Group Contribution Cooperative Co-Evolution Framework for CEC'2013 Large-Scale Optimization Problems

Klemen Berkovič
klemen.berkovic1@um.si
Faculty of Electrical Engineering and Computer Science, University of Maribor
Koroška cesta 46
SI-2000 Maribor, Slovenia

Borko Bošković
borko.boskovic@um.si
Faculty of Electrical Engineering and Computer Science, University of Maribor
Koroška cesta 46
SI-2000 Maribor, Slovenia

Janez Brest
janez.brest@um.si
Faculty of Electrical Engineering and Computer Science, University of Maribor
Koroška cesta 46
SI-2000 Maribor, Slovenia

ABSTRACT

Solving black-box large-scale global optimization problems presents a significant challenge for conventional optimization algorithms. In this article, we introduce a new cooperative co-evolution framework and a modified component grouping algorithm. The new framework employs a divide-and-conquer strategy based on the modified component grouping algorithm. The modified component grouping algorithm breaks down a complex problem into manageable groups. These smaller groups are then optimized efficiently using an algorithm from the NiaPy framework. The proposed framework is highly versatile, it is capable of utilizing various optimization and component grouping algorithms. To assess its performance, we conducted a comparative study against multiple algorithms. We used fifteen benchmark functions from the CEC'2013 large-scale global optimization benchmark. Our findings highlight the potential of our framework in enhancing optimization algorithms' effectiveness when dealing with black-box large-scale global optimization problems.

KEYWORDS

Large-scale global optimization, black-box optimization, CEC'2013 LSGO, component grouping, cooperative co-evolution

1 INTRODUCTION

In the field of optimization, we encounter intricate challenges, demanding inventive solutions for real-world issues. Four interconnected domains stand out: 1) Black-Box (BB) optimization, 2) Large-Scale Global Optimization (LSGO), 3) Cooperative Co-evolution (CC) frameworks and 4) Component Grouping (CG) methods.

BB optimization deals with a hidden formulation of a fitness function, so it relies exclusively on evaluations of the fitness function, for finding solutions. This mirrors complex real-world scenarios where traditional methods have difficulties. LSGO tackles the complexities of a problem that has a high number of components and is very time-consuming to optimize. The high number of components need to be optimized in search of a good solution to the problem. LSGO problems can be found in fields like urban planning [2] and healthcare [10]. CG methods provide a crucial thread, disentangling a complex problem with overlapping components. Our goal is to combine the CG method within the CC framework to effectively optimize the BB LSGO problems. We will use the idea of recursive differential grouping as a CG method to generate groups that represent sub-problems for easier problem-solving. The proposed framework is implemented for the NiaPy optimization framework [11], which includes many nature-inspired algorithms. Our goal is to be able to use any optimization algorithm from that framework within the proposed CC framework.

The rest of the paper is organized as follows: The related work on used algorithms, CC and CG methods are introduced in Section 2. Then, the proposed CC framework and modified CG algorithm are described in Section 3. Experimental studies are presented in Section 4. Finally, conclusions and future work are described in Section 5.

2 RELATED WORK

In the field of tackling BB optimization challenges, CC frameworks have emerged as a powerful tool. In [7] authors used the combined strengths of CC framework with genetic algorithm (GA) to navigate the complexities inherent in such problems. The CCGA-1 algorithm was developed and tested on known optimization functions, where the number of components was set to 0 and 0. They showed that the CCGA-1 algorithm was better than the original GA algorithm.

The CG method focuses on efficient problem decomposition through problems' components interaction understanding. An effective CG method minimizes inter-group and maximizes intra-group interactions [12]. Two primary methods for detecting these interactions are: 1) non-monotonicity [5] and 2) non-linearity [9] detection.

Authors in [6] introduced a CC algorithm with a differential CG method for LSGO, laying the foundation for integrating a differential CG method as a central mechanism in the CC framework. They introduced the Differential Grouping algorithm, which uses non-linearity components interaction detection. With automated decomposition of the problem through CG, this pioneering approach set a precedent for subsequent research. They showed that a near-optimal CG can significantly improve the solution quality for the BB LSGO problems.

