Neural Comput & Applic (2001)10:214–230
Ownership and Copyright
© Springer-Verlag London Limited 2001

Neural
Computing
& Applications

A Robust Evolutionary Algorithm for Training Neural Networks

Jinn-Moon Yang[1] and Cheng-Yan Kao[2]

[1]Department of Biological Science and Technology, National Chiao Tung University, Hsinchu, Taiwan
[2]Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

A new evolutionary algorithm is introduced for training both feedforward and recurrent neural networks. The proposed approach, called the Family Competition Evolutionary Algorithm (FCEA), automatically achieves the balance of the solution quality and convergence speed by integrating multiple mutations, family competition and adaptive rules. We experimentally analyse the proposed approach by showing that its components can cooperate with one another, and possess good local and global properties. Following the description of implementation details, our approach is then applied to several benchmark problems, including an artificial ant problem, parity problems and a two-spiral problem. Experimental results indicate that the new approach is able to stably solve these problems, and is very competitive with the comparative evolutionary algorithms.

Keywords: Adaptive mutations; Evolutionary algorithm; Family competition; Multiple mutations; Neural networks

1. Introduction

Artificial Neural Networks (ANNs) have been applied widely in many application domains. In addition to approximation capabilities for multilayer networks in numerous functions [1], ANNs avoid the bias of a designer in shaping system development owing to their flexibility, robustness and tolerance of noise. Learning the weights of ANNs with fixed architectures can be formulated as a weight training process. This process is to minimise an objective function in order to achieve an optimal set of connection weights of an ANN which can be employed to solve the desired problems.

The standard back propagation learning algorithm [2] and many improved back propagation learning algorithms [3,4] are the widely used approaches. They are the gradient descent techniques which often try to minimise the objective function based on the total error between the actual output and the target output of an ANN. This error is used to guide the search of a back propagation approach in the weight space. However, the drawbacks with a back propagation algorithm do exist due to its gradient descent nature. It may get trapped in a local optima of the objective function, and is inefficient in searching for a global minimum of an objective function which is vast, multimodal and non-differentiable [5]. In addition, the back propagation approach needs to predetermine the learning parameters. Particularly, these applications where gradient methods are not directly applicable. Furthermore, from a theoretical perspective, gradient methods often produce worse recurrent networks than non-gradient methods when an application requires memory retention [6].

An evolutionary algorithm is a non-gradient method, and is a very promising approach for training ANNs. It is considered to be able to reduce the ill effect of the back propagation algorithm, because it does not require gradient and differentiable information. Evolutionary algorithms have been successfully applied to train or evolve ANNs in many application domains [5].

Evolutionary methodologies can be categorised as

Correspondence and offprint requests to: J.-M. Yang, Department of Biological Science and Technology, National Chiao Tung University, Hsinchu, Taiwan. E-mail: moon@csie.ntu.edu.tw

genetic algorithms [7], evolutionary programming [8] and evolution strategies [9]. Applying genetic algorithms to train neural networks may be unsatisfactory because recombination operators incur several problems, such as competing conventions [5] and the epistasis effect [10]. For better performance, real-coded genetic algorithms [11,12] have been introduced. However, they generally employ random-based mutations, and hence still require lengthy local searches near a local optima. In contrast, evolution strategies and evolutionary programming mainly use real-valued representation, and focus on self-adaptive Gaussian mutations. This type of mutation has succeeded in continuous optimisation, and has been widely regarded as a good operator for local searches [8,9]. Unfortunately, experiments [13] show that self-adaptive Gaussian mutation leaves individuals trapped near local optima for rugged functions.

Because none of these three types of original evolutionary algorithms is very efficient, many modifications have been proposed to improve the solution quality, and to speed up convergence. In particular, a popular method [14,15] is to incorporate local search techniques, such as the back propagation approach, into evolutionary algorithms. Such a hybrid approach possesses both the global optimality of the genetic algorithms, and also the convergence of the local searches. In other words, a hybrid approach can usually make a better trade-off between computational cost and the global optimality of the solution. However, for existing hybrid methods, local search techniques and genetic operators often work separately during the search process.

Another technique is to use multiple genetic operators [16,17]. This approach works by assigning a list of parameters to determine the probability of using each operator. Then, an adaptive mechanism is applied to change these probabilities to reflect the performance of the operators. The main disadvantage of this method is that the mechanism for adapting the probabilities may mislead evolutionary algorithms toward local optima.

To further improve the above approaches, in this paper a new method, called the Family Competition Evolutionary Algorithm (FCEA), is proposed for training the weights of neural networks. FCEA is a multi-operator approach which combines three mutation operators: decreasing-based Gaussian mutation; self-adaptive Gaussian mutation; and self-adaptive Cauchy mutation. The performance of these three mutations heavily depends upon the same factor, called *step size*. FCEA incorporates family competition [18] and adaptive rules for controlling *step sizes* to construct the relationship among these three mutation operators. Family competition is derived from $(1 + \lambda)$-ES [9], and acts as a local search procedure. Self-adaptive mutation adapts its step sizes with a stochastic mechanism based on performance. In contrast, decreasing-based mutation decreases the step sizes by applying a fixed rate γ where $\gamma < 1$.

FCEA markedly differs from previous approaches, because these mutation operators are sequentially applied with an equal probability of 1. In addition, each operator is designed to compensate for the disadvantages of the others, to balance the search power of exploration and exploitation. To the best of our knowledge, FCEA is the first successful attempt to integrate self-adaptive mutations and decreasing-based mutations by using family competition and adaptive rules. Our previous work [19] has demonstrated that FCEA is competitive with the back propagation approach, and is more robust than several well-known evolutionary algorithms for regular language recognition. In this paper, we modify the mutations to train both feedforward and recurrent networks for three complex benchmark problems. Experimental results show that FCEA is able to robustly solve these complex problems.

This paper is organised as follows. Section 2 describes the model of artificial neural networks trained by our FCEA. Section 3 describes FCEA in detail, and gives motivations and ideas behind various design choices. Next, Section 4 investigates the main characteristics of FCEA. We demonstrate experimentally how FCEA balances the trade-off between exploitation and exploration of the search. In Section 5, FCEA is applied to train a recurrent network for an artificial ant problem. Sections 6 and 7 present the experimental results of FCEA employed to train feedforward networks for a two-spiral problem and N parity problems, where N ranges from 7 to 10. Conclusions are finally made in Section 8.

2. Artificial Neural Networks

Figure 1 shows three general three-layer neural architectures that are able to arbitrarily approximate functions [1]. Figures 1(a) and (b) depict a fully connected feedforward network and a fully connected feedforward network with shortcuts, respectively. Figure 1(c) shows a fully connected recurrent neural network with shortcuts. This network can be used to save contextual information via internal states, so that it becomes appropriate for tasks which must store and update contextual information. A shortcut

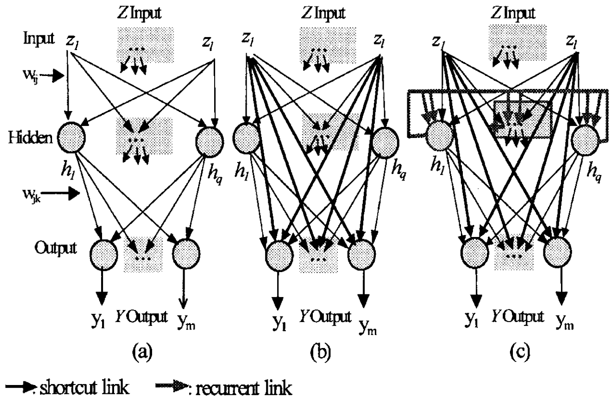


Fig. 1. Three kinds of general three-layer neural networks. (a) A fully connected feedforward network; (b) a fully connected feedforward network with shortcuts; and (c) a fully connected recurrent network with shortcuts.

is a connection link, which is directly connected from an input node to an output node. In this paper, our approach is applied to train the feedforward network shown in Fig. 1(a) for parity problems, the network shown in Fig. 1(b) with three hidden layers for a two-spiral problem, and the recurrent network shown in Fig. 1(c) for an artificial ant problem.

In FCEA, the formulation of training these architectures in Fig. 1 is similar. Thus, only the architecture shown in Fig. 1(a) is considered when formulating the problem of training the weights of an ANN. Z, z_1, …, z_l, and Y, y_1, …, y_m, are the inputs with l elements and the outputs with m nodes, respectively. The output values of the nodes in the hidden layer and in the output layer can be formulated as

$$h_j = f\left(\sum_{i=1}^{l} w_{ij} z_i\right), \quad 1 \le j \le q \tag{1}$$

and

$$y_k = f\left(\sum_{j=1}^{q} w_{jk} h_j\right), \quad 1 \le k \le m \tag{2}$$

respectively, where f is the following sigmoid function:

$$f(\eta) = (1 + e^{-\eta})^{-1} \tag{3}$$

w_{ij} denotes the weights between the input nodes and hidden nodes, w_{jk} denotes the weights between the hidden nodes and output nodes, and q is the number of hidden nodes.

