Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 419372, 14 pages
http://dx.doi.org/10.1155/2013/419372

Research Article

Differential Evolution Algorithm with Self-Adaptive Population Resizing Mechanism

Xu Wang and Shuguang Zhao

College of Information Science and Technology, Donghua University, Shanghai 201620, China

Correspondence should be addressed to Xu Wang; daisy.wangx@gmail.com

Received 4 December 2012; Revised 30 January 2013; Accepted 30 January 2013

Academic Editor: Yang Tang

A differential evolution (DE) algorithm with self-adaptive population resizing mechanism, SapsDE, is proposed to enhance the performance of DE by dynamically choosing one of two mutation strategies and tuning control parameters in a self-adaptive manner. More specifically, more appropriate mutation strategies along with its parameter settings can be determined adaptively according to the previous status at different stages of the evolution process. To verify the performance of SapsDE, 17 benchmark functions with a wide range of dimensions, and diverse complexities are used. Nonparametric statistical procedures were performed for multiple comparisons between the proposed algorithm and five well-known DE variants from the literature. Simulation results show that SapsDE is effective and efficient. It also exhibits much more superiorresults than the other five algorithms employed in the comparison in most of the cases.

1. Introduction

Evolutionary algorithms (EAs), inspired by biological evolutionary mechanism in nature, have achieved great success on many numerical and combinatorial optimizations in diverse fields [1–3]. During the past two decades, EAs have become a hot topic. When implementing the EAs, users need to solve several points, for example, the appropriate encoding schemes, evolutionary operators, and the suitable parameter settings, to ensure the success of the algorithms. The earlier EAs have some disadvantages, such as complex procedure, stagnation, and poor search ability. To overcome such disadvantages, on one hand, some researchers proposed other related methods (e.g., particle swarm optimization (PSO) [4, 5], differential evolution (DE) [6]) which have better global search ability. On the other hand, the effects of setting the parameters of EAs have also been the subject of extensive research [7] by the EA community, and recently there are substantial self-adaptive EAs, which can adjust their parameters along with iterations (see, e.g., [2, 8] for a review).

DE is proposed by Storn and Price [6]. Like other EAs, DE is a population-based stochastic search technique as well, but it is simpler and it can be implemented more easily than other EAs. Besides that, DE [9, 10] is an effective and versatile function optimizer. Owing to simplicity of code, practitioners from other fields can simply apply it to solve their domain-specific problems even if they are not good at programming. Moreover, in traditional DE, there are only three crucial control parameters, that is, scaling factor F, crossover rate C_r, and population size NP, which are fewer than other EAs' (e.g., [8, 11]). It is clear that the appropriate settings of the three control parameters ensure successful functioning of DE [12]. However, results in [6, 13] potentially confuse scientists and engineers who may try to utilize DE to solve scientific and practical problems. Further, while some objective functions are very sensitive to parameter settings, it will be difficult to set the parameter values according to prior experience. Hence, a great deal of publications [14–21] have been devoted to the adjustment of parameters of variation operators. Brest et al. [15] proposed a self-adaptive differential evolution algorithm (called jDE) based on the self-adapting control parameter scheme, which produced control parameters F and C_r into a new parent vector and adjusted them with probability. Qin et al. in [20, 21] proposed a SaDE algorithm, in which there was

a mutation strategy candidate pool. Specifically, in SaDE algorithm, one trial vector generation strategy and associated parameter (F and C_r) settings were adjusted according to their previous experiences of generating promising solutions. Moreover, researchers developed the performance of DE by applying opposition-based learning [22] or local search [23].

In most existing DEs, the population size remains constant over the run. However, there are biological and experimental reasonings to expect that a variable population size would work better. In a natural environment, population sizes of species change and incline to steady state due to natural resources and ecological factors. Technically, the population size in a biological system is the most flexible element. And it can be calibrated more easily than recombination. Bäck et al. [24] have indicated that calibrating the population size during iterative process could be more rewarding than changing the operator parameters in genetic algorithms. Unfortunately, the DEs with variable population size (e.g., [25, 26]) have not received much attention despite their various applications in real world, and there is still a lot of research space. Hence in this paper, we will focus on DE with variable population size scheme in which the population size can be adjusted dynamically based on the online solution-search status. In this algorithm, we introduce three population adjustment mechanisms to obtain the appropriate value of NP according to the desired population distribution. Specifically, while the fitness is improved, it may increase the population size to explore. While short term lacks improvement, it may sink the population size. But if stagnation is over a longer period, the population will grow again. Along with those, two trial vector generation strategies will be adopted adaptively during evolution process.

The remainder of this paper is organized as follows. Section 2 gives a brief review of traditional DE and JADE algorithms. Section 3 introduces the DE with self-adaptive population size scheme—SapsDE. Our mutating and adaptive resizing strategies will also be described. Section 4 describes our studies compared with the traditional DE and several state-of-the-art adaptive DE variants and presents the experimental results on a diverse set of test functions with up to 100 dimensions. Finally, Section 5 concludes this paper with some remarks and future research directions.

2. Differential Evolution and JADE Algorithm

In this section, we present an overview of the basic concepts of DE and JADE algorithm necessary for a better understanding of our proposed algorithm.

2.1. Differential Evolution. DE is a population-based algorithm, and a reliable and versatile function optimizer, which evolves a population of NP D-dimensional individuals towards the global optimum by enhancing the differences of the individuals. In brief, after initialization, DE repeats mutation, crossover, and selection operations to produce a trail vector and select one of those vectors with the best fitness value for each vector until satisfying some specific termination criteria. Conveniently, subsequent generation in DE is denoted by $G = 0, 1, \ldots, G_{\max}$. We notate the ith vector of the population at the current generation as follows:

$$\vec{X}_i^G = \left(x_{i,1}^G, x_{i,2}^G, \ldots, x_{i,j}^G\right), \quad j = 1, 2, \ldots, D. \tag{1}$$

Initialization. First of all, uniformly randomize NP individuals within a D-dimensional real parameter search space. And the initial population should cover the entire search space constrained by the prescribed minimum and maximum bounds: $\vec{X}_{\min} = (x_{\min,1}, x_{\min,2}, \ldots, x_{\min,D})$, $\vec{X}_{\max} = (x_{\max,1}, x_{\max,2}, \ldots, x_{\max,j})$. Hence, the initialization of the jth element in the ith vector is taken as follows:

$$x_{i,j}^0 = x_{\min,j} + \text{rand}\,(0,1) \cdot \left(x_{\max,j} - x_{\min,j}\right), \quad j = 1, 2, \ldots, D, \tag{2}$$

where rand$(0, 1)$ represents a uniformly distributed random variable within the range $[0, 1]$, and it is instantiated independently for each component of the ith vector.

Mutation Operation. In the existing literature on DE, mutate vector V_i, called donor vector, is obtained through the differential mutation operation with respect to each individual $\vec{X}_i$, known as target vector, in the current population. For each target vector from the current population, the donor vector is created via certain mutation strategy. Several mutation strategies have been proposed. Here we list one of the most popular and simplest forms of DE-mutation as follows:

$$\vec{V}_i^G = \vec{X}_{r_1}^G + F \cdot \left(\vec{X}_{r_2}^G - \vec{X}_{r_3}^G\right). \tag{3}$$

The indices r_1, r_2, and r_3 are mutually exclusive integers randomly generated within the range $[1, NP]$, which are different from the base vector index i. These indices are randomly generated for each mutant vector. Now, the difference of any two of these three vectors is scaled by a mutation weighting factor F which typically lies in the interval $[0.4, 1]$ in the existing DE literature, and the scaled difference is added to the third one to obtain a donor vector V.

