

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics



BACHELOR THESIS

Optimization Using Neural Gas

Prague, 2009

Author: Pavel Tomáško

Declaration

I declare my bachelor thesis to be written entirely by myself and that I used only sources (literature, other projects, software etc.) mentioned in the attached list.

V Praze dne 9.7.2009

Pavel Šomáček
podpis

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze dne 9.7.2009

Pavel Tomáško
podpis

Poděkování

Děkuji zde v první řadě svému vedoucím bakalářské práce - *Petru Pošíkovi* - za jeho trpělivost, kterou se mnou měl a ochotu k rychlému řešení jakéhokoliv problému, který se v průběhu jejího vypracovávání objevil.

Dále děkuji své rodině a přátelům, že na mě nezanevřeli a vždy mě podpořili i v četných vypjatých situacích, které během práce nastaly.

Abstract

Continuous global blackbox optimization is one of the important problems in today's science and engineering routine. Although there exist many algorithms, which are able to solve the problem, all of them is applicable only on relatively constrained class of optimized functions.

The present tendency is to look for the algorithm, which could solve larger class of problems - preferably larger, than all other algorithms - and which would work for all functions in this class with acceptable power.

I am focusing on not yet well-explored way of the optimization in this work. *The Neural Gas* - despite it falls into another algorithm class - looks like to be acceptable for being used as a global optimizer after some changes thanks to its properties.

After a brief introduction, I present two ways of how the Neural Gas can be used for optimization, found in the literature.

Next I try to reproduce the authors' reached results, which they present in their articles.

In the last part of the work I compare the algorithms with each other to see which is better and their advantages and disadvantages.

During the work it becomes obvious, that the authors of both articles evidently did some mistakes during writing them, because not even one of the algorithms works as it should. This made me often deep in thought about trustfulness of technical articles.

Nevertheless the Neural Gas showed that can be quite good, compared with other algorithms, in the future.

Abstrakt

Spojité globální blackbox optimalizace je jedním z důležitých problémů v současné vědě i inženýrské praxi. Přestože již existuje řada algoritmů, které tento problém dokážou řešit, každý z nich je aplikovatelný pouze na určitou omezenou třídu optimalizovaných funkcí.

Současný trend je hledat takový algoritmus, který by měl tuto třídu co nejšířší - pokud možno širší, než ostatní - a pracoval na všech funkcích v této třídě s přijatelným výkonem.

V této práci se zaměřuji na dosud málo probádaný způsob optimalizace. *Neuronový plyn* - ač spadá do jiné kategorie algoritmů - svými vlastnostmi vybízí k jeho úpravě a využití jako optimalizátoru.

Po stručném úvodu do problematiky představuji dva možné přístupy, jakými je možno plyn využít pro optimalizaci, nalezené v literatuře.

Dále se pokouším reprodukovat dosažené výsledky, které autoři algoritmů ve svých článcích uvádějí.

V poslední části stavím algoritmy vedle sebe a snažím se poukázat na výhody a nevýhody každého z nich.

V průběhu práce se ukáže, že autoři v obou článcích velmi pravděpodobně udělali chybu, která způsobuje, že ani jeden z algoritmů nepracuje tak, jak má. Tento fakt ve mě vzbudil četná zamyšlení nad důvěryhodností technických článků.

I přes tato úskalí ale Neuronový plyn podle mého názoru může v budoucnu obsadit dobré místo mezi ostatními optimalizátory.

BACHELOR PROJECT ASSIGNMENT

Student: Pavel T o m á š k o

Study programme: Electrical Engineering and Information Technology

Specialisation: Cybernetics and Measurement

Title of Bachelor Project: Optimization Using Neural Gass


Guidelines:

1. Learn the principles of the (growing) neural gas learning method.
2. Survey the applications of the neural gas to solve optimization tasks.
3. Build an optimization algorithm using the neural gas.
4. Evaluate the algorithm, and compare it to other optimization methods.


Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Petr Pošík, Ph.D.

Valid until: the end of the winter semester of academic year 2009/2010


prof. Ing. Vladimír Mařík, DrSc.
Head of Department




doc. Ing. Boris Šimák, CSc.
Head

Prague, Februar 23, 2009

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Pavel T o m á š k o

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný

Obor: Kybernetika a měření

Název tématu: Optimalizace pomocí neuronového plynu

Pokyny pro vypracování:

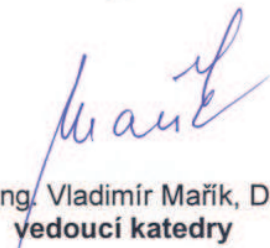
1. Seznamte se s principy (rostoucího) neuronového plynu.
2. Prostudujte, zda a jak se neuronový plyn (a příbuzné metody) používají v optimalizaci, a navrhnete vlastní způsob využití.
3. Zkonstruujte optimalizační algoritmus využívající neuronový plyn.
4. Algoritmus otestujte, porovnejte s jinými optimalizátory a výsledky statisticky vyhodnoťte.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Petr Pošík, Ph.D.

Platnost zadání: do konce zimního semestru 2009/2010




prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry


doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 23. 2. 2009

Contents

1	Introduction	1
2	Optimization using Neural Gas	2
2.1	Neural Gas Network	2
2.2	Milano Approach	3
2.3	Huhse Approach	4
2.4	Conceptual Comparison of Milano and Huhse approach	5
2.4.1	Evolutionary/Non-evolutionary	5
2.4.2	Particle rating	6
2.4.3	Optima count	6
2.4.4	Neighborhood function	6
2.4.5	Adaptation	6
2.4.6	Premature convergence protection	6
3	Implementation and Testing	7
3.1	Milano	7
3.1.1	Experiment Settings	7
3.1.2	Results	7
3.2	Huhse	14
3.2.1	Experiment Settings	14
3.2.2	Results	15
3.3	Summary of Implementation Results	15
4	Experimental Comparison	20
4.1	Experiment settings	20
4.1.1	Milano	20
4.1.2	Huhse	20
4.2	Both algorithms	21
4.3	Comparison results	21
5	Summary and Conclusions	27
6	Appendix	28
6.1	Used testing functions	28
6.1.1	Functions used by Milano	28
6.1.2	Some functions used by Huhse	28
6.2	Used software	29
6.3	Contents of attached CD	29

1 Introduction

Optimization is a mathematical discipline concerning on searching such a set of given parameters to fulfill given criteria. In my case the goal is to find such \vec{x} that $\vec{x} = \arg \min_{\vec{x} \in D} f(\vec{x})$ with D constraining the area where to find the solution. Function f is the function to be optimized and is usually called *objective function*, or *fitness function* in the context of evolutionary algorithms, and is often considered to be computationally expensive, hence the fewer times called is the better.

Optimization task defined this way is called *global optimization*, which means, that the minimum found is the best one and no one with a lower value can be found anywhere over the feasible search space D .

With the view of terminology: The solution \vec{x}_a is *better than* \vec{x}_b when $f(\vec{x}_a) < f(\vec{x}_b)$. Then *local optima* are the points \vec{x} , better than points from their small neighborhood.

Existence of such points introduces classifying all fitness functions into two basic groups:

- *Unimodal functions* having one local optimum, which is also a global optimum.
- *Multimodal functions* having at least two local optima.

This work is basically concerned with a *black-box* optimization, where the fitness function properties, like derivations, count and existence of local and global optima, are generally unknown. I must mention, that in this work I actually know the count of local/global optima.

There are some methods, aimed to find local optima. For example Hill climbing [4].

Other algorithms are stochastic, I can enumerate Simulated Annealing [6] or population-based like CMA-ES [5], Genetic Algorithms [9], Particle Swarm Optimization [7] and Differential Evolution [8] as representatives.

Main problem of global view methods is prematurely switching to the local view, omitting yet unexplored locations in the search space and converging to some local optimum. Authors of new approaches make efforts to keep possibility to find global optima, while focusing on locations supposing to be a global optimum, but being a local one for example by many methods trying to keep large population diversity.

Self Organizing Maps with Neighborhood Attraction [10], [3], are one kind of *machine learning methods*. They are all tools to capture unknown probability distribution in an N -dimensional space by adapting to the learning set, consisting of points of this distribution and also to capture the topology of this distribution and often even project it onto lower-dimensional space.

The motivation of why to use them in global optimization is their global view of the whole search space. Even if - in the extreme case - the map models probability concentrated into one small subspace of the searchspace (this means, that major part of map nodes cumulated in this small space), there is still nonzero probability elsewhere due to presence of dispersed minority of map nodes. This fact induces the NG to be used for modelling "promisability" - after fitting it in the fitness function some way, it can answer the question 'Where am I most likely to find the global optimum?' by the response 'Probably here, but there is still chance to find it over there.'.

In the chapter 2 I describe what the Neural Gas is and how can be used for a global optimization. There you can find described the only two algorithms, found in the literature and compared in a formal way.

In the chapter 3 I write about how my implementation of both algorithms works and how my results differ from the authors' results. There are also some considerations about parameter settings.

In the chapter 4 are the algorithms compared to each other and their mutual drawbacks and advantages are described.

2 Optimization using Neural Gas

2.1 Neural Gas Network

The Neural Gas [3] is a method usually used for *vector quantization*. This allows modelling of often complicated probability density function, which carries large amount of information by distributing a number of representative individuals (also referred as *neurons* or *nodes*), which at the cost of some information loss, endeavours to pick up the most essential outlines of original function.

Why neural? Because it is set of particles, each characterized of its perception field (the Voronoi polygon). Stimulus, incoming to the network activates the neuron, in whose region it lies.

Why gas? However neural networks mostly have fixed topology and their connections do not change over time (analogy with solid substances), neural gas approach changes its topology dynamicaly as a reaction to input data character - analogy with gas.¹

Why network? Strictly as defined in the original Neural Gas [3], the nodes can be connected by edges. An edge is created/kept up between two neurons nearest to the stimulus if it does not exist already. Edges are aged and cease after some time, if not kept up. At the end of the algorithm, edges connect every two nodes, whose associated data are similar. Mean edge count by one neuron in particular space region is specific for this region's dimensionality.

Neural Gas learning consists of "stimulating" the net of nodes, initially with uniform distribution by random signals of an unknown distribution. Each node location in the space is described by its so-called *weight vector*. Weight vectors of all nodes are originally adapted using these formulas [3]:

$$w_{i,new}^{\vec{}} = w_{i,old}^{\vec{}} + \epsilon(t) \cdot e^{-i/\lambda(t)} \cdot (\vec{v} - w_{i,old}^{\vec{}}) \quad (1)$$

$$\epsilon(t) = \alpha_i \cdot \left(\frac{\alpha_f}{\alpha_i}\right)^{t/t_{max}} \quad (2)$$

$$\lambda(t) = \lambda_i \cdot \left(\frac{\lambda_f}{\lambda_i}\right)^{t/t_{max}} \quad (3)$$

With \vec{v} being the input stimulus, i is the node order in ordered node set, while all Neural Gas nodes are sorted according to their increasing Euclidean distance

¹Once looking on neural gas in action, one has feeling more on viscosious liquid than gas.

from \vec{v} . For further reading, $\epsilon(t)$ is called *learning rate*, influencing the adaptation velocity and $\lambda(t)$ is *neighborhood range*, influencing the area of attraction.

$\epsilon(t)$ and $\lambda(t)$ depend on time t exponentially. Their value is ϵ_i, λ_i in the beginning² (for *time* $t = 0$) and reaches ϵ_f, λ_f at the end (for time $t = t_{max}$).

$\epsilon_i, \lambda_i, \epsilon_f, \lambda_f, t_{max}$ are algorithm parameters.

According to the formula, the net nodes move towards each input \vec{v} , jumping a step of length lowering with the distance of the node from the stimulus. The point is to locate all nodes so that each posterior stimulus will have some nodes around and its distance from nodes will fall and their count close around the stimulus will grow with growing probability of occurring stimulus on that place.

In the optimal case, after enough signals adapted the net, the nodes will eventually estimate the input signal distribution.

Using The Neural Gas for optimization is not common so far. There were only two articles found in the literature. Both mention neural gas as their basic, but both are totally different.

One thing we can do is to declare modelled probability function to be the probability distribution of possible global optimum occurrence. Then we ask the neural gas about where it has the highest neuron concentration and focus here searching tries to find the best possible solution. This is the approach [1] which Milano used in his algorithm. Detailed study follows in section 2.2.

Other way is to use the neural gas adaptation rule as a kind of crossover operator in an evolutionary strategy. This is how Huhse [2] uses neural gas attraction rules in her evolution strategy. Detailed study runs from section 2.3.

2.2 Milano Approach

The algorithm of Milano [1] is actually the classical Neural Gas network, but without interconnecting neurons by edges (because it is useless for the purpose, they are totally ignored in the new algorithm) and without time-dependency of learning rate and neighborhood range, which is originally part of the Neural Gas algorithm (see section 2.1). These functions of time are replaced by constant learning rate α and neighborhood range λ .

The algorithm stands as following:

1. Initialization - Lay out the NG particles randomly in the search space. Assign the first best parameters - use \mathcal{M}_{NG} . \mathcal{M}_{NG} is the sampling operator, which first selects one neuron \vec{p} uniformly randomly, then lets \vec{p}_s be the point chosen using normal random distribution with the mean vector being \vec{p} and the standard deviation being the shortest euclidean distance to the nearest neuron. Evaluate $f(\vec{p})$ at the sampled point, store \vec{p} and $f(\vec{p})$ as \vec{p}_{best} and f_{best} . f_{best} is the best currently known fitness value (*BSF - Best So Far value*) and \vec{p}_{best} is the point in the searchspace, where the BSF was found.
2. Repeat following steps until the final condition \mathcal{F} passes. Final condition \mathcal{F} passes when $f_{best} < T$ where T is objective function threshold value, considered to be good enough for a given problem. This condition also often contains the fitness evaluation count limitation: passes also if the fitness evaluation count reaches some given number.

