Differential Evolution and Cooperative Evolution

Armand Maree
120 178 00
Department of Computer Science
University of Pretoria

I. Introduction

In the field of Evolutionary Algorithms (EAs), Differential Evolution (DE) is an optimization algorithm [1]. It borrows concepts from biological evolution and natural selection to guide a population of individuals scattered across the search space to some optimal value. DE does not make any assumptions of the fitness landscape, like the gradient at any position.

II. Background

DE differs from most other EAs in the manner in which cross over and mutation occurs.

A. DE Operators

1) Mutation: During the reproduction process a mutation operator is applied in order to introduce diversity in the population. This is done by randomly choosing two individuals x_{i_2} and x_{i_3} from the population and performing component wise addition to the vectors that represent the individuals. A scalar, β, called the scaling factor, is then multiplied to each component of the resulting vector. Component wise addition is then done between this and another individual, x_{i_1}, called the target vector. This processes is repeated for all $x \in [1, n]$, where n is the number of individuals in the population. Equation 1 [2][3] shows this calculation. The result of this calculation, u_i, is called the trial vector and will be used in the cross over operation.

$$\begin{aligned} \mathbf{u}(t)_i = \mathbf{x}(t)_{i_1} \oplus \beta(\mathbf{x}(t)_{i_2} \ominus \mathbf{x}(t)_{i_3}) \\ \text{where}, \mathbf{x}(t)_{i_1} \neq \mathbf{x}(t)_{i_2} \neq \mathbf{x}(t)_{i_3} \end{aligned} \tag{1}$$

During this report the scaling factor, β, was chosen to be 0.5. The reason for this is that a scaling factor larger than 0.5 can result in some components of the trial vector to lie outside the search space.

2) Cross Over: During the cross over process the trial vector, u_i, and target vector, x_i, is used to produce a new single individual as offspring. A set of cross over points, $\mathcal{J}$, is used to decide which elements of the offspring vector should be selected from the trial vector and which should be selected from the target vector. Algorithm 1 [2] shows how to calculate the cross over set.

In algorithm 1 the value p_r is referred to as the cross over probability and was chosen to be 0.5 for this report.

Once the cross over set has been found, the offspring, $\mathbf{x}'_i$, is produced by using the cross over set to determine whether the a specific component is taken from the trial or the vector, by using equation 2. This form of cross over is referred to as binomial cross over [2].

Algorithm 1: Calculate Cross Over Set

```
j* ~ U(1, n_x)
J ← J ∩ {j*}
foreach j ∈ {1, .., n_x} do
    if U(0,1) < p_r and j ≠ j* then
        J ← J ∩ {j}
    end
end
```

$$x'_{ij}(t), \forall j \in \{1, .., n_x\} = \begin{cases} u_{ij}(t), & \text{if } j \in \mathcal{J} \\ x_{ij}(t), & \text{otherwise} \end{cases} \tag{2}$$

B. Cooperative DE

Cooperative DE is an algorithm that splits the dimensions for the problem into a group of sub-populations in an attempt to break a large problem into a set of small problems. The splitting of the dimensions can happen in a variety of ways which will be discussed in the literature review (section III) and later in the implementation section (section IV) of this report.

III. Literature Review

Very often real world problems that has a large amount of dimensions are split into smaller more manageable problems [4]. A potential problem with the split in the dimensions is the possibility of placing dependent variables in different sub-problems which could lead to a failure in the EA [4]. There has been various methods found to deal with the discovery and adaption of variable dependencies.

One method to discover variable dependencies is the Differential Evolution with Cooperative Co-evolution using Delta-Grouping (DECC-D) algorithm [5]. In this method variables are arranged in descending order of change. That is, the amount of change that a variable j experiences between two consecutive iterations, $\bar{\delta}_j$, is used to sort the variables. Equation 3 [5] is used to calculate the amount of change for each variable.

$$\begin{aligned} \bar{\delta}_j = \frac{\sum_{j=1}^{n_x} \delta_{ji}}{n_x} \\ \text{where}, j \in \{1, .., n_x\} \\ \text{and}, i \in \{1, .., numIndividuals\} \end{aligned} \tag{3}$$

Variables that experience a similar amount of change is then grouped together in subcomponents, which will then in turn be solved by a separate EA. This method seems to be quite successful if only one group of dependent variables exist, about an 80% success rate of grouping dependent variables [4]. This method does fail quite miserably when multiple groups of dependent variables exists [4].