Our approach is to learn the weights of ANNs based on evolutionary algorithms. In FCEA, we optimise the weights (e.g. w_{ij} and w_{jk} in Fig. 1(a)) to minimise the mean square error over a validation set containing T patterns:

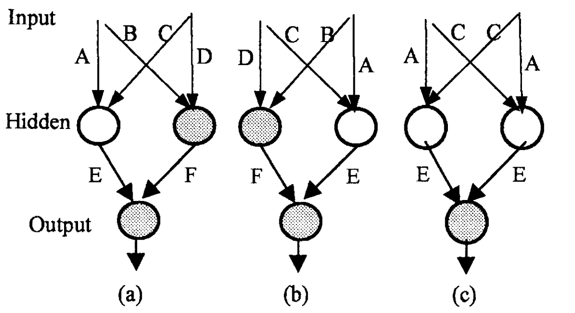


Fig. 2. The competing conventions problem. The parents, (a) and (b), perform the same function and exhibit the same fitness value. Recombination creates an offspring (c) with two hidden neurons that perform nearly the same function.

$$F = \frac{1}{Tm} \sum_{t=1}^{T} \sum_{k=1}^{m} (Y_k(I_t) - O_k(I_t))^2 \tag{4}$$

where m is the number of output nodes, and $Y_k(I_t)$ and $O_k(I_t)$ are the actual and desired outputs of the output node k for the input pattern I_t. The formulation is used as the fitness function of each individual (an ANN) in FCEA, except for the artificial ant problem.

Applying recombination operators to train neural networks creates a particular problem called competing conventions [5]. Under these circumstances, the objective function may become a many-to-one function, because different networks may perform the same function and exhibit the same fitness value, as illustrated by two examples shown in Figs 2(a) and (b). Given two such networks, recombination creates an offspring with two hidden neurons that perform nearly the same function. The performance of the offspring (Fig. 2(c)) is worse than its parents (i.e. the networks shown in Figs 2(a) and (b)), because it is unable to perform the function of the other hidden neuron. The number of competing conventions grows exponentially with the number of hidden neurons [5].

3. Family Competition Evolutionary Algorithm

In this section, we present details of the Family Competition Evolutionary Algorithm (FCEA) for training both feedforward and recurrent neural networks. The basic structure of the FCEA is as follows (Fig. 3): N individuals (ANNs) are generated as the initial population. Then FCEA enters the main evolutionary loop, consisting of three stages in every iteration: decreasing-based Gaussian mutation; self-adaptive Cauchy mutation; and self-adaptive Gaussian mutation. Each stage is realised by generating

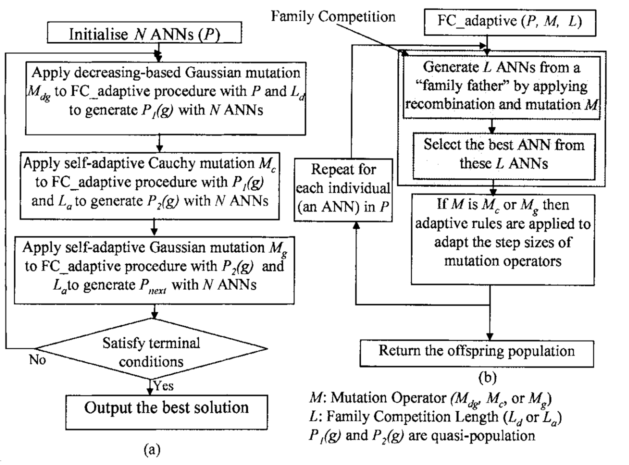

Initialise N ANNs (P)
Apply decreasing-based Gaussian mutation M_{dg} to FC_adaptive procedure with P and L_d to generate $P_1(g)$ with N ANNs
Apply self-adaptive Cauchy mutation M_c to FC_adaptive procedure with $P_1(g)$ and L_a to generate $P_2(g)$ with N ANNs
Apply self-adaptive Gaussian mutation M_g to FC_adaptive procedure with $P_2(g)$ and L_a to generate P_{next} with N ANNs
Satisfy terminal conditions
No
Yes
Output the best solution
(a)
Family Competition
FC_adaptive (P, M, L)
Generate L ANNs from a "family father" by applying recombination and mutation M
Select the best ANN from these L ANNs
Repeat for each individual (an ANN) in P
If M is M_c or M_g then adaptive rules are applied to adapt the step sizes of mutation operators
Return the offspring population
(b)
M: Mutation Operator (M_{dg}, M_c, or M_g)
L: Family Competition Length (L_d or L_a)
$P_1(g)$ and $P_2(g)$ are quasi-population

Fig. 3. Overview of our algorithm: (a) FCEA (b) FC_adaptive procedure.

a new quasi-population (with N ANNs) as the parent of the next stage. As shown in Fig. 3, these stages differ only in the mutations used and in some parameters. Hence, we use a general procedure, 'FC_adaptive', to represent the work done by these stages.

The FC_adaptive procedure employs three parameters (the parent population (P, with N solutions), mutation operator (M) and family competition length (L)) to generate a new quasi-population. The main work of FC_adaptive is to produce offspring, and then conduct the family competition (Fig. 4). Each individual in the population sequentially becomes the 'family father'. Here we use I^1 as the 'family father' to describe the procedure of family competition. With a probability p_c, this family father and another ANN (I^1_{m1}) randomly chosen from the rest of the parent population are used as parents to do a recombination operation. Then the new offspring or the family father (if the recombination is not

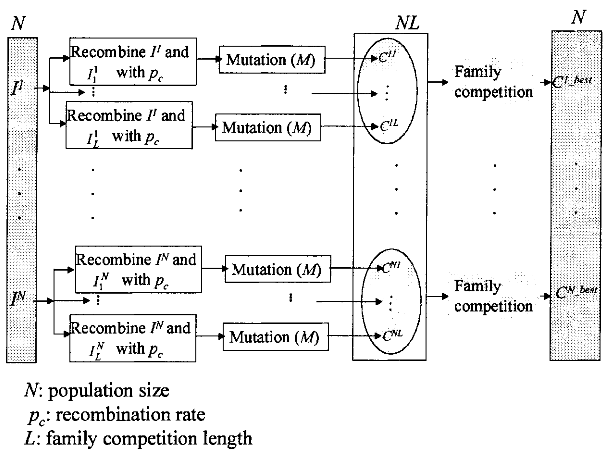

N
NL
N
I^1
Recombine I^1 and I^1_1 with p_c
Mutation (M)
C^{11}
Recombine I^1 and I^1_L with p_c
Mutation (M)
C^{1L}
Family competition
C^{1_best}
I^N
Recombine I^N and I^N_1 with p_c
Mutation (M)
C^{N1}
Recombine I^N and I^N_L with p_c
Mutation (M)
C^{NL}
Family competition
C^{N_best}
N: population size
p_c: recombination rate
L: family competition length

Fig. 4. The main steps of the family competition.

conducted) is operated on by the mutation to generate an offspring (C^{11}). For each family father, such a procedure is repeated L times. Finally, L ANNs (C^{11}, ..., C^{1L}) are produced, but only the one ($C^{1\text{-}best}$) with the lowest objective value survives. Since we create L ANNs from one 'family father' and perform a selection, this is a family competition strategy. We think this is a way not only to avoid the premature, but also to keep the spirit of local searches.

After the family competition, there are N parents and N children left. Based on different stages, we employ various ways of obtaining a new quasi-population with N individuals (ANNs). For both Gaussian and Cauchy self-adaptive mutations, in each of the pairs of father and child, the individual with a better objective value survives. This is the so-called 'family selection'. On the other hand, 'population selection' chooses the best N individuals from all N parents and N children. With a probability P_{ps}, FCEA applies population selection to speed up the convergence when the decreasing-based Gaussian mutation is used. For the probability (1-P_{ps}), family selection is still considered. To reduce the ill effects of greediness on this selection, the initial P_{ps} is set to 0.05, but it is changed to 0.5 when the mean step size of self-adaptive Gaussian mutation is larger than that of decreasing-based Gaussian mutation. Note that the population selection is similar to $(\mu+\mu)$-ES in the traditional evolution strategies. Hence, through the process of selection, the FC_adaptive procedure forces each solution of the starting population to have one final offspring. Note that we create LN offspring in the FC_adaptive procedure but the size of the new quasi-population remains the same as N.

For all three mutation operators, we assign different parameters to control performance. Such parameters must be adjusted through the evolutionary process. We modify them first when mutations are applied. Then, after the family competition is complete, parameters are adapted again to better reflect the performance of the whole FC_adaptive procedure.

In the rest of this section, we explain the chromosome representation and each important component of the FC_adaptive procedure: recombination operators, mutation operations, and rules for adapting step sizes (σ, v and ψ).