Crossover Operation. After the mutation operation, according to the target vector $\vec{X}_i^G$ and its corresponding donor vector $\vec{V}_i^G$, a trail vector $\vec{U}_i^G$ is produced by crossover operation. In traditional version, DE applies the binary defined crossover as follows:

$$u_{i,j}^G = \begin{cases} v_{i,j}^G, & \text{if rand}_{i,j}\,[0,1] \le C_r \text{ or } j = j_{\text{rand}}, \\ x_{i,j}^G, & \text{otherwise}, \end{cases} \tag{4}$$

where C_r is a crossover rate within the range $[0, 1]$, defined by user as a constant, which controls the probability of

(1): **Begin**
(2): Initialization(); Generate uniformly distributed random population of NP individuals
(3): **while** stopping criterion is not satisfied **do**
(4): **for** $i = 1$ to NP **do**
(5): *Select* random indexes r_1, r_2, r_3 *with* $r_1 \neq r_2 \neq r_3 \neq i$
(6): $\vec{V}_i^G = \vec{X}_{r_1}^G + F \cdot (\vec{X}_{r_2}^G - \vec{X}_{r_3}^G)$
(7): $j_{\text{rand}} = [\text{rand}\,[0, 1) * D]$
(8): **for** $j = 1$ to D **do**
(9): **if** $\text{rand}_{i,j}[0, 1] \leq C_r$ *or* $j = j_{\text{rand}}$ **then**
(10): $u_{i,j}^G = v_{i,j}^G$
(11): **else**
(12): $u_{i,j}^G = x_{i,j}^G$
(13): **end if**
(14): **end for**
(15): **if** $f(\vec{U}_i^G) \leq f(\vec{X}_i^G)$ **then**
(16): $\vec{X}_i^{G+1} = \vec{U}_i^G$
(17): **else**
(18): $\vec{X}_i^{G+1} = \vec{X}_i^G$
(19): **end if**
(20): **end for**
(21): $G = G + 1$
(22): **end while**
(23): **End**

Algorithm 1: Differential evolution algorithm.

parameter values employed from the donor vector. j_{rand} is a randomly chosen integer within the range $[1, NP]$ which is introduced to ensure that the trial vector contains at least one parameter from donor vector.

Selection Operation. In classic DE algorithm, a greedy selection is adopted. The fitness of every trail vectors is evaluated and compared with that of its corresponding target vector in the current population. For minimization problem, if the fitness value of trial vector is not more than that of target vector, the target vector will be replaced by the trial vector in the population of the next generation. Otherwise, the target vector will be maintained in the population of the next generation. The selection operation is expressed as follows:

$$\vec{X}_i^{G+1} = \begin{cases} \vec{U}_i^G, & \text{if } f\left(\vec{U}_i^G\right) \leq f\left(\vec{X}_i^G\right), \\ \vec{X}_i^G, & \text{otherwise.} \end{cases} \tag{5}$$

The algorithmic process of DE is depicted in Algorithm 1.

2.2. JADE Algorithm. Zhang and Sanderson [27] introduced adaptive differential evolution with optional external archive, named JADE, in which a neighborhood-based mutation strategy and an optional external archive were employed to improve the performance of DE. It is possible to balance the exploitation and exploration by using multiple best solutions, called DE/current-to-$p$best strategy, which is presented as follows:

$$\mathbf{v}_i^G = \mathbf{x}_i^G + F_i \cdot \left(\mathbf{x}_{\text{best},p}^G - \mathbf{x}_i^G\right) + F_i \cdot \left(\mathbf{x}_{r_1}^G - \tilde{\mathbf{x}}_{r_2}^G\right), \tag{6}$$

where $\mathbf{x}_{\text{best},p}^G$ is randomly selected as one of the top 100 p% individuals of the current population with $p \in (0, 1]$. Meanwhile, $\mathbf{x}_i^G$ and $\mathbf{x}_{r_1}^G$ are diverse and random individuals in the current population $\mathbf{P}$, respectively. $\tilde{\mathbf{x}}_{r_2}^G$ is randomly selected from the union of $\mathbf{P}$ and the archive $\mathbf{A}$. In Particular, $\mathbf{A}$ is a set of achieved inferior solutions in recent generations and its individual number is not more than the population size. At each generation, the mutation factor F_i and the crossover factor C_{r_i} of each individual $\mathbf{x}_i$ are, respectively, updated dynamically according to a Cauchy distribution of mean μ_F and a normal distribution of mean μ_{C_r} as follows:

$$\begin{aligned} F_i &= \text{rand}\, c_i\left(\mu_F, 0.1\right), \\ C_{r_i} &= \text{rand}\, n_i\left(\mu_{C_r}, 0.1\right). \end{aligned} \tag{7}$$

The proposed two location parameters are initialized as 0.5 and then generated at the end of each generation as follows:

$$\begin{aligned} \mu_F &= (1 - c) \cdot \mu_F + c \cdot \text{mean}_L\left(S_F\right), \\ \mu_{C_r} &= (1 - c) \cdot \mu_{C_r} + c \cdot \text{mean}_A\left(S_{C_r}\right), \end{aligned} \tag{8}$$

where c in $(0, 1)$ is a positive constant; S_F/S_{C_r} indicates the set of all successful mutation/crossover factors in generation; $\text{mean}_A(\cdot)$ denotes the usual arithmetic mean, and $\text{mean}_L(\cdot)$ is the Lehmer mean, which is defined as follows:

$$\text{mean}_L\left(S_F\right) = \frac{\sum_{i=1}^{|S_F|} F_i^2}{\sum_{i=1}^{|S_F|} F_i}. \tag{9}$$

3. SapsDE Algorithm

Here we develop a new SapsDE algorithm by introducing a self-adaptive population resizing scheme into JADE. Technically, this scheme can gradually self-adapt *NP* according to the previous experiences of generating promising solutions. In addition, we use two DE mutation strategies in SapsDE. In each iteration, only one DE strategy is activated. The structure of the SapsDE is shown in Algorithm 2.

3.1. Generation Strategies Chooser. DE performs better if it adopts different trail vector generation strategies during different stages of optimization [21]. Hence, in SapsDE, we utilize two mutation strategies, that is, DE/rand-to-best/1 and DE/current-to-*p*best/1.

The DE/rand-to-best/1 strategy benefits from its fast convergence speed but may result in premature convergence due to the resultant reduced population diversity. On the other hand, the DE/current-to-*p*best/1 strategy balances the greediness of the mutation and the diversity of the population. Considering the above two strategies, we introduce a parameter θ to choose one of the strategies in each iteration of the evolutionary process. At an earlier stage of evolutionary process, DE/current-to-best/1 strategy is adopted more to achieve fast convergence speed. For avoiding trapping into a local optimum, as the generation is proceeding further, DE/current-to-*p*best/1 strategy is used more to search for a relatively large region which is biased toward promising progress directions.

As presented in lines 8–15 in Algorithm 2, the DE/rand-to-best/1 strategy is used when Rn is smaller than θ; otherwise, DE/current-to-*p*best/1 strategy is picked where Rn is a random number generated from the continuous uniform distribution on the interval $(0, 1)$. Notice that $\theta \in [0.1, 1]$ is a time-varying variable which diminishes along with the increase of generation and it can be expressed as follows:

$$\theta = \frac{(G_{\max} - G)}{G_{\max}} * \theta_{\max} + \theta_{\min}, \tag{10}$$

where $\theta_{\min} = 0.1$, $\theta_{\max} = 0.9$, and G denotes the generation counter. Hence, DE/rand-to-best/1 strategy can be frequently activated at earlier stage as the random number can easily get smaller than θ. Meanwhile, DE/current-to-*p*best/1 strategy takes over more easily as the generation increases.

3.2. Population Resizing Mechanism. The population resizing mechanism aims at dynamically increasing or decreasing the population size according to the instantaneous solution-searching status. In SapsDE, the dynamic population size adjustment mechanism depends on a *population resizing trigger*. This trigger will activate population-reducing or augmenting strategy in accordance with the improvements of the best fitness in the population. One step further, if a better solution can be found in one iteration process, the algorithm becomes more biased towards exploration augmenting the population size, short-term lack of improvement shrinks the population for exploitation, but stagnation over a longer period causes population to grow again. Further, the population size is monitored to avoid breaking lower bound. As described in lines 23–34 in Algorithm 2, Lb is the lower bound indicator. Correspondingly, $Lbound$ is the lower bounds of the population size. The proposed B, W, and St are three significant trigger variables, which are used to activate one of all dynamic population size adjustment strategies. More specifically, the trigger variable W is set to 1 when the best fitness does not improve, that may lead to a population-reducing strategy to delete poor individuals from current population. Furthermore, similarly, if there is improvement of the best fitness, B and St are assigned 1, respectively. The population-augmenting strategy 1 is used when the trigger variable B is set to 1. Besides, if the population size is not bigger than the lower bound ($Lbound$) in consecutive generations, the lower bound monitor (Lb) variable will be increased in each generation. This process is continuously repeated until it achieves a user-defined value R; that is, the population-augmenting strategy2 will be applied if $Lb > R$ or $St > R$.

Technically, this paper applies three population resizing strategies as follows.