²Symbol i indexing ϵ and λ abbreviates the word *initial*. Do not confuse this abbreviation and the node indexes.

- (a) Use \mathcal{M}_{NG} to sample point \vec{p}_s and evaluate $f(\vec{p}_s)$.
- (b) If $f(\vec{p}_s) < f_{best}$ then:
 - i. Let \vec{p} be the new $p_{best}^{\vec{}}$ and $f(\vec{p})$ be the new f_{best} .
 - ii. Perform NG adaptation step \mathcal{A}_{NG} , which means to stimulate the net by almost the standard way using \vec{p} as a stimulus. See following paragraph.

Milano's Neural Gas attraction rule \mathcal{A}_{NG} means to perform several substeps:

1. Sort all NG particles according to their distance from the stimulus and let $i(\vec{p})$ be the order of each particle in this sorted set A .
2. Adapt \vec{p} of all elements from A using:

$$p_{new}^{\vec{}} = p_{old}^{\vec{}} + \alpha \cdot h_N(i(p_{old}^{\vec{}})) \cdot (p_{best}^{\vec{}} - p_{old}^{\vec{}}) + C_{p_{old}^{\vec{}}}^{\vec{}} \quad (4)$$

where $h_N(i) = \exp(i/\lambda)$ is the neighborhood function,

$$C_{p_{old}^{\vec{}}}^{\vec{}} = \sum_{\vec{j} \in A, \vec{j} \neq p_{old}^{\vec{}}} \frac{K}{\|p_{old}^{\vec{}} - \vec{j}\|^2} (p_{old}^{\vec{}} - \vec{j}) \quad (5)$$

is the repulsive factor, with purpose to avoid premature convergency.

2.3 Huhse Approach

The algorithm [2] appears as a kind of a genetic algorithm with a special crossover operator - the neural gas adaptation rule. Only few particles, which are close to each other, are crossed. I personally think, that in this algorithm the global view of the neural gas of the the search space fades away. Definitely, there is no neural gas used as a probability model.

The run is slightly more complicated, than the one from Milano:

1. Initialization - Lay out the particles randomly in the search space.
2. Repeat following sttpeps while $t < t_{max}$ (this is the stopping contidion \mathcal{F}):
 - (a) Choose a particle \mathbf{p}_p randomly to be a parent particle. There are several ways, how to perform this step. The following options come to my mind:
 - There is one set of particles all the time. After one iteration, the new individual is inserted into the set immediately and so immediately participates every next iteration. The random choice means, that this resulting individual could be the parent even in the next iteration.
 - Similar as previous, except that all old individuals must be randomly choosen and transformed into their children, before anyone from these children can be choosed to be a parent.
 - Similar as previous, except that new children do not influence following iterations until all old ones become parents and produce the new child.

This is how the selection and the whole generation management is done and it was regrettably not explained well in the article. I decided to implement the third variant.

- (b) If all neighbors \mathbf{p}_i of \mathbf{p}_p are worse than \mathbf{p}_p (thus $f(\mathbf{p}_i) > f(\mathbf{p}_p)$) perform mutation \mathcal{M} . The mutation creates n_0 copies of the parent and updates their position:

$$p_{new}^{\vec{}} = p_{old}^{\vec{}} + \frac{1}{N} d_{min} \vec{\mathcal{N}}(0, 1) \quad (6)$$

This means to "noise" them using normally distributed random numbers with the mean at parent and the standard deviation $\frac{1}{N} d_{min}$. N is the problem dimensionality and d_{min} is euclidean distance to the nearest particle. $\vec{\mathcal{N}}(0, 1)$ is a vector of normally distributed random numbers with mean 0 and deviation 1.

- (c) Else perform NG variation \mathcal{A}_{NG} (explained in the paragraph below). .
 (d) Select survivor using \mathcal{S} . The selection operator \mathcal{S} computes $f(\vec{x})$ of the children and chooses the best one. Others and their parent are deleted.

Huhse's Neural Gas variation rule \mathcal{A}_{NG} creates one copy of the parent for each parent's neighbor. The individual is considered to be a neighbor, if it is closer to the parent, than $\bar{d} = \frac{1}{n} \sum_{i=1, i \neq p}^n |p_p^{\vec{}} - p_i^{\vec{}}|$; this is the mean Euclidean distance from parent to all other particles. After that, these just created copies $p_p^{\vec{}}$ are attracted to appropriate neighbors $p_i^{\vec{}}$ by the neural gas attraction rules:

1. Neighbors are first sorted according to their distance from the parent and this makes $i(\vec{p})$ to be their order after this sort.
2. Attract parent copies to the neighbors:

$$\sigma(t) = \sigma_i \cdot (\sigma_f / \sigma_i)^{(t/t_{max})} \quad (7)$$

$$\epsilon(t) = \epsilon_i \cdot (\epsilon_f / \epsilon_i)^{(t/t_{max})} \quad (8)$$

$$p_{new}^{\vec{}} = p_{old}^{\vec{}} + \epsilon(t) \cdot h_{\sigma(t)}(i(\vec{p})) \cdot (p_i^{\vec{}} - p_{old}^{\vec{}}) \quad (9)$$

where $\sigma_i, \sigma_f, \epsilon_i, \epsilon_f, t_{max}$ are algorithm parameters.

2.4 Conceptual Comparison of Milano and Huhse approach

Although the purpose of both algorithms is the same, each of them has its own specific principle. The difference between them looks to be very fundamental, so it evokes, that each will have its own specific behaviour and each could succeed in different use cases.

2.4.1 Evolutionary/Non-evolutionary

Milano This algorithm is non-evolutionary. It uses self-organizing net self only for modelling probability of occurring global optima over the search space and has no parts/sets to be determined as population, parent, children, crossover, etc.

Huhse This algorithm operates with terms of mutation, crossover and selection. The way it works makes it an evolutionary strategy. Particles do not compose a self-organizing net. I would call it "NG approach upside down" with a little disparagement. In the conventional NG strategy, one point in the search space is used to adapt many, possibly all NG particles. By contrast, many points here are used to adapt one point in several ways.

2.4.2 Particle rating

Milano Fitness of NG particles in this approach is not computed. Particles do not represent possible solutions.

Huhse The NG particles in the algorithm are rated computing their fitness. Every particle represents one possible solution.

2.4.3 Optima count

Milano The algorithm is able to find only one global optima. It does not work with a set of solutions, only one (currently best) solution is stored across adaptation steps.

Huhse The algorithm works with large set of solutions, hence theoretically has the ability to find multiple optima.

2.4.4 Neighborhood function

Milano Neighborhood function is nonzero over the whole search space. This causes all particles to be affected during each adaptation step. The function is also time-independent, which does not distinguish between adaptation tuning phases, possibly decreasing quality of found optimum (but according to the current final condition, we are familiar with any solution better than threshold. We should consider found optima quality while changing final condition). This fact is actually induced by time-independent (constant) neighborhood range (λ) function.

Huhse The neighborhood function of a given solution is zero for particles farther (in Euclidean sense) than mean of all particle distances to the given solution. Neighborhood function is also time-dependent because of time-dependent neighborhood range (λ). This distinguishes between starting phase with high mutual particle influence and final precise tuning phase.

2.4.5 Adaptation

Milano The algorithm operates with time-independent (constant) learning rate (α). This could decrease the quality of found optimum. See paragraph about Milano neighborhood function for details.

Huhse The algorithm has time-dependent learning rate (ϵ), also causing run phase distinction mentioned above. Decreasing learning rate and neighborhood range makes final phase a local searching.

2.4.6 Premature convergence protection

Milano The algorithm introduces a remedy against too fast convergence so that it should prevent stuck in local optima. The remedy is imposed via the repulsion factor $C_{\vec{p}oid}$.

Huhse Avoidance is accomplished by the used selection scheme. Even if all particles stuck around local minimum for a time period, best solution from previous iteration never survives the next one, which should lead to escape the local optima sooner or later.

3 Implementation and Testing

3.1 Milano

The foremost thing to do is to test algorithm, if it is implemented well and if there is no mistake in the article.

3.1.1 Experiment Settings

For the Milano, there are originally three sets of algorithm parameters, written in table 1.

Table 1: Original Milano settings (taken from [1])

Set	Node count N	learning rate α	neighborhood range ϵ	repulsion factor K
Set 1	10	0.2	$n/3$	0.001
Set 2	20	0.2	$n/3$	0.001
Set 3	100	0.1	$n/3$	10^{-5}

The authors tested the algorithm using four common functions - equations 10 (Modified Rosenbrock's function), 11 (Griewank's function), 12 (Rastrigin's function) in 2-D domain and 13 (Rosenbrock's function) in 10-D domain.

They declared the threshold of successful convergence to be 0.001 for function 11, 12 and 13 and 40 for 10.

Their results according to the article are caught in table 2.

Table 2: Original Milano results (taken from [1]) - Average fitness evaluation count to converge (30 runs) - (*not performed* - algorithm was not run with this setting)

Set	2-D	2-D	2-D	10-D
	Modified Rosenbrock's f.	Griewank's f.	Rastrigin's f.	Rosenbrock's f.
Set 1	1700 ± 200	180 ± 50	210 ± 50	not performed
Set 2	780 ± 80	150 ± 40	180 ± 40	not performed
Set 3	not performed	not performed	not performed	7500

3.1.2 Results

I tried to reproduce the original results, using the same algorithm and the same settings. However, my results were almost uncomparably worse (table 3).

In addition, we must realize, that the stopping threshold 40 for Modified Rosenbrock's function is too weak criterion, because it settles relatively large area (see the function description for details), so the value under 40 is easily obtainable by trivial random search.

Well, I can proclaim, that the algorithm as described does not converge at all. In my opinion, the reason of this is poor adaptation criterion (to remind: adaptation

Table 3: Reproduced Milano results - Average fitness evaluation count to converge (30 runs), evaluation count limited to 50000 for 2-D, to 10^5 for 10-D - (*does not conv.* means, that the fitness BSF value after reaching the evaluation count limit is worse, than requested 0.001)

Set	2-D Modified Rosenbrock's f.	2-D Griewank's f.	2-D Rastrigin's f.	10-D Rosenbrock's f.
Set 1	3340	does not conv.	does not conv.	not performed
Set 2	2630	does not conv.	does not conv.	not performed
Set 3	not performed	not performed	not performed	does not conv.

occurs only if some new best fitness value is found). This is severe condition, which causes the underlying neural gas to adapt only occasionally, as seen in table 4, which shows constantly low number of adaptations (compare with number of fitness evaluations).

Table 4: Reproduced Milano results - Average;minimal;maximal neural gas adaptation count (10000 runs for 2-D, 1000 runs for 10-D), fitness evaluation count limited to 50000 for 2-D, to 10^5 for 10-D

Set	2-D Modified Rosenbrock's f.	2-D Griewank's f.	2-D Rastrigin's f.	10-D Rosenbrock's f.
Set 1	11; 6; 18	12; 6; 20	11; 5; 17	not performed
Set 2	13; 3; 21	13; 7; 23	12; 6; 20	not performed
Set 3	not performed	not performed	not performed	68; 48; 86

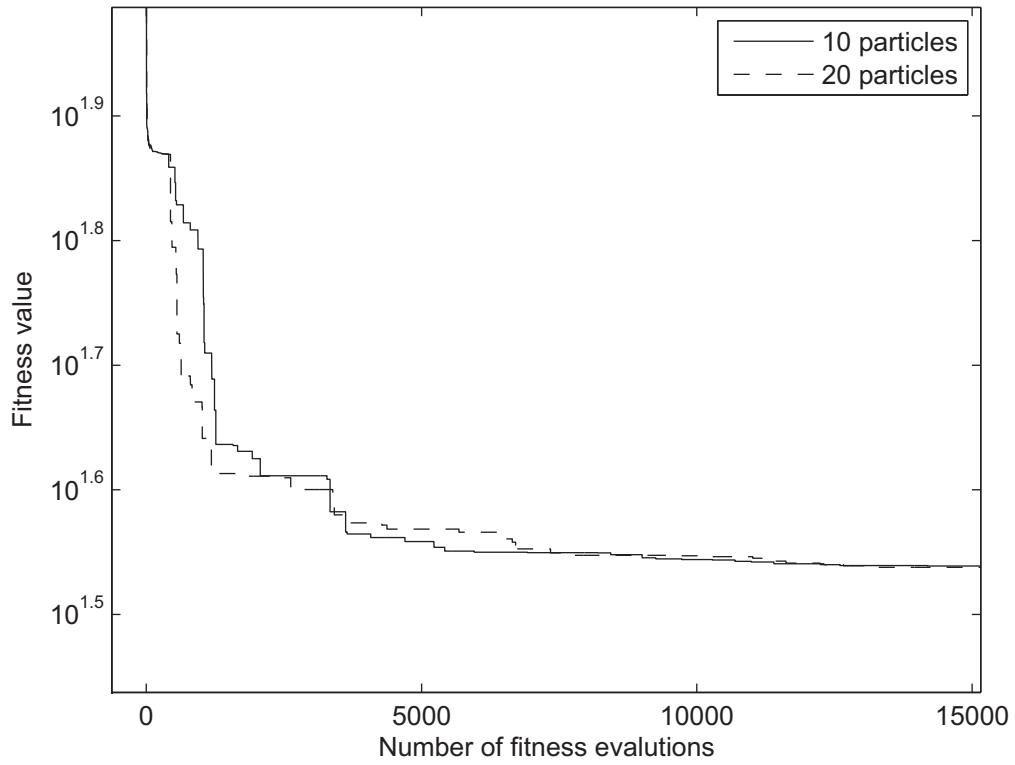
This signifies, that I was not able to reproduce the authors' results at all. The convergency curves are shown on figures 1(a), 1(b), 2(a) and 2(b).

