IGI PUBLISHING
WWW.IGI-GLOBAL.COM

CALL FOR ARTICLES

International Journal of Swarm Intelligence Research

An official publication of the Information Resources Management Association

The Editor-in-Chief of the *International Journal of Swarm Intelligence Research* (IJSIR) would like to invite you to consider submitting a manuscript for inclusion in this scholarly journal.

ISSN 1947-9263
eISSN 1947-9271
Published quarterly

Mission

The mission of the *International Journal of Swarm Intelligence Research* (IJSIR) is to become a leading international and well-referred journal in swarm intelligence, nature-inspired optimization algorithms, and their applications. This journal publishes original and previously unpublished articles including research papers, survey papers, and application papers, to serve as a platform for facilitating and enhancing the information shared among researchers in swarm intelligence research areas ranging from algorithm developments to real-world applications.

Topics of Interest (include but are not limited to):

- Ant colony optimization
- Applications in bioengineering
- Applications in bioinformatics
- Applications in business
- Applications in control systems
- Applications in data mining and data clustering
- Applications in decision making
- Applications in distributed computing
- Applications in evolvable hardware
- Applications in finance and economics
- Applications in games
- Applications in graph partitioning
- Applications in information security
- Applications in machine learning
- Applications in planning and operations in industrial systems, transportation systems, and other systems
- Applications in power system
- Applications in supply-chain management
- Applications in wireless sensor networks
- Artificial immune system
- Constrained optimization
- Culture algorithm
- Differential Evolution
- Foraging algorithm
- Large scale optimization problems
- Modeling and analysis of biological collective systems such as social insects colonies, school, and flocking vertebrates
- Multi-objective optimization
- Optimization in dynamic and uncertain environment
- Other nature-inspired optimization algorithms
- Particle swarm optimization
- Scheduling and timetabling
- Swarm Robotics
- Other nature-inspired optimization algorithms

All submissions should be e-mailed to:
Yuhui Shi, Editor-in-Chief
yuhui.shi@xjtlu.edu.cn

Ideas for Special Theme Issues may be submitted to the Editor-in-Chief.

Please recommend this publication to your librarian. For a convenient easy-to-use library recommendation form, please visit:
http://www.igi-global.com/ijsir

International Journal of Swarm Intelligence Research

October-December 2011, Vol. 2, No. 4

Table of Contents

Research Articles

Chaos-Enhanced Firefly Algorithm with Automatic Parameter Tuning

Xin-She Yang, National Physical Lab, UK

ABSTRACT

Many metaheuristic algorithms are nature-inspired, and most are population-based. Particle swarm optimization is a good example as an efficient metaheuristic algorithm. Inspired by PSO, many new algorithms have been developed in recent years. For example, firefly algorithm was inspired by the flashing behaviour of fireflies. In this paper, the author extends the standard firefly algorithm further to introduce chaos-enhanced firefly algorithm with automatic parameter tuning, which results in two more variants of FA. The author first compares the performance of these algorithms, and then uses them to solve a benchmark design problem in engineering. Results obtained by other methods will be compared and analyzed.

Keywords: *Algorithm, Automatic Parameter Tuning, Chaos, Firefly Algorithm, Metaheuristics, Optimization, Particle Swarm Optimization*

1. INTRODUCTION

Search for optimality in many optimization applications is a challenging task, and search efficiency is one of the most important measure for an optimization algorithm. In addition, an efficient algorithm does not necessarily guarantee the global optimality is reachable. In fact, many optimization algorithms are only efficient in finding local optima. For example, classic hill-climbing or steepest descent method is very efficient for local optimization. Global optimization typically involves objective functions which can be multimodal and highly nonlinear. Thus, it is often very challenging to find global optimality, especially for large-scale optimization problems. Recent studies suggest that metaheuristic algorithms such as particle swarm optimization are promising in solving these tough optimization problems (Kennedy & Eberhart, 1995; Kennedy et al., 2001; Shi & Eberhart, 1998; Eberhart & Shi, 2000; Yang, 2008).

Most metaheuristic algorithms are nature-inspired, from simulated annealing (Kirkpatrick et al., 1983) to firefly algorithm (Yang, 2008, 2010a), and from particle swarm optimization (Kennedy & Eberhart, 1995; Kennedy et al., 2001) to cuckoo search (Yang & Deb, 2010). These algorithms have been applied to almost all areas of optimization, design, scheduling and planning, data mining, machine intelligence, and many others (Gandomi et al., in press; Talbi, 2009; Yang, 2010a). On the other hand,

DOI: 10.4018/jsir.2011100101

chaotic tunneling is an important phenomenon in complex systems (Tomsovic, 1994; Podolskiy & Narmanov, 2003; Kohler et al., 1998; Delande & Zakrzewski, 2003; Shudo & Ikeda, 1998; Shudo et al., 2009). Traditional wisdom in optimization is to avoid numerical instability and chaos. Contemporary studies suggest that chaos can assist some algorithms such as genetic algorithms (Yang & Chen, 2002). For example, metaheuristic algorithms often use randomization techniques to increase the diversity of the solutions generated during search iterations (Talbi, 2009; Yang, 2010a). The most common randomization techniques are probably local random walks and Lévy flights (Gutowski, 2001; Pavlyukevich, 2007; Yang 2010b).

The key challenge for global optimization is that nonlinearity leads to multimodality, which in turns will cause problems to almost all optimization algorithms because the search process may be trapped in any local valley, and thus may cause tremendous difficulty to the search process towards global optimality. Even with most well-established stochastic search algorithms such as simulated annealing (Kirkpatrick et al., 1983), care must be taken to ensure it can escape the local modes/optimality. Premature convergence may occur in many algorithms including simulated annealing and genetic algorithms. The key ability of an efficient global search algorithm is to escape local optima, to visit all modes and to converge subsequently at the global optimality.

In this paper, we will first analyze the recently developed firefly algorithm (FA) (Yang, 2008, 2010b). Under the right conditions, FA can have chaotic behaviour, which can be used as an advantage to enhance the search efficiency, because chaos allow fireflies to sample search space more efficiently. In fact, a chaotic tunnelling feature can be observed in FA simulations when a firefly can tunnel through multimodes and jump from one mode to another modes. This enables the algorithm more versatile in escaping the local optima, and thus can guarantee to find the global optimality. Chaotic tunneling is an important phenomenon in complex systems, but this is the first time that a chaotic tunneling is observed in an optimization algorithm. Through analysis and numerical simulations, we will highlight that intrinsic chaotic characteristics in the FA can enhance the search efficiency. Then, we will introduce automatic parameter tuning to the chaotic firefly algorithm and compare its performance against a set of diverse test functions. Finally, we will apply the FA with automatic parameter tuning to solve a design benchmark whose solutions will be compared with other results in the literature.

2. FIREFLY ALGORITHM

Firefly Algorithm (FA) was developed by Yang (2008, 2010b), which was based on the flashing patterns and behaviour of fireflies. In essence, each firefly will be attracted to brighter ones, while at the same time, it explores and searches for prey randomly. In addition, the brightness of a firefly is determined by the landscape of the objective function.

The movement of a firefly i is attracted to another more attractive (brighter) firefly j is determined by

$$x_i^{t+1} = x_i^t + \beta e^{-\gamma r_{ij}^2}(x_j^t - x_i^t) + \alpha \quad \varepsilon_i^t, \tag{1}$$

where α, β and γ are parameters. α controls the scale of randomization, β controls the attractiveness, while γ is a scaling factor. Here the second term is due to the attraction. The third term is randomization with α being the randomization parameter, and ε_i^t is a vector of random numbers drawn from a Gaussian distribution or other distributions such as Lévy flights. Obviously, for a given firefly, there are often many more attractive fireflies, then we can either go through all of them via a loop or use the most attractive one. For multiple modal problems, using a loop while moving toward each brighter one is usually more effective, though this will lead to a slight increase of algorithm complexity.

Here $\beta \in [0,1]$ is the attractiveness at $r = 0$, and $r_{ij} = \| x_i - x_j \|_2$ is the 2-norm or

Cartesian distance. For other problems such as scheduling, any measure that can effectively characterize the quantities of interest in the optimization problem can be used as the "distance" r. Furthermore, the randomization term can easily be extended to other distributions such as Lévy flights (Reynolds & Rhodes, 2009).

3. CHAOS-ENHANCED FA

In order to see the intrinsic tunneling ability, let us first carry out the convergence analysis for the firefly algorithm in a framework similar to Clerc and Kennedy's dynamical analysis (Clerc & Kennedy, 2002). For simplicity, we start from the equation for firefly motion without the randomness term

$$x_i^{t+1} = x_i^t + \beta e^{-\gamma r_{ij}^2}(x_j^t - x_i^t). \quad (2)$$

If we focus on a single agent, we can replace x_j^t by the global best g found so far, and we have

$$x_i^{t+1} = x_i^t + \beta e^{-\gamma r_i^2}(g - x_i^t), \quad (3)$$

where the distance r_i can be given by the ℓ_2 -norm $r_i^2 = \|g - x_i^t\|_2^2$. In an even simpler 1-D case, we can set $y_t = g - x_i^t$, and we have

$$y_{t+1} = y_t - \beta e^{-\gamma y_t^2} y_t. \quad (4)$$

We can see that γ is a scaling parameter which only affects the scales/size of the firefly movement. In fact, we can let $u_t = \sqrt{\gamma} y_t$ and we have

$$u_{t+1} = u_t[1 - \beta e^{-u_t^2}]. \quad (5)$$

These equations can be analyzed easily using the same methodology for studying the well-known logistic map

$$u_{t+1} = \lambda u_t(1 - u_t). \quad (6)$$

The chaotic map of equation (5) is shown in Figure 1, and the focus on the transition from periodic multiple states to chaotic behaviour is shown in the same figure.

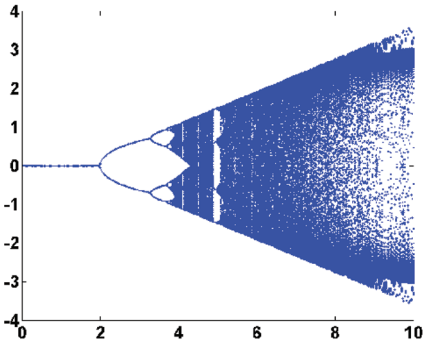
As we can see from Figure 1 that good convergence can be achieved for $\beta < 2$. There is a transition from periodic to chaos at $\beta \approx 4$. This may be surprising, as the aim of designing a metaheuristic algorithm is to try to find the optimal solution efficiently and accurately. However, chaotic behaviour is not necessarily a nuisance; in fact, we can use it to the advantage of the firefly algorithm.

It is worth pointing out that no explicit form of a random variable distribution can be found for the chaotic map of (5). However, simple chaotic characteristics from (6) can often be used as an efficient mixing technique for generating diverse solutions. Statistically, the logistic mapping (6) with $\lambda = 4$ for the initial states in (0,1) corresponds a beta distribution. From the algorithm implementation point of view, we can use higher attractiveness β during the early stage of iterations so that the fireflies can explore, even chaotically, the search space more effectively. As the search continues and convergence approaches, we can reduce the attractiveness β gradually, which may increase the overall efficiency of the algorithm. The simulations presented in the rest of this paper will confirm this.

4. AUTOMATIC PARAMETER TUNING

Apart from the population size n, there are three parameters in the firefly algorithm. They are α, β and γ, which control the randomness, attractiveness and modal scales, respectively. For most implementations, we can take

Figure 1. The chaotic map of the iteration formula (5) in the firefly algorithm and the transition between from periodic/multiple states to chaos



$\beta = O(1)$, $\alpha = O(1)$ and $\gamma = O(1)$. However, randomness reduction technique is often used as iterations continue, and this is often achieved by using an annealing-like exponential function

$$\alpha = \alpha_0 \eta^t, \tag{7}$$

or

$$\alpha \leftarrow \alpha\eta, \tag{8}$$

where $0 < \eta < 1$ is a cooling parameter. Typically, we can use $\alpha_0 = 1$ and $\eta = 0.9 \sim 0.99$. This equivalently introduces a cooling schedule to the firefly algorithm, as used in the traditional simulated annealing. Recently studies showed this works well (Yang, 2008). There may be better ways to tune this parameter and reduce randomness to be discussed later in this section.

It is worth pointing out that (1) is essentially a random walk biased towards the brighter fireflies. If $\beta_0 = 0$, it becomes a simple random walk.

As it is true for all metaheuristic algorithms, algorithm-dependent parameters can affect the performance of the algorithm of interest greatly, a natural question is whether we can automatically tune these parameters? If so, what is the best way to fine-tune these parameters?

For randomness reduction, it should be linked with the diversity of the current solutions. One simple way to automatically tune α is to set α as proportional to the standard deviation of the current solutions. However, for multimodal problems, this standard deviation should be calculated for each local mode among local subgroups of fireflies. For example, for two modes A and B with current best solutions x_a^* and x_b^*, respectively, the population will gradually subdivide into two main subgroups with population sizes of n_1 and n_2, respectively, one around A and one around B. There are two standard deviations σ_A and σ_B which should be calculated among the solutions relative to x_a^* and x_b^*, respectively. Then the overall α should be a function of σ_A and σ_B. The simplest way is to combine them by weighted average

$$\sigma = \frac{\sigma_A n_1 + \sigma_B n_2}{n_1 + n_2}, \quad n_1 + n_2 = n. \tag{9}$$

As iterations continue, σ decreases in general. If we set

$$\alpha = \zeta\sigma, \quad 0 < \zeta < 1, \tag{10}$$

then α is automatically associated with the scale of the problem of interest. In practice, η may be affected by the dimensions d, so in our implementation we used $\zeta = \sqrt{d/(2d+1)}$. The parameter γ should be linked with the scale L of the modes. A simple rule is that the change of the attractiveness term should be O(1) through the search landscape, which provide a simple relationship $\gamma = 1/\sqrt{L}$. Parameter β control the behavior of fireflies, however, its tuning is more subtle. From the above discussion of (5), when β is large, fireflies may experience chaotic behavior, and this can be used to enhance the search capability of the algorithm. In fact, from our intensive simulations, we have observed that fireflies can tunnel through all modes for multimodal function. This chaotic tunnelling effect of the algorithm can help to search the global optimality for highly nonlinear global optimization problems.

To demonstrate this, we now first use a nonlinear multimodal function, namely, Ackley's function:

$$f(x) = 20\exp[-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\,] + \exp[\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)] - (20+e),$$

which has the global maximum $f_* = 0$ at $x_* = (0,0,...,0)$ in the range of $-32.768 \le x_i \le 32.768$ where $i = 1,2,...,d$ and d is the number of dimensions. In the 2D case, Ackley's function is shown in Fig. 2. For 25 fireflies, a snapshot at $t = 15$ of search process using the firefly algorithm is shown in Fig. 3. If we ignore the randomness by setting $\alpha = 0$ and $\beta = 4$ all the time, then we can trace any one particular firefly, say, firefly number 5, its path of x-component displays a random-noise-like path. It is worth pointing out each firefly has the ability of tunneling through all modes, and distance of the tunnelling is controlled by the scaling factor γ and β.

During the iteration, if we reduce β gradually from a higher value, say, $\beta = 4$ to a lower value $\beta = 1$ by $\beta\eta^t + 1$ and also use equation (7), the algorithm can be expected to converge more quickly. So for the same firefly 5, if we reduce β gradually, as the iteration proceeds, this path will gradually settle down and converge to a global optimal point.

Now we have three version of FA: The standard version of FA with α as a cooling schedule, a chaos-enhanced FA with β reduced gradually, and the chaotic FA in combination with automatic parameter tuning (AutoFA). In the rest of the paper, we will carry out more testing and comparison of their performance.

5. NUMERICAL EXPERIMENTS

Various test functions in the literature are designed to test the performance of optimization algorithms. Any new optimization algorithm should also be validated and tested against these benchmark functions. In our simulations, we have used the following test functions.

De Jong's first function is essentially a sphere function

$$f(x) = \sum_{i=1}^{d} x_i^2, \quad x_i \in [-5.12, 5.12], \tag{12}$$

whose global minimum $f_* = 0$ occurs at $x_* = (0,0,...,0)$. Here d is the dimension.

The generalized Rosenbrock's function is given by

$$f(x) = \sum_{i=1}^{d-1}[(1-x_i)^2 + 100(x_{i+1} - x_i^2)^2], \tag{13}$$

which has a unique global minimum $f(x_*) = 0$ at $x_* = (1,1,...,1)$.

Schwefel's test function is multimodal

$$f(\ x) = \sum_{i=1}^{d}[-x_i \sin(\sqrt{|\ x_i\ |}\ )],\ -500 \le x_i \le 500, \tag{14}$$

whose global minimum $f_* = -418.9829d$ is at $x_i^* = 420.9687(i = 1,2,...,d)$.

Rastrigin's test function

$$f(\ x) = 10d + \sum_{i=1}^{d}[x_i^2 - 10\cos(2\pi x_i)], \tag{15}$$

has a unique global minimum $f_* = 0$ at $(0,0,...,0)$ in a hypercube $-5.12 \le x_i \le 5.12$ where $i = 1,2,...,d$.

Easom's test function has a sharp tip

$$f(x,y) = -\cos(x)\cos(y)\exp[-(x-\pi)^2 - (y-\pi)^2], \tag{16}$$

in the domain $(x,y) \in [-100,100] \times [-100,100]$. It has a global minimum of $f_* = -1$ at (π,π) in a very small region.

Rosenbrock's function

$$f(\ x) = \sum_{i=1}^{d-1}[(x_i - 1)^2 + 100(x_{i+1} - x_i^2)^2], \tag{17}$$

whose global minimum $f_* = 0$ occurs at $x_* = (1,1,...,1)$ in the domain $-5 \le x_i \le 5$ where $i = 1,2,...,d$. In the 2D case, it is often written as

$$f(x,y) = (x-1)^2 + 100(y - x^2)^2, \tag{18}$$

which is often referred to as the banana function.

The Michalewicz function

$$f(\ x) = -\sum_{i=1}^{d}\sin(x_i)[\sin(\frac{ix_i^2}{\pi})]^{2m}, \tag{19}$$

where $m = 10$ and $d = 1,2,...$. The global minimum $f_* \approx -1.801$ in 2-D occurs at $(2.20319, 1.57049)$,

Griewangk's test function has many local minima

$$f(\ x) = \frac{1}{4000}\sum_{i=1}^{d}x_i^2 - \prod_{i=1}^{d}\cos(\frac{x_i}{\sqrt{i}}) + 1, \tag{20}$$

but a unique global mimimum $f_* = 0$ at $(0,0,...,0)$ for all $-600 \le x_i \le 600$ where $i = 1,2,...,d$.

Yang's test function (Yang, 2010a)

$$f(\ x) = (\sum_{i=1}^{d}|\ x_i\ |)\exp[-\sum_{i=1}^{d}\sin(x_i^2)],\quad -2\pi \le x_i \le 2\pi, \tag{21}$$

which has a global minimum $f_* = 0$ at $(0,0,...,0)$.

Rosenbrock's stochastic function was extended by Yang (2010a)

$$f(\ x) = \sum_{i=1}^{d-1}[\varepsilon_i(x_i - 1)^2 + 100\varepsilon_{i+1}(x_{i+1} - x_i^2)^2], \tag{22}$$

whose global minimum $f_* = 0$ occurs at $x_* = (1,1,...,1)$ in the domain $-5 \le x_i \le 5$ where $i = 1,2,...,d$.

The functions used in Table 1 are (1) Michaelwicz ($d = 16$), (2) Rosenrbrock ($d = 16$), (3) De Jong ($d = 16$), (4) Schwefel ($d = 8$), (5) Ackley ($d = 16$), (6) Rastrigin, (7) Easom, (8) Griewangk, (9) Yang $d = 16$, (10) Robsenbrock's stochastic function ($d = 8$).

We ran the simulations for 50 times for a given accuracy of $\delta = 10^{-5}$, and the search stops when the best solution is found g_* is near the known solution x_*, that $||\ x_* - g_*|| \le \delta$. We then recorded the number of iterations for finding such best solutions. In this table, the second column corresponds to the average

Figure 2. Ackley's multimodal function

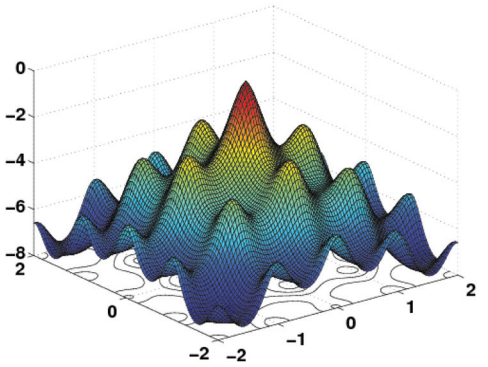


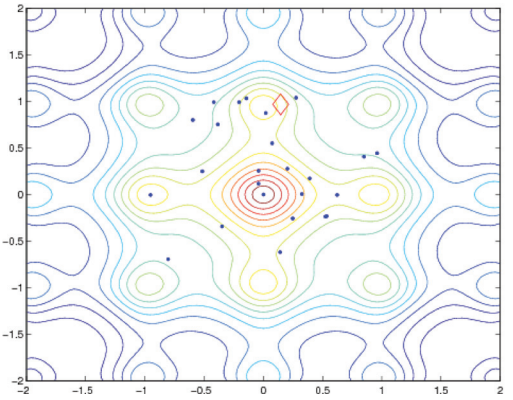
Table 1. Comparison of standard FA, chaotic FA and AutoFA

Test Functions	FA	Chaotic FA (ratio)	AutoFA (ratio)
(1)	3752 ± 725	0.154 ± 0.022	0.108 ± 0.015
(2)	7792 ± 2923	0.175 ± 0.024	0.123 ± 0.017
(3)	2319 ± 337	0.069 ± 0.014	0.054 ± 0.012
(4)	7540 ± 125	0.097 ± 0.018	0.072 ± 0.014
(5)	3172 ± 723	0.071 ± 0.012	0.051 ± 0.010
(6)	11981 ± 970	0.093 ± 0.011	0.069 ± 0.009
(7)	7925 ± 1799	0.145 ± 0.027	0.127 ± 0.024
(8)	12592 ± 3715	0.112 ± 0.019	0.089 ± 0.012
(9)	7390 ± 2189	0.079 ± 0.011	0.057 ± 0.009
(10)	9125 ± 2149	0.037 ± 0.014	0.330 ± 0.049

number of iterations and its standard deviation. The third column is the average ratio of the number of iterations of chaotic FA to the number of iterations for the standard FA when $\beta = 1$ (no chaos). The fourth column is the average ratio of the number of iterations of AutoFA to that of standard FA. These ratios reflect the computational effort saved. For example, if the average ratio is about 0.1, than about 90% of the computing effort is saved, that is the efficiency has been increased by a factor of about 10.

