Evolving the Structure of the Particle Swarm Optimization Algorithms

Laura Dioşan and Mihai Oltean

Department of Computer Science,
Faculty of Mathematics and Computer Science,
Babeş-Bolyai University, Cluj-Napoca, Romania
{lauras, moltean}@cs.ubbcluj.ro

Abstract. A new model for evolving the structure of a Particle Swarm Optimization (PSO) algorithm is proposed in this paper. The model is a hybrid technique that combines a Genetic Algorithm (GA) and a PSO algorithm. Each GA chromosome is an array encoding a meaning for updating the particles of the PSO algorithm. The evolved PSO algorithm is compared to a human-designed PSO algorithm by using ten artificially constructed functions and one real-world problem. Numerical experiments show that the evolved PSO algorithm performs similarly and sometimes even better than standard approaches for the considered problems.

1 Introduction

Particle Swarm Optimization (PSO) is a population based stochastic optimization technique developed by Kennedy and Eberhart in 1995 [8]. Standard PSO algorithm randomly initializes a group of particles (solutions) and then searches for optima by updating all particles along a number of generations. In any iteration, each particle is updated by following some rules [16].

Standard model implies that particles are updated synchronously [16]. This means that the current position and speed for a particle is computed taking into account only information from the previous generation of particles.

A more general model allows updating any particle anytime. This basically means three things:

1. The current state of the swarm is taken into account when a particle is updated. The best global and local values are computed for each particle which is about to be updated, because the previous modifications could affect these two values. This is different from the standard PSO algorithm where the particles were updated taken into account only the information from the previous generation. Modifications performed so far (by a standard PSO) in the current generation had no influence over the modifications performed further in the current generation.
2. Some particles may be updated more often than other particles. For instance, in some cases, is more important to update the best particles several times per generation than to update the worst particles.

J. Gottlieb and G.R. Raidl (Eds.): EvoCOP 2006, LNCS 3906, pp. 25–36, 2006.

3. We will work with only one swarm. Any updated particle will replace its parent. Note that two populations/swarms are used in the standard PSO and the current swarm is filled taken information from the previous generation.

Unlike Parsopoulos' work [13] which use Differential Evolution algorithm (suggested by Storn and Price [14]) for "on fly" adaptation of the PSO parameters, our work looks for designing a new PSO algorithm by taking into account the information from the problem being solved.

Our main purpose is to evolve the structure of a PSO algorithm. This basically means that we want to find which particles should be updated and which is the order in which these particles are updated. In this respect we propose a new technique which is used for evolving the structure of a PSO algorithm. We evolve arrays of integers which provide a meaning for updating the particles within a PSO algorithm during iteration.

Our approach is a hybrid technique that works at two levels: the first (macro) level consists in a steady-state GA [7] whose chromosomes encode the structure of PSO algorithms. In order to compute the quality of a GA chromosome we have to run the PSO encoded into that chromosome. Thus, the second (micro) level consists in a modified PSO algorithm that provides the quality for a GA chromosome.

Firstly, the structure of a PSO algorithm is evolved and later, the obtained algorithm is used for solving eleven difficult function optimization problems The evolved PSO algorithm is compared to a human-designed PSO algorithm by using 10 artificially constructed functions and one real-world problem. Numerical experiments show that the evolved PSO algorithm performs similarly and sometimes even better than standard approaches for several well-known benchmarking problems.

This research was motivated by the need of answering several important questions concerning PSO algorithms. The most important question is: *Can a PSO algorithm be automatically synthesized by using only the information about the problem being solved?* And, if yes, which is the optimal structure of a PSO algorithm (for a given problem)? We better let the evolution find the answer for us.

The rules employed by the evolved PSO during a generation are not preprogrammed. These rules are automatically discovered by the evolution.