I introduced a workaround - new adaptation rule to raise the number of adaptations per fitness evaluations. This helps a lot and makes the algorithm reach the given criteria. But I still did not attain the power specified in the article. This workaround has also one big drawback - it introduces new algorithm constants to be tuned.

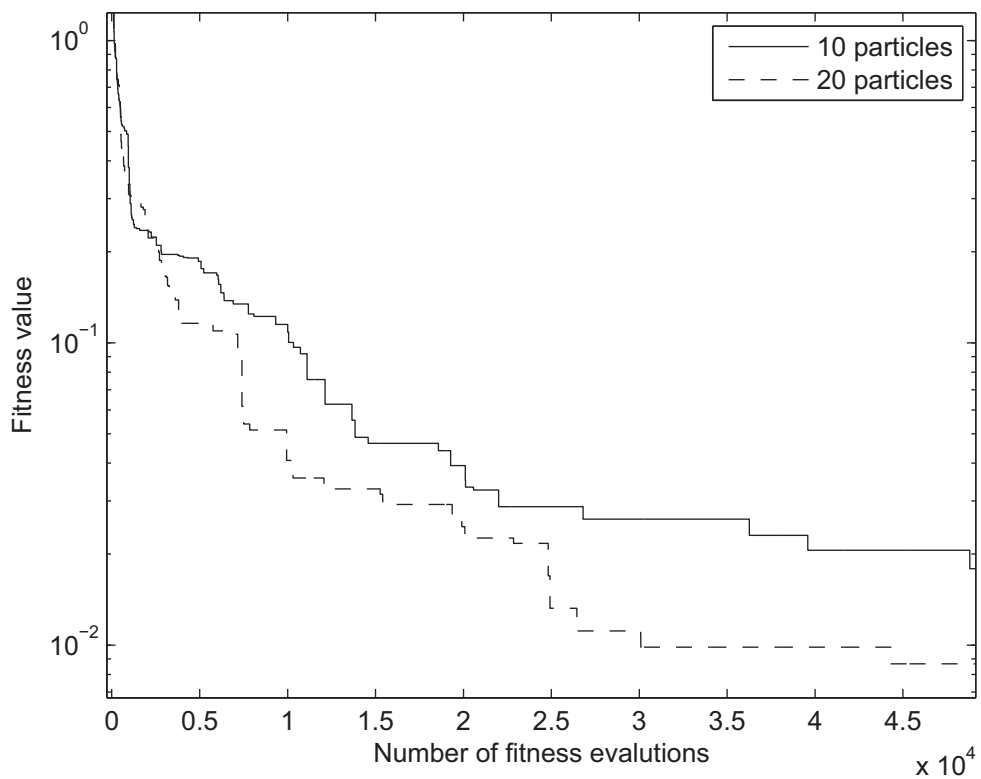
There are actually three versions of the workaround taken into account:

- Adapt whenever $f(\vec{x}) < f_{best} + F_{thr}$ with F_{thr} being a new parameter. There were several experiments performed with this and I rejected the rule, because the threshold F_{thr} is not adaptive with respect to the fitness function surface shape and neural gas nodes positions on it. I was not able to find some "universal" F_{thr} .
- Adapt whenever $f(\vec{x})$ is better than last P fitness trials. This means, that last P fitness trials are saved in an FIFO of length P and every new fitness value is compared to all of them. If the new fitness value was allways better during this comparison, use the point where it was sampled for adaptation.
- Adapt whenever $f(\vec{x})$ is better than P best fitness trials from last Q best fitness trials. This means to put fitness values into FIWO³ of Q elements, sorted

³FIWO stands for *First In Worse Out*. This is a kind of a data structure similar to *FIFO* except that - during taking out procedure - instead the item, which is on the head of the queue, the worse one is removed.

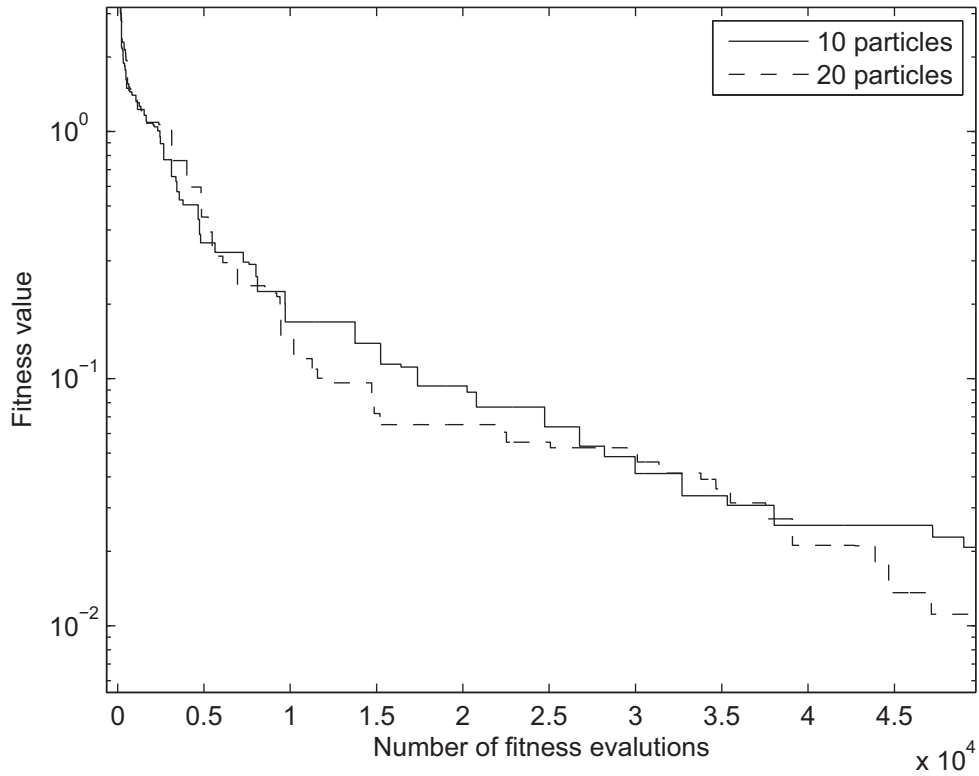


(a) Original results reproduction: 2D Modified Rosenbrock function optimization (Milano) - 10 and 20 particles - median from 30 runs

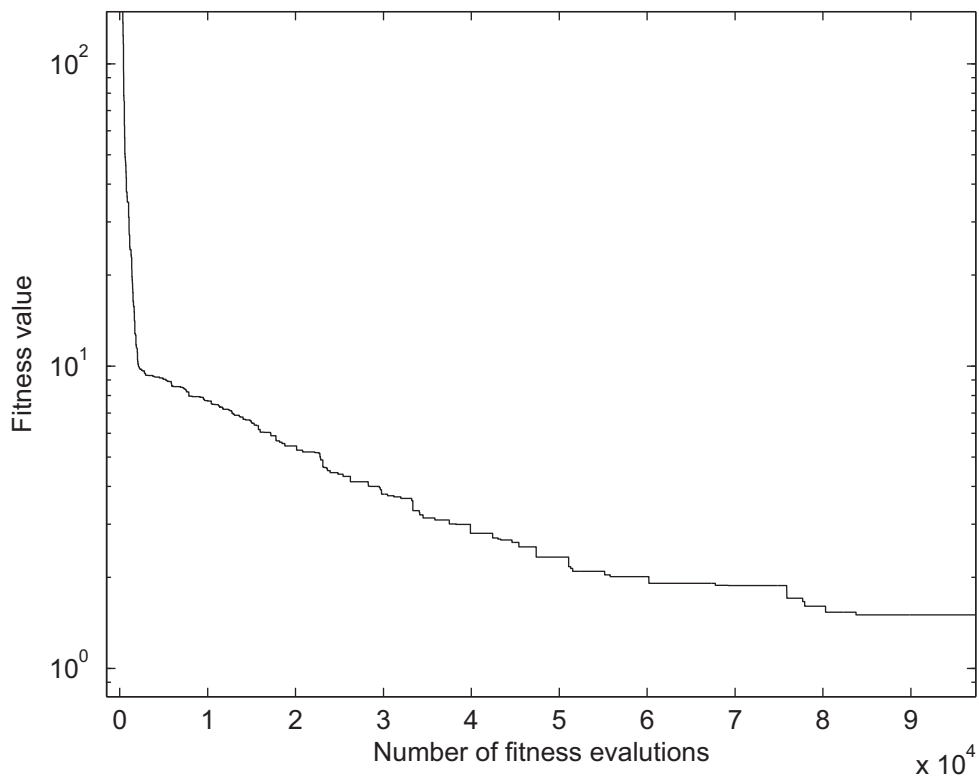


(b) Original results reproduction: 2D Griewank's function optimization (Milano) - 10 and 20 particles - median from 30 runs

Figure 1: Median BSF values dependency on fitness evaluation counts of "Milano" algorithm [1] - original adaptation rule



(a) Original results reproduction: 2D Rastrigin's function optimization (Milano) - 10 and 20 particles - median from 30 runs



(b) Original results reproduction: 10D Generalized Rosenbrock's function optimization (Milano) - 100 particles - median from 30 runs

Figure 2: Median BSF values dependency on fitness evaluation counts of "Milano" algorithm [1] - original adaptation rule

from best to worst, the worst item is removed, current fitness is compared to the P th value and if the current fitness value is better, the point where was sampled is used for adaptation.

I concentrated to the third case.

For the 2-D testing functions I made effort to preserve the original parameters, but there were some changes in repulsion factor. Increasing the repulsion factor was must due to getting stuck in local optima. The factor was set to 0.01 instead of 0.001 for all functions except Rastrigin's function optimized by 10 particles. There are obviously two new parameters added. These can be seen in table 5.

Table 5: Parameters for the new Milano adaptation rule

P	Q
10	120

For the 10-D testing function, however, the original parameters do not work, so that I looked for some working candidates. The best I found for the 10-D Rosenbrock's function are written in table 6, but it was also necessary to increase maximum evaluations count to 2×10^5 . Next thing I discovered is, that for unimodal functions is good, when the deviation size in sampling operator (see section 2.2) is divided by 2 (*deviation halving*). For the Rastrigin's function in 2-D the algorithm with 20 particles does not converge.

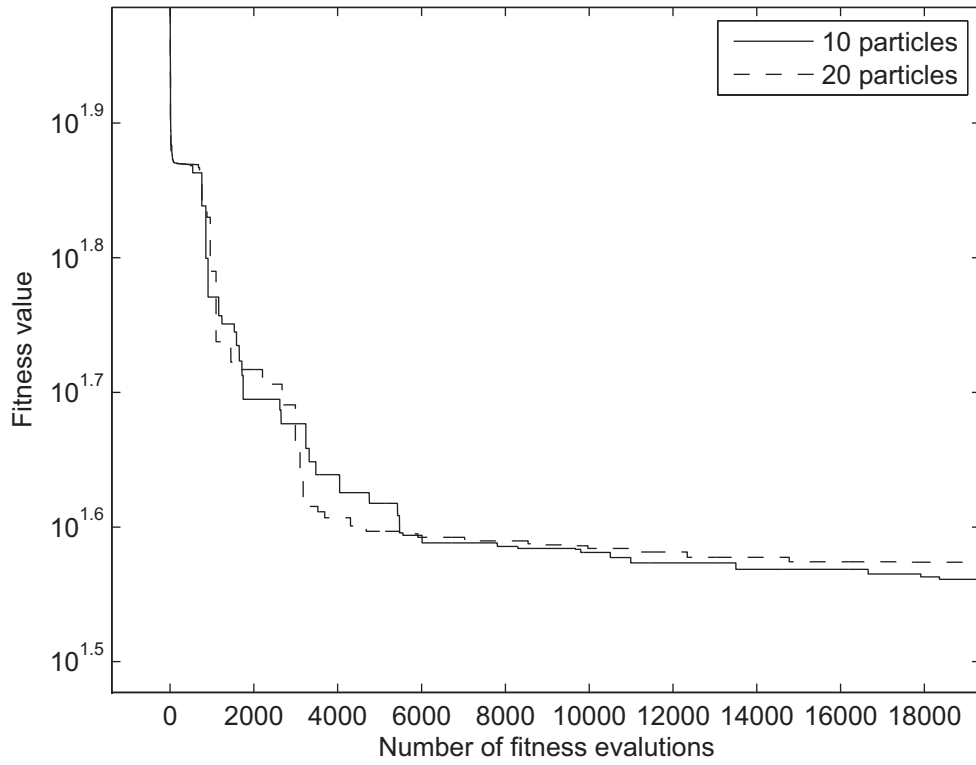
Table 6: Milano settings - best found for 10-D Rosenbrock - (nh. - neighborhood; lr. - learning)

Node count	lr. rate	nh. range	repulsion factor	fitness order	queue length
N	α	ϵ	K	P	Q
150	0.9	$N/5$	10^{-7}	5	25

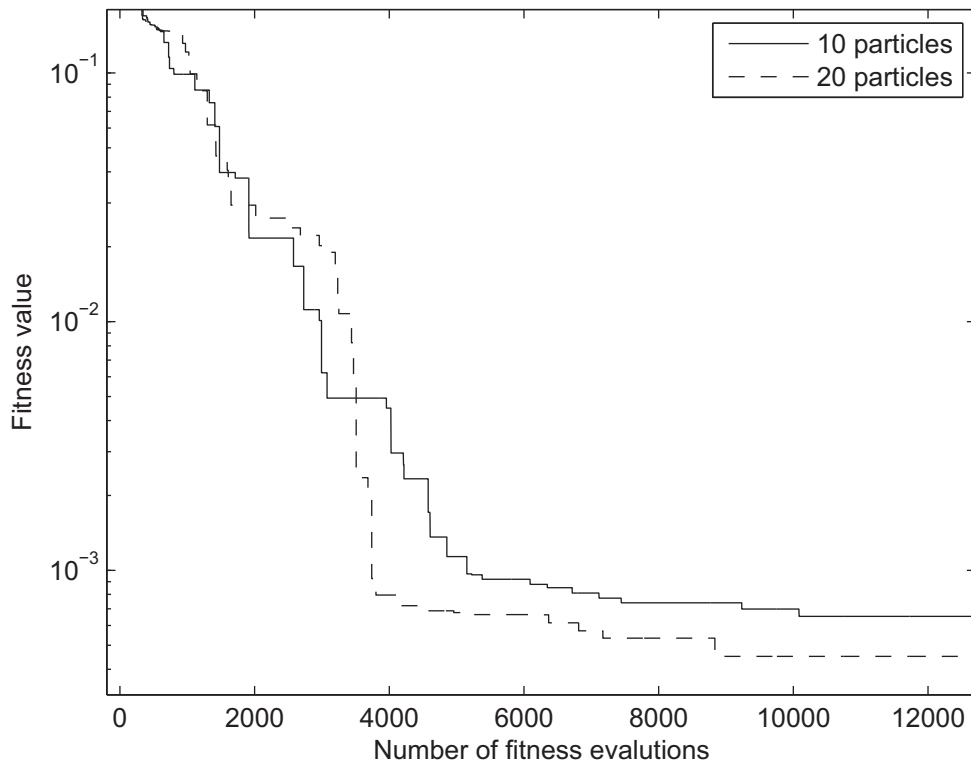
Although the convergence was finally reached for all functions, fitness evaluation count is higher than they accomplished in the article.