We can see that the chaos-enhanced firefly algorithm indeed can improve its search efficiency significantly.

Figure 3. The snapshot of 25 fireflies during iteration $t = 15$



6. DESIGN OPTIMIZATION

There are many design benchmarks in the literature, however, the results are fragmental, as not all results are available and comparable. Here we select a well-known welded beam design, which has many results obtained by other methods in the literature (Ragsdell & Phillips, 1976; Cagnina et al., 2008; Gandomi et al., in press-a, in press-b). The problem typically has four design variables: the width w and length L of the welded area, the depth h and thickness h of the main beam. The objective is to minimise the overall fabrication cost, under the appropriate constraints of shear stress τ, bending stress σ, buckling load P and maximum end deflection δ.

The problem can be written as

$$minimise\ f(\ x) = 1.10471w^2L + 0.04811dh(14.0 + L), \tag{23}$$

subject to

$$\begin{aligned} &g_1(\ x) = w - h \le 0,\\ &g_2(\ x) = \delta(\ x) - 0.25 \le 0,\\ &g_3(\ x) = \tau(\ x) - 13,600 \le 0,\\ &g_4(\ x) = \sigma(\ x) - 30,000 \le 0,\\ &g_5(\ x) = 0.10471w^2 + 0.04811hd(14 + L) - 5.0 \le 0,\\ &g_6(\ x) = 0.125 - w \le 0,\\ &g_7(\ x) = 6000 - P(\ x) \le 0, \end{aligned} \tag{24}$$

where

$$\sigma(\ x) = \frac{504,000}{hd^2}, \qquad Q = 6000(14 + \frac{L}{2}),$$

$$D = \frac{1}{2}\sqrt{L^2 + (w + d)^2}, \qquad J = \sqrt{2}\ wL[\frac{L^2}{6} + \frac{(w + d)^2}{2}],$$

$$\delta = \frac{65,856}{30,000hd^3}, \qquad \beta = \frac{QD}{J},$$

$$\alpha = \frac{6000}{\sqrt{2}wL}, \qquad \tau(\ x) = \sqrt{\alpha^2 + \frac{\alpha\beta L}{D} + \beta^2},$$

$$P = 0.61423 \times 10^6\ \frac{dh^3}{6}(1 - \frac{d\sqrt{30\ /\ 48}}{28}).$$

The simple limits or bounds are $0.1 \le L, d \le 10$ and $0.1 \le w, h \le 2.0$. This benchmark has been solved by many different methods, including simulated annealing (Hedar & Fukushima, 2006), genetic algorithms (Deb, 1991), particle swarm optimization (He et al., 2004; Cagnina et al., 2008), harmony search (Lee & Geem, 2004), differential evolution (Zhang et al., 2008) and firefly algorithm in this study.

It is worth pointing out that the constraints should be handled appropriately. In this case, we have used the penalty functions to incorporate the above nonlinear constraints (Yang, 2010a). Using our chaotic firefly algorithm with automatic parameter tuning, we have the following optimal solution

Table 2. Welded beam design

Refs	Method	w	L	d	h	cost	Number of function evaluations
Deb	GA	0.2489	6.1730	8.1789	0.2533	2.4331	320,080
He et al.	PSO	0.2444	6.2175	8.2915	0.2444	2.3810	30,000
Cagnina et al.	PSO	0.2057	3.4705	9.0366	0.2057	1.7248	24,000
Hedar & Fukushima	SA	0.2444	6.2158	8.2939	0.2444	2.3811	56,243
Lee & Geem	HS	0.2442	6.2231	8.2915	0.2443	2.381	110,000
Zhang et al.	DE	0.2444	6.2175	8.2915	0.2444	2.3810	24,000
This study	AutoFA	0.2057	3.4705	9.0366	0.2057	1.7248	20,000

$$x_* = (w, L, d, h) = (0.20573, 3.47049, 9.03662, 0.20573), \quad (26)$$

with

$$f(x^*)_{\min} = 1.72485. \quad (27)$$

Our results are the same or better than the results obtained by other methods as summarized in Table 2.

From the above validation, comparison and benchmark design, we can see that chaotic FA with automatic parameter tuning is very efficient. Good convergence can be obtained by chaos-assisted tunnelling and automatic parameter adjustment. Effect and improvements become significant for multimodal problems.

6. CONCLUSION

Search for optimality in complex systems and global optimization problems require efficient algorithms. Metaheuristic algorithms such as particle swarm optimization and firefly algorithm are becoming very powerful. We have used a dynamical system approach to study the convergence property of the firefly algorithm and discovered its intrinsic chaotic tunneling ability. This property can be used as an advantage to enhance search efficiency of the algorithm. For multimodal optimization problems, there is a risk for any algorithm to get trapped in local optima. Chaos-assisted tunneling in the firefly algorithm makes it particular suitable for dealing with nonlinear, multimodal optimization problems. Our analysis and numerical experiments indeed demonstrated that chaotic tunneling can increase the search efficiency significantly.

An important topic for further research is to vary the scheme of automatic parameter tuning. The present study presents just one of many ways for automatic tuning of algorithm-dependent parameters. Other methods may be more appropriate and more efficient for different types of problems. In addition, more studies are highly needed to investigate whether this approach can be directly applied to other algorithms for automatic parameter tuning.

Further research can focus on the theoretical framework and extensive numerical studies on how an algorithm can be enhanced by chaotic tunneling, and thus may show insight into the

working of an efficient algorithm. Such studies may help to design new generation truly intelligent optimization algorithms.

REFERENCES

Cagnina, L. C., Esquivel, S. C., & Coello, C. A. (2008). Solving engineering optimization problems with the simple constrained particle swarm optimizer. *Informatica*, *32*, 319–326.

Clerc, M., & Kennedy, J. (2002). The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, *6*, 58–73. doi:10.1109/4235.985692

Deb, K. (1991). Optimal design of a welded beam via genetic algorithms. *AIAA Journal*, *29*(11), 2013–2015. doi:10.2514/3.10834

Delande, D., & Zakrzewski, J. (2003). Experimentally attainable example of chaotic tunneling: The hydrogen atom in parallel static electric and magnetic fields. *Physical Review A.*, *68*(6), 062110. doi:10.1103/PhysRevA.68.062110

Eberhart, E. C., & Shi, Y. (2000). Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation* (Vol. 1, pp. 84-88).

Gandomi, A. H., Yang, X. S., & Alavi, A. H. (in press). -a). Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems. *Engineering with Computers*.

Gandomi, A. H., Yang, X. S., & Alavi, A. H. (in press). -b). Mixed variable structural optimization using firefly algorithm. *Computers & Structures*.

Gutowski, M. (2001). *Lévy flights as an underlying mechanism for global optimization algorithms.* Retrieved from http://arxiv.org/abs/math-ph/0106003

He, S., Prempain, E., & Wu, Q. H. (2004). An improved particle swarm optimizer for mechanical design optimization problems. *Engineering Optimization*, *36*(5), 585–605. doi:10.1080/03052150410001704854

Hedar, A. R., & Fukushima, M. (2006). Derivative-free simulated annealing method for constrained continuous global optimization. *Journal of Global Optimization*, *35*(4), 521–649. doi:10.1007/s10898-005-3693-z

Kennedy, J., & Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks* (pp. 1942-1948).

Kennedy, J., Eberhart, R. C., & Shi, Y. (2001). *Swarm intelligence*. San Francisco, CA: Morgan Kaufmann.

Kirkpatrick, S., Gellat, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 670–680. doi:10.1126/science.220.4598.671

Kohler, S., Utermann, R., Hagnni, R., & Dittrich, T. (1998). Coherent and incoherent chaotic tunneling near singlet-doublet crossings. *Physical Review E: Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, *58*, 7219–7230. doi:10.1103/PhysRevE.58.7219

Lee, K. S., & Geem, Z. W. (2004). A new meta-heuristic algorithm for continues engineering optimization: harmony search theory and practice. *Computer Methods in Applied Mechanics and Engineering*, *194*, 3902–3933. doi:10.1016/j.cma.2004.09.007

Pavlyukevich, I. (2007). Lévy flights, non-local search and simulated annealing. *Journal of Computational Physics*, *226*, 1830–1844. doi:10.1016/j.jcp.2007.06.008

Podolskiy, V. A., & Narmanov, E. E. (2003). Semiclassical description of chaos-assisted tunneling. *Physical Review Letters*, *91*, 263601. doi:10.1103/PhysRevLett.91.263601

Ragsdell, K., & Phillips, D. (1976). Optimal design of a class of welded structures using geometric programming. *Journal of Engineering for Industry*, *98*, 1021–1025. doi:10.1115/1.3438995

Reynolds, A. M., & Rhodes, C. J. (2009). The Lévy flight paradigm: random search patterns and mechanisms. *Ecology*, *90*, 877–887. doi:10.1890/08-0153.1

Shi, Y., & Eberhart, R. C. (1998). A modified particle swarm optimizer. In *Proceedings of the IEEE International Conference on Evolutionary Computation* (pp. 69-73).

Shudo, A., & Ikeda, K. S. (1998). Chaotic tunneling: a remarkable manifestation of complex classical dynamics in non-integrable quantum phenomena. *Physica D. Nonlinear Phenomena*, *115*, 234–292. doi:10.1016/S0167-2789(97)00239-X

Shudo, A., Ishii, Y., & Ikeda, K. S. (2009). Julia sets and chaotic tunneling: II. *Journal of Physics A . Mathematical and Theoretical*, *42*, 265102. doi:10.1088/1751-8113/42/26/265102

Talbi, E.-G. (2009). *Metaheuristics: From design to implementation*. New York, NY: John Wiley & Sons.

Tomsovic, S. (1994). Chao-assisted tunneling. *Physical Review E: Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, *50*, 145–162. doi:10.1103/PhysRevE.50.145

Yang, L. J., & Chen, T. L. (2002). Applications of chaos in genetic algorithms. *Communications in Theoretical Physics*, *38*, 168–192.

Yang, X. S. (2008). *Nature-inspired metaheuristic algorithms*. Beckington, UK: Luniver Press.

Yang, X. S. (2010a). *Engineering optimization: An introduction with metaheuristic applications*. New York, NY: John Wiley & Sons. doi:10.1002/9780470640425

Yang, X. S. (2010b). Firefly algorithm, stochastic test functions and design optimisation. *International Journal of Bio-Inspired Computation*, *2*, 78–84. doi:10.1504/IJBIC.2010.032124

Yang, X. S., & Deb, S. (2010). Engineering optimization by cuckoo search. *International Journal of Mathematical Modelling & Numerical Optimization*, *1*, 330–343. doi:10.1504/IJMMNO.2010.035430

Zhang, M., Luo, W., & Wang, X. (2008). Differential evolution with dynamic stochastic selection for constrained optimization. *Information Science*, *178*(15), 3043–3074. doi:10.1016/j.ins.2008.02.014

Xin-She Yang received his DPhil in Applied Mathematics from Oxford University, and he has been the recipient of Garside Senior Scholar Award in Mathematics of Oxford University. He worked at Cambridge University for 5 years and is now a Senior Research Scientist at National Physical Laboratory. He has written 7 books and published more than 110 papers. He is the Editor-in-Chief of Int. J. Mathematical Modelling and Numerical Optimisation. He is also a Guest Professor of Harbin Engineering University, China. He is the vice chair of IEEE CIS task force on business intelligence and knowledge management. He is the inventor of a few metaheuristic algorithms, including bat algorithm, eagle strategy, firefly algorithm, cuckoo search and virtual bee algorithm.

Swarm Intelligence for Non-Negative Matrix Factorization

Andreas Janecek, University of Vienna, Austria

Ying Tan, Peking University, China

ABSTRACT

The Non-negative Matrix Factorization (NMF) is a special low-rank approximation which allows for an additive parts-based and interpretable representation of the data. This article presents efforts to improve the convergence, approximation quality, and classification accuracy of NMF using five different meta-heuristics based on swarm intelligence. Several properties of the NMF objective function motivate the utilization of meta-heuristics: this function is non-convex, discontinuous, and may possess many local minima. The proposed optimization strategies are two-fold: On the one hand, a new initialization strategy for NMF is presented in order to initialize the NMF factors prior to the factorization; on the other hand, an iterative update strategy is proposed, which improves the accuracy per runtime for the multiplicative update NMF algorithm. The success of the proposed optimization strategies are shown by applying them on synthetic data and data sets coming from the areas of spam filtering/email classification, and evaluate them also in their application context. Experimental results show that both optimization strategies are able to improve NMF in terms of faster convergence, lower approximation error, and better classification accuracy. Especially the initialization strategy leads to significant reductions of the runtime per accuracy ratio for both, the NMF approximation as well as the classification results achieved with NMF.

Keywords: *Differential Evolution, Email Classification, Fireworks Algorithm, Fish School Search, Genetic Algorithms, NMF Initialization, Nonnegative Matrix Factorization (NMF), Particle Swarm Optimization*

1. INTRODUCTION

Low-rank approximations are utilized in several content based retrieval and data mining applications, such as text and multimedia mining, web search, etc. and achieve a more compact representation of the data with only limited loss in information. They reduce storage and runtime requirements, and also reduce redundancy and noise in the data representation while capturing the essential associations. The Non-negative Matrix Factorization (NMF) (Lee & Seung, 1999) leads to a low-rank approximation which satisfies non-negativity constraints. NMF approximates a data matrix A by $A \approx WH$, where W and H are the *NMF factors*. NMF requires *all entries* in A, W and H to be zero or positive. Contrary to other low-rank approximations such as the Singular Value Decomposition (SVD), these constraints force NMF to produce so-called "additive parts-based" representations. This is an impressive

DOI: 10.4018/jsir.2011100102

benefit of NMF, since it makes the interpretation of the NMF factors much easier than for factors containing positive and negative entries (Berry, Browne, Langville, Pauca, & Plemmons, 2007; Janecek & Gansterer, 2010; Lee & Seung, 1999).

The NMF is usually not unique if different initializations of the factors W and H are used. Moreover, there are several different NMF algorithms which all follow different strategies (e.g., mean squared error, least squares, gradient descent,...) and produce different results. Mathematically, the goal of NMF is to find a "good" (ideally the best) solution of an optimization problem with bound constraints in the form $\min_{x\in\Omega} f(x)$, where $f : \mathbb{R}^N \rightarrow \mathbb{R}$ is the nonlinear objective function of NMF, and Ω is the feasible region (for NMF, Ω is restricted to non-negative values). f is usually not convex, discontinuous and may possess many local minima (Stadlthanner, Lutter, Theis, Lang, Tome, Georgieva, & Puntonet, 2007). Since meta-heuristic optimization algorithms are known to be able to deal well with such difficulties they seem to be a promising choice for improving the quality of NMF. Over the last decades nature-inspired meta-heuristics, including those based on swarm intelligence, have gained much popularity due to their applicability for various optimization problems. They benefit from the fact that they are able to find acceptable results within a reasonable amount of time for many complex, large and dynamic problems (Blackwell, 2007). Although they lack the ability to guarantee the optimal solution for a given problem (comparably to NMF), it has been shown that they are able to tackle various kinds of real-world optimization problems (Chiong, 2009). Meta-heuristics as well as the principles of NMF are in accordance with the *law of sufficiency* (Kennedy, Eberhart, & Shi, 2001): If a solution to a problem is good enough, fast enough and cheap enough, then it is sufficient.

In this article we present two different strategies for improving the NMF using five optimization algorithms based on swarm intelligence and evolutionary computing: Particle Swarm Optimization (PSO), Genetic Algorithms (GA), Fish School Search (FSS), Differential Evolution (DE), and Fireworks Algorithm (FWA). All algorithms are population based and can be categorized into the fields of *swarm intelligence* (PSO, FSS, FWA), *evolutionary algorithms* (GA), and a combination thereof (DE). The goal is to find a solution with smaller overall error at convergence, and/or to speed up convergence of NMF (i.e., smaller approximation error for a given number of NMF iterations) compared to identical NMF algorithms without applied optimization strategy. Another goal is to increase the classification accuracy in cases where NMF is used as dimensionality reduction method for machine learning applications. The concepts of the two optimization strategies are the following: In the first strategy, meta-heuristics are used to initialize the factors W and H in order to minimize the NMF objective function *prior* to the factorization. The second strategy aims at iteratively improving the approximation quality of NMF during the first iterations.

The proposed optimization strategies can be considered successful if they are able to improve the NMF in terms of either (*i*) faster convergence (i.e., better accuracy per runtime) (*ii*) lower final approximation error, (*iii*) or better classification accuracy. The optimization of different rows of W and different columns of H can be split up into several partly independent sub-tasks and can thus be executed concurrently. Since this allows for a parallel and/or distributed computation of both update strategies, we also discuss parallel implementations of the proposed optimization strategies. Experimental results show that both strategies, the initialization of NMF factors as well as an iterative update during the first iterations, are able to improve the NMF in terms of faster convergence, lower approximation error, and/or better classification accuracy.

1.1. Related Work

The work by Lee and Seung (1999) is known as a standard reference for NMF. The original *Multiplicative Update* (MU) algorithm introduced

in this article provides a good baselines against which other algorithms, e.g., the *Alternating Least Squares* algorithm (Paatero & Tapper, 1994), the *Gradient Descent* algorithm (Lin, 2007), ALSPGRAD (Lin, 2007), quasi Newton-type NMF (Kim & Park, 2008), fastNMF and bayesNMF (Schmidt & Laurberg, 2008), etc. have to be judged. While the MU algorithm is still the fastest NMF algorithm per iteration and a good choice if a very fast and rough approximation is needed, ALSPGRAD, fastNMF and bayesNMF have shown to achieve a better approximation at convergence compared to many other NMF algorithms (Janecek, Schulze-Grotthoff et al., 2011).

NMF Initialization

Only few algorithms for non-random NMF initialization have been published. Wild, Curry, and Dougherty (2004) used spherical k -means clustering to group column vectors of A as input for W . A similar technique was used in Xue, Tong, Chen, and Chen (2008). Another clustering-based method of structured initialization designed to find spatially localized basis images can be found in Kim and Park (2008). Boutsidis and Gallopoulos (2008) used an initialization technique based on two SVD processes called *nonnegative double singular value decomposition* (NNDSVD). Experiments indicate that this method has advantages over the centroid initialization in Wild, Curry, and Dougherty (2004) in terms of faster convergence.

NMF and Meta-Heuristics

So far, only few studies can be found that aim at combining NMF and meta-heuristics, most of them are based on Genetic Algorithms (GAs). In Stadlthanner et al. (2007), the authors have investigated the application of GAs on sparse NMF for microarray analysis, while Snásel, Platos, and Kromer (2008) have applied GAs for boolean matrix factorization, a variant of NMF for binary data based on Boolean algebra. However, the methods presented in these studies are barely connected to the techniques presented in this article. In two preceding studies (Janecek & Tan 2011a, 2011b), we have introduced the basic concepts of the proposed update strategies.

In this article we extend our preliminary work in several ways by the following new contributions. At first, we evaluate our methods on synthetic data as well as on data sets coming from the areas of spam filtering/email classification. This allows us to evaluate the proposed methods in the application context of the applied data sets. In other words, we are now able to investigate the quality of the NMF not only in terms of approximation accuracy but also in terms of *classification* accuracy achieved with the approximated data sets as well as with the basic vectors of the NMF factor W. Within this evaluation process we consider two different classification settings, a static setting where NMF is computed on the complete data set (training and test data), and a dynamic setting where NMF can be applied dynamically to new data. Moreover, we present a detailed evaluation of the runtime performance of the proposed update strategies, and, finally, we are able to compare the performance of our strategies with each other using the same parameter settings, data sets, and hardware set-up.

1.2. Notation

A matrix is represented by an uppercase italic letter (A , B , Σ , …), a vector by a lowercase bold letter ($\mathbf{u}$, $\mathbf{x}$,, $\mathbf{q}_1$, …), and a scalar by a lowercase Greek letter (λ , μ , …). The i^{th} row vector of a matrix D is represented as $\mathbf{d}_i^r$, and the j^{th} column vector of D as $\mathbf{d}_j^c$. Matrix-matrix multiplications are denoted by "*", element-wise multiplications by "$\cdot$", and element-wise divisions by "$./$".

1.3. Synopsis

In Section 2 we briefly review low-rank approximations and NMF algorithms. In Section 3 we summarize the swarm intelligence algorithms used in this article, and in Section 4 we present the proposed optimization strategies for NMF based on them. Moreover, we discuss differ-

ent classification methods based on NMF. In Sections 5 and 6 we evaluate our methods and discuss the achieved results. Finally, in Section 7 we conclude our work and summarize ongoing and future research activities in this area.

2. LOW RANK APPROXIMATIONS

Given a data matrix $A \in \mathbb{R}^{m \times n}$ whose n columns represent instances and whose m rows contain the values of a certain feature for the instances, most low-rank approximations reduce the dimensionality by representing the original data as accurately as possible with linear combinations of the original instances and/or features. Mathematically, A is replaced with another matrix A_k with usually much smaller rank. In general, a closer approximation means a better factorization. However, it is highly likely that in some applications specific factorizations might be more desirable compared to other solutions.

The most important low-rank approximation techniques are the Singular Value Decomposition (SVD) (Berry, 1992) and the closely related Principal Component Analysis (PCA) (Jolliffe, 2002). Traditionally, the PCA uses the eigenvalue decomposition to find eigenvalues and eigenvectors of the covariance matrix Cov(A) of A. Then the original data matrix A can be approximated by $A_k := AQ_k$, with $Q_k = [\mathbf{q}_1, ..., \mathbf{q}_k]$, where $\mathbf{q}_1, ..., \mathbf{q}_k$ are the first k eigenvectors of Cov(A). The SVD decomposes A into a product of three matrices such that $A = U\Sigma V^\top$, where Σ contains the singular values along the diagonal, and U and V are the singular vectors. The reduced rank SVD to A can be found by setting all but the first k largest singular values equal to zero and using only the first k columns of U and V, such that $A_k := U_k \Sigma_k V_k^\top$. Other well-known low-rank approximation techniques comprise Factor Analysis, Independent Components Analysis, Multidimensional Scaling such as Fastmap or ISOMAP, or Locally Linear Embedding (LLE), which are all summarized in Tan, Steinbach, and Kumar (2005).