Population Reducing Strategy. The purpose of the population-reducing strategy is to make the search concentrating more on exploitation by removing the redundant inferior individuals when there is no improvement of the best fitness in short term. More specifically, in population reducing strategy, the first step is to evaluate fitness function values of individuals. Second step is to arrange the population with its fitness function values from small to large. For minimization problems, the smaller the value of fitness function is, the better the individual performs. Consequently, the third step is to remove some individuals with large values from the current population. The scheme of population reduction is presented in Algorithm 3, where ρ_1 denotes the number of deleted individuals.

Population Augmenting Strategy1. Population Augmenting strategy1 is intended to bias more towards exploration in addition to exploitation. Here it applies DE/best/2 mutation strategy to generate a new individual increasing the population size on fitness improvement. The pseudocode of the population-increasing scheme is shown in Algorithm 4.

Population Augmenting Strategy2. The population size is increased by population augmenting strategy2, shown in Algorithm 5 if there is no improvement during the last R number of evaluations. The second growing strategy is supposed to initiate renewed exploration when the population is stuck in local optima. So, here it applies DE/rand/1 mutation scheme. In theory, the size to increase the population in this step can be defined independently from others, but in fact we use the same growth rate s as one of the population-reducing strategy.

4. Experiments and Results

4.1. Test Functions. The SapsDE algorithm was tested on benchmark suite consisting of 17 unconstrained single-objective benchmark functions which were all minimization problems. The first 8 benchmark functions are chosen from

(1): **Begin**
(2): $G = 0$
(3): Randomly initialize a population of NP vectors uniformly distributed in the
(4): range $[X_{\min}, X_{\max}]$
(5): Evaluate the fitness values of the population
(6): **while** termination criterion is not satisfied **do**
(7): **for** $i = 1$ to NP **do**
(8): Generate donor vector $\vec{X}_i^G$
(9): **if** $Rn < \theta$ **then**
(10): $\mathbf{v}_i^G = \mathbf{x}_i^G + F_i \cdot (\mathbf{x}_{\text{best}}^G - \mathbf{x}_i^G) + F_i \cdot (\mathbf{x}_{r_1}^G - \tilde{\mathbf{x}}_{r_2}^G)$
(11): // mutate with DE/rand-to-best/1 strategy
(12): **else**
(13): $\mathbf{v}_i^G = \mathbf{x}_i^G + F_i \cdot (\mathbf{x}_{\text{best},p}^G - \mathbf{x}_i^G) + F_i \cdot (\mathbf{x}_{r_1}^G - \tilde{\mathbf{x}}_{r_2}^G)$
(14): // mutate with DE/current-to-$p$best/1 strategy (JADE)
(15): **end if**
(16): Through *crossover* operation to generate trial vector
(17): $\vec{U}_i^G$ = Strategy (i, pop)
(18): Evaluate the trial vector $\vec{U}_i^G$
(19): **if** $f(\vec{U}_i^G) \le f(\vec{X}_i^G)$ **then**
(20): $\vec{X}_i^{G+1} = \vec{U}_i^G$ // Save index for replacement
(21): **end if**
(22): **end for**
(23): //ϕ is the minimum value of function evaluations in the last generation
(24): //Lb is the low bound of population size
(25): **if** $\min(f(\vec{X}_i^G)) < \phi$ **then**
(26): $B = 1$;
(27): $\phi = \min(f(\vec{X}_i^G))$
(28): **else**
(29): $W = 1$;
(30): $St = St + 1$;
(31): **end if**
(32): **if** $popsize \le Lbound$ **then**
(33): $Lb = Lb + 1$
(34): **end if**
(35): **if** $(W == 1)$ **and** $(St \le R)$ **then**
(36): **Population_Reducing_Strategy()**
(37): $W = 0$
(38): **end if**
(39): **if** $(B == 1)$ **then**
(40): **Population_Augmenting_Strategy1()**
(41): Evaluate the new additional individuals
(42): $B = 0$
(43): $St = 0$
(44): **end if**
(45): **if** $(St > R)$ **or** $(Lb > R)$ **then**
(46): **Population_Augmenting_Strategy2()**
(47): Evaluate the new additional individuals
(48): $St = 0$
(49): $Lb = 0$
(50): **end if**
(51): $G = G + 1$
(52): **end while**
(53): **End**

Algorithm 2: The SapsDE algorithm.

(1): **Begin**
(2): Arrange the population with its fitness function values in ascending order
(3): $\vec{\mathbf{F}}(f(\vec{x}_1), f(\vec{x}_2), \ldots, f(\vec{x}_{SP})) = (f_{\min}, f_{\min+1}, \ldots, f_{\max})$
(4): $\rho_1 = \lfloor s\% \times NP \rfloor$
(5): Remove the last ρ_1 individuals from current population
(6): **End**

Algorithm 3: Population reducing strategy.

(1): **Begin**
(2): Select the best individual from the current population
(3): Randomly select four different vectors $x^G_{r_1} \neq x^G_{r_2} \neq x^G_{r_3} \neq x^G_{r_4}$ from the
(4): current population
(5): $x^G_{i,b} = x^G_{\text{best}} + F \cdot (x^G_{r_1} - x^G_{r_2}) + F \cdot (x^G_{r_3} - x^G_{r_4})$
(6): Store $x^G_{i,b}$ into BA
(7): Add the individual of BA into the current population
(8): Empty BA
(9): **End**

Algorithm 4: Population augmenting strategy1.

the literature and the rest are selected from CEC 2005 Special Session on real-parameter optimization [28]. The detailed description of the function can be found in [29, 30]. In Table 1, our test suite is presented, among which functions f_1, f_2, and f_9–f_{12} are unimodal and functions f_3–f_8 and f_{13}–f_{17} are multimodal.

4.2. Parameter Settings. SapsDE is compared with three state-of-the-art DE variants (JADE, *j*DE, and SaDE) and the classic DE with DE/rand/1/bin/strategy. To evaluate the performance of algorithms, experiments were conducted on the test suite. We adopt the solution error measure ($f(x) - f(x^*)$), where x is the best solution obtained by algorithms in one run and x^* is well-known global optimum of each benchmark function. The dimensions (D) of function are 30, 50, and 100, respectively. The maximum number of function evaluations (FEs), the terminal criteria, is set to $10\,000 \times D$, all experiments for each function and each algorithm run 30 times independently.

In our experimentation, we follow the same parameter settings in the original paper of JADE, *j*DE, and SADE. For DE/rand/1/bin, the parameters are also studied in [31]. The details are shown as follows:

(1) the original DE algorithm with DE/rand/1/*bin* strategy, $F = 0.9$, $C_r = 0.9$ and P (population size) = N (dimension);

(2) JADE, $p = 0.05$, $c = 0.1$;

(3) *j*DE, $\tau_1 = \tau_2 = 0.1$;

(4) SaDE, F: $\text{rand}_N(0.5, 0.3)$, $LP = 50$.

For SapsDE algorithm, the configuration is listed as follows: the *Lbound* is set to 50. Initial population size is set to 50. The adjustment factor of population size s is fixed to 1. The threshold variable of boundary and stagnation variable R is set to 4.

All experiments were performed on a computer with Core 2 2.26-GHz CPU, 2-GB memory, and Windows XP operating system.

4.3. Comparison with Different DEs. Intending to show how well the SapsDE performs, we compared it with the conventional DE and three adaptive DE variants. We first evaluate the performance of different DEs to optimize the 30-dimensional numerical functions $f_1(x)$–$f_{17}(x)$. Table 2 reports the mean and standard deviation of function values over 30 independent runs with 300 000 FES. The best results are typed in bold. For a thorough comparison, the two-tailed t-test with a significance level of 0.05 has been carried out between the SapsDE and other DE variants in this paper. Rows "+ (Better)," "= (Same)," and "– (Worse)" give the number of functions that the SapsDE performs significantly better than, almost the same as, and significantly worse than the compared algorithm on fitness values in 30 runs, respectively. Row total score records the number of +'s and the number of –'s to show an overall comparison between the two algorithms. Table 2 presents the total score on every function. In order to further evaluate the performance of SapsDE, we report the results of SapsDE and other four DE variants on test functions at $D = 50$. The experimental results are summarized in Table 3.

From Table 2, SapsDE is significantly better than other comparisonal algorithms on thirteen functions while it is outperformed by JADE on functions f_{10} and f_{12} and by SaDE on functions f_1 and f_{17}. Obviously, SapsDE obtains the best average ranking among the five algorithms.