Another technique proposed in [6] is called Multilevel Cooperative Coevolution (MLCC). This technique utilizes a set of "decomposers" which specify how the objective vector is divided into sub-components. Each decomposer has a performance score associated with it. Every predetermined amount of iterations (called cycles), a decomposer from the set is chosen based on its performance score. After a number of iterations have passed, the performance score of the decomposer is assigned a new value which corresponds to the performance of the cycle that just ended. After this the selection process is dome again and the process is repeated.

The MLCC used in [6] was set up in such a way to use the Self-adaptive Differential Evolution with Neighborhood Search (SaNSDE) proposed in [3] with group sizes $\mathbf{S} = \{5, 10, 25, 50, 100\}$. The SaNSDE algorithm is very similar to the standard DE, except for the mutation operator which is replaced with equation 4 [3]. In this equation $N(0.5, 0.5)$ refers to a random number from the Gaussian distribution with a mean and standard deviation of 0.5, and δ refers to a random number from the Cauchy distribution with a scale parameter of $t = 1$ [3].

$$\mathbf{u}(t)_i = \mathbf{x}(t)_{i_1} + \begin{cases} \mathbf{d}_i(t) \times N(0.5, 0.5), & \text{if } U(0,1) < 0.5 \\ \mathbf{d}_i(t) \times \delta, & \text{otherwise} \end{cases}$$
$$\text{where}, \mathbf{x}(t)_{i_1} \neq \mathbf{x}(t)_{i_2} \neq \mathbf{x}(t)_{i_3}$$
$$\text{and}, \mathbf{d}_i(t) = \mathbf{x}(t)_{i_2} - \mathbf{x}(t)_{i_3} \quad (4)$$

The self-adaptive part of SaNSDE refers to its ability to self adjust parameters. It tunes the cross over probability, p_o, and the scaling factor, β, automatically during run time [3].

Based on the results obtained in [6], the MLCC performed quite well, although the newer DECC-D obtained better results for most of the functions tested in [5].

IV. Implementation

Each of the below implementations was executed 30 times and each execution was run for 5000 iterations. The results for the dimensions and functions tested is included in the ODF spreadsheet. Due to time constraints not all of the optimization functions could be tested and not all of the desired dimensions could be tested.

A. DE/rand/1/bin

The first implementation for this report was the simple DE/rand/1/bin. This implementation of DE uses random selection of parents, and performs binomial cross over (which is discussed in section II-A2). The "1" in the name of this DE algorithm refers to the number of difference vectors used when creating the trial vector during the mutation process, this is the process described in section II-A1.

This DE form creates 30 individuals where each individual's size is equal to the number of dimensions that has to be solved. The algorithm then runs until 5000 iterations has been performed and returns the best individual.

Function Name	Number of Dimensions			
	50	100	150	500
Alpine	$k = 0.1$	$k = 0.1$	$k = 0.1$	$k = 0.1$
Eggholder	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Griewank	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Norwegian	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Rosenbrock	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Saloman	$k = 0.5$	$k = 0.1$	$k = 0.1$	
Schaffer6	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Schwefel2.22	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Shubert	$k = 0.1$	$k = 0.1$	$k = 0.1$	
Vincent	$k = 0.1$	$k = 0.1$	$k = 0.1$	

Table I
k VALUES FOR BASIC COOPERATIVE DE

B. Basic Cooperative DE

Due to the *curse-of-dimensionality* [4], the more dimensions are added to a problem, the harder it is for an EA to solve the problem[6]. The problem is thus split into more manageable smaller sub problems.

For this Basic Cooperative DE the number of dimensions were split into sub-populations of size $n_x \times k, k \in [0.0, 1.0]$, where k is the percentage of dimensions per sub-population. The values for k that was tested was $k \in \{0.1, 0.3, 0.5\}$. Overlapping was not allowed, thus each component was only allowed to occur in one sub-population.