Table 7: Original results reproduction for the "Milano" algorithm - original adaptation rule - statistics for fitness BSF values after end of run

Function	Settings	min	$p_{0.25}$	med	$p_{0.75}$	max
2-D Modified Rosenbrock	Set 1	34.0493	34.0785	34.1413	34.4626	37.0912
2-D Griewank	Set 1	2.7×10^{-5}	0.0051	0.0179	0.0377	0.1606
2-D Rastrigin	Set 1	8.3×10^{-4}	0.0133	0.0208	0.0797	0.1910
2-D Modified Rosenbrock	Set 2	34.0438	34.1068	34.1384	34.3620	34.7353
2-D Griewank	Set 2	8.9×10^{-5}	0.0013	0.0087	0.0216	0.1694
2-D Rastrigin	Set 2	0.0010	0.0058	0.0111	0.0524	0.5308
2-D Generalized Rosenbrock	Set 3	0.6505	1.6782	2.3353	3.3727	5.9009

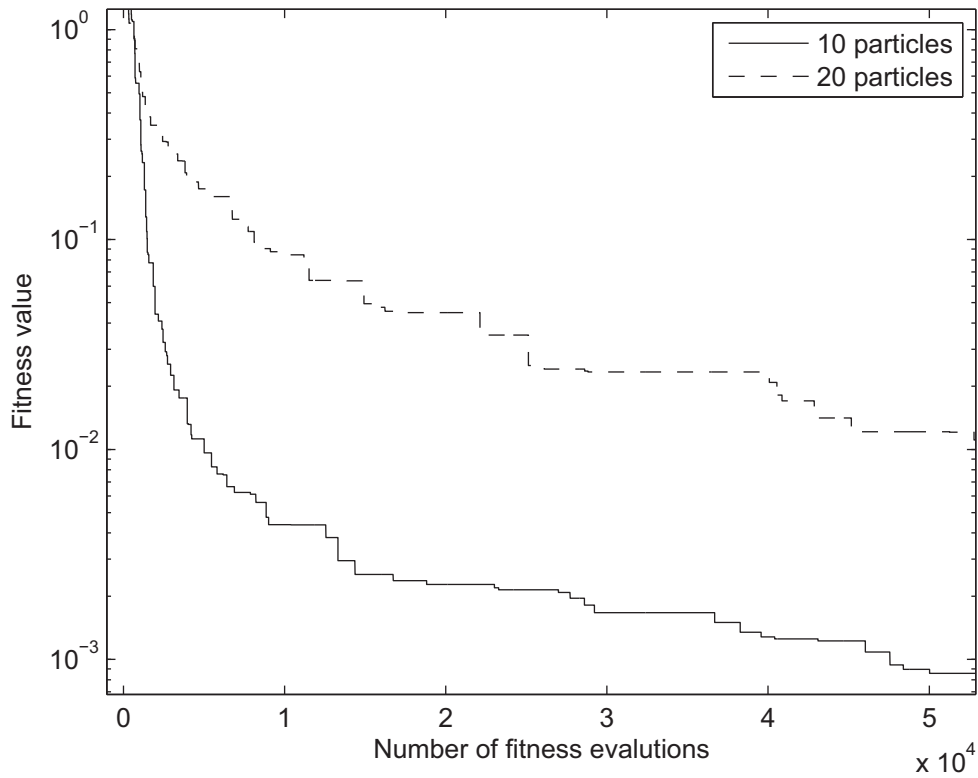


(a) Original results reproduction: 2D Modified Rosenbrock function optimization (Milano) - 10 and 20 particles - median from 30 runs

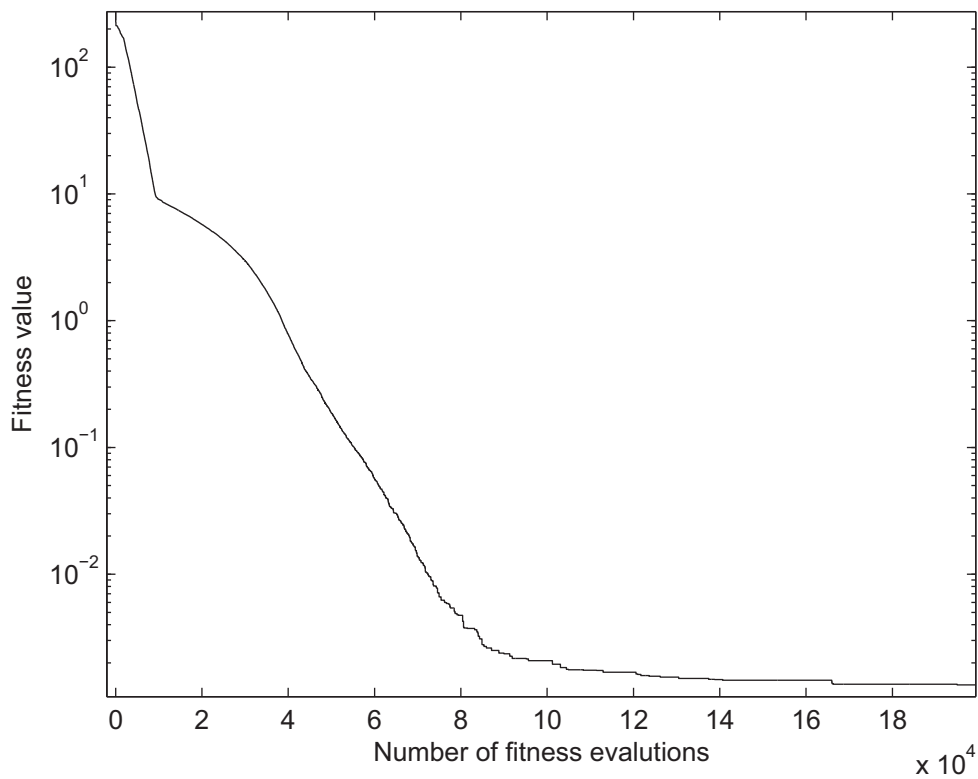


(b) Original results reproduction: 2D Griewank's function optimization (Milano) - 10 and 20 particles - median from 30 runs

Figure 3: Median BSF values dependency on fitness evaluation counts of "Milano" algorithm [1] - new adaptation rule



(a) Original results reproduction: 2D Rastrigin's function optimization (Milano) - 10 and 20 particles - median from 30 runs



(b) Original results reproduction: 10D Generalized Rosenbrock's function optimization (Milano) - new settings were found for this function, see table 6

Figure 4: Median BSF values dependency on fitness evaluation counts of "Milano" algorithm [1] - new adaptation rule

Table 8: Original results reproduction for the "Milano" algorithm - new adaptation rule - statistics for fitness BSF values after end of run

Function	Settings	min	$p_{0.25}$	med	$p_{0.75}$	max
2-D Mod. Ros.	Set 1 - new	34.2950	35.3660	36.4015	38.3448	39.8888
2-D Griewank	Set 1 - new	1.0×10^{-4}	4.7×10^{-4}	6.3×10^{-4}	8.3×10^{-4}	0.0314
2-D Rastrigin	Set 1 - new	3.9×10^{-5}	5.4×10^{-4}	8.6×10^{-4}	0.0018	0.0052
2-D Mod. Ros.	Set 2 - new	34.1129	35.4498	37.4613	38.6536	39.7126
2-D Griewank	Set 2 - new	4.0×10^{-5}	1.6×10^{-4}	4.5×10^{-4}	6.9×10^{-4}	9.5×10^{-4}
2-D Rastrigin	Set 2 - new	8.1×10^{-4}	0.0049	0.0111	0.0190	0.0685
2-D Gen. Ros.	Set 3 - new	8.0×10^{-4}	9.1×10^{-4}	9.8×10^{-4}	0.0013	3.9871

3.2 Huhse

As for Milano, I tried to run the algorithm as it is described, using the authors' parameter settings first.

3.2.1 Experiment Settings

The authors use two settings. These are took down in table 9.

Table 9: Original Huhse settings (N - particle count; n_O - mutation offspring count; t_{max} - stopping time; σ_i - initial neighborhood range; σ_f - final neighborhood range; ϵ_i - initial learning rate; ϵ_f - final learning rate)

Set	N	n_O	t_{max}	σ_i	σ_f	ϵ_i	ϵ_f
Set 1	10	4	30000	5	0.05	0.02	0.01
Set 2	10	4	30000	5	0.05	0.0075	0.00375

They tested the algorithm using 17 standard testing functions. Some of them are quite complicated to implement, and so I used only a subset of them. In addition, the omitted functions are mostly more difficult, than those implemented, for example the Rosenbrock's function is difficult to optimize by the same way as the Fletcher and Powell's function, but F. and P. is harder due to its large nearly constant table, which slopes down to the global optimum very slowly, on the other hand Shekel's functions are highly multimodal, so I suppose that the algorithm will not work for Shekel if it fails for Rastrigin and so on. The functions used are brought out in table 10.

Table 10: Test functions used for Huhse benchmarking and average number of fitness evaluations to convergency for better parameter settings (taken from [2])

Function	Settings	Fitness evaluations
Sphere model	Set 1	30010
Generalized Rosenbrock's function	unavailable	does not conv.
Schweffel's double sum	Set 1	157210
Ackley's function	Set 1	72010
Griewank function	Set 1	44454
Paralel axes hyperellipsoid	unavailable	does not conv.
Sum of different powers	unavailable	does not conv.

All functions were of dimensionality 20. The next fact I suppose is that if the optimization is unsuccessful in 10-D, it will not work neither for higher dimensions.

3.2.2 Results

Running the series of tests did not fare well for the algorithm. Huhse sets the convergency threshold to be 10^{-10} . For 20-D there was actually almost no successful run on any function, neither little bit worse, nor still acceptable (I determined the value 0.001 from the Milano’s article as ”acceptable”). You can find convergency curves on the following pages (figures 5(a), 5(b), 6(a), 6(b), 7(a), 7(b), 8(a)). The approach is quite good for unimodal functions, but totally fails while used on multimodal problems.