Amongst all possible rank k approximations, the approximation A_k calculated by SVD and PCA is the best approximation in the sense that $|| A - A_k ||_F$ is as small as possible (cf. Berry, Drmac, & Jessup, 1999). In other words, SVD and PCA give the closest rank k approximation of a matrix, such that $|| A - A_k ||_F \leq || A - B_k ||_F$, where B_k is *any* matrix of rank k, and $|| . ||_F$ is the Frobenius norm, which is defined as $(\sum |a_{ij}|^2)^{1/2} = || A ||_F$. However, the main drawback of PCA and SVD refers to the interpretability of the transformed features. The resulting orthogonal matrix factors generated by the approximation usually do not allow for direct interpretations in terms of the original features because they contain positive ***and*** negative coefficients (Zhang, Berry, Lamb, & Samuel, 2009). In many application domains, a negative quantification of features is meaningless and the information about how much an original feature contributes in a low-rank approximation is lost. The presence of negative, meaningless components or factors may influence the entire result. This is especially important for applications where the original data matrix contains only positive entries, e.g., in text-mining applications, image classification, etc. If the factor matrices of the low-rank approximation were constrained to contain only positive or zero values, the original meaning of the data could be preserved better.

2.1. Non-negative Matrix Factorization (NMF)

The NMF leads to special low-rank approximations which satisfy these non-negativity constraints. NMF requires that all entries in A, W and H are zero or positive. This makes the interpretation of the NMF factors much easier and enables NMF a non-subtractive combination of parts to form a whole (Lee & Seung, 1999). The NMF consists of reduced rank *non-negative* factors $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ with $k \ll min\{m, n\}$ that approximate a matrix

$A \in \mathbb{R}^{m \times n}$ by $A \approx WH$, where the approximation WH has rank at most k. The nonlinear optimization problem underlying NMF can generally be stated as

$$\min_{W,H} f(W,H) = \min_{W,H} \frac{1}{2} \| A - WH \|_F^2 . \quad (1.1)$$

The Frobenius norm $||\,.\,||_F$ is commonly used to measure the error between the original data A and the approximation WH, but other measures such as the Kullback-Leibler divergence are also possible (Lee & Seung, 2001). The error between A and WH is usually stored in a distance matrix $D = A - WH$ (cf. Figure 1). Unlike the SVD, the NMF is not unique, and convergence is not guaranteed for all NMF algorithms. If they converge, then usually to local minima only (potentially different ones for different algorithms). Nevertheless, the data compression achieved with only local minima has been shown to be of desirable quality for many data mining applications (Langville, Meyer, & Albright, 2006). Moreover, for some specific problem settings a smaller residual $D = A - WH$ (a smaller error) may not necessarily improve of the solution of the actual application (e.g., classification task) compared to a rather coarse approximation. However, as analyzed in Janecek and Gansterer (2010) a closer NMF approximation leads to qualitatively better classification results and turns out to achieve significantly more stable results.

NMF Initialization

Algorithms for computing NMF are iterative and require initialization of the factors W and H. NMF unavoidably converges to local minima, probably different ones for different initialization (cf. Boutsidis & Gallopoulos, 2008). Hence, random initialization makes the experiments unrepeatable since the solution to Equ.1.1 is not unique in this case. A proper non-random initialization can lead to faster error reduction and better overall error at convergence. Moreover, it makes the experiments repeatable. Although the benefits of good NMF initialization techniques are well known in the literature, most studies use random initialization (cf. Boutsidis & Gallopoulos, 2008). Since some initialization procedures can be rather costly in terms of runtime the trade-off between computational cost in the initialization step and the computational cost of the actual NMF algorithm need to be balanced carefully. In some situations, an expensive preprocessing step may overwhelm the cost savings in the subsequent NMF update steps.

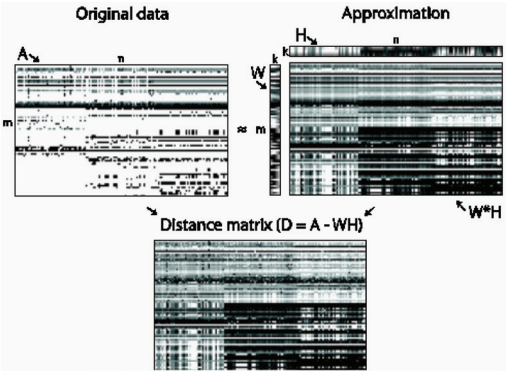
General Structure of NMF

In the basic form of NMF (*Algorithm 1*), W and H are initialized randomly and the whole algorithm is repeated several times (*maxrepetition*). In each repetition, NMF update steps are processed until a maximum number of iterations is reached (*maxiter*). These update steps are algorithm specific and differ from one NMF variant to the other. Termination criteria: If the approximation error drops below a pre-defined threshold, or if the shift between two iterations is very small, the algorithm might stop before all iterations are processed.

Multiplicative Update (MU) Algorithm

To give an example of the update steps for a specific NMF algorithm we provide the update steps for the MU algorithm in *Algorithm 2*. MU is one of the two original NMF algorithms presented in Lee and Seung (1999) and still one of the fastest NMF algorithms per iteration. The update steps are based on the mean squared error objective function and consist of multiplying the current factors by a measure of the quality of the current approximation. The divisions in *Algorithm 2* are to be performed *element-wise*. ε is used to avoid division by zero ($\varepsilon \approx 10^{-9}$).

Figure 1. Scheme of very coarse NMF approximation with very low rank k. Although k is significantly smaller than m and n, the typical structure of the original data matrix can be retained (note the three different groups of data objects in the left, middle, and right part of A)



Algorithm 1. General structure of NMF algorithms

1: given matrix $A \in \mathbb{R}^{m \times n}$ and $k \ll min\{m, n\}$:
2: **for** $rep = 1$ to $maxrepetition$ **do**
3: $W = \text{rand}(m, k)$;
4: $(H = \text{rand}(k, n);)$
5: **for** $i = 1$ to $maxiter$ **do**
6: perform algorithm specific NMF update steps
7: check termination criterion
8: **end for**
9: **end for**

3. SWARM INTELLIGENCE OPTIMIZATION

Optimization techniques inspired by swarm intelligence (SI) have become increasingly popular and benefit from their robustness and flexibility (Chiong, 2009). Swarm intelligence is characterized by a decentralized design paradigm that mimics the behavior of swarms of social insects, flocks of birds, or schools of fish. Optimization techniques inspired by swarm intelligence have shown to be able to successfully deal with increasingly complex problems (Blackwell, 2007). In this article we use five different optimization algorithms. Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995) is a classical swarm intelligence algorithm, while Fish School Search (FSS) (Bastos Filho et al., 2009) and Fireworks Algorithm (FWA) (Tan & Zhu, 2010) are two recently developed swarm intelligence methods. These three algorithms are compared to a Genetic Algorithm (GA) (Haupt & Haupt, 2005), a classical evolutionary algorithm, and Differential Evolution (DE) (Price, Storn, & Lampinen, 2005), which shares some features with swarm intelligence but can also be considered as an evolutionary algorithm. Since

Algorithm 2. Update steps of the multiplicative update algorithm

1: $H = H .* (W^\top A) ./ (W^\top W H + \varepsilon)$;
2: $W = W .* (A H^\top) ./ (W H H^\top + \varepsilon)$;

Algorithm 3. Pseudo code of the Fish School Search algorithm

1: Randomly initialize locations (x_i) of all fish, set all weights (w_i) to 1;
2: **repeat**
3: *Swimming 1:* Compute random individual movement for each fish;
4: *Feeding:* update weights for all fish based on new locations;
5: *Swimming 2:* Collective instinctive movement towards overall direction;
6: *Swimming 3:* Collective volitive movement dilation/contraction;
7: **until** termination (time, max. number of fitness evals., convergence, ...)

Algorithm 4. Pseudo code of the Fireworks Algorithm

1: Randomly initialize locations (x_i) of n fireworks;
2: **repeat**
3: Set off n fireworks respectively at the n locations
4: Calculate number $\hat{s}_i$ and location of sparks for each x_i
5: Generate $\hat{m}$ specific sparks, each for a randomly selected firework
6: Keep best location and select $n-1$ locations for next iteration
7: **until** termination (time, max. number of fitness evals., convergence, ...)

PSO, GA and DE are well known optimization techniques we will not summarize them here; instead the interested reader is referred to the references given.

Fish School Search is a recently developed swarm intelligence algorithm (Algorithm 3) that mimics the movements of schools of fish. The main operators are *feeding* (fish can gain/lose weight, depending on the region they swim in) and *swimming* (there are three different swimming movements).

The ***Fireworks Algorithm*** (Algorithm 4) is a novel swarm intelligence algorithm that is inspired by observing fireworks explosion. Two different types of explosion (search) processes are used in order to ensure diversity of resulting sparks, which are similar to particles in PSO or fish in FSS.

4. IMPROVING NMF WITH SWARM INTELLIGENCE OPTIMIZATION

Before describing our two optimization strategies for NMF based on swarm intelligence, we discuss some properties of the Frobenius norm (cf. Berry, Drmac, & Jessup, 1999). We use the Frobenius norm (1.1) as NMF objective function (i.e., to measure the error between A and WH) because it offers some properties that are beneficial for combining NMF and optimization algorithms. The following statements about the Frobenius norm are valid for any real matrix. However, in the following we assume that D refers to a distance matrix storing the distance (error of the approximation) between the original data and the approximation, $D = A - WH$. The Frobenius norm of a matrix $D \in \mathbb{R}^{m \times n}$ is defined as

$$|| D ||_F = \left(\sum_{i=1}^{min(m,n)} \sigma_i \right)^{1/2} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} | \mathbf{d}_{ij} |^2 \right)^{1/2}, \quad (1.2)$$

where σ_i are the singular values of D, and $\mathbf{d}_{ij}$ is the element in the i^{th} row and j^{th} column of D. The Frobenius norm can also be computed row wise or column wise. The *row wise* calculation is

$$|| D ||_F^{RW} = \left(\sum_{i=1}^{m} | \mathbf{d}_i^r |^2 \right)^{1/2}, \quad (1.3)$$

where $| \mathbf{d}_i^r |$ is the norm of the i^{th} row vector of D, i.e., $| \mathbf{d}_i^r | = (\sum_{j=1}^{n} | r_j^i |^2)^{1/2}$, and r_j^i is the j^{th} element in row i. The *column wise* calculation is

$$|| D ||_F^{CW} = \left(\sum_{j=1}^{n} | \mathbf{d}_j^c |^2 \right)^{1/2}, \quad (1.4)$$

with $| \mathbf{d}_j^c |$ being the norm of the j^{th} column vector of D, i.e., $| \mathbf{d}_j^c | = (\sum_{i=1}^{m} | c_i^j |^2)^{1/2}$, and c_i^j being the i^{th} element in column j. Obviously, a reduction of the Frobenius norm of any row or any column of D leads to a reduction of the total Frobenius norm $|| D ||_F$.

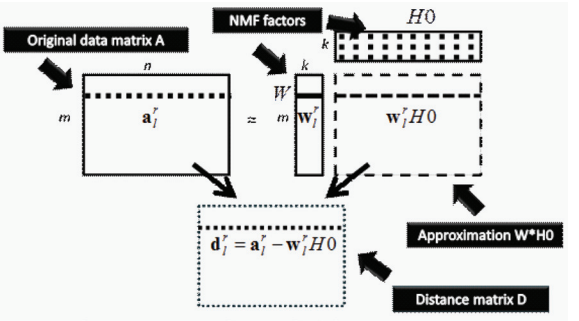
In the following we exploit these properties of the Frobenius norm for the proposed NMF optimization strategies. While strategy 1 aims at finding heuristically optimal starting points for the NMF factors, strategy 2 aims at iteratively improving the quality of NMF during the first iterations. All meta-heuristics mentioned in Section 3 can be used within both strategies. Before discussing the optimization strategies we illustrate the basic optimization procedure for a specific row (row l) of W in Figure 2. This procedure is similar for both optimization strategies.

Parameters: Global parameters used for all optimization algorithms are upper/lower bound of the search space and the initialization, the number of particles (chromosomes, fish, ...), and maximum number of fitness evaluations. Parameter settings are discussed in Sections 5. For all meta-heuristics, the problem dimension is equal to the rank k of the NMF, i.e., if, for example, k = 10, a row/column vector with 10 continuous entries is returned by the optimization algorithms.

4.1. Optimization Strategy 1 – Initialization

The goal of this optimization strategy is to find heuristically optimal starting points for the rows of W and the columns of H respectively, i.e., *prior* to the factorization process. *Algorithm 5* shows the pseudo code for the initialization procedure. In the beginning, $H0$ needs to be initialized randomly using a non-negative lower bound (preferably 0) for the initialization. In the first loop, W is initialized row wise, i.e., row $\mathbf{w}_i^r$ is optimized in order to minimize the

Figure 2. Illustration of the optimization process for row l of the NMF factor W. The l^{th}row of A (a_l^r) and all columns of H0 are the **input** *for the optimization algorithms. The* **output** *is a row-vector $\mathbf{w}_l^r$ (the l^{th}row of W) which minimizes the norm of $\mathbf{d}_l^r$, the l^{th}row of the distance matrix D. The norm of $\mathbf{d}_l^r$ is the fitness function for the optimization algorithms (minimization problem)*



Frobenius norm of the i^{th} row $\mathbf{d}_i^r$ of D, which is defined as $\mathbf{d}_i^r = \mathbf{a}_i^r - \mathbf{w}_i^r H0$. Since the optimization of any row of W is independent to the optimization of any other row of W, all $\mathbf{w}_i^r$ can be optimized concurrently. In the second loop, the columns of H are initialized using on the previously computed and already optimized rows of W, which need to be gathered beforehand (in line 7 of the algorithm). H is initialized column wise, i.e., column $\mathbf{h}_j^c$ is optimized in order to minimize the Frobenius norm of the j^{th} column $\mathbf{d}_j^c$ of D, which is defined as $\mathbf{d}_j^c = \mathbf{a}_j^c - W\mathbf{h}_j^c$. The optimization of the columns of H can be performed concurrently as well.

4.2. Optimization Strategy 2 – Iterative Optimization

The second optimization strategy aims at iteratively optimizing the NMF factors W and H *during the first iterations* of the NMF. Compared to the first strategy not all rows of W and all columns of H are optimized – instead the optimization is only performed on selected rows/columns. In order to improve the approximation as fast as possible we identify rows of D with highest norm (the approximation of this row is worse than for other rows of D) and optimize the corresponding rows of W. The same procedure is used to identify the columns of H that should be optimized. Our experiments showed that not all NMF algorithms are suited for this iterative optimization procedure. For many NMF algorithms there was no improvement with respect to the convergence or a reduction of the overall error after a fixed number of iterations. However, for the multiplicative update (MU) algorithm – which is one of the most widely used NMF algorithms – this strategy is able to improve the quality of the factorization. Hence, *Algorithm 6* shows the pseudo code for the iterative optimization of the NMF factors during the first iterations using the update steps of the MU algorithm described in Section 2.1. As shown in Section 6, this update strategy is able to significantly reduce the approximation error per iteration for the MU algorithm. Due to the relatively high computational cost of the meta-heuristics the optimization procedure is only applied in the first *m* iterations and only on *c* selected rows/columns of the NMF factors. Similar to strategy one the optimization of all rows of W are independent from each other (identical for columns of H), which allows for a parallel implementation of the proposed method. In the following we describe the variables and functions (for updating rows of W) of *Algorithm 6*. Updating columns of H is similar to updating the rows of W.

Algorithm 5. Pseudo code for the initialization procedure for NMF factors W and H. The two for-loops in lines 4 and 10 can be executed concurrently. SIO = Swarm Intelligence Optimization

1: Given matrix $A \in \mathbb{R}^{m \times n}$ and $k \ll min\{m, n\}$;
2: $H0 = \text{rand}(k, n)$;
3: **% Compute in parallel**
4: **for** $i = 1$ to m **do**
5: Use SIO to find $\mathbf{w}_i^r$ that minimizes $||\mathbf{a}_i^r - \mathbf{w}_i^r H0||_F$, (min $||.||_F$ of row i of D);
6: **end for**;
7: **% Gather**
8: $W = [\mathbf{w}_1^r; \ldots; \mathbf{w}_m^r]$;
9: **% Compute in parallel**
10: **for** $j = 1$ to n **do**
11: Use SIO to find $\mathbf{h}_j^c$ that minimizes $||\mathbf{a}_j^c - W\mathbf{h}_j^c||_F$, (min $||.||_F$ of col j of D);
12: **end for**
13: **% Gather**
14: $H = [\mathbf{h}_1^c, \ldots, \mathbf{h}_n^c]$;

- m : the number of iterations in which the optimization using meta-heuristics is applied
- c : the number of rows and/or columns that are optimized in the current iteration.
- Δc : the value of c is decreased by Δc in each iteration. $\Delta c = round(c_{initial} / m)$
- $[Val, IX\_W] = sort(norm(\mathbf{d}_i^r), 'descend')$: returns the values Val and the corresponding indices (IX_W) of the norm of all row vectors $\mathbf{d}_i^r$ of D in descending order.
- $IX\_W = IX\_W(1:c)$: returns only the first c elements of the vector IX_W .
- minimize $|| \mathbf{a}_i^r - \mathbf{w}_i^r H ||_F$: see Figure 2 and optimization strategy 1

4.3. Using NMF for Classification Problems

As already mentioned before, we also investigate the performance of NMF when applied for classification tasks. In this article, we use two different classification methods for evaluating the classification accuracy of NMF based on the optimization strategies discussed in Sections 4.1 and 4.2. Both classification methods have shown to work well for different application areas (Janecek, 2010).

Static Classification

In the first approach we analyze the classification accuracy achieved with the basis vectors (i.e., features in W). In this setting the NMF needs to be computed on the complete dataset (training and test data) which makes this technique only applicable on test data that is already available before the approximation/classification. However, the advantage of this approach is that any freely chosen classification method can be applied on the basis features.

If the original data matrix $A \in \mathbb{R}^{m \times n}$ is an instance $\times$ feature matrix, then the NMF factor W is a $m \times k$ matrix, where every instance is described by k basis *features*, i.e., every column

of W corresponds to a basis feature. Note that this setup is different to the one discussed at the beginning of Section 2! By applying a classification algorithm on the rows of W instead on the rows of A we can significantly reduce the dimension of the classification problem and thus decrease the computational cost for both, building the classification model and testing new data.

Dynamic Classification

The second approach can be applied dynamically to new data. Here the factorization of the data (NMF) and the classification process are separated from each other (i.e., the NMF is performed on labeled training data – the unlabeled test data does not have to be available at the time of performing the NMF). This approach is called *NMF-LSI* and is based on an adaptation of latent semantic indexing which is a variant of the well-known vector space model.

A *vector space model* (VSM) (Raghavan & Wong, 1999) is a widely used algebraic model for representing objects as vectors in a potentially very high dimensional metric vector space. The distance of a query vector **q** to all objects in a given *feature* × *instance* matrix A are usually measured in terms of the cosines of the angles between **q** and the columns of A such that $cos\varphi_i = \frac{e_i^\top A^\top q}{|| Ae_i ||_2 || q ||_2}$.

Latent semantic indexing (LSI) (Berry, Drmac, & Jessup, 1999) is a variant of the basic VSM that replaces the original matrix A with a low-rank approximation A_k of A. In the standard version of LSI the SVD (Section 2) is used to construct A_k, and $cos\varphi_i$ can be approximated as $cos\varphi_i \approx \frac{e_i^\top V_k \Sigma_k U_k^\top q}{|| U_k \Sigma_k V_k^\top e_i ||_2 || q ||_2}$. LSI has computational advantages resulting in lower storage and computational cost, and often gives a cleaner and more efficient representation of the (latent) relationship between data elements.

NMF-LSI: The approximation within LSI can be replaced with other approximations. Instead of using the truncated SVD ($A_k := U_k \Sigma_k V_k^\top$), we approximate A with $A_k := W_k H_k$ (the NMF). When using NMF, the value of k must be fixed prior to the approximation. The cosine of the angle between **q** and the i^{th} column of A can then be approximated as $cos\varphi_i \approx \frac{e_i^\top H_k^\top W_k^\top q}{|| W_k H_k e_i ||_2 || q ||_2}$. In order to save computational cost, the left term in the numerator ($e_i^\top H_k^\top$) and the left part of the denominator ($|| W_k H_k e_i ||_2$) can be computed a priori. In all three methods (VSM and both LSI variants) a query instance **q** is assigned to the same class as the majority of its k-closest (in terms of cosine similarity) instances in A.

5. SETUP

Software

All software is written in Matlab. We used only publicly available NMF implementations: Multiplicative Update (MU, Matlab's Statistics Toolbox since v6.2, *nnmf()*). ALS using Projected Gradient (ALSPG) (Lin, 2007), and BayesNMF and FastNMF (Schmidt & Laurberg, 2008). Matlab code for NNDSVD (Section 1.1) is also publicly available (cf. Boutsidis & Gallopoulos, 2008). Codes for PSO and DE were adapted from Pedersen (2010), and code for GA from the appendix of Haupt and Haupt (2005). For FWA we used the same implementation as in the introductory paper Tan and Zhu (2010), and FSS was self-implemented following the algorithm provided in Bastos Filho et al. (2009).

Hardware

All experiments were performed on a SUN FIRE X4600 M2 with eight AMD Opteron quad-core processors (32 cores overall) with 3.2 GHz, 2MB L3 cache, and 32GB of main memory (DDR-II 666).

Parallel Implementation

We implemented parallel variants of the optimization algorithms exploiting Matlab's paral-

Algorithm 6. Pseudo code for the iterative optimization for the Multiplicative Update algorithm. SIO = Swarm Intelligence Optimization. The methods used in this algorithm are explained.

```
for iter = 1 to maxiter do
  % perform MU specific update steps
  W = W · (AH^⊤)./(WHH^⊤ + ε);
  H = H · (W^⊤A)./(W^⊤WH + ε);
  if (iter < m) then
    % Update rows of W ++++++++++++++++++++++++++++++++++++++++++
      d_i^r is the i^th row vector of D = A − WH;
      [Val, IX_W]= sort(norm(d_i^r),'descend');
      IX_W = IX_W(1 : c);
      % Compute in parallel
      ∀ i ∈ IX_W:
        Use SIO to find w_i^r that minimizes ||a_i^r − w_i^r H0||_F;
      % Gather
      W = [w_1^r; . . . ; w_m^r];
    % Update columns of H ++++++++++++++++++++++++++++++++++++++
      d_j^c is the j^th column vector of D = A − WH;
      [Val, IX_H] = sort(norm(d_j^c),'descend');
      IX_H = IX_H(1 : c);
      % Compute in parallel
      ∀ j ∈ IX_H:
        Use SIO to find h_j^c that minimizes ||a_j^c − Wh_j^c||_F;
      % Gather
      H = [h_1^c, . . . , h_n^c];
    c = c − Δc;
  end if
end for
```


lel computing potential. Matlab's Distributed Computing Server (which requires a separate license) allows for parallelizing the optimization process over a large number (currently up to 64) of workers (threads). These workers can be nodes in multi-core computers, GPUs, or a node in a cluster of simple desktop PCs. Matlab's Parallel Computing Toolbox (which is included in the basic version of Matlab) allows running up to eight workers concurrently, but is limited to local workers, i.e., nodes on a multi-core machine or local GPUs, but no cluster support.