Several attempts for evolving Evolutionary Algorithms (EAs) using similar techniques were made in the past. A non-generational EA was evolved [11] using the Multi Expression Programming (MEP) technique [11]. A generational EA was evolved [12] using the Linear Genetic Programming (LGP) technique. Numerical experiments have shown [11, 12] that the evolved EAs perform similarly and sometimes even better than the standard evolutionary approaches with which they have been compared. A theoretical model for evolving EAs has been proposed in [15].

The paper is structured as follows: section 2 describes, in detail, the proposed model. Several numerical experiments are performed in section 3. Test functions are given in section 3.1. Conclusions and further work directions are given in section 4.

2 Proposed Model

2.1 Representation

Standard PSO algorithm works with a group of particles (solutions) and then searches for optima by updating them during each generation.

During iteration, each particle is updated by following two "best" values. The first one is the location of the best solution that a particle has achieved so far. This value is called *pBest*. Another "best" value is the location of the best solution that any neighbor of a particle has achieved so far. This best value is a neighborhood best and called *nBest*.

In a standard PSO algorithm all particles will be updated once during the course of iteration.

In real-world swarm (such as flock of birds) not all particles are updated in the same time. Some of them are updated more often and others are updated later or not at all. Our purpose is to simulate this, more complex, behavior. In this case we were interested to discover (evolve) a model which can tell us which particles must be updated and which is the optimal order for updating them.

We will use a GA [7] for evolving this structure. Each GA individual is a fixed-length string of genes. Each gene is an integer number, in the interval $[0...SwarmSize-1]$. These values represent indexes of the particles that will be updated during PSO iteration.

Some particles could be updated more often and some of them are not updated at all. Therefore, a GA chromosome must be transformed so that it has to contain only the values from 0 to Max, where Max represents the number of different genes within the current array.

Example. Suppose that we want to evolve the structure of a PSO algorithm with 8 particles. This means that the $SwarmSize = 8$ and all chromosomes of our macro level algorithm will have 8 genes whose values are in the [0, 7] range. A GA chromosome with 8 genes can be:

$$C_1 = (2, 0, 4, 1, 7, 5, 6, 3).$$

For computing the fitness of this chromosome we will use a swarm with 8 individuals and we will perform, during one generation, the following updates:

update(Swarm[2]),	update(Swarm[7]),
update(Swarm[0]),	update(Swarm[5]),
update(Swarm[4]),	update(Swarm[6]),
update(Swarm[1]),	update(Swarm[3]).

In this example all 8 particles have been updated once per generation.

Let us consider another example which consists of a chromosome C_2 with 8 genes that contain only 5 different values.

$$C_2 = (6, 2, 1, 4, 7, 1, 6, 2)$$

In this case particles 1, 2 and 6 are updated 2 times each and particles 0, 3, 5 are not updated at all. Because of that it is necessari to remove the useless

particles and to scale the genes of the GA chromosome to the interval [0 ... 4]. The obtained chromosome is:

$$C_2' = (3, 1, 0, 2, 4, 0, 3, 1).$$

The quality for this chromosome will be computed using a swarm of size 5 (5 swarm particles), performing the following 8 updates:

update(Swarm[3]),	update(Swarm[4]),
update(Swarm[1]),	update(Swarm[0]),
update(Swarm[0]),	update(Swarm[3]),
update(Swarm[2]),	update(Swarm[1]).

We evolve an array of indexes based on the information taken from a function to be optimized. Note that the proposed mechanism should not be based only on the index of the particles in the *Swarm* array. This means that we should not be interested in updating a particular position since that position can contain (in one run) a very good individual and the same position could hold a very poor individual (during another run). For instance it is easy to see that all GA chromosomes, encoding permutations, perform similarly when averaged over (let's say) 1000 runs.

In order to avoid this problem we sort (after each generation) the *Swarm* array ascending based on the fitness value. The first position will always hold the best particle at the beginning of a generation. The last particle in this array will always hold the worst particle found at the beginning of a generation. In this way we will know that $update(Swarm[0])$, will mean something: not that one of the particles is updated, but that the best particle (at the beginning of the current generation) is updated.