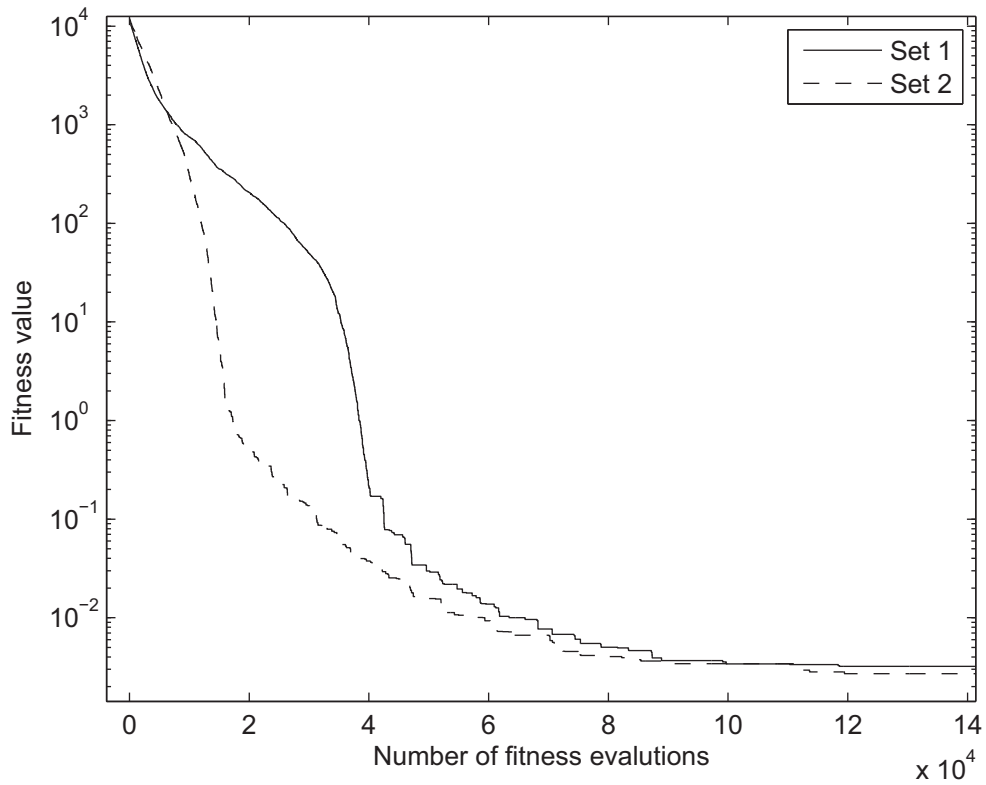
Table 11: Original results reproduction for the ”Huhse” algorithm - statistics for fitness BSF values after end of run (SDS - Schwefel’s Double Sum; SDP - Sum Of Different Powers; Hyp. - Hyperellipsoid; Ros. - Rosenbrock’s function) - All functions are defined in the Appendix

Function	Settings	min	$p_{0.25}$	med	$p_{0.75}$	max
20D Sphere	Set 1	3.0×10^{-5}	5.6×10^{-4}	0.0032	0.0685	0.1852
10D SDS	Set 1	3.8×10^{-7}	8.3×10^{-6}	7.0×10^{-5}	3.8×10^{-4}	0.0092
20D Hyp.	Set 1	7.4×10^{-6}	2.1×10^{-4}	0.0014	0.0033	0.0913
20D SDP	Set 1	2.7×10^{-4}	5.7×10^{-4}	7.5×10^{-4}	0.0012	0.0026
10D Ros.	Set 1	0.8485	1.3413	2.1129	3.0634	6.6296
2D Ackley	Set 1	0.0027	0.0145	0.0256	0.0410	0.0752
2D Griewank	Set 1	0.0023	0.0072	0.0096	0.0165	0.0330
20D Sphere	Set 2	4.3×10^{-4}	9.8×10^{-4}	0.0027	0.0088	0.0563
10D SDS	Set 2	5.1486×10^{-4}	0.0011	0.0015	0.0032	0.0595
20D Hyp.	Set 2	8.2×10^{-5}	3.7×10^{-4}	0.0032	0.0069	0.0299
20D SDP	Set 2	1.5×10^{-4}	6.6×10^{-4}	7.8×10^{-4}	9.5×10^{-4}	0.0019
10D Ros.	Set 2	0.8250	1.4233	7.9292	2.2256	9.1228
2D Ackley	Set 2	0.0048	0.0231	0.0344	0.0652	0.1177
2D Griewank	Set 2	0.0015	0.0074	0.0095	0.0125	0.0539

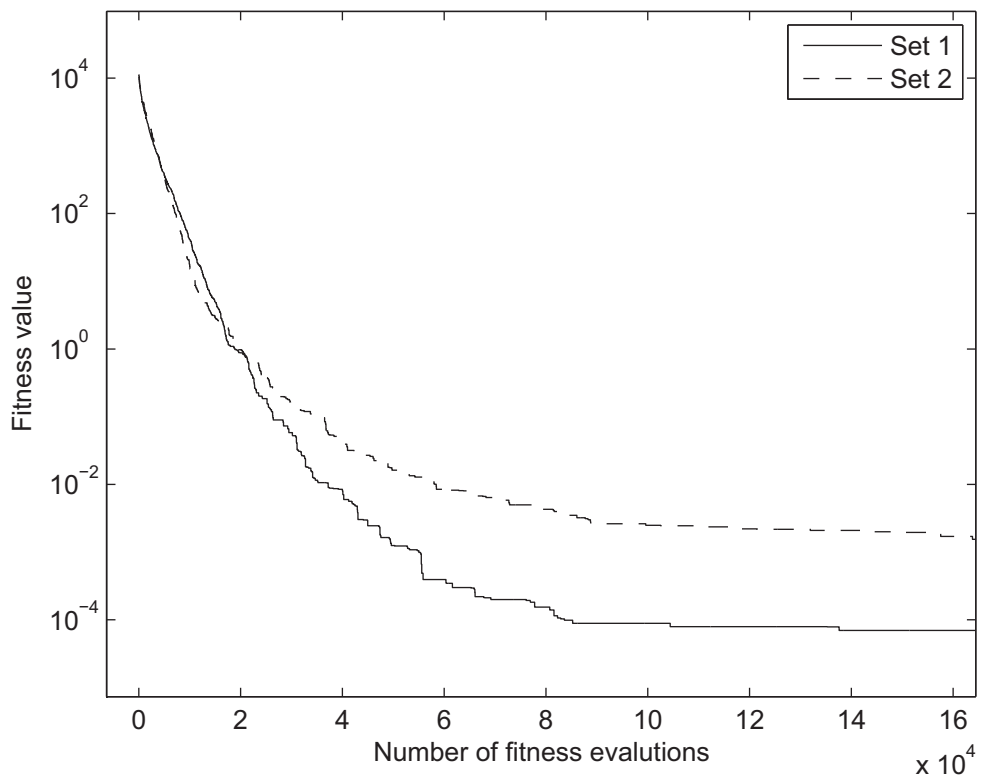
3.3 Summary of Implementation Results

Milano Although the authors of both articles promised rather good performance on both unimodal and multimodal functions, none of the algorithms reached almost even comparable results. As said above, Milano maybe mistaken while specifying the adaptation condition. The metod looks hopeful, but there is still a need for exploration about when to adapt. Adding parameter time dependency should further make the better performance. There was an attempt to contact the authors, but they did not answered.

Huhse There are some not very precise formulations, which obviously caused me to understand things a different way than they were written. Especially the population management and selection is not well described, and there are some more minor unanswered questions. The authors were contacted too, but their response was that it was actual some years ago and the implementation details and source codes are gone now.

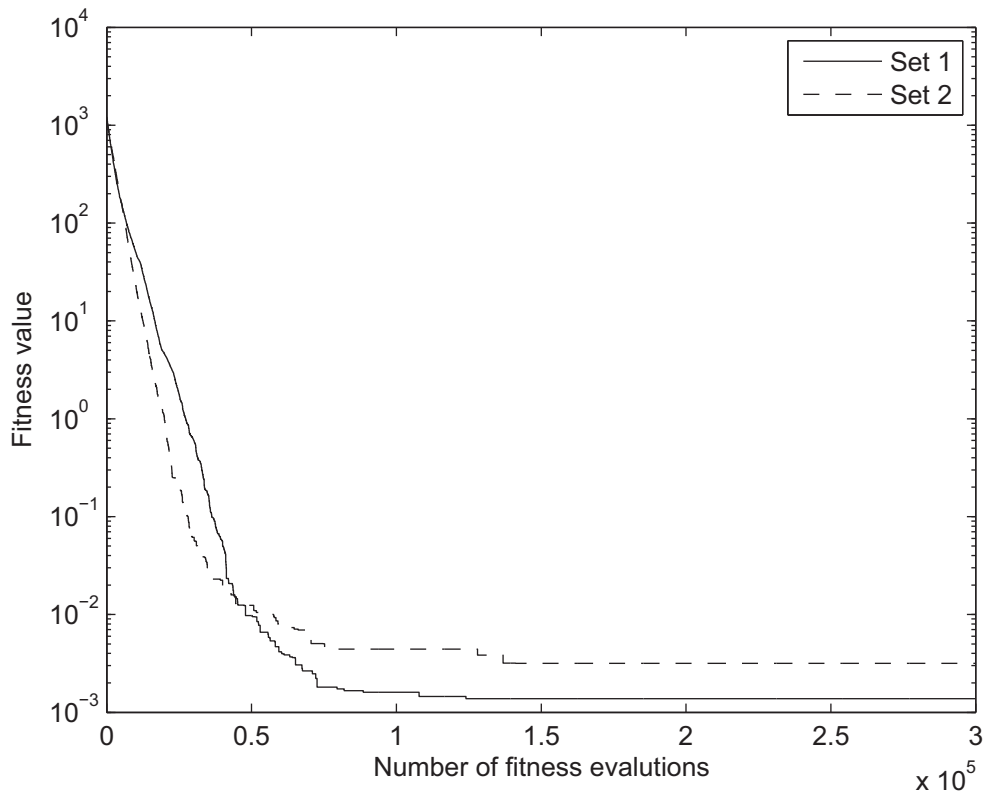


(a) Original results reproduction: 20D Sphere function optimization (Huhse) - both settings - median from 30 runs

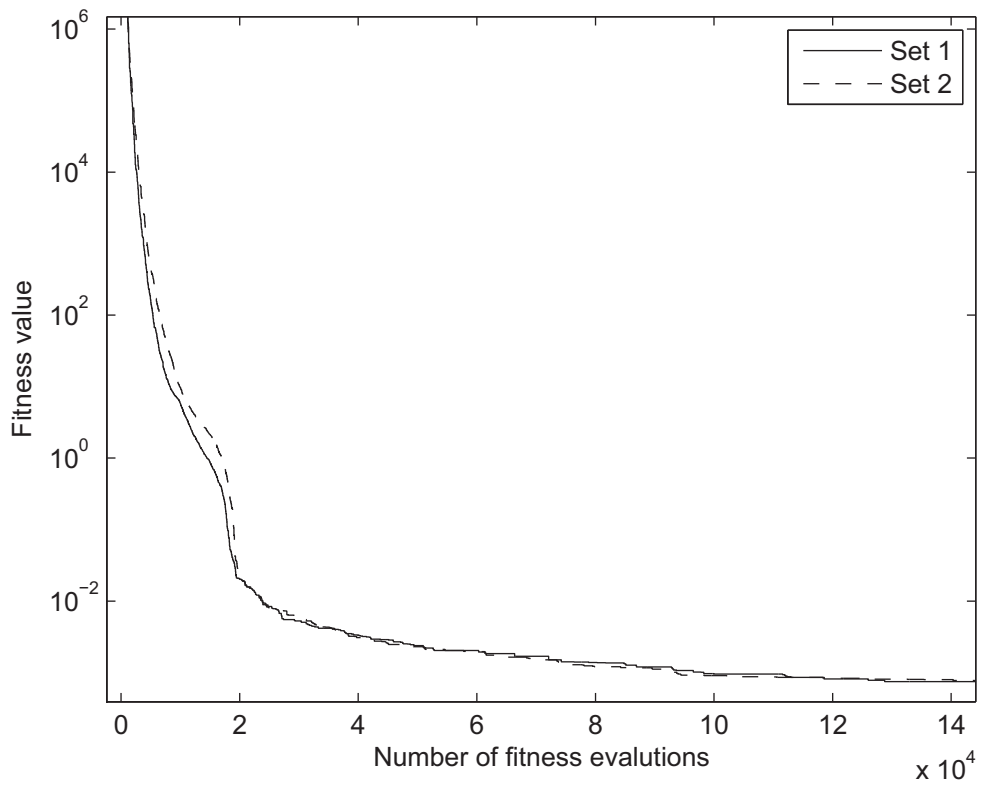


(b) Original results reproduction: 10D Schwefel's Double Sum function optimization (Huhse) - both settings - median from 30 runs

Figure 5: Median BSF values dependency on fitness evaluation counts of "Huhse" algorithm [2]

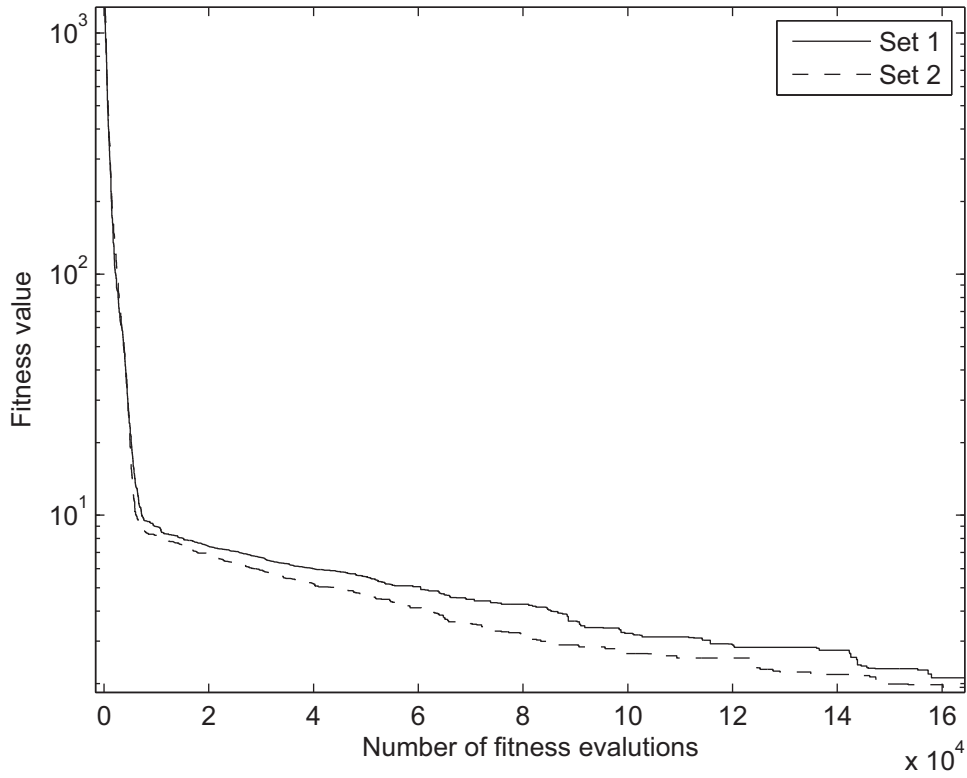


(a) Original results reproduction: 20D Parallel Axes Hyperellipsoid function optimization (Huhse) - both settings - median from 30 runs