Parameter Setup

The dimension of the optimization problem is always identical to the rank k of the NMF (cf.

Section 4). The upper/lower bound of the search space was set to the interval $[0,(4 * max(A))]$ and upper/lower bound of the initialization to $[0,max(A)]$. In order to achieve fair results which are not biased due to excessive parameter tuning we used the same parameter settings for all data sets. These parameter settings were found by running a self-written benchmark program that tested several parameter combinations on randomly generated data. For some optimization strategies (PSO, FSS and FWA) the recommended parameter settings from the literature worked fine. However, for GA and DE the parameter settings that were used in most studies in the literature did not perform very well. For GA we found that a very aggressive (high) mutation rate highly improved the results. For DE we observed a similar behavior and found that the maximum crossover probability (1) achieved the best results. For all experiments in this paper, the following parameter settings were used:

- GA: mutation rate of 0.5; selection rate of 0.65
- PSO: (G_{best} topology) following Bratton and Kennedy (2007) $\omega = 0.8$, and $c_1 = c_2 = 2.05$
- DE: crossover probability (pc) set to upper limit 1
- FSS: $step_{ind\_initial} = 1$, $step_{ind\_final} = 0.001$, $W_{scale} = 10$
- FWA: number of sons (*sonnum*) set to 10

Data Sets

We used three different data sets to evaluate our methods. *DS-RAND* is a randomly created, fully dense 100×100 matrix which is used in order to provide unbiased results. To evaluate the proposed methods in a classification context we further used two data sets from the area of email classification (spam/phishing detection). Data set *DS-SPAM1* consists of 3000 e-mail messages described by 133 features, divided into three groups: spam, phishing and legitimate email. An exact description of this data set can be found in Janecek and Gansterer (2010). Data set *DS-SPAM2* is the *spambase* data set taken from Kjellerstrand (2011) which consists of 1813 spam and 2788 non-spam messages. DS-SPAM1 represents a ternary classification problem; DS-SPAM2 represents a typical binary classification problem.

6. EXPERIMENTAL EVALUATION

The evaluation is split up into two parts. First we evaluate the two optimization strategies proposed in Section 4.1 and Section 4.2, then we evaluate the quality of NMF in a classification context.

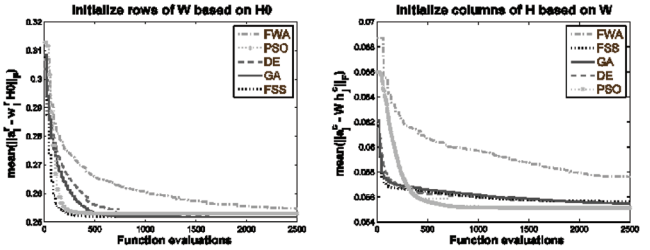
6.1. Evaluation of Optimization Strategy 1

Initialization

Before evaluating the improvement of the NMF approximation quality as such, we first measure the initial error after initializing W and H *(before* running the NMF algorithm). Figure 3 and Figure 4 show the average approximation error (i.e., Frobenius norm / fitness) per row (left) and per column (right) for data set DS-RAND.

The figures on the left side show the *average* (mean) approximation error per *row* after initializing the rows of W (first loop in *Algorithm 5*). The figures on the right side show the *average* (mean) approximation error per *column* after initializing the columns of H (second loop in *Algorithm 5*). The legends are ordered according to the average approximation error achieved after the maximum number of function evaluations for each figure (top = worst, bottom = best). When the NMF rank *k* is small (Figure 3, *k*=5) all optimization algorithms except FWA achieve similar results. Except FWA, all optimization algorithms quickly converge to a good result. With increasing complexity (i.e., increasing rank *k*) FWA clearly improves its results, as shown in Figure 4. The gap between the optimization algorithms is much bigger for

Figure 3. Left hand-side: average approximation error per row (after initializing rows of W). Right hand-side: average approximation error per column (after initializing of H). NMF rank k = 5. Legends are ordered according to approximation error (top = worst, bottom = best)



larger rank k. Note that GA needs more than 2000 evaluations to achieve a low approximation error for initializing the rows of W. When initializing the columns of H, PSO and GA suffer from their high approximation error during the first iterations, which is caused by the relatively sparse factor matrix W for PSO and GA. Although PSO is able to reduce the approximation error significantly during the first 500 iterations, FSS and GA achieve slightly better final results. Generally, FSS achieves the best approximation accuracy after the initialization procedure for large k. However, as shown later the initial approximation error is *not necessarily an indicator* for the approximation quality of NMF or the resulting classification accuracy.

Runtime Performance

When parallelizing a sequential algorithm over p processors the speed-up indicates how much the parallel algorithm can perform specific tasks faster than the sequential algorithm. Speed-up is defined as $S_p = ET_{sequential} / ET_{parallel}$, where ET is the execution time. A linear speed-up is achieved when S_p is equal to p. Efficiency is another metric that estimates how well-utilized the processors are in solving the problem, compared to the cost of communication and synchronization. Efficiency is defined as $E_p = S_p / p$. For algorithms with linear speed-up the efficiency is 1, for algorithms with lower speed-up ratio it is between 0 and 1.

Figure 5 shows the runtime behavior for optimization strategy 1 with increasing number of Matlab workers. Runtimes are shown for the FSS optimization algorithm – however, all optimization algorithms have rather similar runtimes. Due to license limitations we only had Matlab's Parallel Computing Toolbox available which is limited to 8 workers (cf. Section 5). We measured runtimes and speed-up for up to 8 workers (average efficiency of about 0.95) and estimated the behavior of speed-up and runtime for a larger number of workers (based on this efficiency). Upgrading to Matlab's Distributed Computing Server is possible without any code-changes and thus only a license issue. When using eight workers, the NNDSVD initialization (the best NMF initialization strategy from the literature, Section 1.1) is a bit faster, but estimation shows that the proposed initialization strategy is faster when 12 or more workers are used. NNDSVD is already optimized and cannot be parallelized further in its current implementation.

Approximation Quality

For evaluating the approximation results achieved by NMF using the factors W and H initialized by the optimization algorithms, we compare our results to random initialization as well as to NNDSVD. Figure 6 shows the approximation error on the *y*-axis (log scale) after a given number of NMF iterations for four NMF algorithms using different initialization methods (for DS-RAND). The initialization methods in

Figure 4 . Similar information as for Figure 3, but for NMF rank k = 30

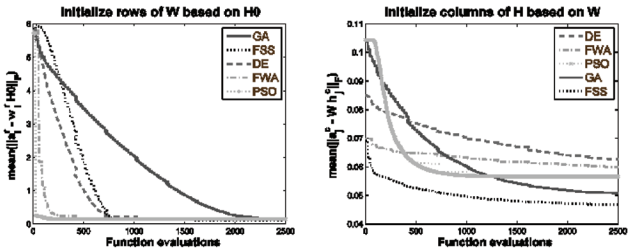
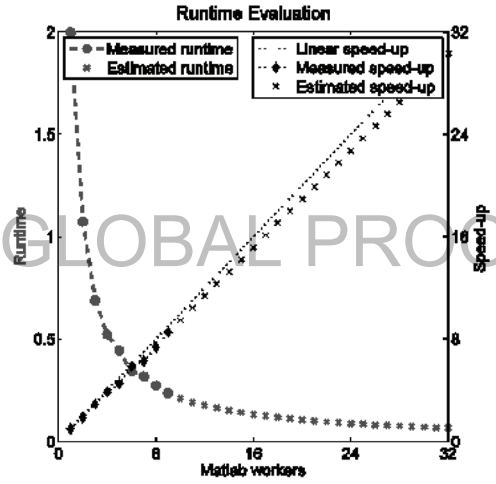
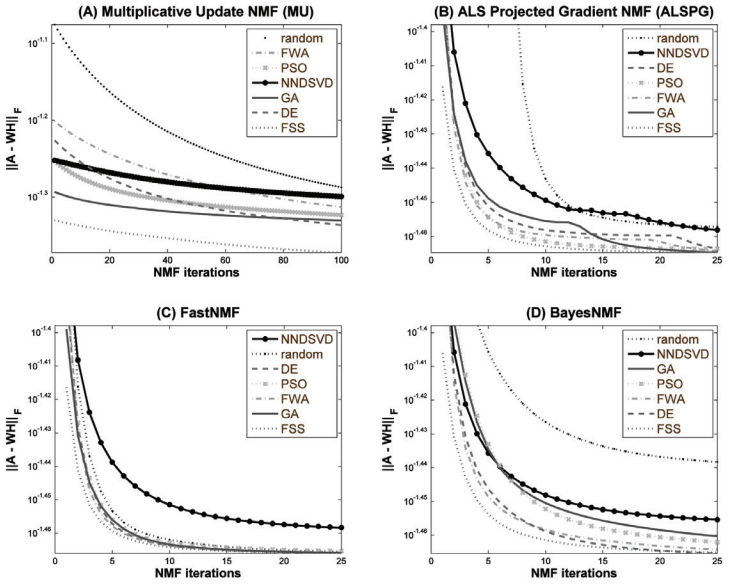


Figure 5. Runtime and speed-up measurement/estimation for DS-RAND using 1500 function evaluations per row/column for k= 5. As a reference, NNDSVD needs about 0.16 seconds for k=5. This indicates that if the number of workers is larger than 12, the proposed optimization strategy is faster than NNDSVD



the legend are ordered (top = worst, bottom = best). Since the MU algorithm (A) has low cost per iteration but converges slowly, the first 100 iterations are shown (for all other algorithms the first 25 iterations are shown). For MU, all initialization variants achieve a smaller approximation error than random initialization. NNDSVD shows slightly better results than PSO and FWA, but GA, DE and especially FSS are able to achieve a smaller error per iteration than NNDSVD. For ALSPG (B), the new initialization strategy achieves better results than random initialization and also achieves a better approximation error than NNDSVD. This improvement is independent of the actual optimization algorithm. The same behavior can be seen for FastNMF (C) and BayesNMF (D). It has to be mentioned that FastNMF and BayesNMF were developed after the NNDSVD initialization. Surprisingly, when using FastNMF, NNDSVD achieves a lower approximation than random initialization. When comparing the different meta-heuristics, FSS achieves the best results amongst all optimization algorithms and achieves the closest approximation after 100 (MU) and 25 (ALSPG, FastNMF,

Figure 6. Approximation error archived by different NMF algorithms using different initialization variants (k=30, after 1500 fitness evaluations)



BayesNMF) iterations, respectively. DE and GA follow with a small gap since they are not as stable as FSS (i.e., they achieve good results for some, but not for all NMF algorithms.

6.2. Evaluation of Optimization Strategy 2

Figure 7 shows the convergence curves for the NMF approximation using optimization strategy 2 for different values of rank k (data set DS-RAND). Due to the relatively high computational cost of the meta-heuristics we applied our optimization procedure here only on the rows of W, while the columns in H remained unchanged. Experiments showed that with this setting the loss in accuracy compared to optimizing both, W *and* H, is relatively small while the runtime can be increased significantly. m was set to 2 which indicates that the optimization is only applied in the first two iterations, and c was set to 20. As can be seen, the approximation error per iteration can be reduced when using optimization strategy 2. For small rank k (left side of Figure 7) the improvement is significant but decreases with increasing values of k (see right side of Figure 7). For larger k (larger than 10) the improvement over the basic MU is only marginal.

Runtime Performance

Figure 8 shows the reduction in runtime for different rank k when the same accuracy as for basic MU should be achieved. Runtimes are shown for a parallel implementation using 32 Matlab workers. Basic MU sets the baseline (1 = 100%), the runtimes of the optimization strategy 2 (using different optimization algorithms) are given as $t_{opt-XX} \,/\, t_{Basic\,MU}$. For example, for small rank k the runtime can often be reduced by more than 50%. With increasing

Figure 7. Accuracy per Iteration when updating only the row of W, m=2, c=20. Left: k=2, right: k=5

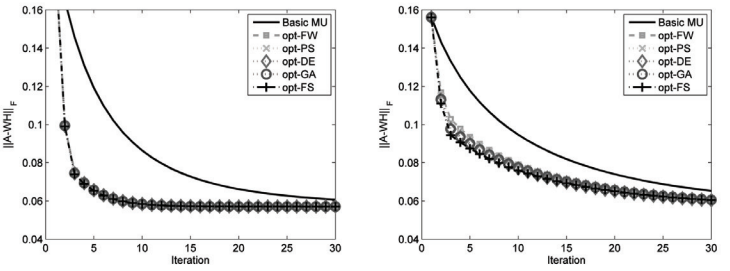
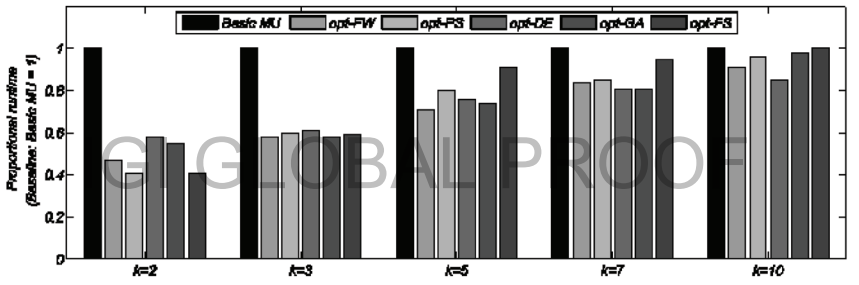


Figure 8. Proportional runtimes for achieving the same accuracy as basic MU after 30 iterations for different values of k when updating only the rows of W. (m=2, c=20)



rank *k* the runtime savings get smaller and are only marginal for *k*=10. For rank *k* larger than 12 the basic MU algorithm is faster than optimization strategy 2.

6.3. Evaluation of the Classification Accuracy

Since optimization strategy 1 (initialization, Sections 4.1 and 6.1) achieves a faster, closer, and more stable approximation as optimization strategy 2 (iterative update, Sections 4.2 and 6.2) we evaluate the classification accuracy for this strategy. In the following, we measure the quality of optimization strategy 1 as pre-processing step for the two classification approaches mentioned in Section 4.3. Within the *static classification approach* any machine learning algorithm can be used for classification, but the approximation used for reducing the dimensionality of the data set (SVD, PCA, NMF) needs to be applied on the complete data set. Contrary, the *dynamic classification approach* can be applied on the training data, the test data does not need to be available at the time of computing the approximation. However, this approach cannot be applied to all classification methods.

Static Classification

We used three classification algorithms from the freely available WEKA toolkit (Witten and Frank 2005) to compare the classification accuracies achieved with the NMF factor W based on different NMF initializations: A sup-

Table 1. Classification results (static classification) for DS-SPAM1

		J4.8			*k*NN(1)			SVM (SMO)		
		all features: 0,973			all features: 0,977			all features: 0,976		
NMF Alg	*Init*	*k = 30*	*k = 15*	*k = 5*	*k = 30*	*k = 15*	*k = 5*	*k = 30*	*k = 15*	*k = 5*
ALSPG	DE	0,968	0,972	0,965	0,974	0,972	0,968	**0,973**	0,956	0,940
ALSPG	FSS	0,961	0,972	0,967	0,971	0,972	**0,969**	**0,973**	0,954	0,939
ALSPG	FWA	**0,973**	0,969	**0,970**	0,972	**0,973**	0,968	0,964	0,954	0,938
ALSPG	GA	0,970	0,968	0,969	0,973	0,970	0,968	**0,973**	**0,957**	**0,947**
ALSPG	PSO	0,971	0,972	0,969	**0,977**	0,971	0,968	0,972	0,954	0,937
ALSPG	NNDSVD	0,963	**0,976**	0,964	0,969	0,972	0,968	0,966	0,952	0,938
ALSPG	RAND	*0,943*	*0,938*	*0,935*	*0,952*	*0,940*	*0,938*	*0,948*	*0,942*	*0,913*
BAYES	DE	**0,971**	0,970	0,970	0,974	0,973	0,968	0,971	0,954	0,946
BAYES	FSS	0,966	**0,973**	**0,971**	**0,976**	0,971	0,969	**0,975**	0,953	**0,947**
BAYES	FWA	0,970	0,970	0,968	0,972	**0,974**	0,968	0,957	0,954	0,941
BAYES	GA	0,966	0,971	0,968	0,974	0,973	0,969	0,972	0,955	**0,947**
BAYES	PSO	0,968	0,967	0,969	0,970	0,971	**0,970**	0,966	**0,957**	0,937
BAYES	NNDSVD	0,968	0,972	0,968	0,970	0,973	0,969	0,966	0,952	0,947
BAYES	RAND	*0,952*	*0,941*	*0,953*	*0,961*	*0,951*	*0,947*	*0,958*	*0,937*	*0,926*
FAST	DE	0,966	0,969	0,969	**0,977**	0,973	0,968	0,970	0,955	0,946
FAST	FSS	0,967	0,971	**0,970**	0,976	0,971	0,969	**0,975**	0,953	**0,947**
FAST	FWA	**0,968**	0,970	0,969	0,971	**0,974**	0,968	0,957	0,954	0,941
FAST	GA	0,966	0,965	0,968	0,973	0,971	0,969	0,973	0,955	**0,947**
FAST	PSO	**0,968**	0,970	**0,970**	0,974	0,971	**0,970**	0,973	**0,956**	0,937
FAST	NNDSVD	0,966	**0,973**	**0,970**	0,970	0,973	0,968	0,966	0,952	0,939
FAST	RAND	*0,954*	*0,949*	*0,937*	*0,958*	*0,951*	*0,941*	*0,957*	*0,935*	*0,917*
MU	DE	0,955	0,952	0,965	0,966	0,959	0,968	0,962	0,953	0,940
MU	FSS	**0,965**	0,960	0,967	0,967	0,964	**0,969**	**0,966**	0,952	0,939
MU	FWA	0,949	0,956	**0,970**	0,964	0,966	0,968	0,959	**0,955**	0,938
MU	GA	0,954	0,961	0,969	0,966	0,966	0,968	0,961	0,944	**0,947**
MU	PSO	0,958	0,939	0,969	0,949	0,946	0,968	0,953	0,940	0,937
MU	NNDSVD	0,964	**0,967**	0,964	**0,972**	**0,973**	0,968	0,963	0,954	0,938
MU	RAND	*0,941*	*0,937*	*0,947*	*0,948*	*0,941*	*0,951*	*0,951*	*0,930*	*0,927*

port vector machine (SVM) based on the sequential minimal optimization (SOM) algorithm using a polynomial kernel with an exponent of 1; a k-nearest neighbor (*k*NN) classifier; and a J4.8 decision tree based on the C4.5 decision tree algorithm. Results were achieved using a 10-fold cross-validation, i.e., by randomly partitioning the data sets into 10 subsamples and then iteratively using one 9 subsamples as training data and 1 for testing.

Table 1 shows the overall classification results achieved with data set DS-SPAM1 using three different values of rank *k* and the three different classification methods mentioned

above. The overall classification accuracy is computed as the number of correct classified email messages divided by the total number of messages. The most-left column indicates the NMF algorithm and the second column the initialization strategy used for computing the NMF (RAND = random initialization). Note that the number of features is reduced to 30, 15 and 5, respectively, compared to 133. This reduction in the number of features significantly speeds up both, the process of building the classification model and the classification process itself. The best result for each NMF algorithm and each rank k is highlighted in bold letters. The proposed initialization strategies achieve better classification results as the state-of-the-art initialization method NNDSVD and significantly better results as random NMF initialization. Among the applied optimization algorithms there is not much difference, though FSS achieves a larger number of best results then the other algorithms. Results for J4.8 and kNN are very stable even for k=5 and are almost identical to the classification result achieved with all features. For SVM, the classification result tends to decrease with decreasing rank k. This behavior has been observed in another study (Janecek, Gansterer, Demel, & Ecker, 2008) where SVM has been applied on data sets from other dimensionality reduction methods (PCA). However, compared to NNDSVD and random initialization the proposed initialization methods achieve better results for all ranks of k. Comparing the different NMF algorithms it can be seen the MU achieves lower classification accuracy compared to ALSPG, FastNMF and BayesNMF.

Table 2 shows the static classification results achieved with data set DS-SPAM2. Results are shown for the FastNMF, which achieved the most stable results of all NMF algorithms for this data set. Again, the proposed initialization strategy again achieves better results as NNDSVD and random initialization. Compared to DS-SPAM1, the results for this data set tend to decrease with decreasing rank k. This indicates that it is important to find a good trade-off between classification accuracy and computational cost.

Dynamic Classification

Table 3 shows the classification results achieved with the dynamic classification approach described in Section 4.3 for DS-SPAM1. In general, the classification accuracies achieved for data set DS-SPAM2 using the dynamic classification approach are rather similar to the results for DS-SPAM1 shown in Table 3. The baseline to which the NMF-LSI variants are compared are given by a standard LSI classification using SVD as approximation algorithm (Section 4.3). A basic vector space model achieves a classification accuracy of 0.911, while LSI achieves 0.911, 0.914 and 0.887, respectively, for rank k set to 30, 15 and 5. Similar to Table 2 (DS-SPAM2) the results are sensible with respect to the value of rank k. For very small values of k (5) the classification results generally tend to decrease. Overall, the initialization strategy based on meta-heuristics achieve much better classification accuracy as NNDSVD and random initialization, and also outperform basic LSI in many cases. The best results are again highlighted in bold letters. Especially GA and FWA achieve good classification results.

7. CONCLUSION

In this article we presented two new optimization strategies for improving the NMF using optimization algorithms based on swarm intelligence. While strategy one uses swarm intelligence algorithms to initialize the factors W and H *prior* to the factorization process of NMF, the second strategy aims at iteratively improving the approximation quality of NMF *during* the first iterations of the factorization. Overall, five different optimization algorithms were used for improving NMF: Particle Swarm Optimization (PSO), Genetic Algorithms (GA), Fish School Search (FSS), Differential Evolution (DE), and Fireworks Algorithm (FWA).