2.2 Fitness Assignment

The model proposed in this paper in divided in two levels: a macro level and a micro level. The macro-level is a GA algorithm that evolves the structure of a PSO algorithm. For this purpose we use a particular function as training problem. The micro level is a PSO algorithm used for computing the quality of a GA chromosome from the macro level.

The array of integers encoded into a GA chromosome represents the order of update for particles used by a PSO algorithm that solves a particular problem. We embed the evolved order within a modified Particle Swarm Optimization algorithm as described in sections 2 and 2.3.

Roughly speaking the fitness of a GA individual is equal to the fitness of the best solution generated by the PSO algorithm encoded into that GA chromosome. But, since the PSO algorithm uses pseudo-random numbers, it is very likely that successive runs of the same algorithm will generate completely different solutions. This problem can be handled in a standard manner: the PSO algorithm encoded by the GA individual is run multiple times (50 runs in fact) and the fitness of the GA chromosome is averaged over all runs.

2.3 The Algorithms

The algorithms used for evolving the PSO structure are described in this section. Because we use a hybrid technique that combines GA and PSO algorithm within a two-level model, we describe two algorithms: one for macro-level (GA) and another for micro-level (PSO algorithm).

The Macro-level Algorithm. The macro level algorithm is a standard GA [7] used for evolving particles order of update. We use steady-state evolutionary model as underlying mechanism for our GA implementation. The GA algorithm starts by creating a random population of individuals. Each individual is a fixed-length array of integer numbers. The following steps are repeated until a given number of generations are reached: Two parents are selected using a standard selection procedure. The parents are recombined (using one-cutting point crossover) in order to obtain two offspring. The offspring are considered for mutation which is performed by replacing some genes with randomly generated values. The best offspring O replaces the worst individual W in the current population if O is better than W.

The Micro-level Algorithm. The micro level algorithm is a modified Particle Swarm Optimization algorithm [16] used for computing the fitness of a GA individual from the macro level.

S_1 Initialize the swarm of particles randomly
S_2 While not stop_condition
S_3 For each gene of the GA chromosome
  S_{31} Compute fitness of the particle specified by the current gene of the GA chromosome
  S_{32} Update *pBest* if the current fitness value is better than *pBest*
  S_{33} Determine *nBest* for the current particle: choose the particle with the best fitness value of all the neighbors as the *nBest*
  S_{34} Calculate particle's velocity according to eq. 1
  S_{35} Update particle's position according to eq. 2
S_4 EndFor
S_5 Sort particles after fitness.

$$v_{id} = w * v_{id} + c_1 * rand() * (p_{id} - x_{id}) + c_2 * rand() * (p_{nd} - x_{id}) \quad (1)$$

$$x_{id} = x_{id} + v_{id} \quad (2)$$

where $rand()$ generates a random real value between 0 and 1.

The above algorithm is quite different from the standard PSO algorithm [16].

Standard PSO algorithm works on two stages: one stage that establishes the fitness, *pBest* and *nBest* values for each particle and another stage that determines the velocity (according to equation 1) and makes update according to equation 1 for each particle. Standard PSO usually works with two populations/swarms. Individuals are updated by computing the *pBest* and *nBest* value using the information from the previous population. The newly obtained individuals are added to the current population.

Our algorithm performs all operations in one stage only: determines the fitness, *pBest*, *nBest* and velocity values only when a particle is about to be updated. In this manner, the update of the current particle takes into account the previous updates in the current generation. Our PSO algorithm uses only one population/swarm. Each updated particle will automatically replace its parent.

3 Experiments

Numerical experiments for evolving a PSO algorithm for function optimization are performed in this section. The obtained PSO algorithm is tested against 11 difficult problems. Several numerical experiments, with a standard Particle Swarm Algorithm [16] are also performed. Finally the results are compared. We evolve the structure of a PSO algorithm and then we asses its performance by comparing it with the standard PSO algorithm.