3.1. Chromosome Representation and Initialisation

Regarding chromosome representation, we present each solution of a population as a set of four n-

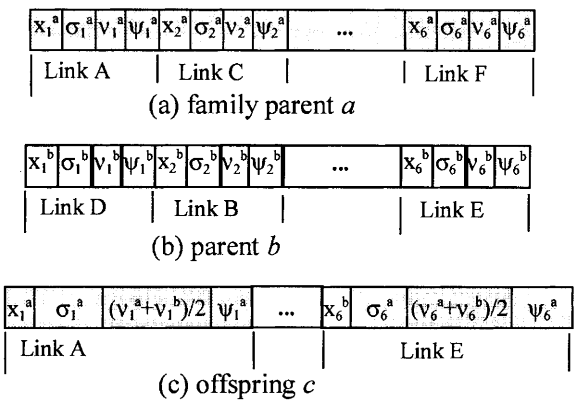

x_1^a σ_1^a v_1^a ψ_1^a x_2^a σ_2^a v_2^a ψ_2^a ... x_6^a σ_6^a v_6^a ψ_6^a
Link A
Link C
Link F
(a) family parent a
x_1^b σ_1^b v_1^b ψ_1^b x_2^b σ_2^b v_2^b ψ_2^b ... x_6^b σ_6^b v_6^b ψ_6^b
Link D
Link B
Link E
(b) parent b
x_1^a σ_1^a $(v_1^a+v_1^b)/2$ ψ_1^a ... x_6^b σ_6^a $(v_6^a+v_6^b)/2$ ψ_6^a
Link A
Link E
(c) offspring c

Fig. 5. Chromosome representation and recombination operators. (a) and (b) represent the networks shown in Figs 2(a) and (b), respectively; (c) is an offspring generated from (a) and (b) in the self-adaptive Gaussian mutation stage.

dimensional vectors (x, σ, v, ψ), where n is the number of weights of an ANN. The vector x is the weights to be optimised; σ, v and ψ are the step-sise vectors of decreasing-based mutations, self-adaptive Gaussian mutation, and self-adaptive Cauchy mutation, respectively. In other words, each solution x is associated with some parameters for step-sise control. Figure 5 depicts three chromosome representations which represent the respective networks shown in Fig. 2. In these cases, n is 6. Here, the initial value of each entry of x is randomly chosen over $[-0.1, 0.1]$, and the initial values of each entries of the vectors σ, v and ψ are set to be 0.4, 0.1 and 0.1, respectively. For easy description of the operators, we use $a = (x^a, \sigma^a, v^a, \psi^a)$ to represent the 'family father' and $b = (x^b, \sigma^b, v^b, \psi^b)$ as another parent (for the recombination operator only). The offspring of each operation is represented as $c = (x^c, \sigma^c, v^c, \psi^c)$. We also use the symbol x_j^d to denote the jth component of an individual d, $\forall j \in \{1, \ldots, n\}$.

3.2. Recombination Operators

FCEA implements two recombination operators to generate offspring: modified discrete recombination; and intermediate recombination [9]. With probabilities 0.9 and 0.1, at each stage only one of the two operators is chosen. Probabilities are set to obtain good performance according to our experimental experience. Here we again mention that recombination operators are activated with only a probability p_c. The optimising connection weights (x) and a step size $(\sigma, v$ or $\psi)$ are recombined in a recombination operator.

Modified discrete recombination: the original discrete recombination [9] generates a child that inherits genes from two parents with equal probability. Here the two parents of the recombination operator are the 'family father' and another solution randomly selected. Our experience indicates that FCEA can be more robust if the child inherits genes from the 'family father' with a higher probability. Therefore, we modified the operator to be as follows:

$$x_j^c = \begin{cases} x_j^a & \text{with probability } 0.8 \\ x_j^b & \text{with probability } 0.2 \end{cases} \tag{5}$$

For a 'family father', applying this operator in the family competition is viewed as a local search procedure, because this operator is designed to preserve the relationship between a child and its 'family father'.

Intermediate recombination: we define intermediate recombination as:

$$x_j^c = x_j^a + 0.5\,(x_j^b - x_j^a) \quad \text{and} \tag{6}$$

$$w_j^c = w_j^a + 0.5\,(w_j^b - w_j^a) \tag{7}$$

where w is v, σ, or ψ based on the mutation operator applied in the family competition. For example, if self-adaptive Gaussian mutation is used in this FC_adaptive procedure, x in Eqs (6) and (7) is v. We follow the work of the evolution strategies community [20] to employ only intermediate recombination on step-size vectors, that is, σ, v and ψ. To be more precise, x is also recombined when the intermediate recombination is chosen.

Figure 5 shows a recombination example in the self-adaptive Gaussian mutation stage. The offspring (c) is generated from the 'family father' (a) and another parent (b) by applying the modified discrete recombination for the connection weights x and the intermediate recombination for the step size v. In other words, the σ and ψ are unchanged in the self-adaptive Gaussian mutation stage.

3.3. Mutation Operators

Mutations are the main operators of the FCEA. After recombination, a mutation operator is applied to the 'family father' or the new offspring generated by a recombination. In FCEA, the mutation is performed independently on each vector element of the selected individual by adding a random value with expectation zero:

$$x_i' = x_i + wD(\cdot) \tag{8}$$

where x_i is the ith connection weight of $_x$, x_i' is the

ith variable of x' mutated from x, $D(\cdot)$ is a random variable, and w is the step size. In this paper, $D(\cdot)$ is evaluated as $N(0, 1)$ or $C(1)$ if the mutations are, respectively, Gaussian mutation or Cauchy mutation.

Self-adaptive Gaussian mutation: we adapted Schwefel's [21] proposal to use self-adaptive Gaussian mutation in training ANNs. The mutation is accomplished by first mutating the step size v_j and then the connection weight x_j:

$$v_j^c = v_j^a \exp[\tau' N(0, 1) + \tau N_j(0, 1)] \tag{9}$$

$$x_j^c = x_j^a + v_j^c N_j(0, 1) \tag{10}$$

where $N(0, 1)$ is the standard normal distribution. $N_j(0, 1)$ is a new value with distribution $N(0, 1)$ that must be regenerated for each index j. For FCEA, we follow Bäck and Schwefel [20] in setting τ and τ' as $(\sqrt{2n})^{-1}$ and $(\sqrt{2\sqrt{n}})^{-1}$, respectively.

Self-adaptive Cauchy mutation: a random variable is said to have the Cauchy distribution ($\sim C(1)$) if it has the following density function:

$$f(x; t) = \frac{t/\pi}{t^2 + x^2}, \quad -\infty < x < \infty \tag{11}$$

We define self-adaptive Cauchy mutation as follows:

$$\psi_j^c = \psi_j^a \exp[\tau' N(0, 1) + \tau N_j(0, 1)] \tag{12}$$

$$x_j^c = x_j^a + \psi_j^c(t) \tag{13}$$

In our experiments, t is 1. Note that self-adaptive Cauchy mutation is similar to self-adaptive Gaussian mutation, except that Eq. (10) is replaced by Eq. (13). That is, they implement the same step-sise control, but use different means of updating x.

Figure 6 compares the density functions of Gaussian distribution ($N(0, 1)$) and Cauchy distributions ($C(1)$). Clearly, Cauchy mutation is able to make a larger perturbation than Gaussian mutation. This implies that Cauchy mutation has a higher probability of escaping from local optima than does Gaussian mutation. However, the order of local convergence is identical for Gaussian and spherical Cauchy distributions, while non-spherical Cauchy mutations lead to slower local convergence [22].

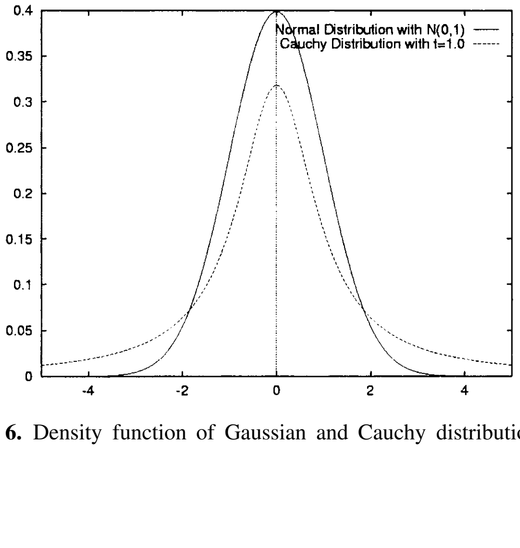


Fig. 6. Density function of Gaussian and Cauchy distributions.

Decreasing-based Gaussian mutations: our decreasing-based Gaussian mutation uses the step-sise vector σ with a fixed decreasing rate $\gamma = 0.97$ as follows:

$$\sigma^c = \gamma \sigma^a \tag{14}$$

$$x_j^c = x_j^a + \sigma^c N_j(0, 1) \tag{15}$$

Previous results [13] have demonstrated that self-adaptive mutations converge faster than decreasing-based mutations, but for rugged functions, self-adaptive mutations are more easily trapped into local optima than decreasing-based mutations.

It can be seen that step sizes are the same for all components of x^a in the decreasing-based mutation, but are different in the self-adaptive mutations. This means two types of mutations have different search behaviour. For decreasing-based mutation, it is like we search for a better child in a hypersphere centred at the parent. However, for self-adaptive mutation, the search space becomes a hyperellipse. Figure 7 illustrates this difference using two-dimensional contour plots.

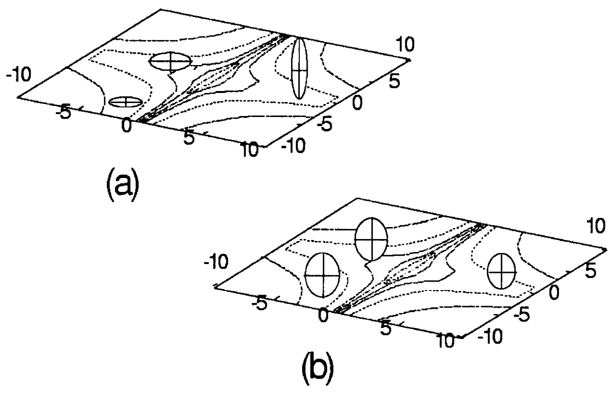


Fig. 7. (a) and (b) show the difference of search space between self-adaptive and decreasing-based mutations. Using two-dimensional contour plots, the search space from parents are ellipses and circles.