Each sub population was then given to a separate DE to optimize. Each DE was placed into their own thread in order to reduce run time. When the fitness of an individual was to be tested, it's components were swapped with a global best individual, called the context vector. If the context vector has a better fitness after the corresponding components were swapped in, then the context vector is updated, other wise it reverts to its old state. The context vector is thus always up to date with the best individual.

This attempts to break the problem into collection of smaller problems which is easier to solve. But this introduces the problem that some variables depend on one another and placing them in two separate sub-populations can cause the algorithm to perform worse.

The values that were tested are: $k \in \{0.1, 0.3, 0.5\}$.

The best values for k found for various different dimensions for the various optimization function can be found in table I.

C. Variable Dependency DE from Literature Review

Unfortunately due to time constraints, this DE was not implemented.

D. Random Grouping Cooperative DE

This algorithm was implemented in the same manner as the Basic Cooperative DE (section IV-B), with the difference

Function Name	Number of Dimensions		
	50	500	1000
Alpine	$k = 0.3$	$k = 0.3$	$k = 0.3$
Saloman	$k = 0.1$	$k = 0.1$	$k = 0.3$

Table II
k VALUES FOR RANDOM GROUPING DE

Function Name	Number of Dimensions		
	50	500	1000
Alpine	$k = 0.3$ $p_o = 0.2$	$k = 0.1$ $p_o = 0.05$	$k = 0.1$ $p_o = 0.05$
Salomon	$k = 0.1$ $p_o = 0.2$	$k = 0.1$ $p_o = 0.05$	$k = 0.1$ $p_o = 0.05$
Vincent	$k = 0.3$ $p_o = 0.2$		

Table III
k AND p_o VALUES FOR RANDOM GROUPING DE (OVERLAP)

coming in with now the components are assigned to various sub-populations. In the basic version, the $n_x \times k$ components are assigned to the first sub-population, the next $n_x \times k$ components are assigned to the second sub-population and so forth. This implementation randomly selects $n_x \times k$ components and assigns them to the first sub-population and then randomly choses the next $n_x \times k$ components and assigns them to the second sub-population. This is repeated until no components remain.

The values that were tested are: $k \in \{0.1, 0.3, 0.5\}$.

This algorithm tries to overcome the potential problem that the previous DE has: dependent variables has to be side by side in the vectors in order for them to be potentially grouped together.

The best values for k found for various different dimensions for the various optimization function can be found in table II.

E. Random Grouping Cooperative DE with Overlap

The implementation for this DE was the same as for the Random Grouping Cooperative DE in section IV-D with the only difference is that there exists a probability, p_o, that a component not included in the original allocation can now be included in this sub-population. Overlap of the sub-populations are thus now allowed.

The values tested are: $k \in \{0.1, 0.3, 0.5\}, p_o \in \{0.05, 0.1, 0.2\}$.

The best values for k and p_o found for various different dimensions for the various optimization function can be found in table III.

F. Decomposition Cooperative DE

This implementation, which is also called a *divide-and-conquer* method [4], requires that every $1 < t < maxIterations$ iterations the number of sub-populations are doubled. Initially the algorithm starts with only 1 sub-population which includes all the dimensions. After t iterations there will be 2 sub-populations and after $2t$ iterations there will be 4 sub-populations. This process will be repeated until there is only 1 component per sub-population. Each sub-population will be optimized by its own DE and each DE will be executed in its own thread. Upon each population split, the components are randomly assigned to the various sub-populations, similar to what the Random Grouping Cooperative DE in section IV-D did.

The idea of this algorithm is to find a general location that proves to contain some optimal and then as time progress each component exploit their immediate surroundings with decreasing consideration for the potential of variable dependencies.

A formula was used to ensure that the decomposition process reaches 1 component per sub-population at a specific iteration, $i_{target} \in \{3000, 4000, 5000\}$. The performance were evaluated separately for each of these target iterations. Equation 5 shows the formula used to calculate t.

$$t = i_{target} \div \log_2 n_x \tag{5}$$

The best values for t found for various different dimensions for the various optimization function can be found in table IV.