3.1 Test Functions

Eleven test problems are used in order to asses the performance of the evolved EA. Functions $f_1 - f_6$ are unimodal test function. Functions $f_7 - f_{10}$ are highly multi modal (the number of the local minimum increases exponentially with problem's dimension [17]). Functions $f_1 - f10$ are given in Table 1. Function f_{11} corresponds to the constrained portfolio optimization problem.

The Portfolio Selection Problem. Modern computational finance has its historical roots in the pioneering portfolio theory of Markowitz [10]. This theory is based on the assumption that investors have an intrinsic desire to maximize return and minimize risk on investment. Mean or expected return is employed as a measure of return, and variance or standard deviation of return is employed as a measure of risk. This framework captures the risk-return tradeoff between a single linear return measure and a single convex nonlinear risk measure. The solution typically proceeds as a two-objective optimization problem where the return is maximized while the risk is constrained to be below a certain threshold. The well-known risk-return efficient frontier is obtained by varying the risk target and maximizing on the return measure.

The Markowitz mean-variance model [10] gives a multi-objective optimization problem, with two output dimensions. A portfolio p consisting of N assets with specific volumes for each asset given by weights w_i is to be found, which minimizes the variance of the portfolio:

$$\sigma_p = \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j \sigma_{ij} \tag{3}$$

maximizes the return of the portfolio:

$$\mu_p = \sum_{i=1}^{N} w_i \mu_i \text{ subject to: } \sum_{i=1}^{N} w_i = 1, 0 \leq w_i \leq 1, \tag{4}$$

Table 1. Test functions used in our experimental study. The parameter n is the space dimension ($n = 5$ in our numerical experiments) and f_{min} is the minimum value of the function. All functions should be minimized.

Test function	**Domain**	f_{min}
$f_1(x) = \sum_{i=1}^{n} (i \cdot x_i^2)$.	[-10, 10]^n	0
$f_2(x) = \sum_{i=1}^{n} x_i^2$.	[-100, 100]^n	0
$f_3(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert$.	[-10, 10]^n	0
$f_4(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2$.	[-100, 100]^n	0
$f_5(x) = \max_i \{x_i, 1 \leq i \leq n\}$.	[-100, 100]^n	0
$f_6(x) = \sum_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2$.	[-30, 30]^n	0
$f_7(x) = 10 \cdot n + \sum_{i=1}^{n} (x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i))$	[-5, 5]^n	0
$f_8(x) = -a \cdot e^{-b\sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}} - e^{\frac{\sum \cos(c \cdot x_i)}{n}} + a + e$.	[-32, 32]^n $a = 20$, $b = 0.2$, $c = 2\pi$.	0
$f_9(x) = \frac{1}{4000} \cdot \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1$.	[-500, 500]^n	0
$f_{10}(x) = \sum_{i=1}^{n} (-x_i \cdot \sin(\sqrt{\lvert x_i \rvert}))$	[-500, 500]^n	-n∗ 418.98
f_{11} = The Portfolio Selection Problem	[0, 1]^n	0

where $i = 1...N$ is the index of the asset, N represents the number of assets available, μ_i the estimated return of asset i and σ_{ij} the estimated covariance between two assets. Usually, μ_i and σ_{ij} are to be estimated from historic data. While the optimization problem given in (3) and (4) is a quadratic optimization problem for which computationally effective algorithms exist, this is not the case if real world constraints are added. In this paper we treat only the cardinality constraints problem.