3.4. Adaptive Rules

The performance of Gaussian and Cauchy mutations is largely influenced by the step sizes. FCEA adjusts the step sizes while mutations are applied (e.g. Eqs (9), (12) and (14)). However, such updates insufficiently consider the performance of the whole family. Therefore, after the family competition, some additional rules are implemented:

1. *A-decrease-rule:* immediately after self-adaptive mutations, if the objective values of all offspring are greater than or equal to that of the 'family father', we decrease the step-sise vectors v (Gaussian) or ψ (Cauchy) of the parent:

$$w_j^a = 0.97 w_j^a \tag{16}$$

   where w^a is the step size vector of the parent. In other words, if there is no improvement after self-adaptive mutations, we may propose a more conservative implementation. That is, a smaller step size tends to result in a better improvement in the next iteration. This follows the 1/5-success rule of $(1+\lambda)$-ES [9].
2. *D-increase-rule:* it is difficult, however, to decide the rate γ of decreasing-based mutations. Unlike self-adaptive mutations, which adjust step sizes automatically, its step size goes to zero as the number of iterations increases. Therefore, it is essential to employ a rule which can enlarge the step size in some situations. The step size of the decreasing-based mutation should not be too small when compared to the step sizes of self-adaptive mutations. Here, we propose to increase σ if either of the two self-adaptive mutations generates better offspring. To be more precise, after a self-adaptive mutation, if the best child with a step size v is better than its 'family father', the step size of the decreasing-based mutation is updated as follows:

$$\sigma_j^c = \max(\psi_j^c, \beta v_{mean}^c) \tag{17}$$

   where v_{mean}^c is the mean value of the vector v, and β is 0.2 in our experiments. Note that this rule is applied in stages of self-adaptive mutations, but not of decreasing-based mutations.

From the above discussions, the main procedure of FCEA is implemented as follows:

```
1. Set the initial step sizes ($\sigma$, $v$ and $\psi$), family competition lengths ($L_d$ and $L_a$), and crossover rate ($p_c$). Let $g = 1$.
2. Randomly generate an initial population, $P(g)$, with $N$ networks. Each network is represented as $(x^i, \sigma^i, v^i, \psi^i)$, $\forall i \in \{1, 2, \ldots, N\}$.
3. Evaluate the fitness of each network in the population $P(g)$.
4. repeat
   4.1 {Decreasing-based Gaussian mutation ($M_{dg}$)}
       • Generate a children set, $C_T$, with $N$ networks by calling FC_Adaptive with parameters: $P(g)$, $M_{dg}$ and $L_d$. That is, $P_1(g)$ = FC_Adaptive($P(g)$, $M_{dg}$, $L_d$).
       • Select the best $N$ networks as a new quasi-population, $P_1(g)$, by population selection from the union set {$P(g) \cup C_T$ with probability $P_{ps}$ or by family selection with $1 - P_{ps}$.
   4.2 {Self-adaptive Cauchy mutation ($M_c$)}: Generate a new children set, $C_T$, by calling FC_Adaptive with parameters: $P_1(g)$, $M_c$ and $L_a$. Apply family selection to select a new quasi-population, $P_2(g)$, with $N$ networks from the union set {$P_1(g) \cup C_T$.
   4.3 {Self-adaptive Gaussian mutation ($M_g$)}: Generate a new children set, $C_T$, by calling FC_Adaptive with parameters: $P_2(g)$, $M_g$ and $L_a$. Apply family selection to select a new quasi-population, $P_{next}$, with $N$ networks from the union set $P_2(g) \cup C_T$.
   4.4 Let $g = g + 1$ and $P(g) = P_{next}$.
until (termination criterion is met).
Output the best solution and the objective function value.
```

On the other hand, the FC_adaptive proceeds along the following steps:

```
{Parameters: $P$ is the working population, $M$ is the applied mutation ($M_{dg}$, $M_g$ or $M_c$), and $L$ denotes the family competition length ($L_a$ or $L_d$).}
1. Let $C$ be an empty set ($C = \theta$).
2. for each network $a$, called family father, in the population with $N$ networks
   2.1 for $l = 1$ to $L$ {Family Competition}
       • Generate an offspring $c$ by using a recombination ($c$ = recombination ($a$,$b$)) with probability $p_c$ or by copying the family father $a$ to $c$ ($c = a$) with probability $1 - p_c$.
       • Generate an offspring $c_l$ by mutating $c$ as follows:
         $c_l = \begin{cases} M_{dg}(c) & \text{if } M \text{ is } M_{dg}; \quad \{\textit{decreasing-based Gaussian mutation}\} \\ Mg(c) & \text{if } M \text{ is } M_g; \quad \{\textit{self-adaptive Gaussian mutation}\} \\ M_c(c) & \text{if } M \text{ is } M_c; \quad \{\textit{self-adaptive Cauchy mutation}\} \end{cases}$
   endfor
   2.2 Select the one ($c^{best}$) with the lowest objective value from $c_1, \ldots, c_L$. {family selection}
   2.3 Apply adaptive rules if $M$ is a self-adaptive mutation operator ($M_c$ or $M_g$)
   • Apply A-decrease-rule to decrease the step sizes ($\psi$ or $v$) of $M_c$ or $M_g$ if the objective value of the family father $a$ is lower than $c^{best}$. That is, $w_j^a = 0.97 w_j^a$.
   • Apply D-increase-rule to increase the step size ($\sigma$) of $M_{dg}$ if the objective value of the family
```

```
    father a is larger than $c^{best}$. That is, $\sigma_j^{best} = \max(\sigma_j^{best}, \beta v_{mean}^{best})$.
  2.4 Add the $c^{best}$ into the set $C$.
endfor
Return the set $C$ with $N$ networks.
```

4. A Study of some Characteristics of FCEA

In this section, we discuss several characteristics of FCEA by numerical experiments and mathematical explanations. First, we experimentally discuss the effectiveness of using the family competition and multiple mutations. Next, we explore the importance of controlling step sizes and of employing adaptive rules, i.e. A-decrease and D-increase rules.

To analyse FCEA, we study a 2-bit adder [23] and a parity problem by using feedforward networks, which are similar to Fig. 1(a). A 4–4–3 network is employed to solve the 2-bit adder problem. The fully connected ANN has 35 connection weights and 16 input patterns. The output pattern is the result of the sum of the two 2-bits input strings. In addition, a 4–4-1 network is used for solving the parity problem. The fully connected ANN has 25 connection weights and 16 input patterns. The output value is 1 if there is an odd number of 1s in the input patterns.

Evolution begins by initialising all the connection weights x of each network to random values between -0.1 and 0.1. The initial values of step sizes for decreasing-based Gaussian mutations, self-adaptive Gaussian mutation and self-adaptive Cauchy mutation are set to 0.4, 0.1 and 0.1, respectively. Table 1 indicates the settings of FCEA parameters, such as the family competition length and the crossover rate (p_c). L_d and σ are the parameters for the decreasing-based mutation; L_a, v and ψ are for self-adaptive mutations. The same parameter settings are used for all testing problems studied in this work.

In this paper, S_r denotes the percentage of an approach classifying all training data correctly, and *FE* denotes the number of average function evaluations of an approach satisfying the terminal conditions. That is, an approach correctly classifies all training data or exhausts the number of maximum function evaluations.

4.1. The Effectiveness of the Family Competition

Because the family competition length is the critical factor in FCEA, we investigate the influence of L_a and L_d. FCEA is implemented on the parity and 2-bit adder problems on various family lengths, i.e. $L_a = L_d = L$, where L ranges from 1 to 9. In other words, the sums of the total length (i.e. $L_d + 2L_a$) are from 3 to 27 in one generation. FCEA executes each problem over 50 runs on each length. The number of maximum function evaluations is 700,000. The S_r and *FE* of FCEA are calculated based on 50 independent runs. Figure 8(a) shows that the performance of FCEA is unstable when the total length is below 9 ($L_d + 2L_a \leq 9$), and FCEA has the worst performance when both L_a and L_d are set to 1. The performance of FCEA becomes stable while the total length exceeds 15.

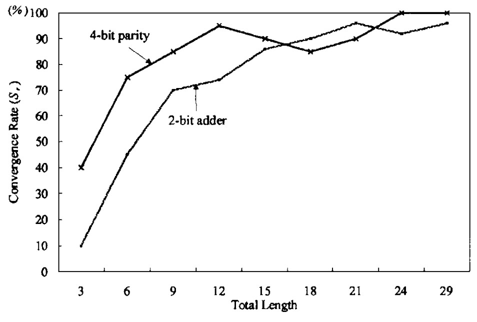
Figures 8(b) and 8(a) show that the *FE* of FCEA is decreasing and the S_r is increasing when the family competition length (L) is increased from 1 to 4. On the other hand, the *FE* of FCEA is increasing, but the S_r is stable while both L_a and L_d exceed 8. From the above observations, we know that the family competition length is important for our FCEA. Therefore, we set L_a to 9 and L_d to 6 for all testing problems studied in this paper.

4.2. The Effectiveness of Multiple Operators

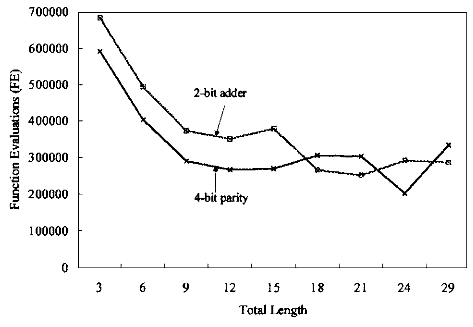
Using multiple mutations in each iteration is one of the main features of FCEA. With family selection, FCEA uses a high selection pressure along with a diversity-preserving mechanism. With a high selection pressure, it become necessary to use highly disruptive search operators, such as the series of three mutation operators used in FCEA. Using

Table 1. Parameter settings and notations.