Table 2. Classification results (static classification) for DS-SPAM2 (FastNMF)

		J4.8			*k*NN(1)			SVM (SMO)		
		all features: 0,921			all features: 0,907			all features: 0,904		
NMF Alg	*Init*	*k = 30*	*k = 15*	*k = 5*	*k = 30*	*k = 15*	*k = 5*	*k = 30*	*k = 15*	*k = 5*
FAST	DE	0,918	0,893	0,863	**0,902**	0,880	0,821	**0,905**	0,865	0,798
FAST	FSS	0,920	**0,920**	0,773	0,895	0,889	0,826	0,894	0,880	0,773
FAST	FWA	0,916	0,916	0,864	0,887	**0,898**	0,797	0,893	0,885	0,757
FAST	GA	0,918	0,914	0,865	0,889	0,896	0,827	0,896	**0,891**	0,778
FAST	PSO	**0,921**	0,911	**0,878**	0,895	0,892	**0,850**	0,896	0,881	**0,827**
FAST	NNDSVD	0,919	0,911	0,811	0,895	0,894	0,816	0,894	0,882	0,766
FAST	RAND	*0,907*	*0,908*	*0,813*	*0,885*	*0,886*	*0,803*	*0,887*	*0,864*	*0,752*

Table 3. Dynamic classification using DS-SPAM1. Basic vector space model (all features): 0,911

Baseline	*LSI*	*0,911*	*0,914*	*0,887*			*LSI*	*0,911*	*0,914*	*0,887*
NMF Alg	Init	k = 30	k = 15	k = 05		NMF Alg	Init	k = 30	k = 15	k = 05
ALSPG	DE	0,911	0,898	0,889		FAST	DE	0,912	0,895	0,888
ALSPG	FSS	**0,943**	0,899	0,877		FAST	FSS	0,926	0,897	0,879
ALSPG	FWA	0,930	**0,914**	0,883		FAST	FWA	0,913	0,912	**0,891**
ALSPG	GA	0,927	0,901	**0,896**		FAST	GA	**0,927**	**0,914**	0,875
ALSPG	PSO	0,918	0,889	0,885		FAST	PSO	0,923	**0,914**	0,847
ALSPG	NNDSVD	0,914	0,911	0,840		FAST	NNDSVD	0,911	0,913	0,846
ALSPG	RAND	0,901	0,886	0,874		FAST	RAND	0,898	0,899	0,838
BAYES	DE	0,911	0,906	0,888		MU	DE	0,893	0,897	0,834
BAYES	FSS	0,926	0,897	0,879		MU	FSS	0,892	0,882	0,807
BAYES	FWA	0,914	0,911	**0,891**		MU	FWA	**0,913**	0,882	**0,843**
BAYES	GA	**0,930**	**0,916**	0,875		MU	GA	0,899	0,899	0,795
BAYES	PSO	0,922	0,915	0,848		MU	PSO	0,922	0,900	0,812
BAYES	NNDSVD	0,904	0,913	0,846		MU	NNDSVD	0,906	**0,908**	0,795
BAYES	RAND	0,898	0,896	0,854		MU	RAND	0,876	0,889	0,817

Both optimization strategies allow for efficiently computing the optimization of single rows of W and/or single columns of H in parallel. The achieved results are evaluated in terms of accuracy per runtime and per iteration, final accuracy after a given number of NMF iterations, and in terms of the classification accuracy achieved with the reduced NMF factors when being applied for machine learning applications. Especially the initialization strategy (optimization strategy 1) is able to significantly improve the approximation results of NMF compared to random initialization and state-of-the-art methods. Among the different

optimization algorithms, the recently developed fish school search algorithm achieves slightly better results than the other heuristics. The iterative strategy (optimization strategy 2) can improve one of the basic NMF algorithms (the multiplicative update strategy) for very small rank k and can thus be used if a rough and very fast approximation method is needed. Moreover, the NMF subsets achieved with optimization strategy 1 have shown to clearly improve the classification accuracy of NMF compared to state-of-the-art initialization strategies, and also achieve better results as feature subsets computed with other low-approximation techniques.

Future Work

Our investigations provide several important and interesting directions for future work. First of all, we will set the focus on developing optimization strategies that update the factor matrices W and H concurrently instead of applying an alternating update fashion where one factor is fixed and the other one is optimized. Moreover, we will apply the optimization strategies on NMF problems were sparseness constraints are enforced, i.e., the optimization strategies are enforced to compute solutions with a certain percentage of zero values. We also plan to use different NMF optimization functions (not based on the Frobenius norm) for our optimization methods and several recently developed NMF algorithms (HALS, multilayer NMF, etc.).

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (NSFC), Grant No. 60875080 and No. 61170057. Andreas wants to thank the *Erasmus Mundus External Coop. Window*, Lot 14 (2009-1650/001-001-ECW).

REFERENCES

Bastos Filho, C. J. A., de Lima Neto, F. B., Lins, A. J. C. C., Nascimento, A. I. S., & Lima, M. P. (2009). Fish school search. R. Chiong (Ed.), *Nature-inspired algorithms for optimisation* (Vol. 193, pp. 261-277). Berlin, Germany: Springer-Verlag.

Berry, M. W. (1992). Large scale singular value computations. *The International Journal of Supercomputer Applications*, *6*, 13–49.

Berry, M. W., Browne, M., Langville, A., Pauca, V., & Plemmons, R. (2007). Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, *52*(1), 155–173. doi:10.1016/j.csda.2006.11.006

Berry, M. W., Drmac, Z., & Jessup, E. R. (1999). Matrices, vector spaces, and information retrieval. *SIAM Review*, *41*(2), 335–362. doi:10.1137/S0036144598347035

Blackwell, T. (2007). Particle swarm optimization in dynamic environments. *Evolutionary Computation in Dynamic and Uncertain Environments*, *1*, 29–49. doi:10.1007/978-3-540-49774-5_2

Boutsidis, C., & Gallopoulos, E. (2008). SVD based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition*, *41*(4), 1350–1362. doi:10.1016/j.patcog.2007.09.010

Bratton, D., & Kennedy, J. (2007). Defining a standard for particle swarm optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium* (pp. 120-127).

Chiong, R. (2009). *Nature-inspired algorithms for optimisation*. New York, NY: Springer.

Haupt, R. L., & Haupt, S. E. (2005). *Practical genetic algorithms* (2nd ed.). New York, NY: John Wiley & Sons.

Janecek, A. (2010). *Efficient feature reduction and classification methods: Applications in drug discovery and email categorization*. Vienna, Austria: Department of Computer Science, University of Vienna.

Janecek, A., & Gansterer, W. N. (2010). Utilizing nonnegative matrix factorization for e-mail classification problems . In Berry, M. W., & Kogan, J. (Eds.), *Survey of text mining III: Application and theory* (pp. 57–80). New York, NY: John Wiley & Sons.

Janecek, A., Gansterer, W. N., Demel, M., & Ecker, G. (2008). On the relationship between feature selection and classi□cation accuracy. *Journal of Machine Learning Research*, *4*, 90–105.

Janecek, A., S. Schulze-Grotthoff, et al. (2011). libNMF - A library for nonnegative matrix factorizatrion. *Computing and Informatic*s, *22*.

Janecek, A., & Tan, Y. (2011a). Iterative improvement of the multiplicative update NMF algorithm using nature-inspired optimization. In *Proceedings of the 7th International Conference on Natural Computation* (pp. 1668-1672).

Janecek, A., & Tan, Y. (2011b). Using population based algorithms for initializing nonnegative matrix factorization. In Y. Tan, Y. Shi, Y. Chai, & G. Wang (Eds.), *Proceedings of the Second International Conference on Advances in Swarm Intelligence* (LNCS 6729, pp. 307-316).

Jolliffe, I. T. (2002). *Principal component analysis*. New York, NY: Springer.

Kennedy, J., & Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks* (Vol. 4, pp. 1942-1948).

Kennedy, J., Eberhart, R. C., & Shi, Y. (2001). *Swarm intelligence*. San Francisco, CA: Morgan Kaufmann.

Kim, H., & Park, H. (2008). Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method. *SIAM Journal on Matrix Analysis and Applications*, *30*, 713–730. doi:10.1137/07069239X

Kjellerstrand, H. (2011). *hakanks hemsida.* Retrieved from http://www.hakank.org/weka/

Langville, A. N., Meyer, C. D., & Albright, R. (2006). Initializations for the nonnegative matrix factorization. In *Proceedings of the 12th ACM International Conference on Knowledge Discovery and Data Mining.*

Lee, D. D., & Seung, H. S. (1999). Learning parts of objects by non-negative matrix factorization. *Nature*, *401*(6755), 788–791. doi:10.1038/44565

Lee, D. D., & Seung, H. S. (2001). Algorithms for non-negative matrix factorization. *Advances in Neural Information Processing Systems*, *13*, 556–562.

Lin, C.-J. (2007). Projected gradient methods for nonnegative matrix factorization. *Neural Computation*, *19*(10), 2756–2779. doi:10.1162/neco.2007.19.10.2756

Paatero, P., & Tapper, U. (1994). Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, *5*(2), 111–126. doi:10.1002/env.3170050203

Pedersen, M. E. H. (2010). *SwarmOps.* Retrieved from http://www.hvass-labs.org/projects/swarmops/cs/files/SwarmOpsCS1_0.pdf

Price, K. V., Storn, R. M., & Lampinen, J. A. (2005). *Differential evolution a practical approach to global optimization*. New York, NY: Springer.

Raghavan, V. V., & Wong, S. K. M. (1999). A critical analysis of vector space model for information retrieval. *Journal of the American Society for Information Science American Society for Information Science*, *37*(5), 279–287.

Schmidt, M. N., & Laurberg, H. (2008). Non-negative matrix factorization with Gaussian process priors. *Computational Intelligence and Neuroscience*, (1): 1–10. doi:10.1155/2008/361705

Snásel, V., Platos, J., & Kromer, P. (2008). Developing genetic algorithms for Boolean matrix factorization. In *Proceedings of the DATESO International Workshop on Current Trends on Databases.*

Stadlthanner, K., Lutter, D., Theis, F. J., Lang, E. W., Tome, A. M., Georgieva, P., & Puntonet, C. G. (2007). Sparse nonnegative matrix factorization with genetic algorithms for microarray analysis. In *Proceedings of the International Joint Conference on Neural Network*s (pp. 294-299).

Tan, P.-N., Steinbach, M., & Kumar, V. (2005). *Introduction to data mining*. Reading, MA: Addison-Wesley.

Tan, Y., & Zhu, Y. (2010). Fireworks algorithm for optimization. In Y. Tan, Y. Shi, & K. C. Tan (Eds.), *Proceeding of the International Conference on Advances in Swarm Intelligence* (LNCS 6145, pp. 355-364).

Wild, S. M., Curry, J. H., & Dougherty, A. (2004). Improving non-negative matrix factorizations through structured initialization. *Pattern Recognition*, *37*(11), 2217–2232. doi:10.1016/j.patcog.2004.02.013

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques*. San Francisco, CA: Morgan Kaufmann.

Xue, Y., Tong, C. S., Chen, Y., & Chen, W. (2008). Clustering-based initialization for non-negative matrix factorization. *Applied Mathematics and Computation*, *205*(2), 525–536. doi:10.1016/j.amc.2008.05.106

Zhang, Q., & Berry, M. W., Lamb, B. T., & Samuel, T. (2009). A parallel nonnegative tensor factorization algorithm for mining global climate data. In G. Allen, J. Nabrzyski, E. Seidel, G. Dick van Albada, J. Dongarra, & P. M. A. Sloot (Eds.), *Proceedings of the 9th International Conference on Computational Science* (LNCS 5545, pp. 405-415).

Andreas Janecek is a post-doctoral researcher at the School of Electronic Engineering and Computer Science, Peking University, China. He received his PhD degree in Computer Science in 2010, and his MS degree in Business Informatics in 2005, both from the University of Vienna, Austria. Besides computational intelligence such as swarm optimization and evolutionary computing, his research activities include data mining and machine learning algorithms, with a focus on high performance and distributed computing aspects of these techniques.

Ying Tan received the BS in 1985, the MS in 1988, and the PhD in signal and information processing from Southeast University in 1997, respectively. Since then, he became a postdoctoral fellow then an associate professor at University of Science and Technology of China. He worked with the Chinese University of Hong Kong in 1999 and in 2004-2005. Now, he is a professor at the Key Laboratory of Machine Perception (MOE), Peking University, and department of Machine Intelligence, EECS, Peking University. He is also the director of Computational Intelligence Laboratory (CIL) of Peking University. He has published more than 200 academic papers in refereed journals and conferences and several books and chapters in book and holds 4 invention patents. His current research interests include computational intelligence, artificial immune system, swarm intelligence, data mining, pattern recognition, and their applications. He was the general chair of International Conference on Swarm Intelligence (ICSI 2010, ICSI 2011) and honored the Second-class Prize of National Natural Science Award of China in 2009.

An Optimization Algorithm Based on Brainstorming Process

Yuhui Shi, Xi'an Jiaotong-Liverpool University, China

ABSTRACT

In this paper, the human brainstorming process is modeled, based on which two versions of Brain Storm Optimization (BSO) algorithm are introduced. Simulation results show that both BSO algorithms perform reasonably well on ten benchmark functions, which validates the effectiveness and usefulness of the proposed BSO algorithms. Simulation results also show that one of the BSO algorithms, BSO-II, performs better than the other BSO algorithm, BSO-I, in general. Furthermore, average inter-cluster distance D_c and inter-cluster diversity D_e are defined, which can be used to measure and monitor the distribution of cluster centroids and information entropy of the population over iterations. Simulation results illustrate that further improvement could be achieved by taking advantage of information revealed by D_c, which points at one direction for future research on BSO algorithms.

Keywords: Algorithm, Brain Storm Optimization, Brainstorming Process, Diversity, Optimization

INTRODUCTION

Many real-world applications can be represented as optimization problems of which algorithms are required to have the capability to search for optimum. Originally, these optimization problems were mathematically represented by continuous and differentiable functions so that algorithms such as hill-climbing algorithms can be designed and/or utilized to solve them. Traditionally, these hill-climbing like algorithms are single-point based algorithms such as gradient decent algorithms which move from the current point along the direction pointed by the negative of the gradient of the function at the current point. These hill-climbing algorithms can find solutions quickly for unimodal problems, but they have the problems of being sensitive to initial search point and being easily trapped into local optimum for nonlinear multimodal problems. Furthermore, these mathematical functions need to be continuous and differentiable, which instead greatly narrows the range of real-world problems that can be solved by hill-climbing algorithms. Recently, evolutionary algorithms have been designed and utilized to solve optimization problems. Different from traditional single-point based algorithms such as hill-climbing algorithms, each evolutionary algorithm is a population-based algorithm, which consists of a set of points (population of individuals). The population of individuals is expected to have high tendency to move towards better and better solution areas iteration over iteration through cooperation and/or competition among themselves. There are

DOI: 10.4018/jsir.2011100103

a lot of evolutionary algorithms out there in the literature. The most popular evolutionary algorithms are evolutionary programming (Fogel, 1962), genetic algorithm (Holland, 1975), evolution strategy (Rechenberg, 1973), and genetic programming (Koza, 1992), which were inspired by biological evolution. In evolutionary algorithms, population of individuals survives into the next iteration. Which individual has higher probability to survive is proportional to its fitness value according to some evaluation function. The survived individuals are then updated by utilizing evolutionary operators such as crossover operator and mutation operator, *etc*. In evolutionary programming and evolution strategy, only the mutation operation is employed, while in genetic algorithms and genetic programming, both the mutation operation and crossover operation are employed. The optimization problems to be optimized by evolutionary algorithms do not need to be mathematically represented as continuous and differentiable functions, they can be represented in any form. Only requirement for representing optimization problems is that each individual can be evaluated as a value called fitness value. Therefore, evolutionary algorithms can be applied to solve more general optimization problems, especially those that are very difficult, if not impossible, for traditional hill-climbing algorithms to solve.

Recently, another kind of algorithms, called swarm intelligence algorithms, is attracting more and more attentions from researchers. Swarm intelligence algorithms are usually nature-inspired optimization algorithms instead of evolution-inspired optimization algorithms such as evolutionary algorithms. Similar to evolutionary algorithms, a swarm intelligence algorithm is also a population-based optimization algorithm. Different from the evolutionary algorithms, each individual in a swarm intelligence algorithm represents a simple object such as ant, bird, fish, *etc*. So far, a lot of swarm intelligence algorithms have been proposed and studied. Among them are particle swarm optimization(PSO) (Eberhart & Shi, 2007; Shi & Eberhart, 1998), ant colony optimization algorithm(ACO) (Dorigo, Maniezzo, & Colorni, 1996), bacterial forging optimization algorithm(BFO) (Passino, 2010), firefly optimization algorithm (FFO) (Yang, 2008), bee colony optimization algorithm (BCO) (Tovey, 2004), artificial immune system (AIS) (de Castro & Von Zuben, 1999), fish school search optimization algorithm(FSO) (Bastos-Filho, De Lima Neto, Lins, Nascimento, & Lima, 2008), shuffled frog-leaping algorithm (SFL) (Eusuff & Lansey, 2006), intelligent water drops algorithm (IWD) (Shah-Hosseini, 2009), to just name a few.

In a swarm intelligence algorithm, an individual represents a simple object such as birds in PSO, ants in ACO, bacteria in BFO, *etc*. These simple objects cooperate and compete among themselves to have a high tendency to move toward better and better search areas. As a consequence, it is the collective behavior of all individuals that makes a swarm intelligence algorithm to be effective in problem optimization.

For example, in PSO, each particle (individual) is associated with a velocity. The velocity of each particle is dynamically updated according to its own historical best performance and its companions' historical best performance. All the particles in the PSO population fly through the solution space in the hope that particles will fly towards better and better search areas with high probability.

Mathematically, the updating process of the population of individuals over iterations can be looked as a mapping process from one population of individuals to another population of individuals from one iteration to the next iteration, which can be represented as $P_{t+1} = f(P_t)$, where P_t is the population of individuals at the iteration t, $f()$ is the mapping function. Different evolutionary algorithm or swarm intelligence algorithm has a different mapping function. Through the mapping function, we expect the population of individuals will update to better and better solutions over iterations. Therefore mapping functions should possess the property of convergence. For nonlinear and complicated problems, mapping functions more

like to move population of individuals toward local minima, which may not be good enough solutions to the optimization problems to be solved. A good mapping function should have not only the capability to converge, but also the capability to diverge when it gets trapped into local minima. As for evolutionary algorithms and swarm intelligence algorithms, they should have the capability to be in convergence or divergence state accordingly. A lot of researches have been done and reported with regards to this. For example, in particle swarm optimization algorithms, diversity has been preserved to keep the algorithm to have good search capability. Different diversity measurements have been defined and monitored (Shi & Eberhart, 2008, 2009). A better designed population-based algorithm should have a good balance of convergence and divergence.

In this paper, we will introduce a new optimization algorithm that is based on the collective behavior of human being, that is, the brainstorming process. It is natural to expect that an optimization algorithm based on human collective behavior could be a better optimization algorithm than existing swarm intelligence algorithms which are based on collective behavior of simple insects, because human beings are social animal and are the most intelligent animals in the world. The designed optimization algorithm will naturally have the capability of both convergence and divergence.

The remaining paper is organized as follows. The human brainstorming process is reviewed. The model of a brainstorming process is proposed and discussed. Two versions of novel optimization algorithms inspired by human brainstorming process are introduced and described, followed by experiments and result discussion on benchmark functions. Finally, conclusions are given.

BRAINSTORMING PROCESS

Brainstorming process has often been utilized for innovative problem solving. It can solve a lot of difficult problems which usually can't be solved by a single person. In a brainstorming process, a group of people with diverse background are gathered together to brainstorm. A facilitator will usually be involved to facilitate the brainstorming process but not directly involved in idea generation himself (or herself). The facilitator usually should have enough facilitation experience but have less knowledge about the problem to be solved so that generated ideas will have less, if not none, biases from the facilitator. The brainstorming process is used to generate many ideas as diverse as possible so that good solutions to solve the problem can be obtained from these ideas. The brainstorming process usually consists of several rounds of idea generation. In each round of idea generation, the brainstorming group is asked to come out a lot of ideas. At the end of each round of idea generation process, better ideas among them will be picked up and will serve as clues to generate ideas in the next round of idea generation process. In the brainstorming process, there is another group of persons that serve the purpose to pick up better ideas from the ideas generated in each round of idea generation process. Through the brainstorming process, hopefully great and un-expectable solution can occur from collective intelligence of human being, and the problem can usually be solved with high probability.

To help generate more diverse ideas, the Osborn's original four rules of idea generation in a brainstorming process (Osborn, 1963; Smith, 2002) should be obeyed. The four rules are listed in the Table 1. One major role of the facilitator is to facilitate the brainstorming group to obey the Osborn's four rules.

The four rules in Table 1 guide the idea generation in each round of idea generation during a brainstorming process. In order to keep the brainstorming group to be open-minded, there is no idea as good idea or bad idea, any idea is welcomed. For any idea generated during each round of idea generation process, there should be no judgment and/or criticism whether it is good idea or bad idea. Any judgment should be held back until the end of this round of idea generation process when better ideas

Table 1. Osborn's original rules for idea generation in a brainstorming process

1. Suspend Judgment
2. Anything Goes
3. Cross-fertilize (Piggyback)
4. Go for Quantity

Table 2. Steps in a brainstorming process

1. Get together a brainstorming group of people with as diverse background as possible;
2. Generate many ideas according to the rules in Table 1;
3. Have several, say 3 or 5, clients act as the owners of the problem to pick up several, say one from each owner, ideas as better ideas for solving the problem;
4. Use the ideas picked up in the Step 3 with higher probability than other ideas as clues, and generate more ideas according to the rules in Table 1;
5. Have the owners to pick up several better ideas generated as did in Step 3;
6. Randomly pick an object and use the functions and appearance of the object as clues, generate more ideas according to the rules in Table 1;
7. Have the owners to pick up several better ideas;
8. Hopefully a good enough solution can be obtained by considering the ideas generated.

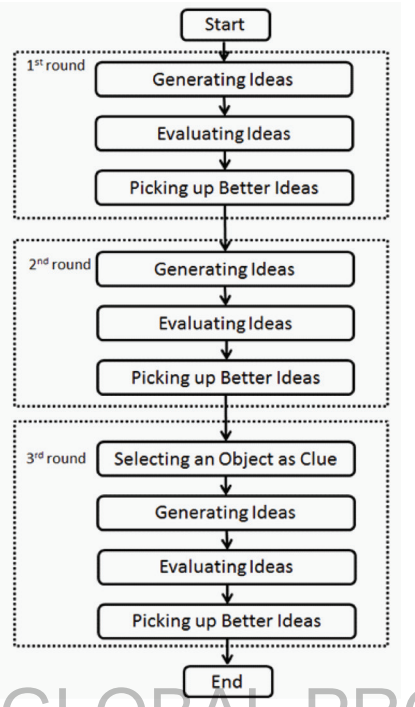
are picked up by problem owners. This is what the Rule 1 "Suspend Judgment" means. The Rule 2 "Anything Goes" means that any thought comes to your mind should be raised and recorded. Don't let any idea or thought pass by without sharing with other brainstorming group members. The Rule 3 "Piggyback" says any generated idea could and should serve as a clue to inspire the brainstorming group to come out more ideas. Ideas are not independently generated. They are related. The late generated ideas are inspired and dependent on the previously generated ideas. The Rule 4 "Go for quantity" says that we focus on generating as many ideas as possible. Hopefully quality of ideas will come out of quantity of idea naturally. Without generating large quantity of ideas, it is naive to believe that good quality ideas will come out.