Cardinality constraints restrict the maximal number of assets used in the portfolio

$$\sum_{i=1}^{N} z_i = K, \text{ where } z_i = sign(w_i). \tag{5}$$

Let K be the desired number of assets in the portfolio, ϵ_i be the minimum proportion that must be held of asset i, $(i = 1, ..., N)$ if any of asset i is held, δ_i be the maximum proportion that can be held of asset i, $(i = 1, ..., N)$ if any of asset i is held, where we must have $0 \leq \epsilon_i \leq \delta_i \leq 1 (i = 1, ..., N)$. In practice, ϵ_i represents a "min-buy" of "minimum transaction level" for asset i and δ_i limits the exposure of the portfolio to asset i.

$$\epsilon_i z_i \leq w_i \leq \delta_i z_i, i = 1, ..., N \quad (6)$$

$$w_i \in [0, 1], i = 1, ..., N. \quad (7)$$

Equation (5) ensures that exactly K assets are held. Equation (6) ensures that if any of asset i is held ($z_i = 1$) its proportion w_i must lie between ϵ_i and δ_i, whilst if none of asset is held ($z_i = 0$) its proportion w_i is zero. Equation (7) is the integrality constraint.

The objective function (equation (3)), involving as it does the covariance matrix, is positive semi-definite and hence we are minimizing a convex function.

The chromosome representation (within a GA algorithm) supposes (conform to [2]) a set Q of K distinct assets and K real numbers s_i, $(0 \leq s_i \leq 1)$, $i \in Q$.

Now, given a set Q of K assets, a fraction $\sum_{j \in Q} \epsilon_j$ of the total portfolio is already accounted for and so we interpret s_i as relating to the share of the *free* portfolio proportion $(1 - \sum_{j \in Q} \epsilon_j)$ associated with asset $i \in Q$.

So, our GA chromosome will encode real numbers s_i and the proportion of asset i from Q in portfolio will be:

$$w_i = \epsilon_i + \frac{s_i}{\sum_{j \in Q} s_j}(1 - \sum_{j \in Q} \epsilon_j) \quad (8)$$

For this experiment we have used the daily rate of exchange for a set of assets quoted to Euronext Stock [6].

Experiment 1. The structure of a PSO algorithm is evolved in this experiment. We use function f_1 as training problem.

For GA we run during 50 generations a population with 50 individuals, each individual having 10 genes. We perform a binary tournament selection, one cutting point recombination (applied with probability 0.8) and weak mutation (applied with probability 0.1). The parameters of the PSO algorithm (micro level) are given in Table 2. The $SwarmSize$ is not included in this table because different PSOs may have different number of particles. However, the number of function evaluations/generation is equal to 10 for all evolved PSO.

Our algorithm uses a randomized inertia weight, selected in the spirit of Clerc's constriction factor [3], [5] (many reports use a linearly decreasing inertia weight which starts at 0.9 and ends at 0.4, but we want to not restrict our

Table 2. The parameters of the PSO algorithm (the micro level algorithm) used for computing the fitness of a GA chromosome

Parameter	**Value**
Number of generations	50
Number of function evaluations/generation	10
Number of dimensions of the function to be optimized	5
Learning factor c_1	2
Learning factor c_2	1.8
Inertia weight	0.5 + rand() / 2

inertia to a fix model: decreasing or increasing function). Learning factors are not identical. Initial we have used same values for this parameters, but recent work [1] reports that it might be even better to choose a larger cognitive parameter, c_1, than a social parameter, c_2, but with $c_1 + c_2 < 4$.

We performed 50 independent runs for evolving order for particles. The results obtained in one of the runs (randomly selected from the set of 50 runs) are presented in Figure 1.

Different orders of particles have been evolved. Two of these orders are represented by the chromosomes: $C_1 = (1112020113)$ and $C_2 = (1502033043)$. The second chromosome (C_2) will be used in the numerical experiments performed in the next section.