	Step Size			Competition Length		Recombination Rate	
Decreasing-based Gaussian mutation	M_{dg}	σ	0.4	L_d	6	p_c	0.2
Self-adaptive Gaussian mutation	M_g	v	0.1	L_a	9		
Self-adaptive Cauchy mutation	M_c	ψ	0.1	L_a	9		



(a) The convergence rates (S_r)



(b) The numbers of average function evaluations (FE)

Fig. 8. The percentages (S_r) of FCEA classifying all training data correctly and the numbers (FE) of average function evaluations of FCEA on different family competition lengths. Each problem is tested over 50 runs for each length.

numerical experiments, we will demonstrate that the three operators cooperate with one another, and possess good local and global properties.

We compare eight different uses of mutation operators in Table 2. Each use combines some of the three operators applied in FCEA: decreasing-based Gaussian mutation (M_{dg}); self-adaptive Cauchy mutation (M_c); and self-adaptive Gaussian mutation (M_g). For example, the M_c approach uses only self-adaptive Cauchy mutation; the $M_{dg} + M_c$ approach integrates decreasing-based Gaussian mutation and self-adaptive Cauchy mutation; and FCEA is an approach integrating M_{dg}, M_c and M_g. The FCEA$_{ncr}$ approach is a special case of FCEA without adaptive rules, i.e. without the A-decrease-rule (Eq. (16)) and D-increase-rule (Eq.(17)). Except for FCEA$_{ncr}$, the others employ adaptive rules. To have a fair comparison, we set the family competition length (L) of all eight approaches to the same value. For example, if $L_d = 6$ and $L_a = 9$ in FCEA, $L = 24$ for one-operator approaches (M_{dg}, M_c and M_g) and $L_d = L_a = 12$ for two-operator approaches ($M_{dg} + M_c$, $M_{dg} + M_g$ and $M_c + M_g$).

There are some observations from experimental results:

1. Generally, strategies with a suitable combination of multiple mutations (FCEA and $M_{dg} + M_g$) perform better than unary-operator strategies, in terms of the solution quality. However, the number of function evaluations does not increase much when using multi-operator approaches. Sometimes, the number even decreases (e.g. FCEA verses M_{dg}). Overall, FCEA has the best performance and the number of function evaluations is very competitive.
2. The adaptive rules applied to control step sizes are useful according to the comparison of FCEA$_{ncr}$ and FCEA. We made similar observations when FCEA was applied in global optimisation [13].
3. Each mutation operator (M_{dg}, M_c and M_g) has a different performance. Table 2 shows that self-adaptive mutations (M_c and M_g) outperform the decreasing-based mutation (M_{dg}) on training neural networks.
4. The approaches combining decreasing-based mutation with self-adaptive mutation ($M_{dg} + M_c$ or $M_{dg} + M_g$) perform better than that combining two self-adaptive mutations (M_c+M_g). These results can be analysed as follows: Figure 7 indicates that the distribution of the one-step perturbation of M_{dg} is different from that of M_c

Table 2. Comparison of various approaches of FCEA on the 2-bit adder problem and the parity problem. M_{dg}, M_c and M_g represent different mutations used in FCEA.

Problem		FCEA	FCEA$_{ncr}$	M_{dg}	M_c	M_g	M_{dg}+M_c	M_{dg}+M_g	M_c+M_g
2-bits adder	S_r[a]	96%	82%	0%	0%	40%	44%	72%	26%
	FE[b]	258981	347905	700000	700000	581115	539842	330871	949585
parity	S_r	100%	96%	0%	70%	60%	94%	90%	80%
	FE	112528	147905	700000	386558	507382	210999	269841	332265

[a]S_r denotes the percentage of an approach classifying all training data correctly.
[b]FE denotes the number of average function evaluations.

or M_g. The former approach ($M_{dg} + M_c$ or $M_{dg} + M_g$) applies decreasing-based mutation with large initial step sizes as a global search strategy and the self-adaptive mutations with the family competition procedure and replacement selection as local search strategies. Therefore, we suggest that a global optimisation method should consist of both global and local search strategies.

4.3. Controlling Step Sizes

Numerical experiments in the previous subsection have not fully shown the importance of controlling the step sizes. Here we would like to further discuss this issue by analysing the mean step sizes and the mean expected improvement in the whole iterative process. First, we denote $A_T(O)$ as the mean value of the step size of an operator O at the Tth generation:

$$A_T(O) \equiv \left(\sum_{k=1}^{N}\left(\sum_{i=0}^{n} w_{ki}\right)/n\right)/N \tag{18}$$

where N is the population size, n represents the number of weights, and w_{ki} is the step size of the ith component of the kth individual in the population. Thus, w is σ for M_{dg} and is ψ for M_c. Secondly, we define $E_T(O)$, the expected improvement of all offspring, by an operator O at the Tth generation:

$$E_T(O) \equiv \left(\sum_{k=1}^{N}\left(\sum_{l=1}^{L} \max(0, f(a_k) - f(c_{kl})\right))/L\right)/N \tag{19}$$

where L is the family competition length and c_{kl} is the lth child generated by the kth 'family father' a_k.

Figures 9–14 show the curves of $A_T(O)$ and $E_T(O)$ on the 2-bit adder problem because FCEA, FCEA$_{ncr}$, M_{dg} and M_g have a greatly different performance on it. Figures 9 and 10 illustrate the behaviour of each mutation in FCEA, while Figs 11–14 present $A_T(O)$ and $E_T(O)$ of M_{dg} and M_g in three different evolutionary processes: M_{dg} or M_g itself, FCEA, and FCEA$_{ncr}$. We do not report M_c in Figs 11–14

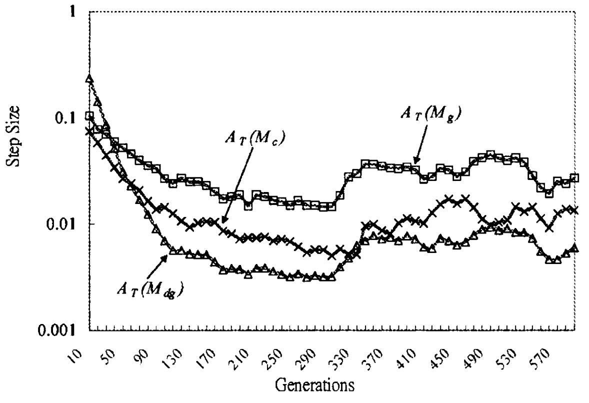


Fig. 9. The average step sizes of various mutations in FCEA.

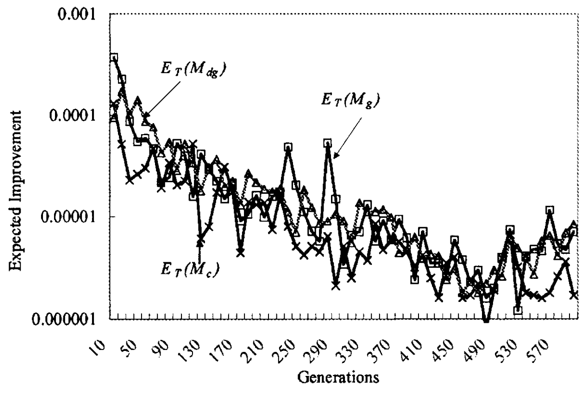


Fig. 10. The average expected improvements of various mutations in FCEA.

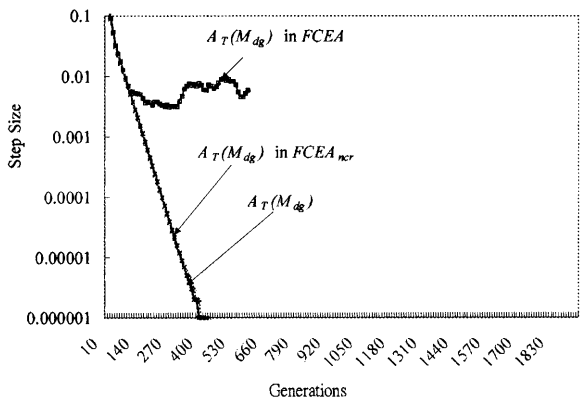


Fig. 11. The average step sizes of decreasing-based Gaussian of FCEA, FCEA$_{ncr}$ and M_{dg}.

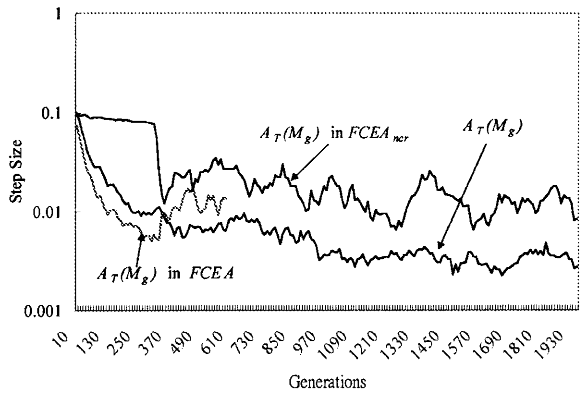


Fig. 12. The average step sizes of self-adaptive Gaussian mutation of FCEA, FCEA$_{ncr}$ and M_g.

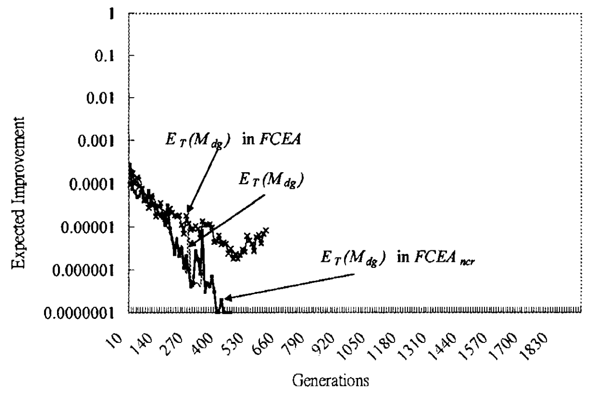


Fig. 13. The average expected improvement of the decreasing-based Gaussian mutation of FCEA, FCEA$_{ncr}$ and M_{dg}.