In [8] a recursive CG method for large-scale continuous optimization was introduced, called the Recursive Differential Grouping 3 (RDG3) algorithm. The RDG3 algorithm uses non-linearity detection to identify components interactions by detecting changes in a fitness function when perturbing components. The RDG3 algorithm showed that a recursive method can break down a complex optimization problem into more manageable sub-problems, without explicitly examining all pairwise components interactions. Notably, the RDG3 algorithm reduces the CG time complexity of n-dimensional problem to (n n). To validate the effectiveness

DOI: https://doi.org/10.26493/scores23.15

of the RDG3 algorithm, a CC algorithm with the RDG3 algorithm was used on CEC'2013 LSGO benchmark.

In summary, these works collectively illustrate the evolution of CC algorithms. They showcase the role of the CG in addressing the challenges of BB LSGO problems. Approaches like recursive CG, adaptive parameter estimation and handling overlapping components underline the versatility and effectiveness of the CC framework. Therefore CC frameworks are highly desirable for optimizing complex systems across various real-world applications.

3 METHODOLOGY

In this section, we describe our Group Contribution CC (GCCC) framework and the Modified Recursive Differential Grouping 3 (MRDG3) algorithm. The advantage of the MRDG3 algorithm is that it has fewer parameters than the RDG3 algorithm. The advantages of the GCCC framework are as follows: 1) it can use any optimization algorithm implemented in the Python programming language for the NiaPy optimization framework, 2) it is capable of using any CG method implemented in the Python programming language for the NiaPy optimization framework and 3) based on previous advantages it enables the rapid testing of new optimization and CG algorithms within the GCCC framework.

3.1 Modified Recursive Differential Grouping 3 algorithm

The MRDG3 algorithm was implemented in the Python programming language for the NiaPy optimization framework and is shown in Algorithm 1. The MRDG3 algorithm has four input parameters: 1) BB fitness function f, 2) lower bound of the search space , 3) upper bound of the search space and 4) threshold value ϵ_n. The function ℓ returns the number of components in a vector or a set. MRDG3 starts by checking interactions with component x , recursively identifying interacting components with algorithm `Interact` [8]. A threshold ϵ_n controls the group size for optimization, which can be seen in line 6. When all interactions between component x and the remaining components in set are identified, MRDG3 continues with the first remaining component in set , what is seen in line 12. Finally, the MRDG3 outputs set with separable components and set with non-separable component groups, what is seen in line 16.

Input: f, , , ϵ_n

1 , , $\mathbf{x}$, , $\mathbf{x}_{,}$, $y_{l,l} \leftarrow$ [], [], [1], [1], , f( );
2 $\mathbf{x}$ $\leftarrow$ [**for** i = **to** ℓ( ) **do** i];
3 **while** ***not*** *empty* **do**
4 $\leftarrow$ [];
5 $\mathbf{x}^*$ $\leftarrow$ `Interact`(f, , , $\mathbf{x}$, $\mathbf{x}$, $\mathbf{x}_{,}$, $y_{l,l}$,);
6 **if** $\ell(\mathbf{x}^*) < \epsilon_n$ ***and*** $\ell(\mathbf{x}^*)$ $\ell(\mathbf{x}$) **then**
7 $\mathbf{x}$, $\mathbf{x}$ = $\mathbf{x}^*$, ;
8 **if** ℓ( ) = 0 **then** `Append`(, $\mathbf{x}$) and **break**;
9 **else**
10 **if** $\ell(\mathbf{x}^*)$ = **then** `Append`(, $\mathbf{x}^*$) **else** `Append`(, $\mathbf{x}^*$);
11 **if** ℓ( ) > **then**
12 $\mathbf{x}$ $\leftarrow$ [[0]];
13 `Delete`(, *[0]*);
14 $\mathbf{x}$ $\leftarrow$;
15 **else if** ℓ( ) = **then** `Append`(, $\mathbf{x}$) and **break**;
16 **return** , ;

Algorithm 1: Modified Recursive Differential Grouping 3 (MRDG3) algorithm.