(1): **Begin**
(2): $\rho_2 = \lceil s\% \times NP \rceil$
(3): Select ρ_2 best individuals from the current population
(4): **for** $i = 1$ to ρ_2 **do**
(5): Randomly generate three different vectors $x_{r_1}^G \neq x_{r_2}^G \neq x_{r_3}^G$ from the
(6): current population
(7): $x_{i,b}^G = x_{r_1}^G + F \cdot (x_{r_2}^G - x_{r_3}^G)$
(8): Store $x_{i,b}^G$ into BA
(9): **end for**
(10): Add the total individuals of BA into the current population, and empty BA
(11): **End**

Algorithm 5: Population augmenting strategy2.

Table 1: Benchmark functions.

Functions	Name	Search space	f_{bias}
$f_1(x)$	Sphere	$[-100, 100]^D$	0
$f_2(x)$	Rosenbrock	$[-100, 100]^D$	0
$f_3(x)$	Ackley	$[-32, 32]^D$	0
$f_4(x)$	Griewank	$[-600, 600]^D$	0
$f_5(x)$	Rastrigin	$[-5, 5]^D$	0
$f_6(x)$	Salomon	$[-100, 100]^D$	0
$f_7(x)$	Generalized Penalized Function 1	$[-50, 50]^D$	0
$f_8(x)$	Generalized Penalized Function 2	$[-50, 50]^D$	0
$f_9(x)$	Shifted Sphere	$[-100, 100]^D$	−450
$f_{10}(x)$	Shifted Schwefel's Problem 1.2	$[-100, 100]^D$	−450
$f_{11}(x)$	Shifted Rotated High Conditioned Elliptic	$[-100, 100]^D$	−450
$f_{12}(x)$	Shifted Schwefel's Problem 1.2 with Noise in Fitness	$[-100, 100]^D$	−450
$f_{13}(x)$	Shifted Rosenbrock	$[-100, 100]^D$	390
$f_{14}(x)$	Shifted Rotated Ackley's Function with Global Optimum on Bounds	$[-32, 32]^D$	−140
$f_{15}(x)$	Shifted Rastrigin	$[-100, 100]^D$	−330
$f_{16}(x)$	Shifted Rotated Rastrigin	$[-5, 5]^D$	−330
$f_{17}(x)$	Shifted Rotated Weierstrass	$[-0.5, 0.5]^D$	90

More specifically, with respect to JADE, the t-test is 4/1/12 in Table 2. It means that SapsDE significantly outperforms JADE on 4 out of 17 benchmark functions. JADE is significantly better than SapsDE on one function f_{12}. For the rest of benchmark functions, there is no significant difference between SapsDE and JADE. Similarly, we can get that SapsDE performs significantly better than DE, jDE, and SaDE on 14, 11, and 14 out of 17 test functions, respectively. Thus, SapsDE is the winner of test. The reason is that SapsDE implements the adaptive population resizing scheme, which can help the algorithm to search the optimum as well as maintaining a higher convergence speed when dealing with complex functions.

From Table 3, it can be seen that SapsDE is significantly better than other algorithms on the 14 (f_1–f_3, f_5, and f_7–f_{16}) out of 17 functions. On the two functions f_4 and f_6, jDE outperforms SapsDE. And for test function f_{17}, the best solution is achieved by SaDE.

Particularly, SapsDE provides the best performance among the five algorithms on all unimodal functions. The three inferior solutions of SapsDE are all on the multimodal functions. However, in general, it offers more improved performance than all of the compared DEs. The t-test is summarized in the last three rows of Table 3. In fact, SapsDE performs better than DE, JADE, jDE, and SaDE on 16, 8, 11, and 16 out of 17 test functions, respectively. In general, our

Table 2: Experimental results of 30-dimensional problems $f_1(x)$–$f_{17}(x)$, averaged over 30 independent runs with 300 000 FES.

Function	DE	JADE	*j*DE	SaDE	SapsDE
	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)
1	8.70*e*−032 (1.07*e*−31)+	8.52*e*−123 (4.49*e*−122)+	2.14*e* − 61 (3.68*e* − 61)+	**1.07e − 130 (4.99e − 130)**−	4.38*e*−126 (2.27*e*−125)
2	8.14*e* + 07 (5.94*e* + 07)+	9.42*e* − 01 (4.47*e* − 01)+	1.18*e* + 01 (1.06*e* + 01)+	3.62*e* + 01 (3.20*e* + 01)+	**1.32e − 01 (7.27e − 01)**
3	4.32*e* − 15 (1.80*e* − 15)+	**2.66e − 15 (0.00e + 00)**	2.90*e* − 15 (9.01*e* − 16)=	1.51*e* − 01 (4.00*e* − 01)+	**2.66e − 15 (0.00e + 00)**
4	5.75*e*−004 (2.21*e*−03)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	6.14*e* − 03 (1.22*e* − 02)+	**0.00e + 00 (0.00e + 00)**
5	1.33*e* + 02 (2.13*e* + 01)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	6.63*e* − 02 (2.52*e* − 01)+	**0.00e + 00 (0.00e + 00)**
6	1.89*e* − 01 (3.00*e* − 02)=	1.90*e* − 01 (3.05*e* − 02)=	1.97*e* − 01 (1.83*e* − 02)+	2.53*e* − 01 (7.30*e* − 02)+	**1.77e − 01 (3.94e − 02)**
7	2.08*e* − 32 (1.21*e* − 32)+	**1.57e − 32 (5.56e − 48)**	**1.57e − 32 (5.57e − 48)**	6.91*e* − 03 (2.63*e* − 02)+	**1.57e − 32 (5.56e − 48)**
8	8.16*e* − 32 (8.89*e* − 32)+	**1.34e − 32 (5.56e − 48)**	1.35*e* − 32 (5.57*e* − 48)+	1.10*e* − 03 (3.35*e* − 03)+	**1.34e − 32 (5.56e − 48)**
9	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**
10	3.99*e* − 05 (2.98*e* − 05)+	**1.10e − 28 (9.20e − 29)=**	9.55*e* − 07 (1.32*e* − 06)+	1.06*e* − 05 (3.92*e* − 05)+	1.17*e* − 28 (1.07*e* − 28)
11	1.01*e* + 08 (2.13*e* + 07)+	1.06*e* + 04 (8.07*e* + 03)+	2.29*e* + 05 (1.47*e* + 05)+	5.25*e* + 05 (2.099*e* + 05)+	**7.00e + 03 (4.29e + 03)**
12	1.44*e* − 02 (1.33*e* − 02)+	**2.33e − 16 (6.61e − 16)**−	5.55*e* − 02 (1.54*e* − 01)+	2.5.*e* + 02 (2.79*e* + 02)+	4.22*e* − 15 (1.25*e* − 14)
13	2.46*e* + 00 (1.61*e* + 00)=	3.02*e* + 00 (9.38*e* + 00)=	1.83*e* + 01 (2.04*e* + 01)+	5.74*e* + 01 (2.95*e* + 01)+	**1.49e + 00 (5.68e + 00)**
14	2.09*e* + 01 (4.56*e* − 02)+	2.08*e* + 01 (2.47*e* − 01)=	2.10*e* + 01 (4.17*e* − 02)+	2.10*e* + 01 (3.46*e* − 02)+	**2.07e + 01 (3.98e − 01)**
15	1.20*e* + 02 (2.64*e* + 01)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	1.33*e* − 01 (3.44*e* − 01)+	**0.00e + 00 (0.00e + 00)**
16	1.79*E* + 02 (9.37*e* + 00)+	2.32*e* + 01 (4.08*e* + 00)=	5.66*e* + 01 (1.00*e* + 01)+	4.62*e* + 01 (1.02*e* + 01)+	**2.18e + 01 (3.58e + 00)**
17	3.92*e* + 01 (1.20*e* + 00)+	2.49*e* + 01 (2.28*e* + 00)+	2.78*e* + 01 (1.87*e* + 00)+	**1.72e + 01 (2.53e + 00)**−	2.42*e* + 01 (1.68*e* + 00)

Total score	DE	JADE	*j*DE	SaDE	**SapsDE**
+	14	4	11	14	*
−	0	1	0	2	*
=	3	12	6	1	*

proposed SapsDE performs better than other DE variants on benchmark functions at $D = 50$ in terms of the quality of the final results.

We can see that the proposed SapsDE is effective and efficient on optimization of low dimensional functions. However, high dimensional problems are typically harder to solve and a common practice is to employ a larger population size. In order to test the ability of the SapsDE in high dimensional problems, we compared it with other four DE variants for our test functions at $D = 100$. In Table 4, the experimental results of 100-dimensional problems f_1–f_{17} are summarized.