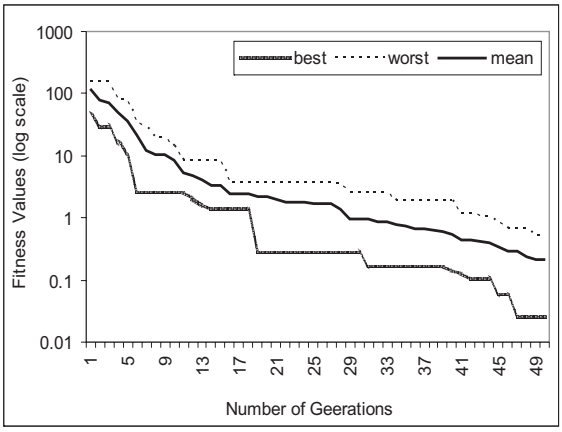

Fig. 1. The evolution of the fitness of the best/worst GA individual, and the average fitness (of all GA individuals in the population) in a particular run

Experiment 2. For assessing the performance of the evolved PSO we will compare it with a standard PSO algorithm. For this comparison we use the test functions given in Table 1.

In order to make a fair comparison we have to perform the same number of function evaluations in both evolved PSO and standard PSO. The evolved PSO has 6 particles, but because some of them are updated more times, it will perform 10 function evaluations / generation (it means that in each generation will be processed 10 updates - some particles will be updated more times). The standard PSO [16] has 10 particles and thus it will perform 10 function evaluations/generation (for standard PSO each particle will be updated one time - 10 updates in total). The other parameters of the standard PSO are similar to those used by the evolved PSO and are given in Table 2.

Taking into account the averaged values we can see in Table 3 that the evolved PSO algorithm performs better than the standard PSO algorithm in 7 cases (out of 11). When taking into account the solution obtained in the best run,

Table 3. The results obtained by the evolved PSO algorithm and the standard PSO algorithm for the considered test functions. $Best/Worst$ stands for the fitness of the best individual in the best/worst run. The results are averaged over 500 runs.

Functions	Evolved PSO				Standard PSO			
	Worst	**Best**	**Mean**	**StdDev**	**Worst**	**Best**	**Mean**	**StdDev**
f_1	2.717	0.001	0.526	0.723	7.123	0.282	2.471	1.517
f_2	3.413	0.070	0.757	0.718	3.908	0.077	1.495	0.995
f_3	2.332	0.340	1.389	0.638	1.639	0.395	0.846	0.344
f_4	632.667	5.549	151.433	134.168	547.968	2.360	81.114	127.010
f_5	3.106	0.457	0.766	0.410	9.298	0.457	1.278	1.268
f_6	1883.010	11.585	281.541	407.174	1625.180	4.254	88.371	230.074
f_7	23.295	6.790	12.766	4.884	29.973	5.743	15.619	5.084
f_8	3.639	0.165	1.601	1.248	3.491	0.469	1.960	0.681
f_9	13.987	8.749	11.911	1.252	41.282	3.233	18.044	9.370
f_{10}	-773.118	-910.367	-853.202	30.990	-512.661	-1563.899	-880.475	207.074
f_{11}	3.412	0.001	0.785	1.168	3.689	0.180	1.285	0.801

the evolved PSO algorithm performs better than the standard PSO algorithm in 5 cases (out of 11) and tied in 1 case. When taking into account the solution obtained in the worst run, the evolved PSO algorithm performs better than the standard PSO algorithm in 7 cases.

We have also compared the evolved PSO to another PSO algorithm that updates all particles one by one (i.e. the order of update is 0, 1, ... $SwarmSize-1$). In 9 cases the evolved PSO performed better (on average) than the other algorithm.

In order to determine whether the differences between the evolved PSO algorithm and the standard PSO algorithm are statistically significant, we use a t-test with a 0.05 level of significance. Before applying the T-test, an F-test has been used for determining whether the compared data have the same variance. The P-values of a two-tailed T-test with 499 degrees of freedom are given in Table 4. Table 4 shows that the differences between the results obtained by standard PSO and by the evolved PSO are statistically significant ($P < 0.05$) in 9 cases (out of 11).