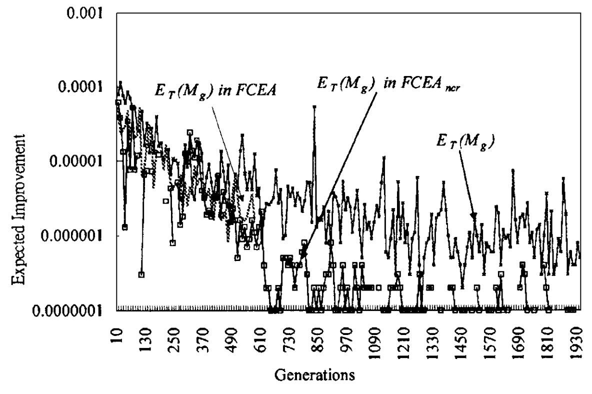


Fig. 14. The average expected improvement of self-adaptive Gaussian mutation of FCEA, FCEA$_{ncr}$ and M_g.

because its behaviour is similar to M_g. Some interesting observations are given below.

1. The control of step sizes is important because the performance of mutations used in FCEA heavily depend upon step sizes based on Figs 9–14. For example, Fig. 11 shows that the step sizes of decreasing-based Gaussian mutations in FCEA$_{ncr}$ and in M_{dg} approach zero while the number of the generations exceeds 450. Figure 13 shows that their respective expected improvement also approaches zero.
2. Self-adaptive mechanisms and family competition are applied to adjust step sizes and to retain children with better function values. From Fig. 9, it can be seen that, in FCEA, the step size of M_c is always smaller than that of M_g. Note that $C_j(1)$ in Eq. (13) tends to be larger than $N_j(0, 1)$ in Eq. (10). However, it seems that a small perturbation from the parent decreases the function value more easily than a large one. For example, Fig. 12 indicates that $A_T(M_g)$ in FCEA$_{ncr}$ is larger than that in M_g. Figure 14 shows that the respective $E_T(M_g)$ in FCEA$_{ncr}$ is smaller than that in M_g. Hence, a self-adaptive mechanism and the family competition cause children with smaller step sizes to survive after Cauchy mutation.
3. FCEA performs better than FCEA$_{ncr}$ due to the implementation of the D-increasing and A-decreasing rules. Figure 11 shows that the average step size, $A_T(M_{dg})$ in FCEA, of the decreasing-based mutation is controlled by the D-increasing rule after the 120th generation. In FCEA$_{ncr}$, the average step size is decreased with a fixed rate and becomes very small because the D-increasing rule is not applied. Furthermore, Fig. 12 indicates that the average step size of M_g in FCEA$_{ncr}$ is larger than FCEA, because the A-decreasing rule is not applied in FCEA$_{ncr}$. Figure 14 shows that $E_T(M_g)$ in FCEA$_{ncr}$ is smaller than that in M_g. These observations may explain the effectiveness of adaptive rules.
4. The mutation operators used in FCEA have a similar average expected improvement, although their step sizes are different according to Figs 10 and 9. This implies that each mutation is able to improve solution quality by adapting its step sizes in the whole searching process.

In summary, the above discussion has shown that the main ideas of FCEA, employing multiple mutation operators in one iteration and coordinating them by adaptive rules, are very useful.

4.4. Comparison with Related Works

Finally, we compare FCEA with three GENITOR style genetic algorithms [23], including a bit-string genetic algorithm (GENITOR), a distributed genetic algorithm (GENITOR II) and a real-valued genetic algorithm. GENITOR, proposed by Whitley et al. [23] to train ANNs, is one of the most widely used genetic algorithms. GENITOR encoded weights as a binary string to solve 2-bit adder problem. This approach found solutions on 17 out of 30 runs, i.e. $S_r = 56\%$, by using a population of 5000 and two million function evaluations. GENITOR II [23] used a combination of adaptive mutation and a distributed genetic algorithm. It was able to increase the S_r to 93% on a 2-bit adder problem. However, its population size was also 5000, and the number of function evaluations also reached two million. A back propagation method was implemented [23] to solve the 2-bit adder problem, and its convergence rate is 90%.

At the same time, Whitley et al. duplicated a real-valued genetic algorithm, proposed by Montana and Davis [16]. This algorithm is able to evolve both

Table 3. Comparison of FCEA with three GENITOR style genetic algorithms, including GENITOR, GENITOR II and real-value representation GENITOR on the 2-bit adder problem.

Problem		FCEA	GENITOR[c]	GENITOR II	Real-valued GENITOR
2-bits adder	*S*[a]	96%	56%	93%	90%
	FE[b]	258,981	2,000,000	2,000,000	42,500

[a]S_r denotes the percentage of an approach classifying all training data correctly.
[b]*FE* denotes the number of average function evaluations.
[c]All results of GENITOR style genetic algorithms are directly summarised from Whitley et al. [23].

the architecture and the weights. In this experiment, the convergence (S_r) is 90% on a 2-bit adder problem using a population of 50. The number of average function evaluations was 42,500.

In contrast to these approaches, FCEA needs 258,981 function evaluations, and the S_r is up to 96% using small population size, i.e. 30. These results are summarised in Table 3.

5. The Artificial Ant Problem

In this section, we study an artificial ant problem, i.e. tracker task 'John Muir Trail' [24]. In this problem, a simulated ant is placed on a two-dimensional toroidal grid that contains a trail of food. The ant traverses the grid to collect any food encountered along the trail. This task requires us to train a neural network (i.e. a simulated ant) that collects the maximum number of pieces of food during the given time steps. Figure 15 presents this trail. Each black box in the trail stands for a food unit. According to the environment of Jefferson et al. [24], the ant stands on one cell, facing one of the cardinal directions; it can sense only the cell ahead of it. After sensing the cell ahead of it, the ant must take one of four actions: move forward one step, turn right 90°, turn left 90°, and no-op (do nothing). In the optimal trail, there are 89 food cells, 38 no food cells and 20 turns. Therefore, the number of minimum steps for eating all food is 147 time steps.

To compare with previous research, we follow the work of Jefferson et al. [24]. That investigation used not only finite state machines and recurrent neural networks to represent the problem, but also a traditional bit-string genetic algorithm to train the architectures. The recurrent network used for controlling the simulated ant is the full connection with shortcuts architecture shown in Fig. 1(c). Each simulated ant is controlled by a network having two input nodes and four output nodes. The 'food' input is 1 when the food is present in the cell ahead of the ant; and the second 'no-food' is 1 in the absence of the food in the cell in front of the ant. Each output unit corresponds to a unique action: move forward one step, turn right 90°, turn left 90°, or no-op. Each input node is connected to each of the five hidden nodes and to each of the four output nodes. The five hidden nodes are fully connected in the hidden layer. Therefore, this architecture is a

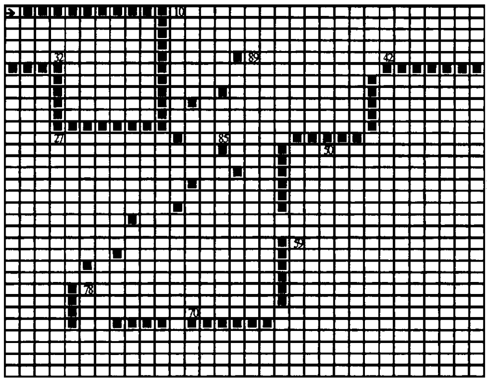


Fig. 15. The 'John Muir Trail' artificial ant problem. The trail is a 32 × 32 toroidal grid where the right edge is connected to left edge. The symbol ■ denotes a food piece on the trail, and → denotes the start position and starting facing direction of an ant.

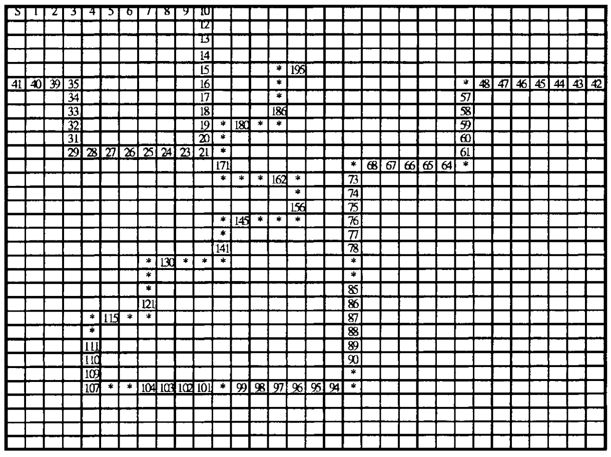


Fig. 16. A travelled solution of the 'John Muir Trail' within 195 time steps of a simulated ant controlled by our evolved neural controller. The number in the cell is the order in which the ant eats the food. The symbol * in the entry represents a cell travelled by the ant.

Table 4. Comparison among the genetic algorithm, evolutionary programming and our FCEA on the 'John Muir Trail' ant problem. The first number in parentheses is the number of runs finding all 89 food peices, and the second number is the number of total runs.

Method	N[a]	FE[b]	Best performance	Average performance
Genetic algorithms [24]	65536	6,553,600	89	N/A
Evolutionary programming [25]	100	184,250	82	81.5
FCEA	50	126,000	89 (20/25)	88.68
	100	284,000	89 (24/25)	88.96

[a]N is the population size. N/A denotes the result not available in the literature.
[b]FE denotes the average numbers of function evaluations.

full connection with shortcuts recurrent neural network; its total number of links with bias input is 72. To compare with previous results, the fitness is defined as the number of pieces of food eaten within 200 time units.