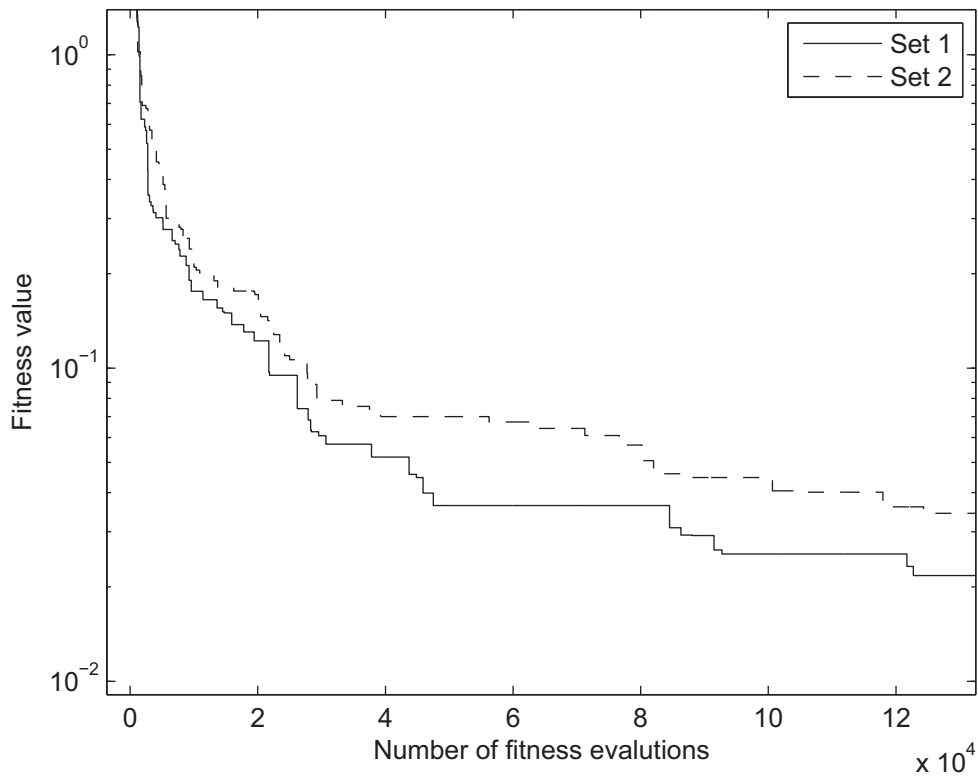


(b) Original results reproduction: 20D Sum of Different Powers function optimization (Huhse) - both settings - median from 30 runs

Figure 6: Median BSF values dependency on fitness evaluation counts of "Huhse" algorithm [2]

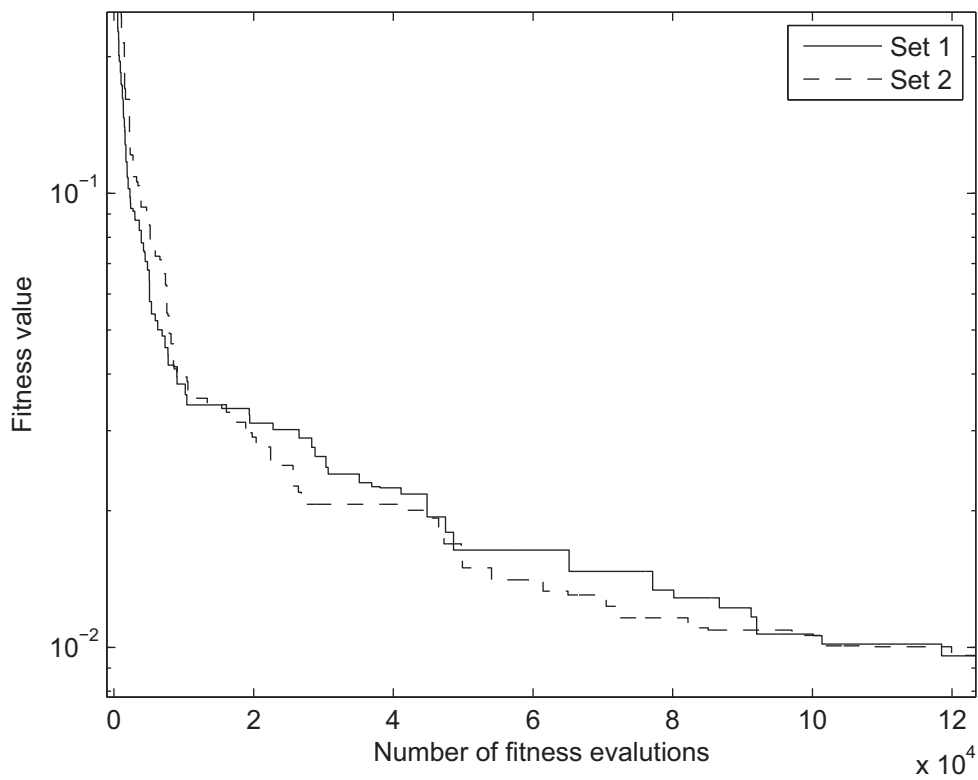


(a) Original results reproduction: 10D Rosenbrock's function optimization (Huhse) - both settings - median from 30 runs



(b) Original results reproduction: 2D Ackley's function optimization (Huhse) - both settings - median from 30 runs

Figure 7: Median BSF values dependency on fitness evaluation counts of "Huhse" algorithm [2]



(a) Original results reproduction: 2D Griewank function optimization (Huhse) - median from 30 runs

Figure 8: Median BSF values dependency on fitness evaluation counts of "Huhse" algorithm [2]

4 Experimental Comparison

The primary purpose of this section should have been to compare the robustness and performance of both algorithms. If they had work as described, there would have been comparison of the algorithms for more functions of various dimensionalities both uni- and multimodal.

As the algorithms do not perform as promised in the articles, I concentrated to use only 8 functions used in the articles, where the power of the algorithms is rather comparable.

4.1 Experiment settings

4.1.1 Milano

There was a lot of time and effort spent on finding some working and universal set of parameters. During this time I considered, that the Milano algorithm is quite sensitive to even small changes of parameters. For example increasing P by one can cause the algorithm to be unable to converge.

There were also found basically two ways, how to set up the Milano approach:

- Learning rate α lower than approx. 0.45 and neighborhood range ϵ somewhere around $n/3$ - rather useless for multimodal functions, while this neighborhood range causes too fast contraction. This can be tried to be outweighed by increasing the repulsion factor K , but the contraction is often needed in the final iterations to concentrate trials utmost close to the optimum.
- Learning rate α equal or higher than 0.9 and neighborhood range ϵ set approx. in the interval $(n/25, n/5)$ - Nodes close to the stimulus are strongly attracted, but this strength quickly decreases with distance. Quite good for multimodal functions, because of this quick decreasing, which leaves far nodes almost unaffected, so even if there are many stimuli from one point, the far nodes preserve a global view of the search space.

I actually found some setting, which makes the algorithm able to optimize 10-D Ackley, but this was tuned only for this function and did not work for others. So after all I finally experimented with setting, working for used multimodal functions only in 2-D (quite easy to find working setting for this dimensionality), because the Huhse approach was unable to successfully optimize even 2-D multimodal functions so that there would be nothing to compare to.

For unimodal functions I used a setting found for 10-D Rosenbrock (table 6, including sampling deviation halving (described in section 3.1.2 on page 11), which showed itself to be applicable to the rest of unimodal functions.

For multimodal functions I used the original settings (table 1) with the new adaptation rule with queue settings as written in table 5.

4.1.2 Huhse

For each tried setting, this approach highly suffers from getting stuck in local optima, although the author claim, that the used selection scheme should help to avoid that. You can find a snapshot in figure 4.2 of particle being stucked while optimizing 2-D Ackley function, using setting 1 (see table 9). Remember that this setting uses 10 particles, but the picture appears as only with six particles. This is one of the

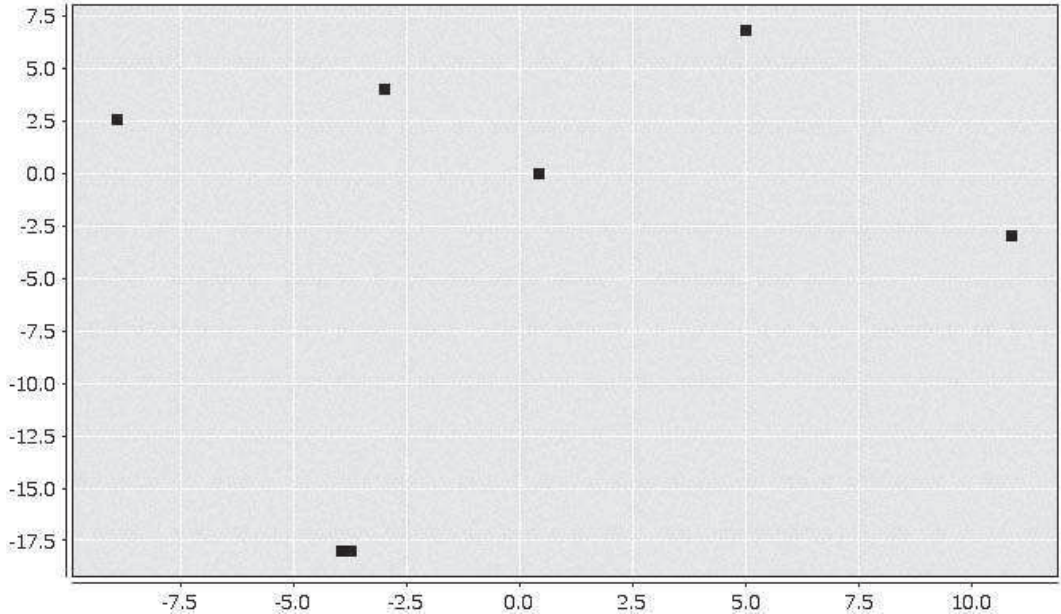


Figure 9: Snapshot of particle positions after 300000 fitness function evaluations - "Huhse" algorithm optimizing 2-D Ackley function

biggest drawbacks of the algorithm identified. There are 10 particles, but some of them are stuck together as seen on the picture near $[-5, -15.5]$. This fact lowers the population diversity. More particles did not help, it only caused more of them to stick together.

There is actually a setting, making this method rather good for unimodal functions. It is presented in table 12. The diversity of population may be point of discussion, because this setting uses only 3 particles.

Table 12: Huhse settings, good for unimodal functions (N - particle count; n_O - mutation offspring count; t_{max} - stopping time; σ_i - initial neighborhood range; σ_f - final neighborhood range; ϵ_i - initial learning rate; ϵ_f - final learning rate)

Set	N	n_O	t_{max}	σ_i	σ_f	ϵ_i	ϵ_f
Set U	3	10	60000	3	2	0.7	0.6

So, for unimodal functions I used this setting or Set 1 from table 9. For multimodal functions I used Set 1 from table 9.

4.2 Both algorithms

Considering the behaviour of the algorithms, I decided to implement fitness value limit 0.001 in most cases for both. It is the original limitation of the Milano algorithm and it is true, that the algorithm is hardly able to reach better results for all tested functions. I only switched this criterion off for Huhse on some unimodal functions, where I considered it to behave quite well.

4.3 Comparison results

According to the articles, the Huhse approach should be better than Milano. But because both algorithms do not work as expected, the power of each is arguable.

The first obvious fact is, that the Huhse algorithm is better for unimodal functions and totally fails when used on multimodal functions.

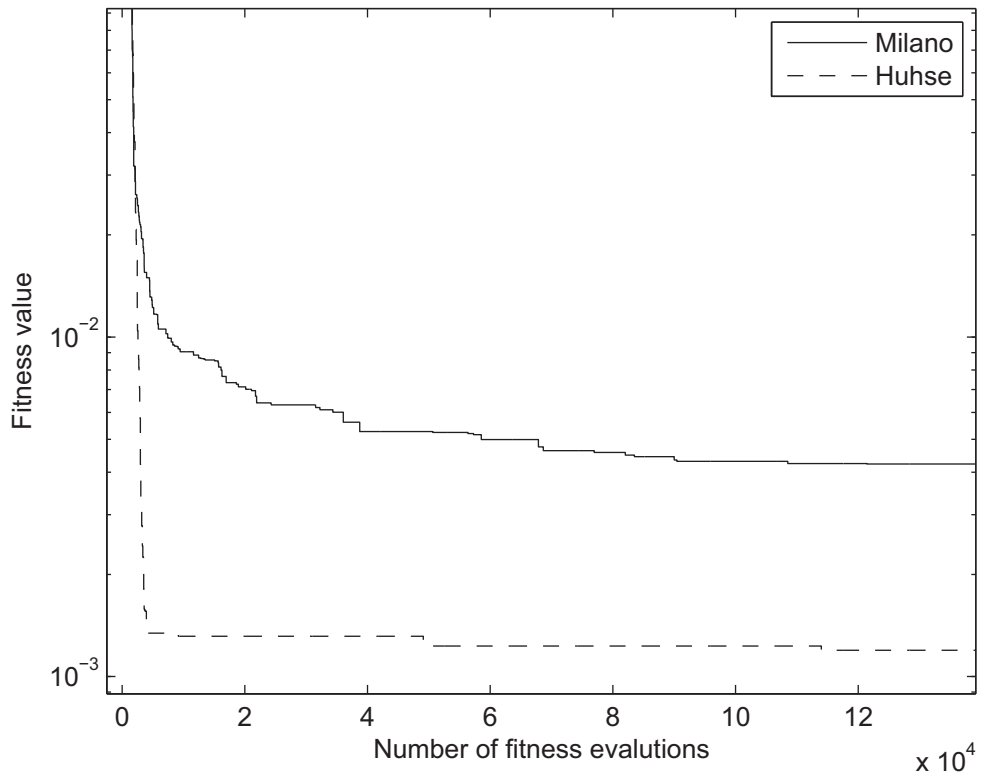
The Milano algorithm is quite good for unimodal as well as for multimodal functions, but the convergence limit 0.001 is too loose for practical applications.