Table 4 shows that the SapsDE provides the best performance on the f_1, f_2, f_5, f_6, f_8, f_{11}, and f_{14}–f_{17} and performs slightly worse than one of four comparisonal algorithms on the rest of the functions. The *j*DE offers the best performance on the f_3, f_4, and f_7. SaDE performs best on the f_9. However, the differences between SapsDE and *j*DE are considered to be not quite statistically significant on the f_3, f_4, and f_7, neither does it between SapsDE and SaDE on the f_9.

To apply the *t*-test, we obtain the statistic results of the experiments at $D = 100$ for all functions. The results of SapsDE are compared with those of DE, JADE, SaDE, and *j*DE. Results of *t*-test are presented in the last three rows of Table 4. It is clear that SapsDE obtains higher "+" values than "−" values in all cases. Unquestionably, SapsDE performs best on 100-dimension benchmark functions.

Comparing the overall performances of five algorithms on all functions, we find that SapsDE obtains the best average ranking, which outperforms the other four algorithms. JADE follows SapsDE in the second best average ranking. The *j*DE can converge to the best solution found so far very quickly though it is easy to stuck in the local optima. The SaDE has

Table 3: Experimental results of 50-dimensional problems $f_1(x)$–$f_{17}(x)$, averaged over 30 independent runs with 500 000 FES.

Function	DE	JADE	*j*DE	SADE	SapsDE
	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)
1	3.51*e* − 35 (4.71*e* − 35)+	1.99*e* − 184 (0.00*e* + 00)+	5.98*e* − 62 (7.49*e* − 62)+	1.67*e*−118 (7.81*e*−118)+	**3.05e − 202 (0.00e + 00)**
2	1.82*e* + 08 (1.16*e* + 08)+	5.32*e* − 01 (1.38*e* + 00)+	2.61*e* + 01 (1.56*e* + 01)+	6.79*e* + 01 (3.58*e* + 01)+	**1.70e − 28 (5.55e − 28)**
3	2.69*e* − 02 (7.63*e* − 03)+	5.98*e* − 15 (9.01*e* − 16)=	6.21*e* − 15 (0.00*e* + 00)+	1.18*e* + 00 (4.68*e* − 01)+	**5.38e − 15 (1.52e − 15)**
4	6.35*e* − 02 (1.62*e* − 01)+	1.56*e* − 03 (5.46*e* − 03)=	**0.00e + 00 (0.00e + 00)**−	1.39*e* − 02 (3.26*e* − 02)+	2.46*e* − 04 (1.35*e* − 03)
5	2.02*e* + 02 (5.08*e* + 01)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	5.97*e* − 01 (7.20*e* − 01)+	**0.00e + 00 (0.00e + 00)**
6	1.78*e* + 00 (2.00*e* − 01)+	3.07*e* − 01 (3.65*e* − 02)+	**2.00e − 01 (3.55e − 11)**−	5.37*e* − 01 (1.13*e* − 01)+	2.73*e* − 01 (3.05*e* − 02)
7	1.24*e* − 02 (4.73*e* − 02)+	6.22*e* − 03 (1.90*e* − 02)+	1.57*e* − 32 (5.57*e* − 48)+	5.39*e* − 02 (1.04*e* − 01)+	**9.42e − 33 (2.78e − 48)**
8	1.37*e* − 32 (9.00*e* − 34)=	1.35*e* − 32 (5.57*e* − 48)=	1.35*e* − 32 (5.57*e* − 48)=	5.43*e* − 02 (2.91*e* − 01)+	**1.34e − 32 (5.56e − 48)**
9	1.40*e* − 28 (1.34*e* − 28)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	1.68*e* − 30 (9.22*e* − 30)+	**0.00e + 00 (0.00e + 00)**
10	3.08*e* + 00 (1.90*e* + 00)+	4.20*e* − 27 (2.47*e* − 27)=	6.20*e* − 07 (7.93*e* − 07)+	1.22*e* − 01 (1.98*e* − 01)+	**3.54e − 27 (2.97e − 27)**
11	4.61*e* + 08 (8.15*e* + 07)+	3.63*e* + 04 (2.18*e* + 04)+	5.31*e* + 05 (2.42*e* + 04)+	1.07*e* + 06 (3.67*e* + 05)+	**1.23e + 04 (5.78e + 03)**
12	2.36*e* + 02 (1.52*e* + 02)+	8.77*e* − 01 (2.10*e* + 00)=	4.83*e* + 02 (4.62*e* + 02)+	6.49*e* + 03 (2.89*e* + 03)+	**5.24e − 01 (6.83e − 01)**
13	2.99*e* + 01 (1.97*e* + 01)+	1.26*e* + 01 (4.03*e* + 01)+	2.63*e* + 01 (2.70*e* + 01)+	8.97*e* + 01 (4.56*e* + 01)+	**9.30e − 01 (1.71e + 00)**
14	2.11*e* + 01 (2.82*e* − 02)+	2.11*e* + 01 (2.01*e* − 01)+	2.10*e* + 01 (5.57*e* − 02)+	2.11*e* + 01 (3.94*e* − 02)+	**2.07e + 01 (5.03e − 01)**
15	1.93*e* + 02 (3.67*e* + 01)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	1.89*e* + 00 (9.90*e* − 01)+	**0.00e + 00 (0.00e + 00)**
16	3.57*e* + 02 (1.19*e* + 01)+	4.84*e* + 01 (7.99*e* + 00)=	5.47*e* + 01 (1.03*e* + 01)+	1.26*e* + 02 (2.10*e* + 01)+	**4.70e + 01 (7.86e + 00)**
17	7.29*e* + 01 (1.19*e* + 00)+	5.19*e* + 01 (2.17*e* + 00)+	5.40*e* + 01 (2.94*e* + 00)+	**3.85e + 01 (4.07e + 00)**−	5.00*e* + 01 (2.69*e* + 00)

Total score	DE	JADE	*j*DE	SADE	**SAPSDE**
+	16	8	11	16	*
−	0	0	2	1	*
=	1	9	4	0	*

good global search ability and slow convergence speed. The DE can perform well on some functions. The SapsDE has good local search ability and global search ability at the same time.

4.4. Comparable Analysis of Convergence Speed and Success Rate. Table 5 reports the success rate (Sr) and the average number of function evaluations (NFE) by applying the five algorithms to optimize the 30-D, 50-D, and 100-D numerical functions f_2, f_3, f_{10}, and f_{15}. The success of an algorithm means that the algorithms can obtain a solution error measure no worse than the prespecified optimal value, that is, error measures $+1e-14$ for all problem with the number of FES less than the prespecified maximum number. The success rate is the proportion of success runs number among the total runs number. NFE is the average number of function evaluations required to find the global optima within the prespecified maximum number of FES when an algorithm is a success. Because the convergence graphs of the 30-D problems are similar to their 50-D and 100-D counterparts, they are given as represented here. Figure 1 illustrates the convergence characteristics of each algorithm for the four 30-dimensional benchmark functions.

From Table 5, we find that SapsDE almost achieves the fastest convergence speed at all functions, which is also displayed visually in Figure 1. Moreover, SapsDE obtains a higher success rate than the four DE variants in most instances.

4.5. Compared with ATPS. Recently, in [26], Zhu et al. proposed an adaptive population tuning scheme (ATPS) for DE. The ATPS adopts a dynamic population strategy to remove redundant individuals from the population according to its ranking order, perturb the population, and generate good

Table 4: Experimental results of 100-dimensional problems $f_1(x)$–$f_{17}(x)$, averaged over 30 independent runs with 1 000 000 FES.

