/cec13-lsgo.

[19] M. Tezuka, M. Munetomo, and K. Akama, "Linkage identification by nonlinearity check for real-coded genetic algorithms," in *Proc. of Genetic and Evolutionary Computation Conference*, ser. Lecture Notes in Computer Science, vol. 3103. Springer, 2004, pp. 222–233.

[20] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998.

[21] Z. Yang, K. Tang, and X. Yao, "Self-adaptive differential evolution with neighborhood search," in *Proc. of IEEE Congress on Evolutionary Computation*, 2008, pp. 1110–1116.

[22] R. Storn and K. Price, "Differential evolution . a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization 11 (4)*, pp. 341–359, 1995.

[23] K. Tang, X. Yao, P. N. Suganthan, C. MacNish, Y. P. Chen, C. M. Chen, , and Z. Yang, "Benchmark functions for the CEC'2008 special session and competition on large scale global optimization," Nature Inspired Computation and Applications Laboratory, USTC, China, Tech. Rep., 2007, http://nical.ustc.edu.cn/cec08ss.php.

[24] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, Ann Arbor, MI, USA, 1975.

[25] N. Hansen, S. Finck, R. Ros, and A. Auger, "Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions," INRIA, Tech. Rep. RR-6829, 2010.

[26] M. Molga and C. Smutnicki, "Test functions for optimization needs," 2005, www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf.

[27] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. CRC Press, 2003.