Table 4. The results of F-test and T-test

Functions	F-test	T-test	Functions	F-test	T-test
f_1	7.40E-07	5.13E-13	f_7	7.80E-01	2.58E-03
f_2	2.43E-02	2.41E-05	f_8	4.06E-05	3.87E-02
f_3	3.14E-05	3.63E-07	f_9	4.13E-30	6.64E-06
f_4	7.03E-01	4.18E-03	f_{10}	8.55E-28	1.80E-01
f_5	5.76E-13	3.90E-03	f_{11}	9.57E-03	7.12E-03
f_6	1.06E-04	2.17E-03			

4 Conclusion and Further Work

A new hybrid technique for evolving the structure of a PSO algorithm has been proposed in this paper. The model has been used for evolving PSO algorithms for function optimization. Numerical experiments have shown that the evolved PSO algorithm performs similarly and sometimes even better than the standard PSO algorithm for the considered test functions.

Note that according to the No Free Lunch theorems [18] we cannot expect to design a perfect PSO which performs the best for all the optimization problems. This is why any claim about the generalization ability of the evolved PSO should be made only based on some numerical experiments.

Further work will be focused on: finding patterns in the evolved structures. This will help us design PSO algorithms that use larger swarms, evolving better PSO algorithms for optimization, evolving PSO algorithms for other difficult problems.

References

1. A. Carlisle, G. Dozier, "An Off-the-shelf PSO", *Proceedings of the Particle Swarm Optimization Workshop*, pp. 1-6, 2001.
2. T. -J. Chang, (et al.), "Heuristics for cardinality constrained portfolio optimisation" Comp. & Opns. Res. 27, pp. 1271-1302, 2000.
3. M. Clerc, "The swarm and the queen: towards a deterministic and adaptive particle swarm optimization", *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1999)*, pp. 1951-1957, 1999.
4. R. C. Eberhart, Y. Shi, "Comparison Between Genetic Algorithms and Particle Swarm Optimization", *Evolutionary Programming VII: Proceedings of the Seventh International Conference*, pp. 611-616, 1998.
5. R. C. Eberhart, Y. Shi, "Particle swarm optimization: developments, applications and resources", *Proceedings of the CEC*, 2001.
6. http://www.euronext.com
7. D. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, Boston, USA, 1989.
8. J. Kennedy, R. C. Eberhart, "Particle Swarm Optimization", *Proceedings of the 1995 IEEE International Conference on Neural Networks*, pages 1942-1948, 1995.
9. J. R. Koza, *Genetic programming, On the programming of computers by means of natural selection*, MIT Press, Cambridge, MA, 1992.
10. H. Markowitz, "Portfolio Selection", *Journal of Finance*, 7, pp. 77-91,1952.
11. M. Oltean, C. Groşan, "Evolving EAs using Multi Expression Programming", *Proceedings of the European Conference on Artificial Life*, pp. 651-658, 2003.
12. M. Oltean, "Evolving evolutionary algorithms using Linear Genetic Programming", *Evolutionary Computation*, MIT Press, Cambridge, Vol. 13, Issue 3, 2005.
13. K. E. Parsopoulos, M. N. Vrahatis, "Recent approaches to global optimization problems through Particle Swarm Optimization", *Natural Computing*, Vol. 1, pp. 235-306, 2002.
14. R. Storn, K. Price, "Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces", *Global Optimization*, Vol. 11, pp. 341-359, 1997.

15. J. Tavares, (et al.), "On the evolution of evolutionary algorithms", in Keijzer, M. (et al.) editors, *European Conference on Genetic Programming*, pp. 389-398, Springer-Verlag, Berlin, 2004.
16. H. Xiaohui, S. Yuhui, R. Eberhart, "Recent Advances in Particle Swarm", *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 90 - 97, 2004.
17. X. Yao, Y. Liu, and G. Lin, "Evolutionary Programming Made Faster", *IEEE Transaction on Evolutionary Computation*, pp. 82-102, 1999.
18. D. H. Wolpert and W. G. McReady, "No Free Lunch Theorems for Optimization", *IEEE Transaction on Evolutionary Computation*, Nr. 1, pp. 67-82, IEEE Press, NY, USA, 1997.