You can find the algorithms compared by their convergence curves on the figures 10(a), 10(b), 11(a), 11(b), 12(a), 12(b), 13(a), 13(b). Charts are zoomed the way, that there is only the interesting part of the behaviour visible. All runs had maximum fitness evaluation count restricted to 3×10^5 , excepting 10-D Rosenbrock's function for Milano, where this count was 2×10^5 .

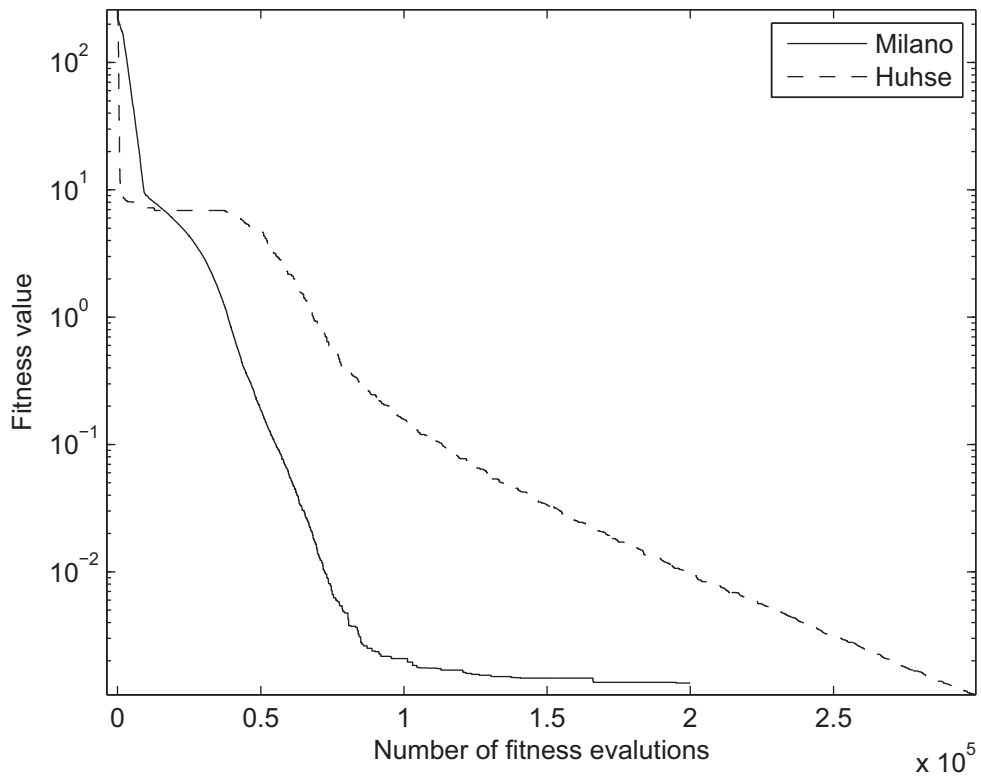
Numeric form of the results can also be found in the table 13.

Table 13: Statistical comparison of the "Miano" and "Huhse" approach optimizing some well known test functions (SDS - Schwefel's double sum; SDP - Sum of different powers; Hyp. - Hyperellipsoid; Ros. - Rosenbrock) - (M - Milano; H - Huhse; R - setting for 10-D Rosenbrock (table 6); U - Setting for unimodal functions (table 12))

Function	Settings	min	$p_{0.25}$	med	$p_{0.75}$	max
10D Hyp.	M-Set R	0.0013	0.0023	0.0030	0.0037	0.0051
	H-Set U	0.0010	0.0011	0.0012	0.0015	110.2555
10D Ros.	M-Set R	7.8×10^{-4}	0.0010	0.0013	0.0015	3.9880
	H-Set U	0.0010	0.0010	0.0011	3.9885	623.3862
10D SDS	M-Set R	3.1×10^{-4}	7.6×10^{-4}	8.9×10^{-4}	0.0010	0.0013
	H-Set 1	3.8×10^{-7}	8.3×10^{-6}	7.0×10^{-5}	3.8×10^{-4}	0.0092
10D Sphere	M-Set R	2.5×10^{-4}	8.1×10^{-4}	9.0×10^{-4}	9.7×10^{-4}	0.0011
	H-Set 1	2.0×10^{-12}	3.1×10^{-10}	6.9×10^{-9}	1.5×10^{-6}	5.8×10^{-5}
20D SDP	M-Set R	2.7×10^{-4}	5.7×10^{-4}	7.5×10^{-4}	0.0012	0.0026
	H-Set U	0.0010	0.0011	0.0014	0.0018	7.6×10^9
2D Ackley	M-Set 1	6.0×10^{-5}	5.1×10^{-4}	6.6×10^{-4}	9.1×10^{-4}	9.8×10^{-4}
	H-Set 1	0.0027	0.0145	0.0258	0.0410	0.0752
2D Griewank	M-Set 1	1.0×10^{-4}	8.7×10^{-4}	9.1×10^{-4}	9.7×10^{-4}	0.0010
	H-Set 1	0.0023	0.0072	0.0096	0.0165	0.0330
2D Rastrigin	M-Set 1	2.6×10^{-6}	2.8×10^{-4}	5.7×10^{-4}	8.3×10^{-4}	0.9950
	H-Set 1	2.3×10^{-5}	0.0151	0.9954	1.0073	2.0112

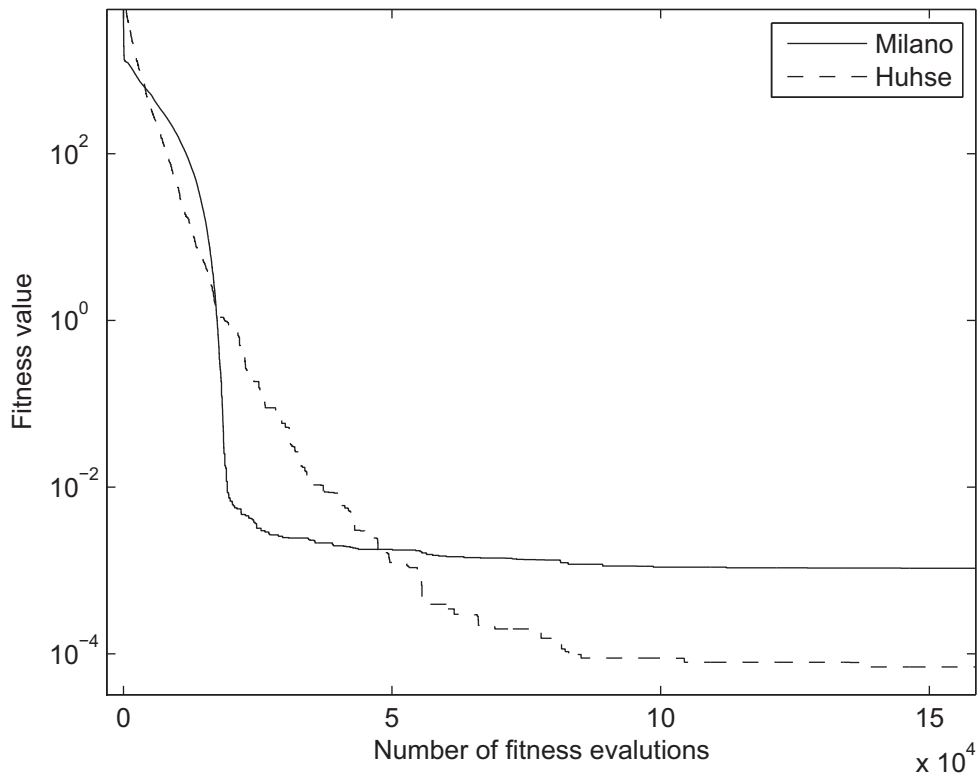


(a) Comparison of the algorithms optimizing 10-D Hyperellipsoid

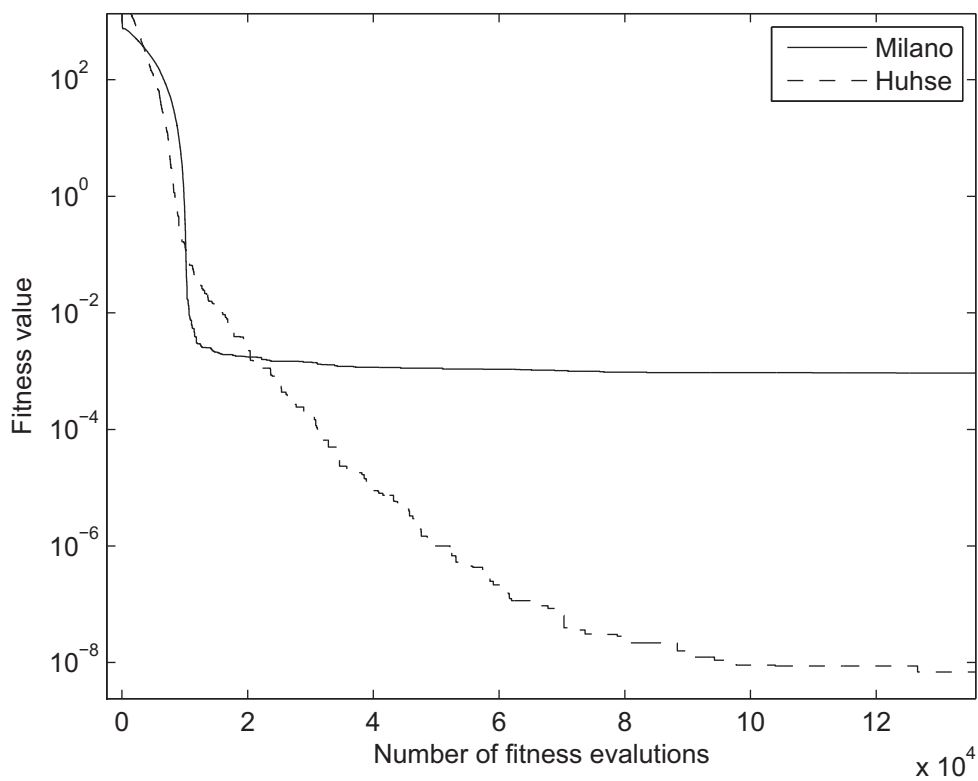


(b) Comparison of the algorithms optimizing 10-D Rosenbrock's function

Figure 10: Comparison of "Milano" and "Huhse" algorithms for 10-D Hyperellipsoid and 10-D Rosenbrock's function - Median of BSF fitness values - 30 runs