Function	DE	JADE	*j*DE	SaDE	SapsDE
	Mean error (std Dev)	Mean error (std Dev)	Mean error (std Dev)	Mean error (std Dev)	Mean error (std Dev)
1	1.21e − 39 (1.21e − 39)+	6.89e − 189 (0.00e + 00)=	1.15e − 97 (1.34e − 97)+	2.82e − 92 (1.48e − 91)+	**1.70e − 198 (0.00e + 00)**
2	3.81e + 08 (1.57e + 08)+	1.20e + 00 (1.86e + 00)+	1.03e + 02 (4.06e + 01)+	1.75e + 02 (7.03e + 01)+	**2.65e − 01 (1.01e + 00)**
3	8.96e + 00 (7.84e − 01)+	9.68e − 01 (5.18e − 01)+	**1.28e − 14 (3.70e − 15)=**	2.92e + 00 (5.56e − 01)+	1.36e − 14 (2.07e − 15)
4	3.98e + 01 (9.40e + 00)+	1.97e − 03 (4.21e − 03)=	**0.00e + 00 (0.00e + 00)=**	3.82e − 02 (5.92e − 02)+	1.07e − 03 (3.28e − 03)
5	8.52e + 02 (5.26e + 01)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**	9.95e + 00 (2.75e + 00)+	**0.00e + 00 (0.00e + 00)**
6	1.00e + 01 (7.87e − 01)+	7.23e − 01 (9.71e − 02)+	3.67e − 01 (4.79e − 02)+	1.46e + 00 (2.97e − 01)+	**2.96e − 01 (3.17e − 02)**
7	5.57e + 05 (5.85e + 05)+	3.73e − 02 (8.14e − 02)=	**1.04e − 03 (5.68e − 03)=**	4.36e − 02 (1.12e − 01)=	1.24e − 02 (3.70e − 02)
8	3.69e + 06 (2.40e + 06)+	1.07e − 01 (4.05e − 01)=	1.01e − 01 (5.52e − 01)=	1.62e − 01 (6.86e − 01)=	**1.34e − 32 (5.56e − 48)**
9	3.71e + 03 (9.46e + 02)+	1.05e − 29 (2.43e − 29)=	2.73e − 29 (6.31e − 29)=	**1.68e − 30 (9.22e − 30)=**	5.47e − 30 (1.68e − 29)
10	3.82e + 05 (2.72e + 04)+	**6.51e − 15 (1.25e − 14)=**	3.04e + 01 (1.84e + 01)+	1.15e + 02 (3.38e + 01)+	8.18e − 15 (1.63e − 14)
11	2.29e + 09 (3.39e + 08)+	3.25e + 05 (1.42e + 05)+	2.28e + 06 (6.14e + 05)+	4.26e + 06 (9.98e + 05)+	**2.20e + 05 (8.44e + 04)**
12	4.47e + 05 (3.67e + 04)+	**7.80e + 03 (3.46e + 03)=**	2.69e + 04 (9.63e + 03)+	4.83e + 04 (1.06e + 04)+	8.59e + 03 (4.50e + 03)
13	2.40e + 08 (8.93e + 07)+	**1.20e + 00 (1.86e + 00)=**	9.59e + 01 (4.48e + 01)+	1.73e + 02 (4.20e + 01)+	1.72e + 00 (2.00e + 00)
14	2.13e + 01 (2.27e − 02)+	2.13e + 01 (8.41e − 02)+	2.13e + 01 (2.47e − 02)+	2.13e + 01 (2.12e − 02)+	**2.03e + 01 (4.56e − 01)**
15	8.58e + 02 (5.87e + 01)+	1.78e − 16 (5.42e − 16)=	**0.00e + 00 (0.00e + 00)**	2.57e + 01 (7.00e + 00)+	**0.00e + 00 (0.00e + 00)**
16	1.09e + 03 (3.06e + 01)+	1.55e + 02 (3.51e + 01)+	2.05e + 02 (2.94e + 01)+	4.36e + 02 (6.18e + 01)+	**4.80e + 01 (9.04e + 00)**
17	1.60e + 02 (1.70e + 00)+	1.14e + 02 (9.12e + 00)+	1.27e + 02 (3.55e + 00)+	1.05e + 02 (7.30e + 00)+	**5.22e + 01 (2.21e + 00)**

Total score	DE	JADE	*j*DE	SaDE	**SapsDE**
+	17	7	10	14	*
−	0	0	0	0	*
=	0	10	7	3	*

individuals. this APTS framework could be incorporated into several recently reported DE variants and achieves good performance on function optimization. In this section, we compare the SapsDE with the ATPS-DE variants, which are reported in [26], on nine 30-D benchmark functions. The parameter settings are the same as those in [26]. The averaged results of 30 independent runs are shown in the Table 6 (results for ATPS-DEs are taken from [26]). Obviously, from Table 6, we can observe that the number of "+" is more than that of "−". On the whole, the SapsDE outperforms the ATPS-DEs, respectively.

4.6. Time Complexity of SapsDE. In this section, we analyze time complexity of the SapsDE algorithm by using power regression $y = at^b$. Here t is dimension D, and y is the time in Table 7. Table 7 lists all time of the function with 30, 50, and 100 dimensions in 30 runs. After calculations, the results of power regression are also listed in Table 7. It can be found that the time complexity of our algorithm is less than $O(D^2)$.

4.7. Parameter Study. In this section, we use 30-D test functions f_1–f_{17} to investigate the impact of the initial NP value on SapsDE algorithm. The SapsDE algorithm runs 30 times on each function with three different initial NP values of 50, 100, and 200. We here present the results of mean and standard deviation for these functions in Table 8.

Furthermore, we study the robustness of s value by testing the 30-D benchmark functions f_1–f_{17}. Similarly, the SapsDE algorithm runs 30 times on each function with three different s values of 1, 3, and 5. The result of experiments is shown in Table 9.

Table 5: Experiment results of convergence speed and success rate.

	SapsDE NFE	Sr	JADE NFE	Sr	DE NFE	Sr	*j*DE NFE	Sr	SaDE NFE	Sr
$D = 30$										
f_2	99641	96.67%	131700	93.33%	—	0%	—	0%	—	0%
f_3	42162	100%	73600	100%	256200	100%	143700	100%	66250	86.70%
f_{10}	98897	100%	103800	100%	—	0%	—	0%	—	0%
f_{15}	83882	100%	171200	100%	—	0%	138700	100%	83800	86.70%
$D = 50$										
f_2	174475	100%	267000	87%	—	0%	—	0%	—	0%
f_3	57610	100%	91800	100%	—	0%	206800	100%	348200	10%
f_{10}	232765	100%	275400	100%	—	0%	—	0%	—	0%
f_{15}	131032	100%	281700	100%	—	0%	217400	100%	149100	6.67%
$D = 100$										
f_2	629805	93.30%	802900	70%	—	0%	—	0%	—	0%
f_3	765400	3.33%	—	0%	—	0%	543700	13.30%	—	0%
f_{10}	946216	76.70%	977200	83.30%	—	0%	—	0%	—	0%
f_{15}	209975	100%	424600	100%	—	0%	382200	100%	—	0%

Table 6: Experimental results of SapsDE and ATPS-DEs.

Function	*j*DE-APTS Mean error (std Dev)	CoDE-APTS Mean error (std Dev)	JADE-APTS Mean error (std Dev)	SapsDE Mean error (std Dev)
9	**0.00e + 00 (0.00e + 00)**	8.82*e* − 17 (7.06*e* − 16)+	**0.00e + 00 (0.00e + 00)**	**0.00e + 00 (0.00e + 00)**
10	7.93*e* − 11 (2.31*e* − 11)+	2.70*e* − 03 (6.73*e* − 03)+	**2.10e − 29 (4.86e − 29)**−	1.17*e* − 28 (1.07*e* − 28)
11	9.83*e* + 04 (4.42*e* + 04)+	1.06*e* + 05 (8.03*e* + 05)+	7.43*e* + 03 (6.04*e* + 03)=	**7.00e + 03 (4.29e + 03)**
12	9.17*e* − 02 (8.02*e* − 02)+	1.55*e* − 03 (3.98*e* − 02)=	**1.79e − 23 (9.76e − 23)**=	4.22*e* − 15 (1.25*e* − 14)
13	**3.87e − 01 (7.22e − 01)**=	6.12*e* + 00 (4.39*e* + 00)+	8.59*e* + 00 (3.08*e* + 01)+	1.49*e* + 00 (5.68*e* + 00)
14	2.09*e* + 01 (1.09*e* − 02)+	**2.07e + 01 (2.44e − 01)**+	2.09*e* + 01 (6.43*e* − 02)+	**2.07e + 01 (3.98e − 01)**
15	**0.00e + 00 (0.00e + 00)**	5.01*e* − 01 (3.40*e* − 01)+	5.42*e* − 12 (2.24*e* − 12)+	**0.00e + 00 (0.00e + 00)**
16	5.59*e* + 01 (2.47*e* + 01)+	5.10*e* + 01 (1.55*e* + 01)+	3.94*e* + 01 (1.16*e* + 01)+	**2.18e + 01 (3.58e + 00)**
17	**7.90e + 00 (6.99e + 00)**−	1.22*e* + 01 (6.30*e* + 00)−	2.19*e* + 01 (2.84*e* + 00)−	2.42*e* + 01 (1.68*e* + 00)
Total score	*j*DE-APTS	CoDE-APTS	JADE-APTS	SapsDE
+	5	7	4	*
−	1	1	2	*
=	3	1	3	*

From Tables 8 and 9, we find that SapsDE algorithm obtains similar optimization solutions on most of the functions by using three different initial NP values and s values, respectively. Therefore, the performance of SapsDE algorithm is less sensitive to the parameter NP initial value between 50 and 200 and s value between 1 and 5.