Input: f, , , ϵ_n

1 $\leftarrow$ `MRDG3`(f, , , ϵ_n); `// Algorithm 1`
2 , $\leftarrow$ [**foreach** g ***in*** **do** Initialize algorithm a with starting population based on group g];
3 * $\leftarrow$ Get best individuals from initialized populations for each algorithm based on fitness values;
4 $\mathbf{x}^*$ $\leftarrow$ Get best individual from * based on fitness values;
5 **while** *stopping condition meet* **do**
6 **for** i = **to** ℓ( ) **do**
7 [i], $\mathbf{x}^*$ $\leftarrow$ `RunGeneration`([i], [i], f, ,);
8 **if** $f(\mathbf{x}^*) < f($ $^*[i]$) **then**
9 $^*[i] \leftarrow \mathbf{x}^*$;
10 **if** $f(\mathbf{x}^*) < f(\mathbf{x}^*)$ **then** $\mathbf{x}^* \leftarrow \mathbf{x}^*$;
11 **if** *Any group found new local best individual* **then**
12 **foreach** g ***in*** **do** $\mathbf{x}^*[g] \leftarrow$ $^*[g]$;
13 **if** $f(\mathbf{x}^*) < f(\mathbf{x}^*)$ **then** $\mathbf{x}^* = \mathbf{x}^*$;
14 **return** $\mathbf{x}^*$, $f(\mathbf{x}^*)$;

Algorithm 2: Group Contribution Cooperative Co-evolution (GCCC) framework.

3.2 Group Contribution Cooperative Co-evolution framework

As the CG method aims to decompose and divide the BB LSGO problem into smaller subgroups for smarter optimization, this part describes the GCCC framework. GCCC uses decomposition information for optimization algorithms initialization, algorithms population initialization, and for running a generation of an algorithm. Our proposed GCCC framework's pseudocode is shown in Algorithm 2. GCCC was implemented in the Python programming language for the NiaPy optimization framework.

The GCCC framework works as follows. First, groups of components for independent optimizations are retrieved with the help of the CG method, seen in line 1. In line 2, algorithms and starting populations are initialized based on groups . All algorithms initialize their initial population, which is used and updated by the algorithms during the execution of one generation of the algorithm. In line 3 for each initialized population local best individual is found so that every algorithm has it's own local best individual. Then global best individual is found from all the local best individuals. Search for global best is faster because we are using only a small part of all individuals, which is depicted in line 4.

Our main novelty of the GCCC framework is seen when the optimization stage begins in line 5. In line 7, we perform only one generation of the algorithm a. After the algorithm a ends a generation, a new population for the algorithm a is returned with the new local best individual $\mathbf{x}^*$. Line 10 checks if the new local best individual is the new global best individual and updates the global best individual if that is true. After all algorithms in finish their own generation, line 11 checks if the new individual has to be constructed. If so, the new individual is composed of all local

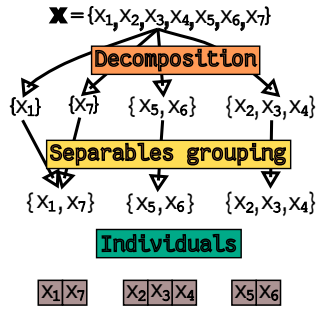

Figure 1: Components grouping and individual construction for Equation 1.

best individuals * components values, which is seen in line 12. Line 13 updates the global best individual if the composed individual is better than the current global best individual. After the GCCC returns to line 5 and starts the new iteration of the algorithm and conducts previous steps until the stopping condition is not achieved.