Function Name	Number of Dimensions		
	50	500	1000
Salomon	$t = 885$	$t = 446$	$t = 401$
Vincent		$t = 334$	

Table IV
t VALUES FOR DECOMPOSITION COOPERATIVE DE

Function Name	Number of Dimensions		
	50	500	1000
Alpine	$t = 557$	$t = 557$	$t = 501$

Table V
t VALUES FOR DECOMPOSITION COOPERATIVE DE WITH RANDOM GROUPING

G. Decomposition Cooperative DE with Random Grouping

This implementation is a combination between the Decomposition Cooperative DE (section IV-F) and the Random Grouping Cooperative DE (section IV-D). The over all operation is the exact same as the Decomposition Cooperative DE, except that the components assigned to each sub-population is recalculated after every iteration instead of every time the sub-populations split.

The best values for t found for various different dimensions for the various optimization function can be found in table V.

H. Merging Cooperative DE

The Merging Cooperative DE is the opposite of the Decomposition Cooperative DE (section IV-F). Where the Decomposition Cooperative DE starts with one sub-population that contains all the components and ends with one component per sub-population, the Merging Cooperative DE starts with n_x sub-populations that each contains one component and every t iterations the number of sub-populations are halved and the number of components per sub-populations are then doubled. This process is repeated until only one sub-population exists that contains all the dimensions. Other than this, the algorithm functions the same as the Decomposition Cooperative DE.

The idea of this algorithm is firstly to find a general good position for each component and then as time progress take

Function Name	Number of Dimensions		
	50	500	1000
Alpine	$t = 885$	$t = 557$	$t = 501$
Saloman	$t = 885$	$t = 557$	$t =$
Vincent	$t = 885$	$t = 334$	$t =$

Table VI
t VALUES FOR MERGING COOPERATIVE DE

Function Name	Number of Dimensions		
	50	500	1000
Alpine	$t = 885$	$t = 557$	$t = 501$
Saloman	$t = 708$	$t = 446$	$t = 501$
Vincent	$t = 885$	$t = 557$	

Table VII
t VALUES FOR MERGING COOPERATIVE DE WITH RANDOM GROUPING

more and more potential variable dependencies into account but creating larger sub-populations.

A formula was used to ensure that the decomposition process reaches n_x components per sub-population at a specific iteration, $i_{target} \in \{3000, 4000, 5000\}$. The performance was evaluated separately for each of these target iterations. Equation 5 shows the formula used to calculate t.

The best values for t found for various different dimensions for the various optimization function can be found in table VI.

I. Merging Cooperative DE with Random Grouping

Lastly the Merging Cooperative DE that uses random grouping was implemented the exact same as the Merging Cooperative DE (section IV-H) except that the assignment of components to sub-populations was done on each iteration instead of each time the sub-populations merge. Other than this difference, the algorithms function in the same manner.

The best values for t found for various different dimensions for the various optimization function can be found in table VII.

V. RESULTS

The results for the parameters can be found in the implementation sectio (section IV) and the results for the algorithms can be found in the results spreadsheet.

VI. CONCLUSION

From the few experiments that were run, it is quite clear that the merge strategy produced the best results. This is shortly followed by the decomposition strategy.

The results from this report is as expected and lines up with some of the research done in the field.

Due to time constraints not a lot of functions could be optimized, especially since a stringer focus has been placed on the parameter tuning aspect of each algorithm.

Some improvements to the speed of the optimization process were done near the end of this project, which would've allowed for more results if this have been done earlier. Unfortunately due to the large scope and limited time, an initial focus was placed on implementing all the algorithms first.

REFERENCES

[1] Janez Brest, Sao Greiner, Borko Boskovic, Marjan Mernik, and Viljem Zumer. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE transactions on evolutionary computation*, 10(6):646–657, 2006.

[2] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[3] Zhenyu Yang, Ke Tang, and Xin Yao. Differential evolution for high-dimensional function optimization. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 3523–3530. IEEE, 2007.

[4] Guangming Dai, Xiaoyu Chen, Liang Chen, Maocai Wang, and Lei Peng. Cooperative coevolution with dependency identification grouping for large scale global optimization. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 5201–5208. IEEE, 2016.

[5] Mohammad Nabi Omidvar, Xiaodong Li, and Xin Yao. Cooperative co-evolution with delta grouping for large scale non-separable function optimization. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[6] Zhenyu Yang, Ke Tang, and Xin Yao. Multilevel cooperative coevolution for large scale optimization. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 1663–1670. IEEE, 2008.