Figure 16 depicts a typical search behaviour and the travelled path of a simulated ant that is controlled by our evolved neural network. The number in the cell is the time step to eat the food. The symbol * denotes a cell travelled by an ant when the cell is empty. Figure 16 indicates that the ant requires 195 time steps to seek all 89 food pieces in the environment.

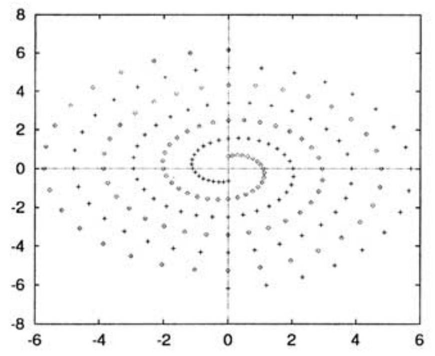
Table 4 compares our FCEA, evolutionary programming [25], and genetic algorithm [26] on the artificial ant problem. Jefferson et al. [24] used traditional genetic algorithms to solve the 'John Muir Trail'. That investigation encoded the problem with 448 bits, and used a population of 65,536 to achieve the task in 100 generations. Their approach required 6,553,600 networks to forage 89 food pieces within exactly 200 time steps. In contrast to Jefferson et al.'s solution, our FCEA uses population sizes of 50 and 100, and only requires about 126,000 and 284,000 function evaluations, respectively, to eat 89 food pieces within 195 time steps. Table 4 also indicates that FCEA performs much better than evolutionary programming [25], which uses a population of 100, and the average number of function evaluations is about 184,250. The best fitness value of evolutionary programming is 82, and its solution quality is worse than FCEA.

6. The Two-spiral Problem

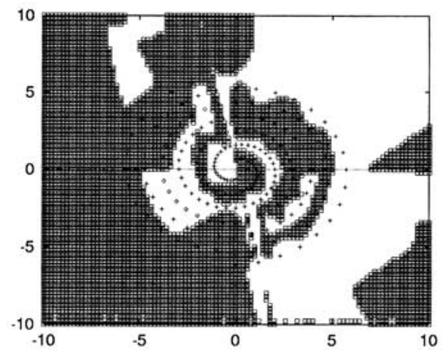
In the neural network community, learning to tell two spirals apart is a benchmark task which is an extremely hard classification task [3,4,26,27]. The learning goal is to properly classify all the training data (97 points on each spiral, as shown in Fig. 17(a)) which lie on two distinct spirals in the x-y plane. These spirals coil three times around the origin and around one another. The data of the two-spiral problem is electronically available from the Carnegie-Mellon University connectionist benchmark collection. We follow the suggestion of Fahlman and Lebiere [3] to use 40-20-40 criterion. That is, an output is considered to be a logical 0 if it lies in [0, 0.4], to be a logical 1 if the output lies in [0.6, 1.0], and indeterminate if it lies in [0.4, 0.6]. In the testing phase, 20,000 points are chosen regularly from the space (i.e. $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$), and the output of the system is defined as in the training phase.

Lang and Withbrock [26] used a 2-5-5-5-1 network with shortcut connections. Each node is connected to all nodes in all subsequent layers. With one additional bias connection for each node, therefore, there is a total of 138 trainable weights in the network. They employed the standard back propagation approach to train this network, and considered the task to be completed when each of the 194 points in the training set responded to within 0.4 of its target output value.

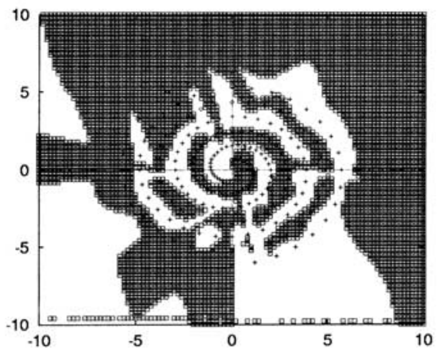
We follow the work of Lang and Withbrock [26] to solve the two-spiral problem by using the 2-5-5-5-1 network. Equation (4) is employed as the fitness function, and the population size is 30. A training input pattern is classified correctly if the tolerance of $|Y_k(I_t) - O_k(I_t)|$ is below 0.4, where $1 \leq k \leq m$ and $1 \leq t \leq T$. In this problem, T is 194 and m is 1, where $Y_k(I_t)$, $O_k(I_t)$, T and m are defined in Eq. (4). FCEA executes 10 independent runs, and it successfully classifies all 194 training points in seven runs. Figure 18 indicates a convergence curve of FCEA for the two-spiral problem. It correctly classifies 152 and 186 training points at the 5000th and 15,000th generation, respectively; it required 25,300 generations to learn all 194 training points.



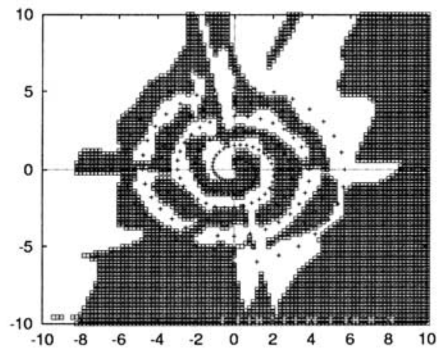
(a) The training data set of two-spiral problem with 194 examples



(b) The classification at the 5000th generation (learned points are 152)

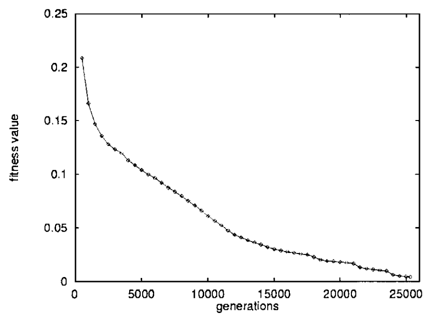


(c) The classification at the 15000th generation (learned points are 186)

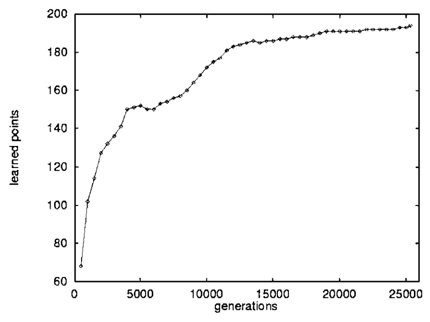


(d) The classification at the 25300th generation (learned points are 194)

Fig. 17. The two-spiral problem and classification solutions created by the ANNs evolved by FCEA at different numbers of the generations.



(a) Fitness value vs. generations



(b) Learned points vs. generations

Fig. 18. The convergence of FCEA for the two-spiral problem.

Figures 17(b)–(d) show the two-spiral response patterns of FCEA at the 5000th, 15,000th and 25,300th generation, respectively.

For every learning method, an important practical aspect is the number of pattern presentations necessary to achieve the necessary performance. In the case of a finite training set, a common measure is the number of cycles through all training patterns, also called the *epoch*. Table 5 compares our FCEA with previous approaches [3,4,26] on the two-spiral problem based on the averaged number of epochs and the ANN architectures. As can be seen, FCEA needs more epochs than the constructive algorithm [3,4], evolving both ANN architectures and connec-

Table 5. Comparison FCEA with several previous studies in the two-spiral problem.

Method	Number of epochs[a]	Hidden Nodes	Number of Connection weights[a]
Back propagation [26]	20000	15	138
Cascade-correlation [3]	1700	12–19	N/A[b]
Projection pursuit learning [4]	N/A	11–13	187
FCEA (this paper)	27318 (generations)	15	138

[a]The values in 'number of epochs' and 'connection weights' are the average values.
[b]N/A denotes 'not applicable' in the original paper.

tion weights simultaneously, and its convergence speed is also slower than the back propagation approach.

In general, the performance of the back propagation approach is more sensitive to initial weights of an ANN than evolutionary algorithms. The architectures obtained by constructive algorithms seem to be larger than the 2-5-5-5-1 network. To the best of our knowledge, FCEA is the first evolutionary algorithm to stably solve the two-spiral problem by employing the fixed 2-5-5-5-1 network. FCEA is also more robust than genetic programming [28] whose learned points was about 180, on average.

7. The Parity Problems

In this section, FCEA trains feedforward networks shown in Fig. 1(a) for N parity problems [3,29,30] where N ranges from 7 to 10. All 2^N patterns are used in the training phase, and no validation set is used. The learning goal is to train an ANN to classify all training data, consisting of an N-binary input data and a respective one-binary output for each training data. The output value is 1 if there is an odd number of 1s in the input pattern. We employ an ANN with N input nodes, N or $2N$ hidden nodes, and 1 output node. The amount of training data and the connection weights of the N-N-1 and N-$2N$-1 architectures are summarised in Table 6. FCEA employs the same parameter values shown in Table 1, except that population size is 30. The fitness function is defined as in Eq. (4). A training input pattern is classified correctly if the tolerance of $|Y_i(I_t) - O_i(I_t)|$ is below 0.1 for each output neuron, where $Y_i(I_t)$ and $O_i(I_t)$ are defined in Eq. (4). Then, a network is convergent if a network classifies all training input patterns (i.e. 2^N).

The experimental results of FCEA are averaged over ten runs for Parity-7 and Parity-8; and five runs for Parity-9 and Parity-10. These results are summarised in Table 6. The convergent rates (S_r) are 100% for parity problems with a different problem size when FCEA uses the N-$2N$-1 architectures. On the other hand, the S_r exceeds 60% if FCEA employs the N-N-1 architectures.