To demonstrate how the GCCC constructs the individuals of optimization algorithms and how it evaluates an individual, let's take the optimization problem formulated as:

$$f(\mathbf{x}) = x \quad (x \ -x\ ) \quad (x \ -x\ ) \quad (x \quad x\ ) \quad x\ . \qquad (1)$$

We will optimize Equation 1 as a BB problem with a lower bound equal to − and an upper bound equal to for all seven components. First, the GCCC decomposes the problem into four groups and then joins separable components into one group. So after the decomposition stage, we have three groups. This is depicted in Figure 1. In the initialization stage, three algorithms are initialized with initial populations. Each algorithm in initialized individuals with the length of one individual equal to the number of components in a group that the algorithm is going to optimize. When the algorithm a in runs one generation of the optimization algorithm, fitness function f, fills the missing components with the lower bound of the search space. In this example, for the first algorithm, which is optimizing group containing components x and x , components x , x , x , x and x are filled with value − . In line 12 the GCCC constructs an individual $\mathbf{x}^*$ that has all seven components filled. So when evaluating an individual $\mathbf{x}^*$ at line 13, there is no need to fill any components.

4 EXPERIMENT

In this section, the results of the GCCC framework with two different optimization algorithms from the NiaPy framework are presented. Test functions were taken from the CEC'2013 LSGO benchmark competition [3] and are implemented in the C++ programming language. Python code of the performed experiments in this work is available via a connection[1]. For comparison, we used algorithms PSO [1] and SCA [4], which are implemented in the NiaPy framework. Algorithms' parameters are presented in Table 1 and were set based on experimental results on test functions that are part of the NiaPy framework.

In Table 2, the obtained results of CG for each test function in the CEC'2013 LSGO benchmark are shown. The results show that

[1]https://github.com/kb2623/scores23

Table 1: Algorithms' parameters for conducted experiment.

Algorithm	Parameter values
MRDG3	$\epsilon_n =$ 0
SCA	= , $a =$, r = 0, $r_x =$
PSO	= , c = , c = , $w = 0.$, v = − . , $v_x =$.

Table 2: Results of the CG for the MRDG3 algorithm on the CEC'2013 LSGO benchmark functions.

	Number of groups	Number of separable components	Number of evaluations
f	1	0	5992
f	0	1000	2998
f	2	1	5987
f	1	0	5992
f	6	800	8287
f	5	750	8569
f	1	0	5992
f	1	0	5992
f	1	0	5992
f_0	21	151	19357
f	1	0	5992
f	1	0	5992
f	1	0	5992
f	1	0	5992
f	1	0	5992

the number of evaluations is fairly small. MRDG3 does not need n number of evaluations to decompose a BB problem, where n represents the number of components and for the CEC'2013 LSGO benchmark $n =$ 000. For fully separable functions, only f is 00 correct. The result of decomposition on function f is missing seven groups. MRDG3 has incorrect results on function f_0 because this function has no separable sub-components. CG results for functions f , f and f are 00 correct, because functions have overlapping sub-components. The result of decomposition for the function f is 00 correct because that function is fully non-separable.

For each function from the CEC'2013 LSGO benchmark, we performed 50 runs for each algorithm. Results are shown in Table 3 where we reported three values for each benchmark function. The top value in each cell represents the minimum, the middle value is the median and the bottom value is the standard deviation. Underlined values represent better results when comparing the basic algorithm to its CC-based algorithm. For each of the algorithms used with the GCCC framework, we added a sign that indicates if the GCCC framework is better ( ), similar ( ), or worse (−). This assumption was made based on the Wilcoxon signed-rank test from SciPy[2]. We used $\alpha = 0.0$ for detecting statistical differences. The only similarity we detected was between algorithm SCA and SCA-GCCC on function f . From the last row in Table 3, we can see that SCA-GCCC compared to SCA is better on eleven functions, worse on three functions, and one function had a similar result. When comparing PSO-GCCC to PSO, we detected that PSO-GCCC was better on twelve functions, and on three functions was worse than PSO.

[2]https://scipy.org/

Table 3: Algorithms results for the CEC'2013 LSGO benchmark.