5. Conclusion

This paper presents a SapsDE algorithm which utilizes two DE strategies and three novel population size adjustment strategies. As evolution processes, the algorithm selects a more suitable mutation strategy. Meanwhile, three popula-

Table 7: Time complexity of SapsDE.

D	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
30	112.163	112.1049	99.51729	119.3885	109.8877	98.08094	128.1295	129.8256	105.0963
50	250.8682	214.8441	194.698	229.0735	220.8529	192.6483	264.1902	269.5691	210.9053
100	763.8241	733.7201	736.4833	827.2118	735.0107	651.3032	942.6226	947.4914	688.3214
a	$4.94E-01$	0.504	0.30967	0.45001	0.47659	0.42848	0.42052	0.43884	0.48612
b	1.5941	1.5726	1.6773	1.6219	1.5874	1.5831	1.6678	1.6603	1.5694
rss	6.0578	1391.343	1925.493	2275.271	772.9743	842.2943	1561.575	1335.442	596.6701
D	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	
30	132.4887	133.8268	141.0859	129.3506	140.4079	125.1071	136.9139	2292.366	
50	303.5054	290.2551	270.1778	249.1615	302.0771	246.1471	295.1372	6133.686	
100	1263.261	857.5675	1233.553	696.3027	914.3774	682.6456	879.7776	23393.98	
a	0.20834	0.69751	0.25784	1.0708	0.6923	1.0071	0.70463	3.2359	
b	1.8836	1.554	1.8234	1.403	1.5587	1.4129	1.5469	1.9295	
rss	2740.002	18.8848	11110.21	234.2553	90.6071	122.6194	45.898	110.6387	

Table 8: Robustness of NP value.

Function	$NP = 50$	$NP = 100$	$NP = 200$
	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)
1	$4.38e-126$ ($2.27e-125$)	$9.59e-126$ ($4.64e-125$)	$3.32e-120$ ($1.24e-119$)
2	$1.32e-01$ ($7.27e-01$)	$1.32e-01$ ($7.27e-01$)	$6.64e-01$ ($1.51e+00$)
3	$2.66e-15$ ($0.00e+00$)	$2.66e-15$ ($0.00e+00$)	$2.66e-15$ ($0.00e+00$)
4	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)
5	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)
6	$1.77e-01$ ($3.94e-02$)	$1.78e-01$ ($4.01e-02$)	$1.83e-01$ ($3.38e-02$)
7	$1.57e-32$ ($5.56e-48$)	$1.57e-32$ ($5.56e-48$)	$1.57e-32$ ($5.56e-32$)
8	$1.34e-32$ ($5.56e-48$)	$1.34e-32$ ($5.56e-48$)	$1.34e-32$ ($5.56e-48$)
9	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)
10	$1.17e-28$ ($1.07e-28$)	$9.85e-29$ ($9.93e-29$)	$9.19e-29$ ($1.00e-28$)
11	$7.00e+03$ ($4.29e+03$)	$7.02e+03$ ($5.74e+03$)	$7.16e+03$ ($5.09e+03$)
12	$4.22e-15$ ($1.25e-14$)	$6.17e-14$ ($2.32e-13$)	$5.16e-15$ ($1.37e-14$)
13	$1.49e+00$ ($5.68e+00$)	$2.65e+00$ ($7.29e+00$)	$1.64e+00$ ($5.76e+00$)
14	$2.07e+01$ ($3.98e-01$)	$2.07e+01$ ($3.74e-01$)	$2.08e+01$ ($3.19e-01$)
15	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)	$0.00e+00$ ($0.00e+00$)
16	$2.18e+01$ ($3.58e+00$)	$2.11e+01$ ($4.28e+00$)	$2.33e+01$ ($4.82e+00$)
17	$2.42e+01$ ($1.68e+00$)	$2.41e+01$ ($1.95e+00$)	$2.41e+01$ ($2.34e+00$)

tion size adjustment strategies will dynamically tune the NP value according to the improvement status learned from the previous generations. We have investigated the characteristics of SapsDE. Experiments show that SapsDE has low time complexity, fast convergence, and high success rate. The sensitivity analysis of initial NP value and s indicates that they have insignificant impact on the performance of the SapsDE.

We have compared the performance of SapsDE with the classic DE and four adaptive DE variants over a set of benchmark functions chosen from the existing literature and CEC 2005 special session on real-parameter optimization

Table 9: Robustness of s value.

Function	$s = 1$	$s = 3$	$s = 5$
	Mean error (Std Dev)	Mean error (Std Dev)	Mean error (Std Dev)
1	4.38e − 126 (2.27e − 125)	6.67e − 119 (3.65e − 118)	3.84e − 143 (1.57e − 142)
2	1.32e − 01 (7.27e − 01)	3.98e − 01 (1.21e + 00)	6.64e − 01 (1.51e + 00)
3	2.66e − 15 (0.00e + 00)	3.01e − 15 (1.08e − 15)	2.78e − 15 (6.48e − 16)
4	0.00e + 00 (0.00e + 00)	1.47e − 3 (3.46e − 03)	1.56e − 03 (4.45e − 03)
5	0.00e + 00 (0.00e + 00)	0.00e + 00 (0.00e + 00)	0.00e + 00 (0.00e + 00)
6	1.77e − 01 (3.94e − 02)	1.96e − 01 (1.82e − 02)	1.96e − 01 (1.82e − 02)
7	1.57e − 32 (5.56e − 48)	1.57e − 32 (5.56e − 48)	1.57e − 32 (5.56e − 48)
8	1.34e − 32 (5.56e − 48)	1.34e − 32 (5.56e − 48)	1.34e − 32 (5.56e − 48)
9	0.00e + 00 (0.00e + 00)	0.00e + 00 (0.00e + 00)	0.00e + 00 (0.00e + 00)
10	1.17e − 28 (1.07e − 28)	5.08e − 28 (2.51e − 28)	6.92e − 28 (5.60e − 28)
11	7.00e + 03 (4.29e + 03)	5.19e + 03 (4.64e + 03)	5.09e + 03 (3.17e + 03)
12	4.22e − 15 (1.25e − 14)	1.03e − 06 (2.14e − 06)	1.07e − 06 (3.07e − 06)
13	1.49e + 00 (5.68e + 00)	3.04e + 00 (1.22e + 01)	5.31e − 01 (1.37e + 00)
14	2.07e + 01 (3.98e − 01)	2.05e + 01 (4.07e − 01)	2.06e + 01 (3.24e − 01)
15	0.00e + 00 (0.00e + 00)	0.00e + 00 (0.00e + 00)	0.00e + 00 (0.00e + 00)
16	2.18e + 01 (3.58e + 00)	3.16e + 01 (6.60e + 00)	3.34e + 01 (7.00e + 00)
17	2.42e + 01 (1.68e + 00)	1.91e + 01 (4.04e + 00)	1.81e + 01 (4.14e + 00)