The purpose to generate ideas according to rules in Table 1 is to keep the brainstorming group to be open-minded as much as possible so that they will generate ideas as diverse as possible. A brainstorming process generally follows the steps listed in Table 2 (Shi, 2011). After some time of brainstorming, the brainstorming group will become tired and narrow-minded, and therefore it becomes harder to come out new diverse ideas. The operation of picking up an object in Step 6 in Table 2 serves for the purpose to help brainstorming group to diverge from previously generated ideas therefore to avoid being trapped by the previously generated ideas. Picking up several good ideas from ideas generated so far is to cause the brainstorming group to pay more attention to the better ideas which the brainstorming group believes to be. The ideas picked-up works like point-attraction for the idea generation process while ideas generation works like point-expansion. Therefore, there are attraction and expansion embedded in the brainstorming process naturally.

MODELING BRAINSTORMING PROCESS

The procedure of a brainstorming process listed in Table 2 can be described by the flow chart shown in Figure 1. There are three rounds of idea generation involved in a brainstorming process in general. In each round of brainstorming process, there are several steps. For example, in the first round, there are idea generations, idea evaluations, and idea picking up. The idea evaluation step serves the purpose of finding

Figure 1. Flow chart of a brainstorming process



out better ideas. By idea evaluation, good ideas could be identified and picked up in the Picking up Better Ideas step, which simulates picking up good ideas by problem owners. The first round simulates Step 2 &3 in Table 2. The second round is the same as the first round which simulates Step 4 & 5 in Table 2. The third round is the same as the first two rounds except that one additional step Selecting an Object as Clue is added to simulate randomly picking up an object as clues in Step 6 in Table 2. Each step in a brainstorming process therefore can be modeled (and/or simulated) and put together as a model for the brainstorming process as shown in Figure 1, which will be further explained and modified in the following sub-sections.

Population

A solution to a problem with d variables to be optimized can be looked as a point in the d dimensional solution space. An idea can be considered as a potential solution, i.e., a point in the solution space. Therefore to find a good solution is equivalent to find a point or a solution in the solution space. A group of ideas can therefore be considered as a population of solutions or individuals in the solution space. If for every round of idea generation in the brainstorming process, a fixed number of n ideas will be generated before the problem owners pick up good ideas, then these n ideas can be considered as a population of individuals (or solutions) with population size being n in the solution space. Therefore, the human brainstorming process can be considered as generating a population of individuals iteratively three times as shown in Figure 1. One round of idea generation can be considered as one iteration of individual generations in population-based optimization algorithms such as particle swarm optimization algorithm. The difference between them is the way how new population of individuals is generated based on the current population of individuals.

Initialization

The Generating Ideas step in the very first round of idea generations can be considered as the population initialization in any population-based optimization algorithm. During the population initialization, to gather a group of people with as diverse background as possible can be considered as initializing the population of individuals randomly with uniform distribution over the dynamic range of the solution space. The whole population of individuals can be totally randomly generated or only portion of the population is randomly generated and the rest of the population of individuals will be generated by adding noise to the already randomly generated individuals. To preserve the initialized population to be diversified, usually *a priori* domain knowledge should not be utilized in the initialization process, unless when computation cost is the first priority, in which the domain knowledge should be utilized to initialize the population to find good solution quickly at the risk of premature convergence.

Clustering

Each round of idea generation generates enough ideas, but not necessary too many ideas because otherwise all the generated ideas will more like to diverge, and therefore will be far away from expected ideas which are close to expected solutions. To have diverse ideas is good to seek around all possible ideas to help find good potential solutions, but there should be a tradeoff between divergence and focus. We also need to pull the brainstorming group back to concentrate on generating ideas around some areas with high potential to speed up searching for good enough ideas. The problem owners in the brainstorming process serve this purpose. They are asked to pick good ideas from generated ideas. Because every problem owner has different expertise and knowledge, therefore the picked ideas will be different. They represent potential good ideas that have been generated so far. Next round of idea generation should better be conducted with focus on them. Certainly, it does not exclude idea generation by piggybacking other ideas, but with small probability. One way to simulate the idea picking up by problem owners is to use clustering algorithms. All the individuals (ideas) in the population are clustered into several clusters. The number of clusters corresponds to the number of problem owners. The cluster center of each cluster corresponds to the idea(s) picked up by a problem owner. The cluster center for each cluster can be the best performed individual within this cluster. It can also be the centroid of the cluster.

One possible clustering algorithm is the k-means clustering algorithm (MacQueen, 1967), which requires to know the number of clusters *k a priori*. The number *k* corresponds to the number of problem owners, that is, the number of problem owners is fixed. The self-organizing feature map (Kohonen & Honkela, 2007) is another kind of clustering algorithm, in which the number of clusters is unknown before running the algorithm. The number of clusters will be determined by the algorithm itself according to the distribution of individuals in the population. Other clustering algorithms (Xu & Wunsch, 2005) such as partitioning around medoids (Theodoridis & Koutroumbas, 2006), fuzzy c-means (FCM) (Nock & Nielsen, 2006), *etc.* can also be employed.

Individual Generation

For idea generations by piggyback, it is similar to randomly select one or several existing individuals (or ideas) and generate a new individual by adding noise to the selected individual(s). The purpose of doing this is to guarantee that new individuals (ideas) are generated by piggybacking existing individuals as diverse as possible. If a new idea (individual) x_{new} is generated by piggybacking one existing idea (individual) x_{old}, it can be written as

$$x_{new}^{i} = x_{old}^{i} + \xi(t) * random(t) \quad (1)$$

where $x_{new}{}^{i}$ and $x_{old}{}^{i}$ are the ith dimension of x_{new} and x_{old}, respectively; *random(t)* is a random

function; $\xi(t)$ is a coefficient that weights the contribution of random value to the new individual. The formula is similar to the mutation operation in evolutionary programming algorithm. The commonly utilized random function in mutation operation is the Gaussian function (Yao, Liu, & Lin, 1997). Other random functions that can be used are Cauchy function (Yao, Liu, & Lin, 1997), Lévy flights (Pavlyukevich, 2007), *etc*. Compared with Gaussian function, Cauchy function has a longer tail which makes it preferable if wider areas need to be explored (Yao, Liu, & Lin, 1997).

If a new idea (individual) x_{new} is generated by piggybacking two existing ideas (individuals) x_{old1} and x_{old2}, it can be written as

$$x_{new}^{i} = x_{old}^{i} + \xi(t) * random(t) \quad (2a)$$

$$x_{old}^{i} = w_1 * x_{old1}^{i} + w_2 * x_{old2}^{i} \quad (2b)$$

where x_{old}^{i} is the weighted summation of the ith dimension of x_{old1} and x_{old2}; w_1 and w_2 are two coefficients to weight the contribution of two existing individuals. The formula simulates generating new idea by piggybacking two existing ideas. Certainly, a new idea can also be generated by piggybacking more than two existing ideas.

No matter how many existing ideas (individuals) will be piggybacked to generate new ideas (individuals), the cluster centers will have high probability to be chosen to generate new ideas (individuals) compared with the other non-cluster-center ideas (individuals) which usually can be chosen with small probability.

The coefficient $\xi(t)$ weights the contribution of randomly generated value to the new individual. Generally, large $\xi(t)$ value facilitates exploration while small $\xi(t)$ values facilitates exploitation. When global search capability is preferred, for example, at the beginning of search process, $\xi(t)$ should give large value, while when local search capability is preferred, for example, at the end of search process, $\xi(t)$ should give small value. One possible function for $\xi(t)$ is

$$\xi(t) = logsig\left(\frac{\frac{T}{2} - t}{k}\right) * random(t) \quad (3)$$

where *logsig*() is a logarithmic sigmoid transfer function, T is the maximum number of iterations, and t is the current iteration number, k is for changing *logsig*() function's slope, and *random*() is a random value within $(0,1)$.

Disruption

After two rounds of idea generation, the mindset of the brainstorming group usually will be narrowed and therefore it becomes more difficult, if not impossible, for them to come out different ideas efficiently. To further explore whether there are potential good ideas out there somewhere, in the brainstorming process, an object will be randomly picked up, and the brainstorming group will be asked to generate new ideas which are more or less related to the functions and appearance of the object. The purpose of this is to help the brainstorming group to disrupt from their current mindset, which is usually difficult to achieve. This disruption operation can be simulated by replacing selected ideas (individuals) with randomly generated individuals. Therefore, wider areas could be explored with high probability by utilizing disruption operation.

As shown in Figure 1, there are three rounds of idea generation. The first two rounds are identical while the third round serves as the purpose of disruption with the step Selecting an Object as Clue. To further modulate the operations, this disruption operation could be distributed and shared among all three rounds of idea generation. Figure 2 shows the modified flow chart of the brainstorming process, which includes three identical rounds of idea generation. Each round of idea generation is shown in Figure 3 in which the step Selecting an Object as Clue is changed to be the step

Figure 2. Flow chart of a brainstorming process

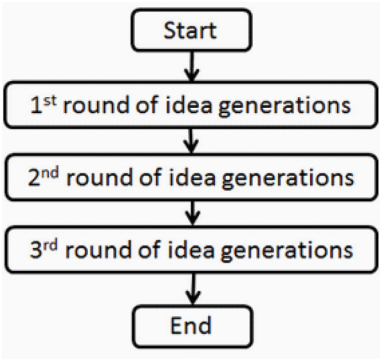
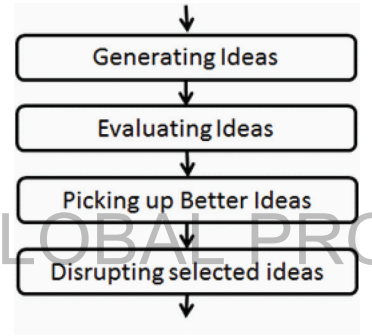


Figure 3. Flow chart of one round of idea generation



Disrupting Selected Ideas and it is put at the end of each round.

Selection

In a population-based optimization algorithm, generally speaking, if it is not because of specific requirements, the population size p is fixed and not changed during the algorithm running time. During each iteration, number of new individuals will be generated, say n $(n \geq p)$, therefore there will exist $p+n$ number of individuals, among which only p will be copied into the next iteration due to the fixed population size. Similar to other population-based algorithms, how to select p from $p+n$ individuals is critical to the optimization algorithm inspired by the brainstorming process. One simple way is that for each existing individual in the population, a new individual is generated. This pair of individuals is compared. The better one will be kept as the individual into the next iteration. Another way could be to randomly pick up p pairs of individuals from the $n+p$ individuals, and the better one of each pair will be kept into the next iteration.

To further take advantage of information embedded in each pair of individuals, crossover operations could also be applied to each pair of individuals to generate two new offspring. The best of the four will then be copied into the next iteration.

In each round of idea generation shown in Figure 3, one more step Selecting Ideas is inserted right below the step Generating Ideas. Figure 4 shows the new flow chart of each round of idea generation.

Figure 4. Flow chart of one round of idea generation

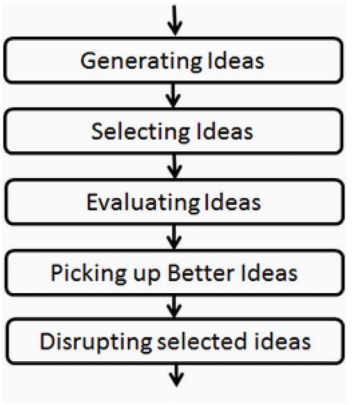
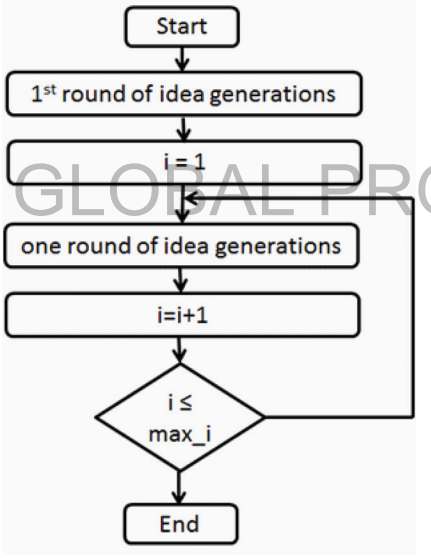
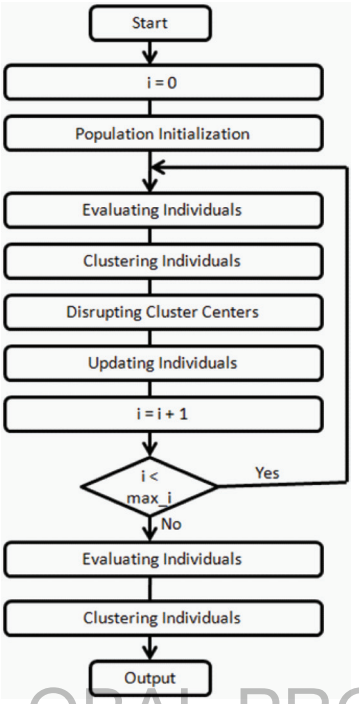


Figure 5. Flow chart of a brainstorming process



In practice, limited time will be taken for a brainstorming process, otherwise, the brainstorming group will be tired to generate new meaningful ideas efficiently. Usually as a good practice, a brainstorming process takes approximately 60 minutes. As shown in the Figure 1, there are only three rounds of idea generation in a brainstorming process. But for a model to be executed by computers, the number of rounds of idea generation can be as large as that we want. Figure 5 shows the flow chart for a brainstorming process that can be simulated by computers. In Figure 5, the step 1st Round of Idea Generations is the same as the step One Round of Idea Generation shown in Figure 5. The purpose to have the extra step 1st Round of Idea Generations at the beginning is to be similar to the Initialization step in population-based algorithms. The *max_i* is the maximum number of rounds of idea generation we want to conduct. Therefore totally, *max_i* rounds of idea generation will be conducted in

Figure 6. An implementation of BSO algorithm



the brainstorming process shown in the Figure 5.

By implementing Figure 5, a model or algorithm to mimic the human being brainstorming process can be built.

BRAIN STORM OPTIMIZATION ALGORITHM

According to the Figure 5, a brain storm optimization (BSO) algorithm can be designed by directly mapping the steps in the Figure 5. By some straightforward rearrangement, one possible flow chart of the BSO algorithm is shown in Figure 6. In Figure 6, there are five main operations among which three operations are unique to the BSO algorithm and the other two operations are similar to those in other evolutionary algorithms.

In the procedure of the Brain Storm Optimization (BSO) algorithm shown in the Figure 6, the first two steps are the initialization step and evaluation step which are the same as that in other swarm intelligence algorithms. In the initialization step, the population of individuals is usually uniformly and randomly initialized within the dynamic range of solution space. The population size n simulates the number of ideas generated in each round of idea generation in the brainstorming process. For the simplicity of the algorithm, the population size usually is set to be a constant number for all iterations in the BSO algorithm. In the evaluation step, each individual will be evaluated. An evaluation value (fitness) will be obtained to measure how good the individual as a potential solution to the problem to be solved. The third step is to cluster the population of individuals into several clusters. Different kind of clustering algorithms could be employed. In this paper, the k-means clustering algorithm will be used as the clustering algorithm. The disruption step randomly selects a cluster center and replace it with a randomly generated individual. This step

Figure 7. One implementation of updating individual operation

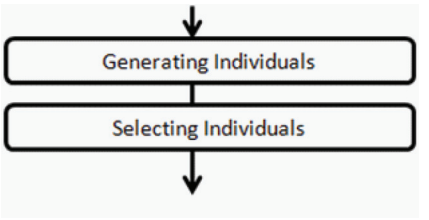
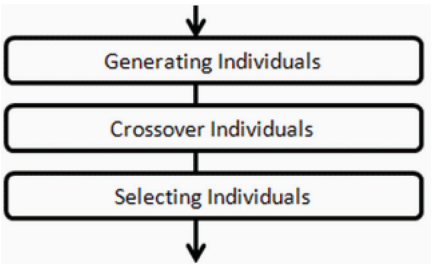


Figure 8. Another implementation of updating individual operation



will not be executed in every iteration, but will only be selected to execute with small probability.

The Updating Individuals step generally includes two sub-operations, i.e., Generating Individuals and Selecting Individuals, which is shown in Figure 7. As discussed in previous section, crossover operation could be utilized to further take advantage of existing search information. Figure 8 shows another possibility of the Updating Individuals operation which adds one additional sub-operation, i.e., crossover operation. One implementation of the BSO algorithm was introduced in Shi (2011) and is given in Table 3 here for convenience, in which the Updating Individuals operation shown in Figure 7 is implemented. By replacing the Step 6.d in Table 3 with two sub-steps shown in Table 4, another implementation of BSO algorithm can be achieved. To distinguish the two different implementations, the first one is noted as BSO-I and the second is noted as BSO-II for the purpose of description convenience. Intuitively, the BSO algorithms should be superior to other swarm intelligence algorithms, which are inspired by collective behaviors of inferior animals, because of the highest intelligence unique to human beings.

In the BSO algorithm, the number of cluster centers is usually set to be a small number, say $m=5$, and the number of generated individuals in each iteration is usually set to be a relatively large number, say $n=100$.

EXPERIMENTS AND DISCUSSIONS

Test Problems

To validate the brain storm optimization algorithms, ten benchmark functions listed in Table 5 are tested. Among them, the first five functions are unimodal functions and the remaining five functions are multimodal functions. They all are minimization problems with minimum zero. The third column in the Table 5 is the dynamic ranges for the ten benchmark functions, which have been used to test population-based algorithms in the literature. For each benchmark function, the tested BSO algorithm will be run 50 times to obtain reasonable statistical results.

Table 3. The procedure of the brain storm optimization algorithm in Shi (2011)

1. Randomly generate n potential solutions (individuals);
2. Evaluate the n individuals;
3. Cluster n individuals into m clusters by k-means clustering algorithm;
4. Rank individuals in each cluster and record the best individual as cluster center in each cluster;
5. Randomly generate a value between 0 and 1;
   a) If the value is smaller than a pre-determined probability $p_{5a,}$
      i. Randomly select a cluster center;
      ii. Randomly generate an individual to replace the selected cluster center;
   b) Otherwise, do nothing.
6. Generate new individuals
   a) Randomly generate a value between 0 and 1;
   b) If the value is less than a probability p_{6b},
      i. Randomly select a cluster with a probability p_{6bi};
      ii. Generate a random value between 0 and 1;
      iii. If the value is smaller than a pre-determined probability p_{6biii},
         1) Select the cluster center and add random values to it to generate new individual.
      iv. Otherwise randomly select an individual from this cluster and add random value to the individual to generate new individual.
   c) Otherwise randomly select two clusters to generate new individual
      i. Generate a random value;
      ii. If it is less than a pre-determined probability p_{6c}, the two cluster centers are combined and then added with random values to generate new individual;
      iii. Otherwise, two individuals from each selected cluster are randomly selected to be combined and added with random values to generate new individual.
   d) The newly generated individual is compared with the existing individual with the same individual index, the better one is kept and recorded as the new individual;
7. If n new individuals have been generated, go to step 8; otherwise go to step 6;
8. Terminate if pre-determined maximum number of iterations has been reached; otherwise go to step 2.

Table 4. Two sub-steps to replace step 6.d in table 3

d) The newly generated individual crossovers with the existing individual with the same individual index to generate two more individuals (offspring);
e) The four individuals are compared, the best one is kept and recorded as the new individual;

Simulations on *k*

In Shi (2011), the BSO-I algorithm was tested on two benchmark functions, i.e., the Sphere function and the Rastrigin function. The parameters are setup as that listed in Table 6. The purpose there is to validate the usefulness and effectiveness of the proposed BSO-I algorithm. Generally speaking, the parameter *k* determines the slope of the *logsig*() functions, therefore it determines the decreasing speed of the step-size $\xi(t)$ over iterations. Different *k* should have different impacts on the performance of BSO algorithms. In order to test the impact of *k* on BSO performance, we change the *k* value while all other parameter values are kept to be the same as that listed in Table 6. For this purpose, again only one unimodal function, Sphere function, and one multimodal function, Rastrigin function, are utilized. The dimension of the two functions is set to be 20.

Table 7 gives the simulation results. The results given in the Table 7 are mean, best, worst function values and their variance at the final iteration over *50* runs. From the Table 7, it can be observed that generally there is no single parameter *k* value with which the BSO algorithm

Table 5. Benchmark functions tested in this paper

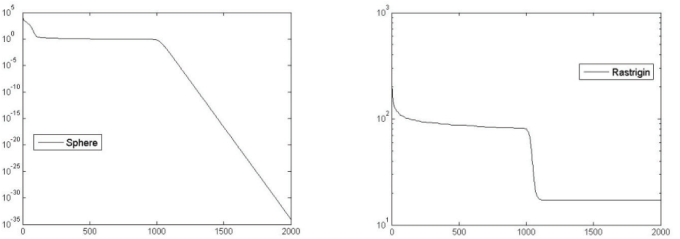
Function	Expressions	Range
Sphere	$f_1 = \sum_{i=1}^{d} x_i^2$	$[-100,100]^d$
Schwefel's P221	$f_2 = \max_i\{\lvert x_i\rvert\}$	$[-100,100]^d$
Step	$f_3 = \sum_{i=1}^{d} (\lfloor x_i + 0.5\rfloor)^2$	$[-100,100]^d$
Schwefel's P222	$f_4 = \sum_{i=1}^{d} \lvert x_i\rvert + \prod_{i=1}^{d} \lvert x_i\rvert$	$[-10,10]^d$
Quartic Noise	$f_5 = \sum_{i=1}^{d} i x_i^4 + \text{random}[0,1)$	$[-1.28,1.28]^d$
Ackely	$f_6 = -20\exp\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)\right) + 20 + e$	$[-32,32]^d$
Rastrigin	$f_7 = \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i) + 10]$	$[-5.12,5.12]^d$
Rosenbrock	$f_8 = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^{\,2})^2 + (x_i - 1)^2]$	$[-30,30]^d$
Schwefel's P226	$f_9 = -\sum_{i=1}^{d} (x_i \sin(\sqrt{\lvert x_i\rvert})) + 418.9829d$	$[-500,500]^d$
Griewank	$f_{10} = \frac{1}{4000}\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$[-600,600]^d$

can have the best performance. From the Table 7, relatively speaking, unimodal function (Sphere function) prefers relatively small *k* value while multimodal function (Rastrigin function) prefers relatively large *k* value. By considering the robustness of the BSO algorithm and from the results given in Table 7 itself, generally speaking, a good choice for the parameter *k* is *25* as a tradeoff between unimodal function and multimodal function. The obtained mean function values over *2000* iterations with parameter *k =25* are shown in Figure 9. From the Figure 9, it can be observed that the BSO-I with *k =25* can converge fast when solving the Sphere function and Rstrigin function. In all simulations, we will set the parameter *k* to be *25* with all other parameters are set as the same as that in Table 6.