	SCA	SCA-GCCC		PSO	PSO-GCCC	
	3.2116e+11	2.0983e+11		2.2449e+11	1.8236e+11	
f	3.5997e+11	2.0983e+11		2.5767e+11	2.0881e+11	
	1.7041e+10	6.1654e-05		1.4521e+10	4.4700e+09	
	1.0988e+05	4.7598e+04		1.8498e+04	2.0991e+04	
f	1.1765e+05	4.7598e+04		2.0399e+04	2.4994e+04	–
	4.0256e+03	7.3498e-12		9.9743e+02	1.7148e+03	
	2.1625e+01	2.1625e+01		2.1540e+01	2.1514e+01	
f	2.1660e+01	2.1659e+01		2.1565e+01	2.1559e+01	
	1.0164e-02	1.0796e-02		1.4204e-02	1.8726e-02	
	6.7784e+12	1.4182e+13		4.5338e+12	4.1230e+12	
f	3.0918e+13	3.4167e+13	–	7.5541e+12	6.9831e+12	
	9.6588e+12	1.1140e+13		1.2660e+12	1.6166e+12	
	5.4753e+07	4.8419e+07		5.5953e+06	5.1143e+06	
f	7.0849e+07	4.8419e+07		8.9464e+06	8.2784e+06	
	5.2834e+06	0.0000e+00		1.9627e+06	1.7852e+06	
	1.0688e+06	1.0688e+06		1.0530e+06	1.0518e+06	
f	1.0717e+06	1.0704e+06		1.0587e+06	1.0594e+06	–
	1.8314e+03	6.2956e+02		1.8514e+03	2.4651e+03	
	5.3201e+14	9.9382e+14		1.5502e+14	1.0261e+14	
f	8.3849e+15	9.9382e+14		4.3684e+14	3.4803e+14	
	6.1855e+15	0.0000e+00		2.3930e+14	1.8312e+14	
	9.6025e+17	9.6025e+17		1.5852e+17	1.0169e+17	
f	1.8327e+18	1.9011e+18	–	3.6707e+17	3.0508e+17	
	5.9387e+17	5.6790e+17		1.0251e+17	8.8785e+16	
	4.0665e+09	4.0665e+09		4.6629e+08	5.0571e+08	
f	5.7516e+09	5.6652e+09		6.8898e+08	7.7170e+08	–
	7.2037e+08	5.1872e+08		1.3255e+08	1.4362e+08	
	9.4583e+07	9.4583e+07		9.2361e+07	9.2574e+07	
f_0	9.6401e+07	9.5732e+07		9.3658e+07	9.4114e+07	–
	4.8204e+05	1.8366e+05		4.2994e+05	4.4469e+05	
	6.3357e+16	6.3357e+16		1.0835e+16	8.1720e+15	
f	6.4911e+17	1.0331e+17		3.7431e+16	3.5241e+16	
	4.4896e+17	6.2676e+15		1.5817e+16	2.1938e+16	
	7.8795e+12	1.7039e+12		5.4019e+12	1.7039e+12	
f	8.3439e+12	1.7039e+12		6.1512e+12	1.7039e+12	
	2.1357e+11	0.0000e+00		2.8899e+11	0.0000e+00	
	5.0621e+16	6.3339e+16		1.1161e+16	1.0884e+16	
f	7.5321e+17	9.4042e+16		3.8680e+16	3.1150e+16	
	4.7974e+17	5.2137e+15		1.7411e+16	1.3835e+16	
	9.3147e+16	9.3147e+16		1.2127e+16	2.1257e+16	
f	1.0716e+18	1.1121e+18	–	6.5235e+16	5.3469e+16	
	7.2936e+17	7.6183e+17		3.2750e+16	2.3430e+16	
	1.4057e+17	5.6572e+11		3.7352e+16	5.6572e+11	
f	7.7989e+17	5.6572e+11		6.9690e+16	5.6572e+11	
	3.1321e+17	0.0000e+00		2.1668e+16	0.0000e+00	
Overall / –:		11/1/3			12/0/3	

5 CONCLUSIONS

We tackled the BB LSGO problems with overlapping components from the CEC'2013 LSGO benchmark suit, using a divide-and-conquer method. We used the GCCC framework, that we implemented in the Python programming language for the NiaPy optimization framework. We demonstrate that the GCCC can be used with many existing algorithms from the same optimization framework. For the decomposition of overlapping problems, we used the MRDG3 algorithm, that we implemented in the Python programming language for the NiaPy optimization framework. To systemically evaluate the efficacy of our CC algorithm, we used multiple algorithms and tested them on the CEC'2013 LSGO benchmark which consists of fifteen test functions. Experimental results showed that the GCCC framework facilitated problem-solving, and outperformed its base versions of used algorithms on some of the test functions.