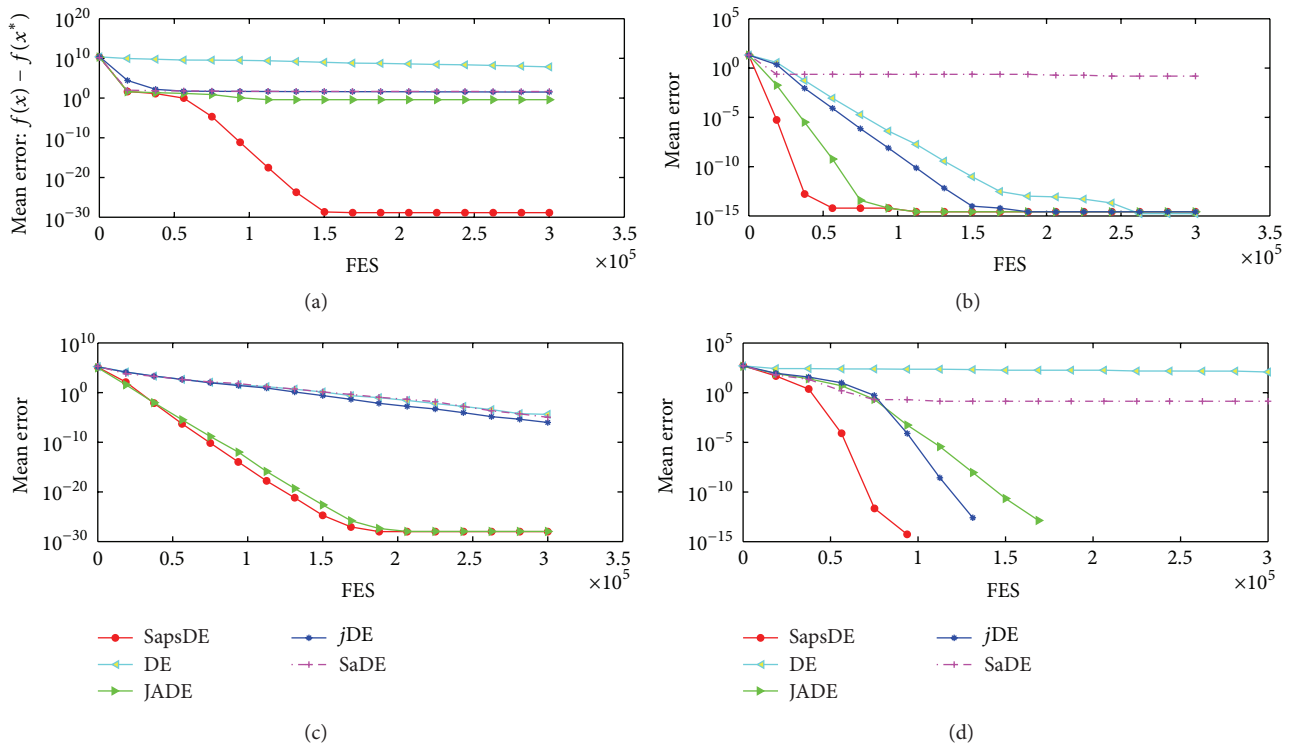

Figure 1: Performance of the algorithms for two 30-dimensional benchmark functions—(a) f_2; (b) f_3; (c) f_{10}; (d) f_{15}.

problems, and we conclude that the SapsDE algorithm is a highly competitive algorithm in 30-, 50-, and 100-dimensional problems. To summarize the results of the tests, the SapsDE algorithm presents significantly better results than the remaining algorithms in most cases. In our future work, we will focus on solving some real-world problems with the SapsDE. We will also tend to modify it for discrete problems.

Acknowledgment

This paper was partially supported by the National Natural Science Foundation of China (61271114).

References

[1] Y. Tang, H. Gao, J. Kurths, and J. Fang, "Evolutionary pinning control and its application in UAV coordination," *IEEE Transactions on Industrial Informatics*, vol. 8, pp. 828–838, 2012.

[2] A. Tuson and P. Ross, "Adapting operator settings in genetic algorithms," *Evolutionary Computation*, vol. 6, no. 2, pp. 161–184, 1998.

[3] Y. Tang, Z. Wang, H. Gao, S. Swift, and J. Kurths, "A constrained evolutioanry computation method for detecting controlling regions of cortical networks," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, pp. 1569–1581, 2012.

[4] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks*, pp. 1942–1948, December 1995.

[5] Y. Tang, Z. Wang, and J. A. Fang, "Controller design for synchronization of an array of delayed neural networks using a controllable probabilistic PSO," *Information Sciences*, vol. 181, no. 20, pp. 4715–4732, 2011.

[6] R. Storn and K. Price, "Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[7] Y. Tang, H. Gao, W. Zou, and J. Kurths, "Identifying controlling nodes in neuronal networks in different scales," *PLoS ONE*, vol. 7, Article ID e41375, 2012.

[8] J. Gomez, D. Dasgupta, and F. Gonzalez, "Using adaptive operators in genetic search," in *Proceedings of the Genetic and Evolutionary Computing Conference*, pp. 1580–1581, Chicago, Ill, USA, July 2003.

[9] S. Das and S. Sil, "Kernel-induced fuzzy clustering of image pixels with an improved differential evolution algorithm," *Information Sciences*, vol. 180, no. 8, pp. 1237–1256, 2010.

[10] S. Das, A. Abraham, and A. Konar, "Automatic clustering using an improved differential evolution algorithm," *IEEE Transactions on Systems, Man, and Cybernetics Part A*, vol. 38, no. 1, pp. 218–237, 2008.

[11] Y. Tang, Z. Wang, and J. Fang, "Feedback learning particle swarm optimization," *Applied Soft Computing*, vol. 11, pp. 4713–4725, 2011.

[12] J. Zhang and A. C. Sanderson, "An approximate Gaussian model of differential evolution with spherical fitness functions," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pp. 2220–2228, Singapore, September 2007.

[13] K. V. Prie, "An introduction to differential evolution," in *New Ideas in Oprimization*, D. Corne, M. Dorigo, and F. Glover, Eds., pp. 79–108, McGraw-Hill, London, UK, 1999.

[14] V. L. Huang, A. K. Qin, and P. N. Suganthan, "Self-adaptive differential evolution algorithm for constrained real-parameter optimization," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '06)*, pp. 17–24, Vancouver, Canada, July 2006.

[15] J. Brest, S. Greiner, B. Bošković, M. Mernik, and V. Zumer, "Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 6, pp. 646–657, 2006.

[16] J. Brest, V. Žumer, and M. S. Maučec, "Self-adaptive differential evolution algorithm in constrained real-parameter optimization," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '06)*, pp. 215–222, Vancouver, Canada, July 2006.

[17] J. Brest, B. Bošković, S. Greiner, V. Žumer, and M. S. Maučec, "Performance comparison of self-adaptive and adaptive differential evolution algorithms," *Soft Computing*, vol. 11, no. 7, pp. 617–629, 2007.

[18] J. Teo, "Exploring dynamic self-adaptive populations in differential evolution," *Soft Computing*, vol. 10, no. 8, pp. 673–686, 2006.

[19] Z. Yang, K. Tang, and X. Yao, "Self-adaptive differential evolution with neighborhood search," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08)*, pp. 1110–1116, Hong Kong, June 2008.

[20] A. K. Qin and P. N. Suganthan, "Self-adaptive differential evolution algorithm for numerical optimization," in *Proceedings of the IEEE Congress on Evolutionary Computation (IEEE CEC '05)*, pp. 1785–1791, Edinburgh, UK, September 2005.

[21] A. K. Qin, V. L. Huang, and P. N. Suganthan, "Differential evolution algorithm with strategy adaptation for global numerical optimization," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 398–417, 2009.

[22] R. S. Rahnamayan, H. R. Tizhoosh, and M. M. A. Salama, "Opposition-based differential evolution," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 1, pp. 64–79, 2008.

[23] N. Noman and H. Iba, "Accelerating differential evolution using an adaptive local search," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 1, pp. 107–125, 2008.

[24] T. Bäck, A. E. Eiben, and N. A. L. van der Vaart, "An empirical study on GAs "without parameters"," in *Proceedings of the 6th Conference on Parallel Problem Solving from Nature*, M. Schoenauer, K. Deb, G. Rudolph et al., Eds., vol. 1917 of *Lecture Notes in Computer Science*, pp. 315–324, Springer, Berlin, Germany, 2000.

[25] W. Zhu, J.-A. Fang, Y. Tang, W. Zhang, and W. Du, "Digital IIR filters design using differential evolution algorithm with a controllable probabilistic population size," *PLoS ONE*, vol. 7, Article ID e40549, 2012.

[26] W. Zhu, Y. Tang, J.-A. Fang, and W. Zhang, "Adaptive population tuning scheme for differential evolution," *Information Sciences*, vol. 223, pp. 164–191, 2013.

[27] J. Zhang and A. C. Sanderson, "JADE: adaptive differential evolution with optional external archive," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 945–958, 2009.

[28] P. N. Suganthan, N. Hansen, J. J. Liang et al., "Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization," KanGAL Report, Nanyang Technological University, Singapore; IIT Kanpur, Kanpur, India, 2005.

[29] X. Yao, Y. Liu, and G. Lin, "Evolutionary programming made faster," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 82–102, 1999.

[30] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*, Springer, Berlin, Germany, 2005.

[31] N. Noman and H. Iba, "Accelerating differential evolution using an adaptive local search," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 1, pp. 107–125, 2008.