Simulations on BSO-I Algorithm

The BSO-I algorithm is tested on the ten benchmark functions listed in Table 5 to illustrate the effectiveness and efficiency of the BSO-I

Figure 9. Obtained Mean Minimum Values vs. Iterations for BSO-I with k = 25



a) 20-D Sphere Function *b) 20-D Rastrigin Function*

Table 6. Set of parameters for BSO algorithm

n	m	p_{5a}	p_{6b}	p_{6biii}	p_{6c}	k	Max_ iteration	μ	σ
100	5	0.2	0.8	0.4	0.5	20	2000	0	1

Table 7. Simulation results of BSO-I with different k

function	k	mean	best	worst	variance
Sphere	10	1.20381E-11	2.55674E-86	5.27132E-10	5.61316E-21
	20	2.30827E-43	1.24079E-43	3.17853E-43	2.5931E-87
	25	9.4726E-35	5.55931E-35	1.33105E-34	3.78414E-70
	30	5.35092E-29	3.01328E-29	7.6509E-29	1.30974E-58
	40	7.70746E-22	4.18609E-22	1.10552E-21	2.74116E-44
	50	1.60782E-17	7.60191E-18	2.22015E-17	1.18045E-35
Rastrigin	10	17.11636	5.969754	29.84873	28.93288
	20	18.00875	8.954632	31.83866	20.98068
	25	17.17298	6.964713	23.879	13.24541
	30	17.15308	7.959667	29.84871	26.04389
	40	15.81984	6.964713	24.87396	17.95025
	50	16.21782	8.954632	25.8689	16.85928

algorithm instead of only two benchmark functions in Shi (2011). Each function is tested with three different dimension setting, *10*, *20*, and *30*, respectively. The experimental results are given in Tables 8 and 9 for unimodal functions and multimodal functions, respectively. From the Table 8, it can be seen that good results can be achieved by BSO-I algorithm, and the results also show that the BSO-I is robust and reliable when it is applied to solve benchmark unimodal functions. From the Table 9, it can be seen that good results can be obtained for function f_6, relatively good results can be obtained for functions f_7 and f_8, but not relatively

Table 8. Simulation results of BSO-I on unimodal functions

Function	Dimension	mean	best	worst	variance
f_1	10	1.3989E-35	3.90855E-36	2.71203E-35	2.75801E-71
	20	9.77845E-35	6.11475E-35	1.37856E-34	3.55418E-70
	30	2.66069E-34	1.79135E-34	3.59892E-34	2.02141E-69
f_2	10	2.31285E-18	1.49658E-18	3.11619E-18	1.47169E-37
	20	5.05671E-18	3.69394E-18	6.40744E-18	4.41064E-37
	30	0.000235	3.18538E-08	0.001718355	1.55583E-07
f_3	10	0	0	0	0
	20	0	0	0	0
	30	0	0	0	0
f_4	10	9.28917E-18	5.51341E-18	1.21942E-17	1.81665E-36
	20	3.4224E-17	2.63097E-17	4.25525E-17	1.03733E-35
	30	1.9978E-06	5.84794E-17	9.94736E-05	1.97869E-10
f_5	10	0.000424	4.52215E-05	0.001140455	6.14016E-08
	20	0.002636	0.000613	0.008465	2.8024E-06
	30	0.00835095	0.001967	0.020706	1.33183E-05

Table 9. Simulation results of BSO-I on multimodal functions

Function	Dimension	mean	best	worst	variance
f_6	10	4.44089E-15	4.44089E-15	4.44089E-15	0
	20	4.44089E-15	4.44089E-15	4.44089E-15	0
	30	5.93303E-15	4.44089E-15	7.99361E-15	3.13741E-30
f_7	10	3.502256	0	5.969754	1.949178
	20	17.75005	8.954632	26.86387	15.12629
	30	34.56484	13.92943	51.7378	51.65143
f_8	10	6.330642	2.587793	29.36235	11.77892
	20	21.60337	15.83735	87.11474	255.4539
	30	42.02786	25.91331	296.7523	2073.832
f_9	10	1350.782	454.0165	2270.172	192322.2
	20	3012.657	1598.991	4501.054	570878.2
	30	4951.779	3652.088	6771.33	563448.4
f_{10}	10	1.35123	0.497182	2.21245	0.158512
	20	0.058446	0	0.9467	0.022289
	30	0.010777	0	0.056496	0.000163

good results are obtained for f_9 which is in general a difficult function to optimize, and for function f_{10}, good results can be obtained for it with dimension *20* and *30*, but not with dimension*10*, for which only relatively good results are obtained instead.

Simulation on BSO-II Algorithm

The BSO-II algorithm further exploits the search areas by generating two new offspring through utilizing crossover operation to crossover the newly generated individual with the existing individual with the same individual index. The BSO-II is applied to the ten benchmark functions with dimensions *10*, *20*, and *30*, respectively. The simulation results are given in Tables 10 and 11 for unimodal functions and multimodal functions, respectively. From the Table 10, we can observe that good results can be achieved by the BSO-II algorithm, and the results also show that the BSO-II is robust and reliable when it is applied to solve benchmark unimodal functions. From the Table 11, it can be seen that good results can be obtained for function f_6, relatively good results can be obtained for functions f_7 with dimension 30 and f_8, but not relatively good results are obtained for f_9 which is in general a difficult function to optimize, and for function f_{10}, reasonable good results can be obtained. Compared with the observation from the BSO-I algorithm, very good results (the optimum) can be obtained for f_7 with dimension *10* and *20*. For f_7 with dimension *30*, the best results over *50* runs is *0* which is the optimum of the problem, but the worst and variance over *50* runs are *3.979836* and *1.010551*, which indicates that the BSO-II is better than the BSO-I, but it is still not robust when solving f_7 function.

To further compare the BSO-I and BSO-II algorithm, Figures 10 through 19 show curves which display the average evaluation function values over *50* runs vs. iterations for the ten benchmark functions tested. From the figures, it can be easily seen that the BSO-II algorithm performs better than the BSO-I algorithm for all the benchmark functions with all three different dimensions except the Griewank function with dimension *20* and *30*. For Griewnak function with dimension *10*, the BSO-I can't obtain very good results but the BSO-II could. Therefore, even for the Griewank function, the BSO-II could be a better choice compared with the BSO-I algorithm. For function f_9, even though still not very good results are obtained by BSO-II, but BSO-II performs much better than BSO-I does.

Diversity

During each iteration, the population of individuals is clustered into *m* clusters. Individuals in each cluster are scattered with different distribution over iterations. To measure and monitor the distribution of individuals in each cluster, the following average intra-cluster distance is defined

$$d_c(x_i, x_j) = \left\| x_i - x_j \right\| \tag{4a}$$

$$\tilde{d}_c(x_i, x_j) = \frac{d_c(x_i, x_j)}{\left\| a - b \right\|} \tag{4b}$$

$$D_c = \frac{2}{q(q-1)} \sum_{i=1}^{q} \sum_{j=i+1}^{q} \tilde{d}(x_i, x_j) \tag{4c}$$

where q is the number of individuals in a cluster; $d(x_i,x_j)$ is the Euclidean distance between individual x_i and x_j; a and b are dynamic range; $\tilde{d}(x_i,x_j)$ is the normalized Euclidean distance between individual x_i and x_j; D_c is the normalized distance for a cluster. For $m=5$, there will be *5* intra-cluster diversities. In addition to m average intra-cluster distances, there will be one average inter-cluster distance to measure and/or monitor the distribution of cluster centers. The formula for calculating average intra-cluster distance can also be utilized to calculate the average inter-cluster distance except that here the x_i is the ith cluster center and number of individuals is m.

Over iterations, the number of individuals in each cluster will change. To measure and monitor the distribution of number of individu-

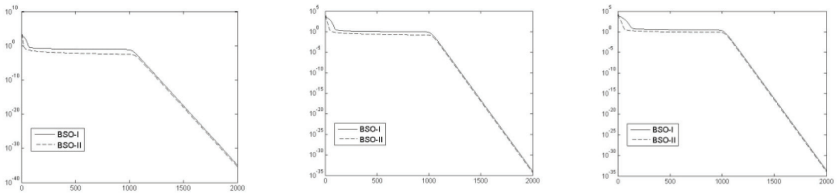
Table 10. Simulation results of BSO-II on unimodal functions

Function	Dimension	mean	best	worst	variance
f_1	10	4.56E-36	2.61244E-36	7.53912E-36	1.13477E-72
	20	4.54E-35	2.98742E-35	6.19235E-35	7.00667E-71
	30	1.33E-34	9.34047E-35	1.65808E-34	2.53466E-70
f_2	10	1.52E-18	1.10811E-18	1.94476E-18	3.80565E-38
	20	3.86E-18	3.23175E-18	4.44916E-18	8.94695E-38
	30	5.85E-18	4.80866E-18	6.91767E-18	2.62081E-37
f_3	10	0	0	0	0
	20	0	0	0	0
	30	0	0	0	0
f_4	10	4.76E-18	3.30555E-18	6.17314E-18	5.02845E-37
	20	2.13E-17	1.54076E-17	2.52198E-17	5.11026E-36
	30	4.49E-17	3.56463E-17	5.22311E-17	1.57E-35
f_5	10	8.85E-05	3.18035E-05	0.000256579	1.69387E-09
	20	0.000319	0.000104	0.000853	2.24337E-08
	30	0.000766	0.000176	0.001733	1.00554E-07

Table 11. Simulation results of BSO-II on multimodal functions

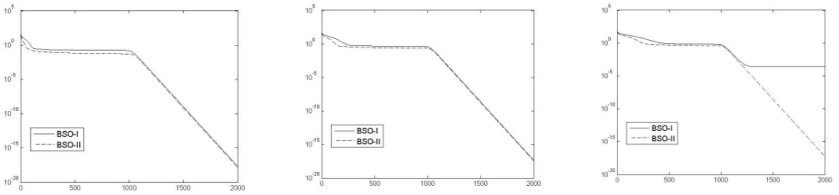
Function	Dimension	mean	best	worst	variance
f_6	10	4.16E-15	8.88178E-16	4.44089E-15	9.47921E-31
	20	4.44E-15	4.44089E-15	4.44089E-15	0
	30	4.44E-15	4.44089E-15	4.44089E-15	0
f_7	10	0	0	0	0
	20	0	0	0	0
	30	0.855665	0	3.979836	1.010551
f_8	10	4.558798	2.019811	9.069095	0.862945
	20	28.514436	15.49093	83.89686	604.2519
	30	34.06948	25.85653	128.9086	505.6776
f_9	10	56.23811	0.000127	236.8768	5855.616
	20	499.023	118.4386	927.8028	48176.18
	30	1128.729	335.5784	1993.748	157231
f_{10}	10	0.150697	0.022151	0.531254	0.019883
	20	0.311937	0.017241	1.519837	0.110482
	30	0.090445	0	0.568672	0.011172

Figure 10. Mean Function Evaluation Values vs. Iterations of Sphere Function



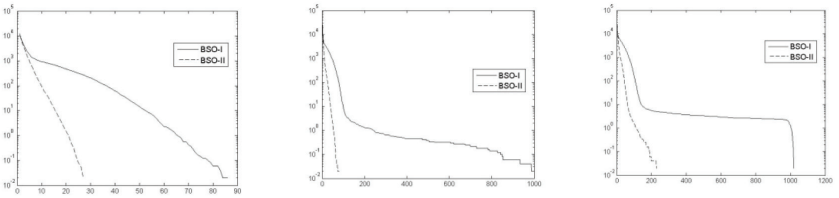
a) 10-D *b) 20-D* *c) 30-D*

Figure 11. Mean Function Evaluation Values vs. Iterations of Schwefel's P221



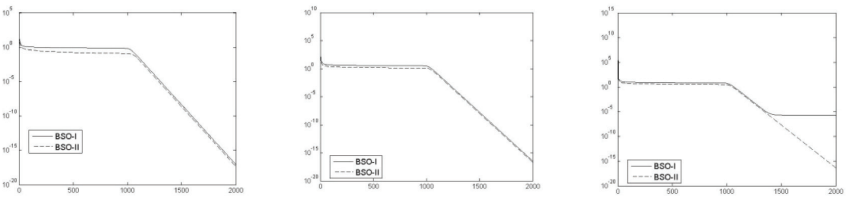
a) 10-D *b) 20-D* *c) 30-D*

Figure 12. Mean Function Evaluation Values vs. Iterations of Step Function



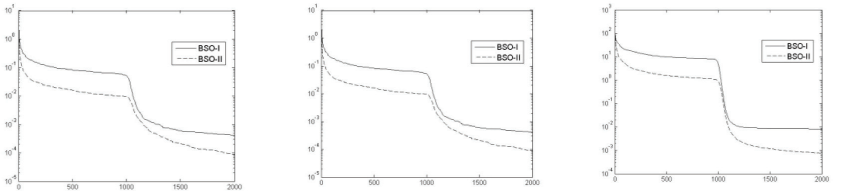
a) 10-D *b) 20-D* *c) 30-D*

Figure 13. Mean Function Evaluation Values vs. Iterations of Schwefel's P222



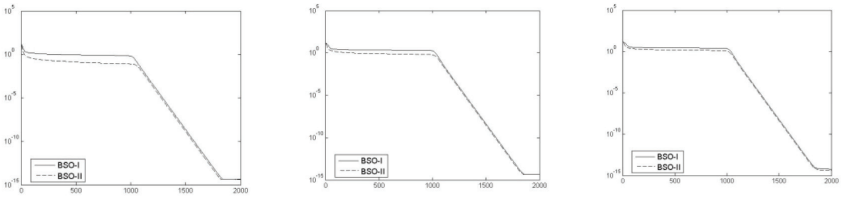
a) 10-D *b) 20-D* *c) 30-D*

Figure 14. Mean Function Evaluation Values vs. Iterations of Quartic Noise



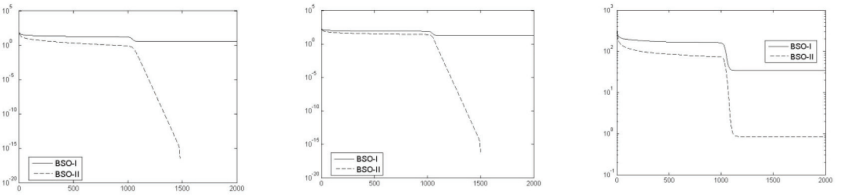
a) 10-D *b) 20-D* *c) 30-D*

Figure 15. Mean Function Evaluation Values vs. Iterations of Ackely Function



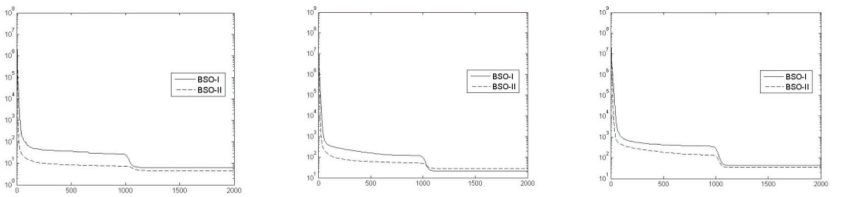
a) 10-D *b) 20-D* *c) 30-D*

Figure 16. Mean Function Evaluation Values vs. Iterations of Rastrigin Function



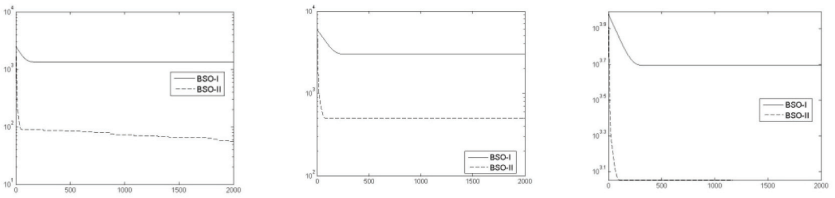
a) 10-D *b) 20-D* *c) 30-D*

Figure 17. Mean Function Evaluation Values vs. Iterations of Rosenbrock Function



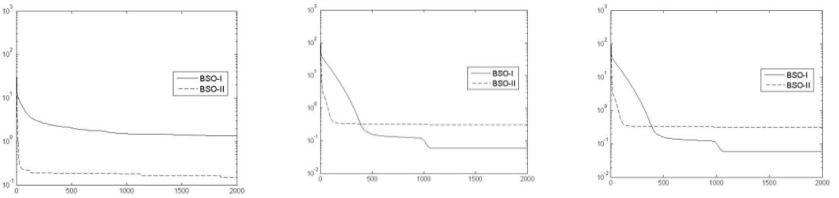
a) 10-D *b) 20-D* *c) 30-D*

Figure 18. Mean Function Evaluation Values vs. Iterations of Schwefel's P226



a) 10-D *b) 20-D* *c) 30-D*

Figure 19. Mean Function Evaluation Values vs. Iterations of Griewank Function



a) 10-D *b) 20-D* *c) 30-D*

als in each cluster over whole population, the following inter-cluster diversity is defined

$$D_v = \sum_{i=1}^{m} \frac{(n_i - \bar{n})^2}{m} \quad ,\bar{n} = \frac{\sum_{i=1}^{m} n_i}{m} \tag{5}$$

where m is the number of clusters, n_i is the number of individuals in the ith cluster. D_v is similar to the definition of variance for distribution of number of individuals in each cluster among the population.

Another similar definition of the inter-cluster diversity can be defined as

$$D_e = -\sum_{i=1}^{m} p_i \log(p_i), \quad p_i = \frac{n_i}{n} \tag{6}$$

where m is the number of clusters, n_i is the number individuals in the ith cluster. Therefore p_i is the percentage of individuals that the ith cluster has over the population. D_e is similar to the definition of information entropy. Therefore, it can be looked as a measurement of information entropy for the population. When all the individuals are located in one cluster, the D_e has the smallest value, which is *0*; when all the individuals are equally distributed into each cluster, D_e has the largest value, which is *log(m)*. If *m =5, D_e = log(5) = 0.699*.

Tables 12 and 13 give the results of average inter-cluster distance D_c and inter-cluster diversity D_e for ten tested benchmark functions at the end of BSO-I running. Tables 14 and 15 give the results of average inter-cluster distance D_c and inter-cluster diversity D_e for ten tested benchmark functions at the end of BSO-II running. Figures 20 through 29 show mean average inter-cluster distance over 50 runs vs. iterations for the ten benchmark functions, respectively. From both the Tables 12 through 15 and Figures 20 through 29, it could be easily observed that the average inter-cluster distance quickly decreases over iterations and gets to very small values way before reaching the prefixed

Table 12. Simulation results of D_c and D_e for BSO-I on unimodal functions

F	d	D_c				D_e			
		mean	short	long	variance	mean	small	large	variance
f_1	10	3.99E-20	3.02E-20	5.45E-20	2.87E-41	0.674	0.576	0.697	0.00056
	20	7.73E-20	5.84E-20	9.83-20	7.02E-41	0.658	0.566	0.697	0.000893
	30	9.92E-20	7.47E-20	1.16E-19	1.10E-40	0.645	0.473	0.695	0.001978
f_2	10	4.6E-20	3.31E-20	5.58E-20	2.71E-41	0.671	0.583	0.695	0.000528
	20	9.11E-20	6.98E-20	1.15E-19	1.17E-40	0.653	0.579	0.693	0.000653
	30	2.2E-19	1.40E-19	3.57E-19	2.36E-39	0.624	0.488	0.694	0.002807
f_3	10	0.006908	0.005538	0.008269	2.80E-07	0.685	0.666	0.697	7.604E-05
	20	0.007145	0.005921	0.007997	1.89E-07	0.666	0.572	0.695	0.000567
	30	0.006818	0.005485	0.007595	2.15E-07	0.653	0.499	0.693	0.001402
f_4	10	4.44E-19	3.24E-19	5.95E-19	3.44E-39	0.674	0.608	0.699	0.000481
	20	8.54E-19	6.73E-19	1.12E-18	1.01E-38	0.652	0.567	0.698	0.000925
	30	1.12E-18	8.86E-19	1.55E-18	1.83E-38	0.640	0.554	0.694	0.001363
f_5	10	0.06774	0.031242	0.107543	0.000272	0.609	0.484	0.681	0.002202
	20	0.041977	0.016624	0.071656	0.000163	0.604	0.394	0.686	0.003644
	30	0.029224	0.010858	0.04638	6.94E-05	0.591	0.389	0.682	0.004305

Table 13. Simulation results of D_c and D_e for BSO-I on multimodal functions

F	d	D_c				D_e			
		mean	short	long	variance	mean	small	large	variance
f_6	10	1.01E-16	7.83E-17	1.24E-16	6.71E-35	0.677	0.617	0.698	0.00023
	20	1.01E-16	7.28E-17	1.25E-16	1.30E-34	0.642	0.560	0.695	0.001086
	30	1.14E-16	1.01E-18	2.21E-16	5.69E-33	0.613	0.384	0.692	0.003186
f_7	10	2.57E-10	1.31E-16	4.39E-10	1.16E-10	0.607	0.450	0.692	0.002966
	20	1.90E-10	4.56E-17	7.13E-10	3.82E-20	0.604	0.448	0.690	0.002892
	30	1.22E-10	1.22E-17	7.10E-10	3.49E-20	0.589	0.378	0.695	0.004818
f_8	10	1.15E-17	6.43E-19	5.77E-17	1.06E-34	0.622	0.505	0.688	0.002294
	20	2.22E-17	1.17E-18	8.01E-17	3.61E-34	0.600	0.456	0.686	0.003408
	30	2.75E-17	1.49E-18	1.81E-16	1.05E-33	0.588	0.349	0.692	0.00625
f_9	10	1.91E-09	5.68E-17	3.89E-09	7.92E-19	0.618	0.115	0.685	0.007802
	20	2.32E-09	3.22E-16	4.14E-09	8.62E-19	0.603	0.378	0.692	0.004574
	30	2.16E-09	6.82E-17	4.30E-09	1.30E-18	0.593	0.097	0.683	0.00893
f_{10}	10	1.21E-11	7.22E-19	8.07E-11	3.72E-22	0.589	0.097	0.693	0.01162
	20	5.01E-11	5.05E-18	9.57E-11	6.74E-22	0.602	0.362	0.689	0.003937
	30	5.21E-11	3.854E-16	9.28E-11	5.74E-22	0.610	0.506	0.696	0.002307