Tesauro and Janssens [29] used the back propagation approach to study N parity functions. They employed an N-$2N$-1 fixed network to reduce the problems of getting stuck in a local optimum for N ranging from 2 to 8. They required an average of 781 and 1953 *epochs*, respectively, for the Parity-7 and Parity-8 problems. FCEA can obtain 100% convergent rates for the N-$2N$-1 architectures, and

Table 6. Summary of the results produced by FCEA on the N parity problems by training N-N-1 and N-$2N$-1 architectures. All results are averaged over 10 independent runs.

Problem	Number of training data	N-N-1			N-$2N$-1		
		Number of connections	Average generations	S_r (%)[a]	Number of connections	Average generations	S_r (%)[a]
Parity-7	128	64	1052	90	127	1050	100
Parity-8	256	81	3650	80	161	1360	100
Parity-9	512	100	6704	80	199	4072	100
Parity-10	1024	121	9896	60	241	7868	100

[a]S_r denotes the percentage of successfully classifying all training data for FCEA on ten independent runs for Parity-7 and Parity-8; and on five independent runs for Parity-9 and Parity-10.

Table 7. Comparison of FCEA with several previous approaches on various parity functions based on epochs.

Problem	FCEA		Back propagation [28]		EPNet [29]	
	Number of links	Average generations	Number of links	Number of epochs	Number of links	Number of epochs
Parity-7	64	1052	127	781	34.7	177417
Parity-8	81	3650	161	1953	55	249625
Parity-9	100	6704	N/A[a]	N/A	N/A	N/A
Parity-10	121	9896	N/A	N/A	N/A	N/A

[a]N/A denotes ‘not applicable’.

90% for the *N*-*N*-1 architectures on these two problems. EPNet [30] is an evolutionary algorithm based on evolutionary programming [8] and combines the architectural evolution with the weights learning. It only evolves feedforward ANNs which are generalised multilayer perceptrons. EPNet can evolve a network whose hidden nodes are 4.6 and connection weights are 55 on average for the Parity-8 problem. FCEA requires 1052 and 3650 generations for the Parity-7 and Parity-8 problems, respectively. FCEA is also able to robustly solve the Parity-9 problem and the Parity-10 problem. Table 7 summarises these results.

8. Conclusions and Future Works

This study demonstrates that FCEA is a stable approach to training both feedforward and recurrent neural networks with the same parameter settings for three complex problems: the artificial ant problem, the two-spirals problem, and parity problems. Our experience suggests that a global optimisation method should consist of both global and local search strategies. For our FCEA, the decreasing-based mutation with large initial step sizes is the global search strategy; the self-adaptive mutations with the family competition procedure and replacement selection are the local search strategies. Based on the family competition and adaptive rules, these mutation operators can closely cooperate with one another.

The experiments on the artificial ant problem verify that the proposed approach is very competitive with other evolutionary algorithms, including genetic algorithms and evolutionary programming. Although FCEA requires more training time than back propagation, FCEA employs the same parameter settings and initial weights to train the neural architectures for the two-spirals problem and parity problems. We believe that the flexibility and robustness of our FCEA makes it a highly effective global optimisation tool for other task domains. According to the experimental results, using an evolutionary algorithm as a replacement for back propagation approach does not seem to be competitive with the best gradient methods (e.g. Quickprop [3]). However, evolutionary algorithms may be a promising learning method when the gradient or error information is not directly applicable, such as in the artificial ant problem.

We believe that FCEA can be applied to real-world problems, because FCEA trains both feedforward and recurrent neural networks with the same parameter values for applications in this paper. Recently, we have studied FCEA to train neural networks on several real-world problems, such as the classification of sonar signals [31], and gene prediction [32]. Our proposed approach was also successfully applied to global optimisation [13] and flexible ligand docking [33] for structure-base drug design.

To further improve the performance qualities of the FCEA, several modifications and extensions should be investigated in the future. We will extend FCEA to automatically evolve both architectures and connection weights simultaneously, because to design a near optimal ANN architecture for some application domains is an important issue. Then, a flexible mechanism is considered to adapt the family competition lengths to improve the performance according to the performance improvement of mutations and the morphology of the landscape. Finally, we will investigate FCEA on applications where gradient methods are not directly applicable.

References

1. Hornik K. Approximation capabilities of multilayer feedforward networks. Neural Networks 1991; 4: 251–257
2. Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. In: DE Rumelhart, JL McClelland, editors, Parallel Distributed

Processing: Explorations in the Microstructures of Cognition. MIT Press 1986; 318–362

3. Fahlamn SE, Lebiere C. The cascade-correlation learning architecture. In: DS Touretzky, editor, Advances in Neural Information Processing Systems II. Morgan-Kaufmann, 1990; 524–532
4. Hwang J-N, You S-S, Lay S-R, Jou I-C. The cascade-correlation learning: A projection pursuit learning perspective. IEEE Trans Neural Networks 1996; 7(2): 278–289
5. Schaffer JD, Whitley D, Eshelman LJ. Combinations of genetic algorithms and neural networks: A survey of the state of the art. Proc Int Workshop on Combinations of Genetic Algorithms and Neural Networks 1992; 1–37
6. Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Networks 1994; 5(2): 157–166
7. Goldberg DE. Genetic algorithms in Search, Optimisation and Machine Learning. Addison-Wesley, 1989
8. Fogel DB. Evolutionary Computation: Toward a New Philosophy of Machine Intelligent. IEEE Press, 1995
9. Bäck T. Evolutionary Algorithms in Theory and Practice. Oxford University Press, 1996
10. Davidor Y. Epistasis variance: Suitability of a representation to genetic algorithms. Complex Systems 1990; 4: 368–383
11. Eshelman LJ, Schaffer JD. Real-coded genetic algorithms and interval-schemata. In: LD Whitley, editor, Foundation of Genetic Algorithms 2. Morgan Kaufmann, 1993; 187–202
12. Mühlenbein H, Schlierkamp-Voosen D. Predictive models for the breeder genetic algorithm I. Continuous parameters optimisation. Evolutionary Computation 1993; 1(1): 24–49
13. Yang J-M, Kao C-Y. Integrating adaptive mutations and family competition into genetic algorithms as function optimizer. *Soft Computing* 2000; 4(2): 89–102
14. Hart WE. Adaptive global optimisation with local search. PhD thesis, University of California, San Diego, 1994
15. Kitano H. Empirical studies on the speed of the convergence of neural network training using genetic algorithms. Proc Int Conf on Artificial Intelligence 1990; 789–795
16. Montana DJ, Davis L. Training feedforward neural networks using genetic algorithms. Proc Eleventh Int Joint Conf on Artificial Intelligence 1989; 762–767
17. Xiao J, Michalewicz Z, Zhang L, Trojanowski K. Adaptive evolutionary planner/navigator for mobile robots. IEEE Trans Evolutionary Computation 1997; 1(1): 18–28
18. Yang J-M, Chen Y-P, Horng J-T, Kao C-Y. Applying family competition to evolution strategies for constrained optimisation. In: Angeline PJ, Reynolds RG, McDonnell JR, Eberhart R, editors, Lecture Notes in Computer Science 1213. 1997; 201–211
19. Yang J-M, Kao C-Y, Horng J-T. Evolving neural induction regular languages using combined evolutionary algorithms. Proc 9th Int Conf on Artificial Intelligence 1996; 162–169
20. Bäck T, Schwefel H-P. An overview of evolution algorithms for parameter optimisation. *Evolutionary Computation* 1993; 1(1): 1–23
21. Schwefel H-P. Numerical Optimisation of Computer Models. Wiley, 1981
22. Rudolph G. Local convergence rates of simple evolutionary algorithms with cauchy mutations. IEEE Trans Evolutionary Computation 1997; 1(4): 249–258
23. Whitley D, Starkweather T, Bogart C. Genetic algorithms and neural networks: Optimizing connections and connectivity. Parallel Computing 1990; 14: 347–361
24. Jefferson D, Collins R, Cooperand C, Dyer M, Flowers M, Korf R, Taylor C, Wang A. Evolution as a theme in artificial life: The genesys/tracker system. Artificial Life II: Proc Workshop on Artificial Life 1990; 549–577
25. Angeline PJ, Saunders GM, Pollack JB. An evolutionary algorithm that constructs recurrent neural networks. IEEE Trans Neural Networks 1994; 5(1): 54–65
26. Lang KJ, Witbrock MJ. Learning to tell two spirals apart. Proc Connections Models Summer School. Morgan Kaufmann 1988; 52–59
27. Śmieja F. The pandemonium system of reflective agents. IEEE Trans Neural Networks 1996; 7(1): 97–106
28. Juillú H, Pollack JB. Co-evolving intertwined spirals. Proc Fifth Annual Conference on Evolutionary Programming 1996
29. Tesauro G, Janssens B. Scaling relationships in back-propagation learning. Complex Systems 1988; 2: 39–84
30. Yao X, Liu Y. A new evolutionary system for evolving artificial neural networks. IEEE Trans Neural Networks 1996
31. Gorman RP, Sejnowski TJ. Analysis of hidden units in a layered network trained to classify sonar targets. Neural Networks 1988; 1: 75–89
32. Xu Y, Mural RJ, Einstein JR, Shah MB, Uberbacher EC. Grail: A multi-agent neural network system for gene identification. Proc IEEE 1996; 84(10): 1544–1552
33. Yang J-M, Kao C-Y. Flexible ligand docking using a robust evolutionary algorithm. J Computational Chemistry 2000; 21(11): 988–998