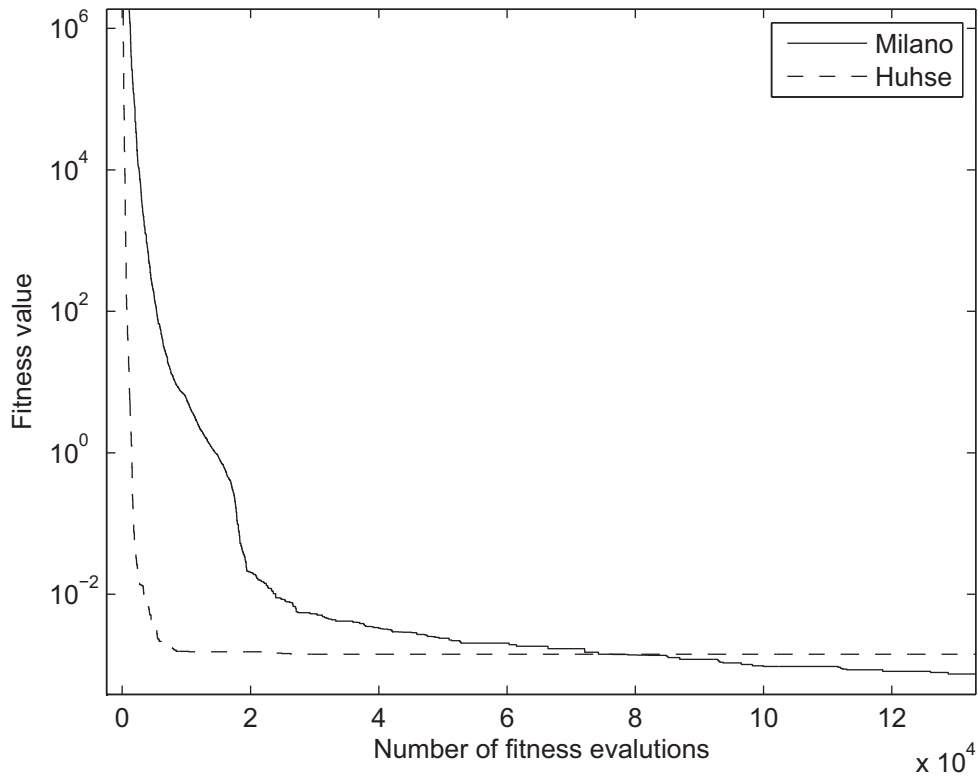


(a) Comparison of the algorithms optimizing 10-D Schwefel's double sum function

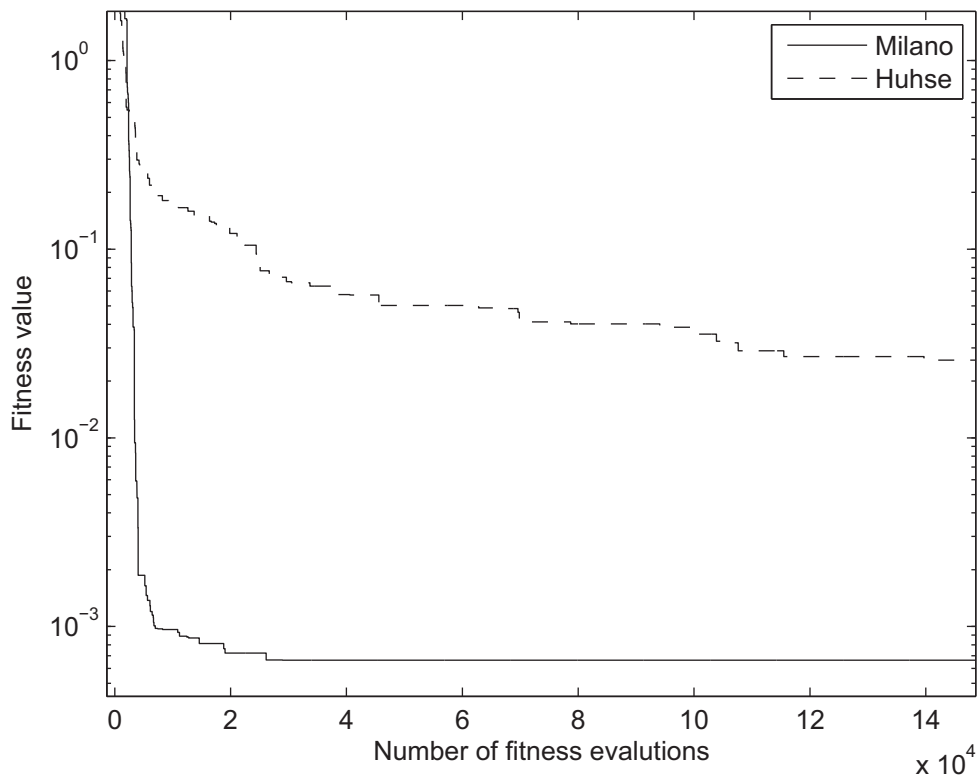


(b) Comparison of the algorithms optimizing 10-D Sphere function

Figure 11: Comparison of "Milano" and "Huhse" algorithms for 10-D Schwefel's double sum function and 10-D Sphere - Median of BSF fitness values - 30 runs

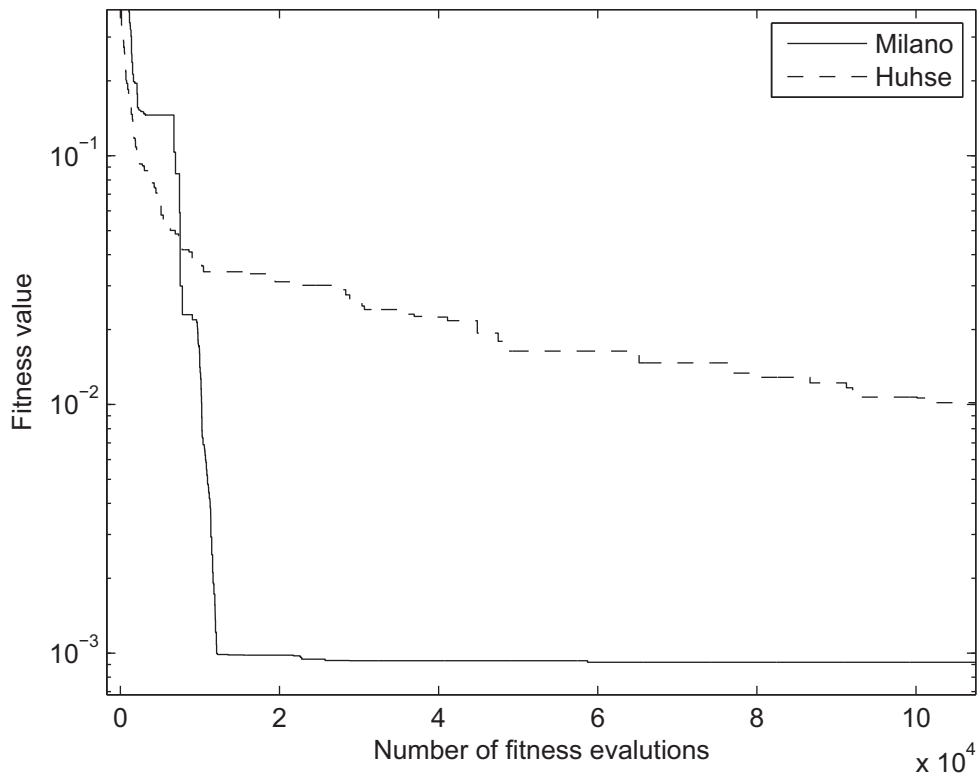


(a) Comparison of the algorithms optimizing 20-D Sum of different powers function

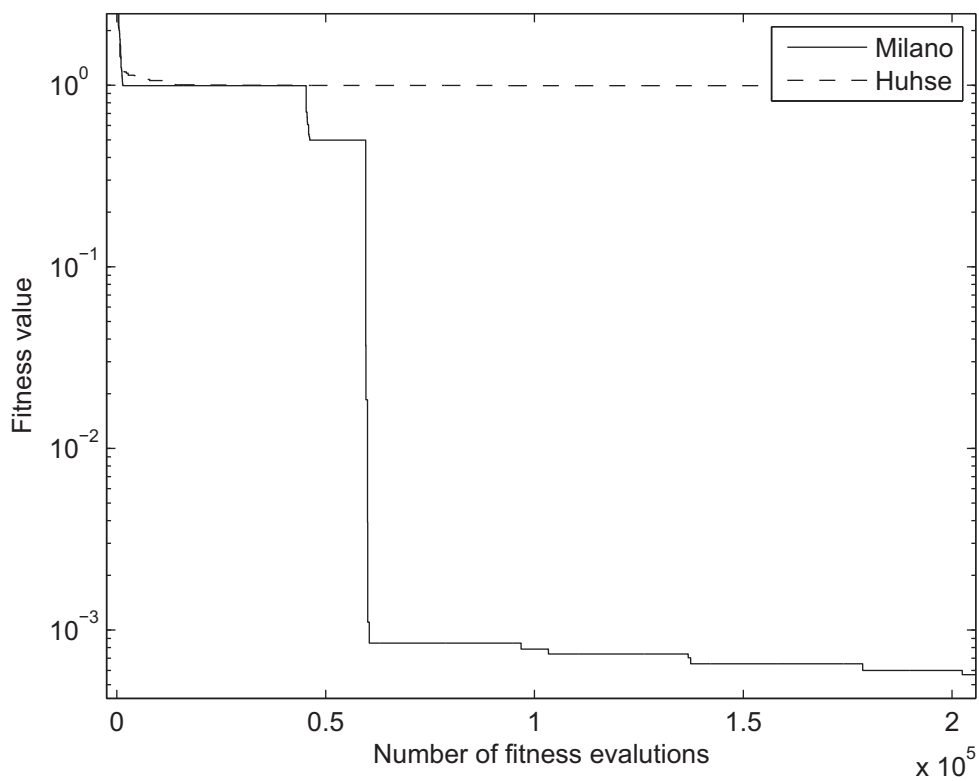


(b) Comparison of the algorithms optimizing 2-D Ackley's function

Figure 12: Comparison of "Milano" and "Huhse" algorithms for 20-D Sum of different powers function and 2-D Ackley's function - Median of BSF fitness values - 30 runs



(a) Comparison of the algorithms optimizing 2-D Griewank's function



(b) Comparison of the algorithms optimizing 2-D Rastrigin's function

Figure 13: Comparison of "Milano" and "Huhse" algorithms for 2-D Griewank's function and 2-D Rastrigin's function - Median of BSF fitness values - 30 runs

5 Summary and Conclusions

There are two algorithms using the Neural Gas by the particular way, found in the literature. I tried to understand them the best I could and consequently implemented them to see how they work.

Both algorithms were run on functions and with settings the authors described, but did not fulfill the expectations and their results were far from results presented in the articles, even with adjusted parameters/adaptation rule.

The algorithms were then compared to see, how they perform on various unimodal/multimodal functions.

The idea of using the Neural Gas self for the continuous optimization is quite nice and theoretically correct, but if it should be comparably powerful as other well-established optimization algorithms, there is still a lot of work to do. Even if the algorithms would be described with more clarity.

For Milano, the first step for making it better could be establishing time dependency, as originate from the Neural Gas algorithm self and adjusting the adaptation rule.

For Huhse, there should be performed some research respecting that the algorithm often gets stuck in local optima and make clear how the selection and the generation management actually works.

6 Appendix

6.1 Used testing functions

6.1.1 Functions used by Milano

- 2-D Modified Rosenbrock function optimization:

$$f(x, y) = 74 + 100(y - x^2)^2 + (1 - x)^2 - 400 \exp\left(-\frac{(x+1)^2 + (y+1)^2}{0.1}\right) \quad (10)$$
$$(x, y) \in (-2, 2) \times (-2, 2)$$

The article says, that the global optimum is at $(-1, -1)$ (calculated as 78) but there is value certainly lower. Consider point $(-0.9, -0.95)$ with value approx. 34.4 .

- 2-D Griewank's function:

$$f(x, y) = 1 + 0.005(x^2 + y^2) - \cos(x) \cos(2^{-0.5}y) \quad (11)$$
$$(x, y) \in (-100, 100) \times (-100, 100)$$

- 2-D Rastrigin's function:

$$f(x, y) = 20 + (x^2 - 10 \cos(2\pi x)) + (y^2 - 10 \cos(2\pi y)) \quad (12)$$
$$(x, y) \in (-5.12, 5.12) \times (-5.12, 5.12)$$

- 10-D Rosenbrock's function:

$$f(\vec{x}) = \sum_{i=1}^9 \left(100(x_{i+1} - x_i^2)^2 + (1 + x_i)^2\right) \quad (13)$$
$$x_i \in (-2, 2)$$
$$i = 1, \dots, 10$$

6.1.2 Some functions used by Huhse

- Sphere model

$$f(\vec{x}) = \sum_{i=1}^N x_i^2 \quad (14)$$
$$x_i \in (-5.12, 5.12)$$

- but I used bigger definition area $(-50, 50)$

- Generalized Rosenbrock's function - see function 13 with its sum working over all i from 1 to $N - 1$.
- Schwefel's double sum function

$$f(\vec{x}) = \sum_{i=1}^n \left(\sum_{j=1}^N x_j^2 \right) \quad (15)$$
$$x_i \in (-5.12, 5.12)$$

- Ackley's function

$$f(\vec{x}) = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2}\right) - \exp\left(\frac{1}{N} \sum_{i=1}^N \cos(c \cdot x_i)\right) + a + \exp(1) \quad (16)$$
$$x_i \in (-5.12, 5.12)$$

- Generalized Rastrigin's function

$$f(\vec{x}) = 10n + \sum_{i=1}^N (x_i^2 - 10 \cos(2\pi x_i))$$

$$x_i \in (-5.12, 5.12)$$
(17)

- Generalized Griewank's function

$$f(\vec{x}) = \frac{1}{4000} \sum_{i=1}^N x_i^2 - \prod_{i=1}^N \cos \frac{x_i}{\sqrt{i}} + 1$$

$$x_i \in (-100, 100)$$
(18)

- but more often described as with definition limits $(-600, 600)$

- Hyperellipsoid, paralel to axes

$$f(\vec{x}) = \sum_{i=1}^N i \cdot x_i^2$$

$$x_i \in (-5.12, 5.12)$$
(19)

- Sum of different powers

$$f(\vec{x}) = \sum_{i=1}^N |x_i|^{i+1}$$

$$x_i \in (-5.12, 5.12)$$
(20)

6.2 Used software

All charts were created using the MATLAB R2008b software.

The algorithms were implemented in the Java programming language using the NetBeans 6.5 IDE.

The document self was typed in TeXnicCenter and rendered by MiKTeX.

6.3 Contents of attached CD

The whole document is on the CD in the PDF format. There is also the NetBeans 6.5 project folder with my implementation of algorithms.

References

- [1] M. Milano, P. Koumoutsakos, and J. Schmidhuber. *Self-Organizing Nets for Optimization*. IEEE Transactions On Neural Networks, vol. 15, No. 3, 2004
- [2] J. Huhse, T. Villmann, P. Merz, and A. Zell. *Evolution Strategy with Neighborhood Attraction Using a Neural Gas Approach*. Lecture Notes In Computer Science; Vol. 2439, Proceedings of the 7th International Conference on Parallel Problem Solving from Nature, Springer-Verlag, 2002
- [3] T. Martinetz, and K. Schulten. *A "Neural-Gas" Network Learns Topologies* . Artificial Neural Networks, Elsevier Science Publishers B.V. (North-Holland), 1991
- [4] S. J. Russel, and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Upper Saddle River, NJ: Prentice Hall, pp. 111-114, 2003
- [5] Hansen, N., Ostermeier, A. *Completely derandomized self-adaptation in evolution strategies*. Evolutionary Computation 9-2, pp. 159-195, 2001
- [6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimization by Simulated Annealing*. Science 220, 1983
- [7] J. Kennedy, and R. Eberhart. *Particle Swarm Optimization* . in Proc. of the IEEE int. Conf. on Neural Networks, Piscataway, NJ, pp. 1942-1948, 1995
- [8] K. Price, R. Storm, and J. Lampinen. *Differential Evolution - A Practical Approach to Global Optimization* . Springer, 2005
- [9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison - Wesley, 1989
- [10] T. Kohonen. *Self-organizing Neural Projections*. Neural Networks, Vol. 19, No. 6, Elsevier Science Ltd., 2006