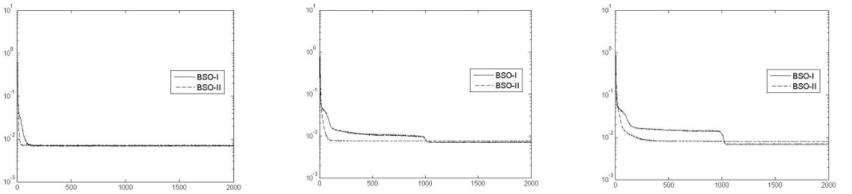
Table 14. Simulation results of D_c and D_e for BSO-II on unimodal functions

F	d	D_c				D_e			
		mean	short	long	variance	mean	small	large	variance
f_1	10	2.35E-20	1.80E-20	3.00E-20	5.86E-42	0.674	0.609	0.696	0.000435
	20	5.3E-20	4.39E-20	6.41E-20	2.37E-41	0.676	0.568	0.696	0.000443
	30	7.5E-20	6.48E-20	8.77E-20	3.51E-41	0.674	0.612	0.695	0.000363
f_2	10	3.16E-20	2.50E-20	4.16E-20	1.36E-41	0.675	0.597	0.697	0.000592
	20	7.4E-20	5.56E-20	8.36E-20	3.47E-41	0.658	0.586	0.694	0.000782
	30	1.06E-19	9.13E-20	1.22E-19	5.64E-41	0.658	0.540	0.696	0.001079
f_3	10	0.007209	0.005138	0.008104	2.81E-07	0.685	0.652	0.698	8.749E-05
	20	0.007577	0.006727	0.008262	1.37E-07	0.685	0.610	0.697	0.000214
	30	0.00797	0.007136	0.008711	1.77E-07	0.677	0.622	0.695	0.000298
f_4	10	2.51E-19	1.77E-19	3.58E-19	1.42E-39	0.667	0.546	0.697	0.000904
	20	5.58E-19	4.59E-19	8.08E-19	5.04E-39	0.658	0.509	0.699	0.001228
	30	8.14E-19	5.89E-19	1.11E-18	9.52E-39	0.662	0.567	0.696	0.001039
f_5	10	0.073914	0.052208	0.104325	0.000129	0.649	0.509	0.695	0.00143
	20	0.066032	0.044808	0.095853	0.000149	0.638	0.515	0.692	0.001513
	30	0.062562	0.041369	0.102545	0.000142	0.624	0.326	0.688	0.003219

Table 15. Simulation results of D_c and D_e for BSO-II on multimodal functions

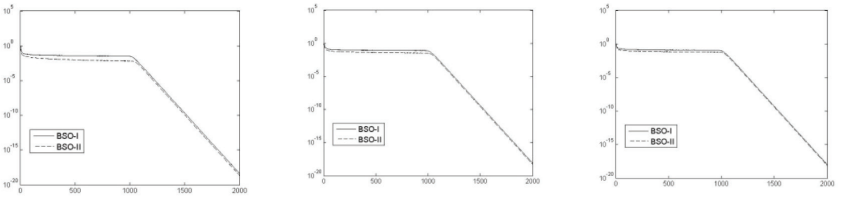
F	d	D_c				D_e			
		mean	short	long	variance	mean	small	large	variance
f_6	10	9.83E-17	1.13E-18	1.25E-16	8.10E-34	0.684	0.616	0.698	0.00025
	20	1.17E-16	7.84E-17	1.47E-16	1.68E-34	0.655	0.570	0.693	0.000735
	30	1.02E-16	5.82E-17	1.38E-16	3.12E-34	0.597	0.419	0.691	0.00392
f_7	10	4.63E-10	3.95E-10	5.28E-10	1.12E-21	0.688	0.665	0.698	5.111E-05
	20	4.9E-10	4.10E-10	5.46E-10	8.55E-22	0.681	0.615	0.698	0.000203
	30	4.85E-10	4.03E-10	5.68E-10	8.28E-22	0.663	0.590	0.698	0.000578
f_8	10	1.93E-13	3.12E-19	9.66E-12	1.87E-24	0.589	0.411	0.687	0.004572
	20	1.52E-16	7.17E-19	6.67E-16	1.98E-32	0.599	0.434	0.683	0.003304
	30	1.38E-16	8.33E-19	6.48E-16	2.28E-32	0.585	0.468	0.668	0.002776
f_9	10	2.13E-09	0	3.22E-09	6.53E-19	0.571	0.097	0.685	0.024857
	20	3.42E-09	0	4.95E-09	1.38E-18	0.595	0.097	0.693	0.024992
	30	5.79E-09	3.21E-09	7.72E-09	9.25E-19	0.634	0.443	0.697	0.003643
f_{10}	10	3.24E-11	1.51E-19	6.07E-11	5.18E-22	0.623	0.421	0.694	0.002855
	20	5.89E-11	1.35E-18	1.04E-10	1.04E-21	0.625	0.493	0.694	0.002619
	30	8.63E-11	2.99E-20	1.24E-10	1.40E-21	0.631	0.499	0.691	0.002384

Figure 21. Mean Average Inter-cluster Distance vs. Iterations of Schwefel's P221



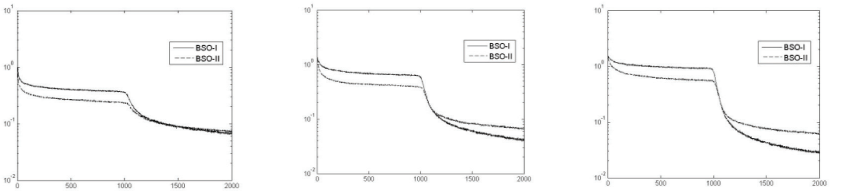
a) 10-D *b) 20-D* *c) 30-D*

Figure 22. Mean Average Inter-cluster Distance vs. Iterations of Step Function



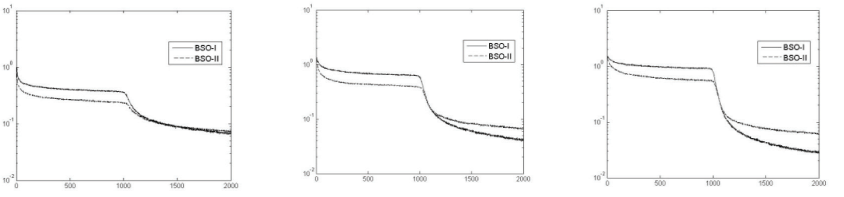
a) 10-D *b) 20-D* *c) 30-D*

Figure 23. Mean Average Inter-cluster Distance vs. Iterations of Schwefel's P222



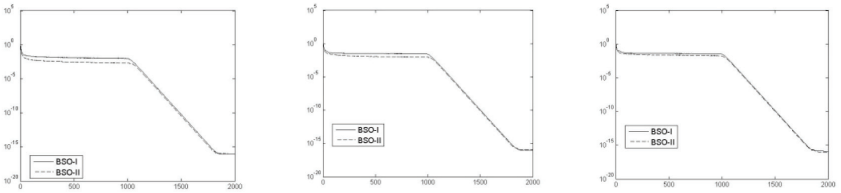
a) 10-D *b) 20-D* *c) 30-D*

Figure 24. Mean Average Inter-cluster Distance vs. Iterations of Quartic Noise



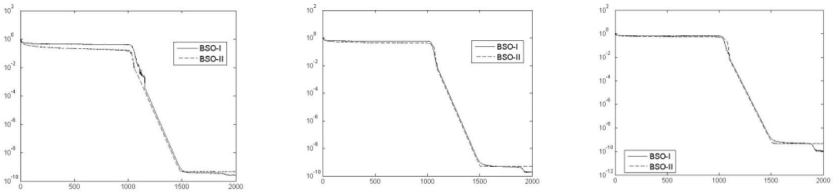
a) 10-D *b) 20-D* *c) 30-D*

Figure 25. Mean Average Inter-cluster Distance vs. Iterations of Ackely Function



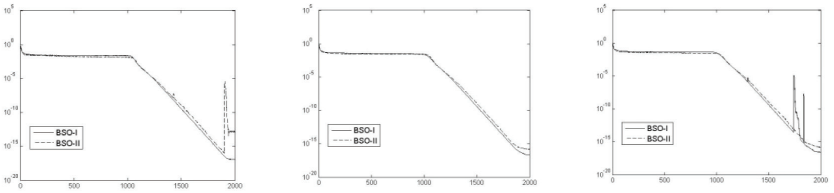
a) 10-D *b) 20-D* *c) 30-D*

Figure 26. Mean Average Inter-cluster Distance vs. Iterations of Rastrigin Function



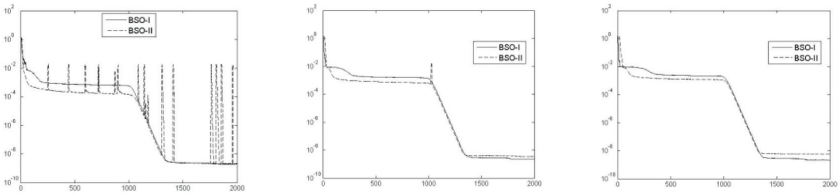
a) 10-D *b) 20-D* *c) 30-D*

Figure 27. Mean Average Inter-cluster Distance vs. Iterations of Rosenbrock Function



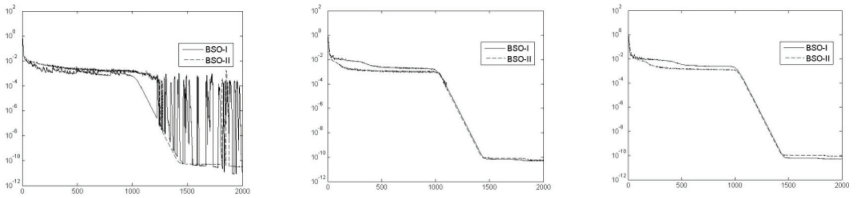
a) 10-D *b) 20-D* *c) 30-D*

Figure 28. Mean Average Inter-cluster Distance vs. Iterations of Schwefel's P226



a) 10-D *b) 20-D* *c) 30-D*

Figure 29. Mean Average Inter-cluster Distance vs. Iterations of Griewank Function



a) 10-D *b) 20-D* *c) 30-D*

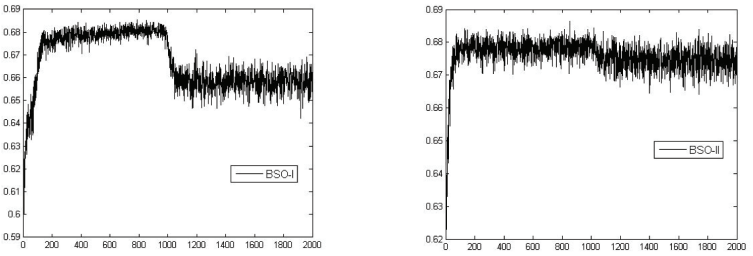
maximum iteration number for both BSO algorithms, which indicates that *m* clusters move close to each other very quickly, and therefore the algorithms may lose their search capabilities quickly, may converge quickly, or may be stuck in (local) optima quickly. By double checking the cluster centers over iterations, the same observation can be obtained. That tells us that when the situation occurs, further improvement could be achieved by randomly move away from current cluster centers and at the same time increase the step-size to a relatively large value which then will be dynamically adjusted according to the formula (3).

The mean inter-cluster diversities of *50* runs over iterations for all benchmark functions seem to have similar behaviors except function f_9 with dimension *10* and *20*. Figures 30 and 31 display the curves of mean inter-cluster diversities over *50* runs vs. iterations for function f_1 as an example for unimodal functions and for function f_7 as an example for multimodal functions. Figure 32 displays the curves of mean inter-cluster diversities over *50* runs vs. iterations for function f_9 with dimension *20*. From Figures 30 and 31, the mean inter-cluster diversities tend to have relatively large values, which indicate that the population of individuals is generally well-uniformly divided into *m* clusters. This may be because the fixed number of clusters and the k-means clustering algorithm with randomly selecting *k* individuals as initial cluster centroid positions are used over iterations in the implementation of the BSO algorithms. If different initialization method for k-means clustering algorithm or a different clustering algorithm especially those with unfixed number of clusters such as the self-organizing feature map is utilized, the mean inter-cluster diversity may behave quite different. From Figure 32 and Table 15, it can be seen that toward the end of BSO-II running for function f_9, the number of individuals in each cluster is not uniformly distributed anymore, but clustered into one cluster with other *4* clusters with only *1* individual, in which the $D_c = -\left(4 * \frac{1}{100} * log_{10}\left(\frac{1}{100}\right) + \frac{96}{100} * log_{10}\left(\frac{96}{100}\right)\right) = 0.097$.

CONCLUSION

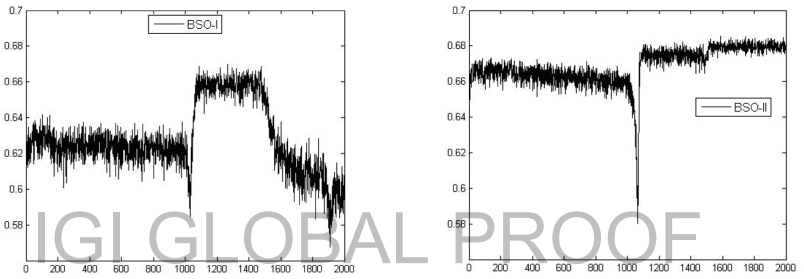
In this paper, we first modeled the human brainstorming process, then introduced two versions of Brain Storm Optimization algorithms It is natural to believe that BSO algorithms should be superior to the optimization algorithms inspired by collective behavior of injects such as ants, birds, *etc*. because the BSO algorithms were inspired by the human brainstorming process. The proposed BSO algorithms were implemented and tested on ten benchmark functions, of which five are unimodal functions and the other five are multimodal functions. Simulation results showed that both BSO algorithms performed reasonably well, and BSO-II performs better than BSO-I does in general. Furthermore, average inter-cluster distance D_c and inter-cluster diversity D_e were defined to measure and monitor the distribution of cluster centroids and the information entropy of the BSO population. Simulation results on D_c

Figure 30. Mean D_e over 50 Runs vs. Iterations for Sphere Function with Dimension d = 20



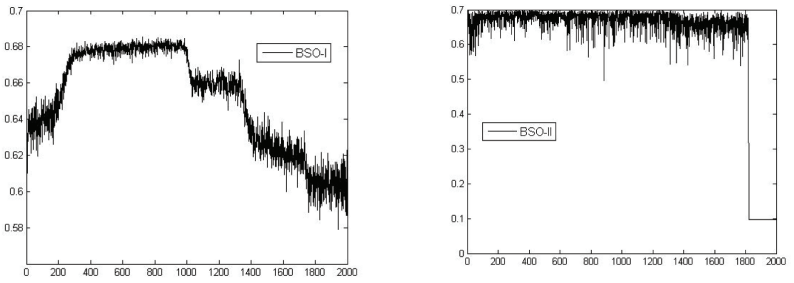
a) BSO-I *b) BSO-II*

Figure 31. Mean D_e over 50 Runs vs. Iterations for Rastrigin Function with Dimension d = 20



a) BSO-I *b) BSO-II*

Figure 32. Mean D_e over 50 Runs vs. Iterations for Schwefel's P226 with Dimension d = 20



a) BSO-I *b) BSO-II*

showed that further performance improvement for the BSO algorithms could be achieved by taking advantage of information revealed by the D_c, which is one of future research directions.

Good optimization algorithms for solving complicated and nonlinear optimization problems should have the capability to converge in order to find better and better solutions, but at the same time, it should have the capability to diverge in order to escape from local optima which are not good enough solutions for the problem to be solved. The BSO algorithm during each iteration involves two opposite operations. One is to converge or contract by utilizing clustering methods to converge to the *m* cluster centers. Another is to diverge or expand by adding noise to generate new individuals. Depending on the amplitude of noise, different scales of areas can be searched by the BSO algorithm. Therefore, the BSO algorithms naturally include contraction and expansion operations during each iteration by design. It should be a good choice for solving complicated and nonlinear optimization problems.

ACKNOWLEDGMENT

This paper is partially supported by National Natural Science Foundation of China under Grant Number 60975080, and by the Suzhou Science and Technology Project under Grant Number SYJG0919.

REFERENCES

Bastos-Filho, C. J. A., De Lima Neto, F. B., Lins, A. J. C. C., Nascimento, A. I. S., & Lima, M. P. (2008). A novel search algorithm based on fish school behavior. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics* (pp. 2646-2651).

de Castro, J. N., & Von Zuben, F. J. (1999). *Artificial immune systems: Part I -Basic theory and applications* (Tech. Rep. No. DCA-RT 01/99). Brazil, Campinas: School of Computing and Electrical Engineering, State University of Campinas.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man . Cybernetics B*, *26*(2), 29–41. doi:10.1109/3477.484436

Eberhart, R. C., & Shi, Y. (2007). *Computational intelligence, concepts to implementation* (1st ed.). San Francisco, CA: Morgan Kaufmann.

Eusuff, M., & Lansey, K. (2006). Shuffled frog-leaping algorithm: A memetic meta-heuristic for discrete optimization. *Engineering Optimization*, *38*(2), 129–154. doi:10.1080/03052150500384759

Fogel, L. J. (1962). Autonomous automata. *Industrial Research*, *4*, 14–19.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.

Kohonen, T., & Honkela, T. (2007). Kohonen network. *Scholarpedia*, *2*(1), 1568. doi:10.4249/scholarpedia.1568

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability* (pp. 281-297).

Nock, R., & Nielsen, F. (2006). On weighting clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *28*(8), 1–13. doi:10.1109/TPAMI.2006.168

Osborn, A. F. (1963). *Applied imagination: Principles and procedures of creative problem solving* (3rd ed.). New York, NY: Charles Scribner's Son.

Passino, K. M. (2010). Bacterial foraging optimization. *International Journal of Swarm Intelligence Research*, *1*(1), 1–16. doi:10.4018/jsir.2010010101

Pavlyukevich, I. (2007). Lévy flights, non-local search and simulated annealing. *Journal of Computational Physics*, *226*, 1830–1844. doi:10.1016/j.jcp.2007.06.008

Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart, Germany: Frommann-Holzboog.

Shah-Hosseini, H. (2009). The intelligent water drops algorithm: a nature-inspired swarm-based optimization algorithm. *International Journal of Bio-inspired Computation*, *1*(1-2), 71–79. doi:10.1504/IJBIC.2009.022775

Shi, Y. (2011, June 11-15). Brain storm optimization algorithm. In Y. Tan, Y. Shi, Y. Chai, & G. Wang (Eds.), *Proceedings of the Second International Conference on Advances in Swarm Intelligence*, Chongqing, China (LNCS 6728, pp. 303-309).

Shi, Y., & Eberhart, R. C. (1998, May 4-9). A modified particle swarm optimizer. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, Anchorage, AK.

Shi, Y., & Eberhart, R. C. (2008). Population diversity of particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation*, Hong Kong, China.

Shi, Y., & Eberhart, R. C. (2009). Monitoring of particle swarm optimization. *Frontiers of Computer Science in China*, *3*(1), 31–37. doi:10.1007/s11704-009-0008-4

Smith, R. (2002). *The 7 levels of change* (2nd ed.). Arlington, VA: Tapeslry Press.

Theodoridis, S., & Koutroumbas, K. (2006). *Pattern recognition* (3rd ed.). New York, NY: Academic Press.

Tovey, C. (2004). The honey bee algorithm: A biological inspired approach to internet server optimization. *Engineering Enterprise, the Alumni Magazine for ISyE at Georgia Institute of Technology*, 13-15.

Xu, R., & Wunsch, D. II. (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, *16*(3), 645–678. doi:10.1109/TNN.2005.845141

Yang, X. (2008). *Nature-inspired metaheuristic algorithms*. Beckington, UK: Luniver Press.

Yao, X., Liu, Y., & Lin, G. (1997). Evolutionary programming made faster. *IEEE Transactions on Evolutionary Computation*, *3*, 82–102.

Yuhui Shi received the PhD degree in electronic engineering from Southeast University, Nanjing, China, in 1992. He is currently a Professor with the Department of Electrical and Electronic Engineering, Xi'an Jiaotong-Liverpool University, Suzhou, China. His current research interests include computational intelligence techniques (including swarm intelligence) and their applications. Dr. Shi is the Editor-in-Chief of the International Journal of Swarm Intelligence Research *and an Associate Editor of the* IEEE Transactions on Evolutionary Computation. *He is the Chair of the IEEE Task Force on Swarm Intelligence.*

International Journal of Swarm Intelligence Research

An official publication of the Information Resources Management Association

Mission

The mission of the *International Journal of Swarm Intelligence Research* (IJSIR) is to become a leading international and well-referred journal in swarm intelligence, nature-inspired optimization algorithms, and their applications. This journal publishes original and previously unpublished articles including research papers, survey papers, and application papers, to serve as a platform for facilitating and enhancing the information shared among researchers in swarm intelligence research areas ranging from algorithm developments to real-world applications.

Subscription Information

IJSIR is published quarterly: January-March; April-June; July-September; October-December by IGI Global. Full subscription information may be found at www.igi-global.com/ijsir. The journal is available in print and electronic formats.

Institutions may also purchase a site license providing access to the full IGI Global journal collection featuring more than 100 topical journals in information/computer science and technology applied to business & public administration, engineering, education, medical & healthcare, and social science. For information visit www.igi-global.com/isj or contact IGI at eresources@igi-global.com.

Copyright

Correspondence and questions:

Editorial: Yuhui Shi
Editor-in-Chief
IJSIR
E-mail: yuhui.shi@xjtlu.edu.cn

Subscriber Info: IGI Global
Customer Service
701 E Chocolate Avenue
Hershey PA 17033-1240, USA
Tel: 717/533-8845 x100
E-mail: cust@igi-global.com

The *International Journal of Swarm Intelligence Research* is currently listed or indexed in: Bacon's Media Directory; DBLP; Google Scholar; MediaFinder; The Standard Periodical Directory; Ulrich's Periodicals Directory