For the feature work, we suggest designing more benchmark problems with a more flexible variable interaction structure and richer sources of overlap. Developing a CC framework that has an option of sharing the same population between all algorithms that are working cooperatively in solving BB LSGO problems.

ACKNOWLEDGMENTS

This work was supported by the Slovenian Research Agency (Computer Systems, Methodologies, and Intelligent Services) under Grant P2-0041.

REFERENCES

[1] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. IEEE, 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[2] Douglass B. Lee. 1994. Retrospective on Large-Scale Urban Models. *Journal of the American Planning Association* 60, 1 (1994), 35–40. https://doi.org/10.1080/01944369408975549

[3] Xiaodong Li, Ke Tang, Mohammad N Omidvar, Zhenyu Yang, Kai Qin, and Hefei China. 2013. Benchmark functions for the CEC 2013 special session and competition on large-scale global optimization. *gene* 7, 33 (2013), 8.

[4] Seyedali Mirjalili. 2016. SCA: A Sine Cosine Algorithm for solving optimization problems. *Knowledge-Based Systems* 96 (2016), 120–133. https://doi.org/10.1016/j.knosys.2015.12.022

[5] Masaharu Munetomo and David E. Goldberg. 1999. Linkage Identification by Non-monotonicity Detection for Overlapping Functions. *Evolutionary Computation* 7, 4 (1999), 377–398. https://doi.org/10.1162/evco.1999.7.4.377

[6] Mohammad Nabi Omidvar, Xiaodong Li, Yi Mei, and Xin Yao. 2014. Cooperative co-evolution with differential grouping for large scale optimization. *IEEE Transactions on evolutionary computation* 18, 3 (2014), 378–393. https://doi.org/10.1109/TEVC.2013.2281543

[7] Mitchell A Potter and Kenneth A De Jong. 1994. A cooperative coevolutionary approach to function optimization. In *International conference on parallel problem solving from nature*. Springer, 249–257. https://doi.org/10.1007/3-540-58484-6_269

[8] Yuan Sun, Xiaodong Li, Andreas Ernst, and Mohammad Nabi Omidvar. 2019. Decomposition for large-scale optimization problems with overlapping components. In *2019 IEEE congress on evolutionary computation (CEC)*. IEEE, 326–333. https://doi.org/10.1109/CEC.2019.8790204

[9] Masaru Tezuka, Masaharu Munetomo, and Kiyoshi Akama. 2004. Linkage Identification by Nonlinearity Check for Real-Coded Genetic Algorithms. In *Genetic and Evolutionary Computation – GECCO 2004*, Kalyanmoy Deb (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–233. https://doi.org/10.1007/978-3-540-24855-2_20

[10] Jerry D VanVactor. 2011. Cognizant healthcare logistics management: ensuring resilience during crisis. *International Journal of Disaster Resilience in the Built Environment* 2, 3 (2011), 245–255. https://doi.org/10.1108/17595901111167114

[11] Grega Vrbančič, Lucija Brezočnik, Uroš Mlakar, Dušan Fister, and Iztok Fister Jr. 2018. NiaPy: Python microframework for building nature-inspired algorithms. *Journal of Open Source Software* 3 (2018). Issue 23. https://doi.org/10.21105/joss.00613

[12] Tian-Li Yu, David E. Goldberg, Kumara Sastry, Claudio F. Lima, and Martin Pelikan. 2009. Dependency Structure Matrix, Genetic Algorithms, and Effective Recombination. *Evol. Comput.* 17, 4 (dec 2009), 595–626. https://doi.org/10.1162/evco.2009.17.4.17409